LARAVEL BEYOND CRUD

Building larger-than-average web applications

Brent Roose

LARAVEL BEYOND CRUD

Learn how to build larger-than-average Laravel applications and maintain them for years to come.

Brent Roose

Table of contents

Foreword

by Freek Van der Herten

It's funny how from the outside, IT is often seen as an exact science. When starting as a developer, I often heard people say: "Oh, you're in IT, so you must be good at mathematics". I never understood why someone who is supposedly good at mathematics is the right person to fix a printer.

When you have some experience building applications, you know that IT is not an exact science. There are multiple valid solutions to a problem. Take five developers and ask them what the best way to set up a blog is. You'll likely get five different answers. Each suggestion will have its own set of tradeoffs.

On a lower level, many interesting choices need to be made that don't have a clear cut answer:

- Can we send a mail from a controller?
- How should this concept be named?
- Where should this piece of logic be saved?
- Should a class be made final?

There are no clear cut answers for these questions. It depends on a couple of things: the size of the project, the preference of individual team members, and the agreements that the team made.

Programmers have particular preferences on how things should be handled. Some prefer to work very strictly by sticking very close to the SOLID rules. Others tend to work more pragmatically. Most devs are not in one camp or the other. I see this as a scale.

There are different kinds of preferences as well. Some consider visual debt. Others don't care about that. Some might want to stick to the default structure of their favorite frameworks. Others like to use a custom organization.

There's no approach that is inherently right or wrong. Beautiful, maintainable projects can be built either more pragmatically, or very strictly. Most projects, however, are not built or maintained by a single person. Instead, there is a collection of people involved, who are all on different positions on the strict/pragmatic scale.

From experience, I can tell you that it's not a good idea to let each member use his or her preferred style on a project. You'll likely get a project that isn't programmed consistently, making it hard to add features or perform maintenance.

In a well-oiled team, members talk and discuss the pros and cons of each of their individual preferences. When reaching a consensus, it's a good idea to write it down, together with the reasons why a certain approach was picked. This can result in a set of guidelines that newcomers to the project or team can refer to. You can read our collection of guidelines on our website. It

contains our agreements on details, such as how to name things, and to use certain Laravel features.

The book you're holding now is an extension of these guidelines. You'll learn how our team organises larger than average projects. The knowledge in this book is the result of countless discussions between all team members. Our team has built and is building multiple large projects. By creating and maintaining these projects, we have discovered where it's safe to be pragmatic and where it's often best to stick to a strict approach.

We consider our guidelines and this book a living document. People, teams, and opinions change over time. Don't dogmatically keep following rules you agreed on in the past, but keep challenging them. New experiences should lead to new and improved agreements.

Instead of blindly following the ideas shared in the book, I encourage you to discuss the content with your colleagues. Maybe your team members are positioned somewhere else on the pragmatic/strict scale, and the agreements you make with your team should reflect that.

I hope you'll enjoy reading this book, and that you'll learn some cool things that can help you and your team build better projects.

Preface

I've been writing and maintaining several larger-than-average web applications for years now. Projects that take a team of developers to work on them for at least a year, oftentimes longer. In these cases, the well-known, default Laravel approach to building CRUD applications doesn't scale all that well. These projects become large, difficult to navigate and maintain.

So over the past few years I've looked at several architectures which would help me and our team improve the maintainability of these projects, as well as help make the development process easier, both for us and our clients. These are architectures and patterns like Domain Driven Design, Hexagonal Architectures, and Event Sourcing, amongst others.

Because most of our projects are large but not ginormous, these architectures as a whole are almost always overkill. So instead we took the best parts from all of them, and combined them in a pragmatic approach.

In this book, I'll write about the knowledge we gained over the years in designing these projects. I will take a close look at the Laravel way and what did and didn't work for us. This book is for you if you're dealing with those kinds of larger projects, and want practical and pragmatic solutions in

managing them. I will talk about theory, patterns, and principles, though everything will be in context of a real-life web application.

In this book I'll give you concrete solutions to real life problems, but its actual goal is to teach you a mindset. I want to share the process we went through, so that you can apply it to your problems, and come up with the solutions that work best for you.

About me

My name is Brent and I'm a 26-year-old web developer living in Belgium. I've been writing PHP professionally for the past 6 years, and have been programming since I was 13 years old.

As a professional, I've mainly worked on medium to large-sized web applications and API's. Right now I'm working with Laravel at a company called Spatie, and before that I worked with both Symfony and company-specific frameworks.

LARAVEL
BEYOND CRUD

Domain oriented Laravel

> Humans think in categories, our code should be a reflection of that.

First things first: I didn't come up with the term *"domain"* myself — I got it from the popular programming paradigm DDD, or *"domain driven design"*. It's a rather generic term, and according to the Oxford Dictionary, a *"domain"* can be described as *"A specified sphere of activity or knowledge"*.

While my use of the word *"domain"* won't exactly have the same meaning as in the DDD community, there are several similarities. If you're familiar with DDD, you'll discover these similarities throughout this book. I tried my best to mention any overlap and differences when relevant.

So, domains. You could also call them *"groups"*, *"modules"*, or as some people call them, *"services"*. Whichever name you prefer, domains describe a set of the business problems you're trying to solve.

Hang on... I realise I just used my first *"enterprisey"* term in this book: *"the business problem"*. Making your way throughout this book, you'll note that I did my best to steer away from the theoretical, upper-management, business side of things. I'm a developer myself and prefer to keep things practical. So another, simpler name would be *"the project"*.

Let's give an example: an application to manage hotel bookings. It has to manage customers, bookings, invoices, hotel inventories, etc.

Modern web frameworks teach you to take one group of related concepts, and split them across multiple places throughout your codebase: controllers with controllers, models with models, and so on. You get the deal.

So let's stop and think about that for a moment.

Has a client ever told you to *"work on the controllers now"*, or to *"focus on the models directory"*? No — they ask you to work on invoicing, customer management or booking features.

These groups are what I call domains. They aim to group concepts within your project that belong together. While this might seem trivial at first, it's more complicated than you might think. That's why part of this book will focus on a set of rules and practices to keep your code nicely ordered.

Obviously there's no mathematical formula I can give you because almost everything depends on the specific project you're working on. So don't think of this book as giving a fixed set of rules. Rather think of it as handing you a collection of ideas that you can use and build upon, however you like.

It's a learning opportunity, much more than a solution you can throw at whatever problem you encounter.

Domains and applications

If we're grouping ideas together, evidently the question arises: how far do we go? You could, for example, group everything invoice-related together: models, controllers, resources, validation rules, jobs, ...

However, this approach raises a problem in classic HTTP applications: there often isn't a one-to-one mapping between controllers and models. Granted, in REST APIs and for the majority of your classic CRUD controllers there might be a strict one-to-one mapping. Unfortunately there are exceptions to the rules, and those will give us a hard time.

Invoices, for example, are simply not handled in isolation; they need a customer to be sent to, they need bookings to invoice, etc.

That's why we need to make a further distinction between what is domain code, and what is not.

On the one hand there's the domain, representing all the business logic, and on the other hand, we have code that uses — that is, consumes — that domain to integrate it with the framework and exposes it to the end-user. Applications provide the infrastructure for end-users to access and manipulate the domain functionality in a user-friendly way.

Now, we will dedicate a chapter diving deeper into the differences between domain and application code, but right now it's already important to know that we will make a distinction between the two. I promise we will address several questions that might be popping up in your head right now, soon.

Domains in practice

So what does all of what I described above look like in practice? Domain code will consist of classes like models, query builders, domain events, validation rules and more; we will look at all these concepts in-depth.

The application layer will be one or several applications. Every application can be seen as an isolated app which is allowed to use all of that domain code. In general, applications don't talk to each other, at least not directly.

One example could be a standard HTTP admin panel, and another one could be a REST API. I also like to think of the console, Laravel's artisan, as an application of its own.

As a high level overview, here's what the folder structure of a domain-oriented project might look like:

```
One specific domain folder per business concept
src/Domain/Invoices/
      ├── Actions
      ├── QueryBuilders
      ├── Collections
      ├── DataTransferObjects
      ├── Events
      ├── Exceptions
      ├── Listeners
      ├── Models
      ├── Rules
      └── States

src/Domain/Customers/
      // …
```

And this is what the application layer would look like:

```
The admin HTTP application
src/App/Admin/
      ├── Controllers
      ├── Middlewares
      ├── Requests
      ├── Resources
      └── ViewModels

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

The console application
src/App/Console/
      └── Commands
```

You might have noticed that the above example doesn't follow the Laravel convention of `\App` as the single root namespace. Since applications are only part of our project, and because there can be several, it doesn't make sense to use `\App` as the root for everything.

If you do prefer to stay closer to Laravel's default structure, you're allowed to do that. Remember, this book isn't about providing you a fixed set of rules, it's about teaching a mindset. You're free to choose whatever patterns and solutions you apply to your problems, and which not.

If you still want to separate the root namespaces, you can do so by making a slight change in `composer.json`:

```json
{
    // …

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

Note that I also have a `Support` root namespace, you can think of that one as the dumping ground for all little helpers that don't belong anywhere. You will see some practical uses of the `Support` namespace in future chapters.

Unfortunately there's one more thing you need to do in order for Laravel to fully support your custom namespaces. By default, Laravel will look at the app/ folder containing all `App\` code, and this default is hard coded in the `\Illuminate\Foundation\Application` class.

Fortunately we can easily make our own version, like so:

```php
namespace App;

class Application extends \Illuminate\Foundation\Application
{
    protected $namespace = 'App\\';
}
```

And overwrite it in `bootstrap/app.php`, like so:

```php
use App\Application;

$app = (new Application(
    $_ENV['APP_BASE_PATH'] ?? dirname(__DIR__)
))->useAppPath('src/App');
```

Again, you're not required to do this if you want to stick with Laravel's default structure.

Whatever folder structure you use, it's very important that you start thinking in groups of related business concepts, rather than in groups of code with the same technical properties.

Within each group — each domain — there's room to structure the code in whatever ways that make sense within that specific group. The first part of this book will look closely at how domains can be structured internally and which patterns can be used to help you keep your codebase maintainable as it grows over time. After that, we'll look at the application layer: how the domain can be consumed by it, and how we improve upon existing Laravel concepts by using, for example, view models.

There's a lot of ground to cover, so let's dive in!

Working with data

I like to think of that simple idea of grouping code in domains — which I explained in the previous chapter — as a conceptual foundation we can use to build upon. You'll notice that throughout the first part of this book, this core idea will return again and again.

The very first building block we'll be laying upon this foundation, is once again so simple in its core, yet so powerful: we're going to model data in a structured way; we're going to make data a first-class citizen of our codebase.

You probably noticed the importance of data modelling at the start of almost every project you do: you don't start building controllers and jobs, you start by building, what Laravel calls, models. Large projects benefit from making ERDs and other kinds of diagrams to conceptualise what data will be handled by the application. Only when that's clear, you can start building the entry points and hooks that work with your data.

The goal of this chapter is to teach you the importance of that data flow. We're not even going to talk about models. Instead, we're going to look at simple plain data to start with, and the goal is that all developers in your team can write code that interacts with this data in a predictable and safe way.

To really appreciate all the benefits we'll get from applying a simple data-oriented pattern, we'll need to dive into PHP's type system first.

Type theory

Not everyone agrees on the vocabulary used when talking about type systems. So let's clarify a few terms in the way that I will use them here.

The strength of a type system — strong or weak types — defines whether a variable can change its type after it was defined. A simple example: given a string variable `$a = 'test';` a weak type system allows you to re-assign that variable to another type, for example `$a = 1`, an integer.

PHP is a weakly typed language. Let's look at what that means in practice.

```php
$id = '1'; // E.g. an id retrieved from the URL

function find(int $id): Model
{
    // The input '1' will automatically be cast to an int
}

find($id);
```

To be clear: it makes sense for PHP to have a weak type system. Being a language that mainly works with a HTTP request, everything is basically a string.

You might think that in modern PHP, you can avoid this behind-the-scenes type switching — type juggling — by using the strict types feature, but that's not completely true. Declaring strict types prevents other types being passed

into a function, but you can still change the value of the variable in the function itself.

```php
declare(strict_types=1);

function find(int $id): Model
{
    $id = '' . $id;

    /*
     * This is perfectly allowed in PHP
     * `$id` is a string now.
     */

    // …
}

find('1'); // This would trigger a TypeError.

find(1); // This would be fine.
```

Even with strict types and type hints, PHP's type system is weak. Type hints only ensure a variable's type at that point in time, without a guarantee about any future value that variable might have.

Like I said before: it makes sense for PHP to have a weak type system, since all input it has to deal with starts out as a string. There is an interesting property to strong types though: they come with a few guarantees. If a variable has a type that's unchangeable, a whole range of unexpected behaviour simply cannot happen anymore.

You see, it's mathematically provable that if a strongly typed program compiles, it's impossible for that program to have a range of bugs which would be able to exist in weakly typed languages. In other words, strong

types give the programmer a better insurance that the code actually behaves how it's supposed to.

As a sidenote: this doesn't mean that a strongly typed language cannot have bugs! You're perfectly able to write a buggy implementation. But when a strongly typed program compiles successfully, you're sure a certain set of type-related bugs and errors can't occur in that program.

> Strong type systems allow developers to have much more insight into the program when writing the code, instead of having to run it.

There's one more concept we need to look at: static and dynamic type systems — and this is where things start to get interesting.

As you're probably aware, PHP is an interpreted language which means that a PHP script is translated to machine code at runtime. When you send a request to a server running PHP, it will take those plain `.php` files, and parse that text into something the processor can execute.

Again, this is one of PHP's strengths: the simplicity of writing a script, refreshing the page, and everything is there. That's a big difference compared to a language that has to be compiled before it can be run.

Obviously there are caching mechanisms which optimise this, so the above statement is an oversimplification but it's good enough to get to the next point though.

That point is that, once again, there is a downside to PHP's approach: since it only checks its types at runtime, there might be type errors that crash the program, while running. You might have a clear error to debug, but still the program has crashed.

This type checking at runtime makes PHP a dynamically typed language. A statically typed language on the other hand will have all its type checks done before the code is executed, usually during compile time.

As of PHP 7.0, its type system has been improved quite a lot. So much so that tools like PHPStan, Phan and Psalm started to become very popular lately. These tools take the dynamic language that is PHP, but run a bunch of statical analyses on your code.

These opt-in libraries can offer quite a lot of insight into your code, without ever having to run it or run unit tests. What's more, an IDE like PhpStorm also has many of these static checks built-in.

With all this background information in mind, it's time to return to the core of our application: data.

Structuring unstructured data

Have you ever had to work with an "array of stuff" that was actually more than just a list? Did you use the array keys as fields? And did you feel the pain of not knowing exactly what was in that array? How about not being sure whether the data in it is actually what you expect it to be, or what fields are available?

Let's visualise what I'm talking about: working with Laravel's requests. Think of this example as a basic CRUD operation to update an existing customer.

```php
function store(CustomerRequest $request, Customer $customer)
{
    $validated = $request->validated();

    $customer->name = $validated['name'];
    $customer->email = $validated['email'];

    // …
}
```

You might already see the problem arising: we don't know exactly what data is available in the `$validated` array. While arrays in PHP are a versatile and powerful data structure, as soon as they are used to represent something other than "a list of things", there are better ways to solve your problem.

Before looking at solutions, here's what you could do to deal with this situation:

- Read the source code
- Read the documentation
- Dump `$validated` to inspect it
- Or use a debugger to inspect it

Now imagine for a minute that you're working with a team of several developers on this project, and that one of your colleagues has written this piece of code five months ago. I can guarantee you that you will not know what data you're working with, without doing any of the cumbersome things listed above.

It turns out that strongly typed systems in combination with static analysis can be a great help in understanding what exactly we're dealing with. Languages like Rust, for example, solve this problem cleanly:

```rust
struct CustomerData {
    name: String,
    email: String,
    birth_date: Date,
}
```

Actually, a struct is exactly what we need but unfortunately PHP doesn't have structs; it has arrays and objects, and that's it.

However... objects and classes might be enough.

```php
class CustomerData
{
    public string $name;
    public string $email;
    public Carbon $birth_date;
}
```

It's a little more verbose, but it basically does the same thing. This simple object could be used like so.

```php
function store(CustomerRequest $request, Customer $customer)
{
    $validated = CustomerData::fromRequest($request);

    $customer->name = $validated->name;
    $customer->email = $validated->email;
    $customer->birth_date = $validated->birth_date;

    // …
}
```

The static analyser built into your IDE would always be able to tell us what data we're dealing with.

This pattern of wrapping unstructured data in types, so that we can use that data in a reliable way, is called *"data transfer objects"*. It's the first concrete pattern I highly recommend you to use in your larger-than-average Laravel projects.

When discussing this book with your colleagues, friends or within the Laravel community, you might stumble upon people who don't share the same vision about strong type systems. There are in fact lots of people who prefer to embrace the dynamic/weak side of PHP, and there's definitely something to say for that.

In my experience though, there are more advantages to the strongly typed approach when working with a team of several developers on a project for serious amounts of time. You have to take every opportunity you can to reduce cognitive load. You don't want developers having to start debugging their code every time they want to know what exactly is in a variable. The information has to be right there at hand, so that developers can focus on what's important: building the application.

Of course, using DTOs comes with a price: there is not only the overhead of defining these classes; you also need to map, for example, request data onto a DTO. But the benefits of using DTOs definitely outweigh this added cost: whatever time you lose initially writing this code, you make up for in the long run.

The question about constructing DTOs from *"external"* data is one that still needs answering though.

DTO factories

I will share two possible ways to construct DTOs, and also explain which one is my personal preference.

The first one is the most correct one: using a dedicated factory.

```php
class CustomerDataFactory
{
    public function fromRequest(
        CustomerRequest $request
    ): CustomerData {
        return new CustomerData([
            'name' => $request->get('name'),
            'email' => $request->get('email'),
            'birth_date' => Carbon::make(
                $request->get('birth_date')
            ),
        ]);
    }
}
```

Having a separated factory keeps your code clean throughout the project. I would say it makes most sense for this factory to live in the application layer, since it has to know about specific requests and other kinds of user input.

While being the correct solution, did you notice I used a shorthand in a previous example? That's right; on the DTO class itself:
`CustomerData::fromRequest()`.

What's wrong with this approach? Well for one, it adds application-specific logic in the domain. The DTO class, living in the domain, now has to know about the `CustomerRequest` class, which lives in the application layer.

```php
use Spatie\DataTransferObject\DataTransferObject;

class CustomerData extends DataTransferObject
{
    // …

    public static function fromRequest(
        CustomerRequest $request
    ): self {
        return new self([
            'name' => $request->get('name'),
            'email' => $request->get('email'),
            'birth_date' => Carbon::make(
                $request->get('birth_date')
            ),
        ]);
    }
}
```

Obviously, mixing application-specific code within the domain isn't the best of ideas. However, it is my preference. There's two reasons for that.

First of all, we already established that DTOs are the entry point for data into the codebase. As soon as we're working with data from the outside, we want to convert it to a DTO. We need to do this mapping *somewhere*, so we might as well do it within the class that it's meant for.

Secondly, and this is the more important reason; I prefer this approach because of one of PHP's own limitations: it doesn't support named parameters — yet.

See, you don't want your DTOs to end up having a constructor with an individual parameter for each property: this doesn't scale, and is very confusing when working with nullable or default-value properties. That's why I prefer the approach of passing an array to the DTO, and have it construct itself based on the data in that array. As an aside: we use our `spatie/data-transfer-object` package to do exactly this.

Because named parameters aren't supported, there's also no static analysis available, meaning you're in the dark about what data is needed whenever you're constructing a DTO. I prefer to keep this "being in the dark" within the DTO class, so that it can be used without an extra thought from the outside.

If PHP were to support something like named parameters though, which it will in PHP 8, I would say the factory pattern is the way to go:

```php
public function fromRequest(
    CustomerRequest $request
): CustomerData {
    return new CustomerData(
        name: $request->get('name'),
        email: $request->get('email'),
        birth_date: Carbon::make(
            $request->get('birth_date')
        ),
    );
}
```

Until PHP supports this, I would choose the pragmatic solution over the theoretically correct one. It's up to you though. Feel free to choose what fits your team best.

An alternative to typed properties

There is an alternative to using typed properties: DocBlocks. Our DTO
package I mentioned earlier also supports them.

```php
use Spatie\DataTransferObject\DataTransferObject;

class CustomerData extends DataTransferObject
{
    /** @var string */
    public $name;

    /** @var string */
    public $email;

    /** @var \Carbon\Carbon */
    public $birth_date;
}
```

In some cases, DocBlocks offer advantages: they support `array of` types and
generics. But by default though, DocBlocks don't give any guarantees that the
data is of the type they say it is. Luckily PHP has its reflection API, and with it,
a lot more is possible.

The solution provided by this package can be thought of as an extension of
PHP's type system. While there's only so much one can do in userland and at
runtime, still it adds value. If you're unable to use PHP 7.4 and want a little
more certainty that your DocBlock types are actually respected, this package
has you covered.

A note on DTO's in PHP 8

PHP 8 will support named arguments, as well as constructor property promotion. Those two features will have an immense impact on the amount of boilerplate code you'll need to write.

Here's what a small DTO class would look like in PHP 7.4.

```php
class CustomerData extends DataTransferObject
{
    public string $name;

    public string $email;

    public Carbon $birth_date;

    public static function fromRequest(
        CustomerRequest $request
    ): self {
        return new self([
            'name' => $request->get('name'),
            'email' => $request->get('email'),
            'birth_date' => Carbon::make(
                $request->get('birth_date')
            ),
        ]);
    }
}

$data = CustomerData::fromRequest($customerRequest);
```

And this is what it would look like in PHP 8.

```php
class CustomerData
{
    public function __construct(
        public string $name,
        public string $email,
        public Carbon $birth_date,
    ) {}
}

$data = new CustomerData(...$customerRequest->validated());
```

Because data lives at the core of almost every project, it's one of the most important building blocks. Data transfer objects offer you a way to work with data in a structured, type safe and predictable way.

You'll note throughout this book that DTOs are used more often than not. That's why it was so important to take an in-depth look at them at the start. Likewise, there's another crucial building block that needs our thorough attention: actions. That's the topic for the next chapter.

Actions

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 don't want to work with random arrays full of data, we also don't want the most critical part of our project, the business functionality, to be spread throughout random functions and classes.

Here's an example: one of the user stories in your project might be for *"an admin to create an invoice"*. This means we'll be saving an invoice in the database, but there's more to do than just that.

Calculate the price of each individual invoice line and the total price

- Generate an invoice number
- Save the invoice to the database
- Create a payment via the payment provider
- Create a PDF with all relevant information
- Send this PDF to the customer

A common practice in Laravel is to create *"fat models"* that will handle all this functionality. In this chapter we will look at another approach to adding this behaviour into our codebase.

Instead of mixing functionality in models or controllers, we will treat these user stories as first class citizens of the project. I tend to call these *"actions"*.

Terminology

Before looking at their use, we need to discuss how actions are structured. For starters, they live in the domain.

Secondly, they are simple classes without any abstractions or interfaces. An action is a class that takes input, does something, and gives output. That's why an action usually only has one public method, and sometimes a constructor.

As a convention in our projects, we decided to suffix all of our classes. For sure `CreateInvoice` sounds nice, but as soon as you're dealing with several hundreds or thousands of classes, you'll want to make sure that no naming collisions can occur. You see, `CreateInvoice` could very well also be the name of an invokable controller, a command, a job or a request. We prefer to eliminate as much confusion as possible, hence, `CreateInvoiceAction` will be the name.

Evidently this means that class names become longer. The reality is that if you're working on larger projects, you can't avoid choosing longer names to make sure no confusion is possible. Here's an extreme example from one of our projects (I'm not kidding):
`CreateOrUpdateHabitantContractUnitPackageAction`.

We hated this name at first. We desperately tried to come up with a shorter one. In the end though, we had to admit that clarity of what a class is about is the most important thing. Our IDE's autocompletion will take care of the inconvenience of the long names anyway.

When we're settled on a class name, the next hurdle to overcome is naming the public method to use our action. One option is to make it invokable, like so:

```php
class CreateInvoiceAction
{
    public function __invoke(InvoiceData $invoiceData): Invoice
    {
        // …
    }
}
```

There's a practical problem with this approach though. To understand it I need to mention something that we'll look at later in this chapter and that is that

sometimes, we'll compose actions out of other actions. It would look
something like this:

```php
class CreateInvoiceAction
{
    private CreateInvoiceLineAction $createInvoiceLineAction;

    public function __construct(
        CreateInvoiceLineAction $createInvoiceLineAction
    ) { /* … */ }

    public function __invoke(InvoiceData $invoiceData): Invoice
    {
        foreach ($invoiceData->lines as $lineData) {
            $invoice->addLine(
                ($this->createInvoiceLineAction)($lineData)
            );
        }
    }
}
```

Can you see the issue? PHP does not allow you to directly invoke an invokable
property on a class, since it is looking for a class method instead. That's why
you'll have to wrap the action in parentheses before calling it.

If you don't want to use that funky syntax, there are alternatives: we could, for
example, use `handle`, which is often used by Laravel as the default name in
these kinds of cases. Once again there's a problem with it, specifically
because Laravel uses it.

Whenever Laravel allows you to use `handle` in, for example, jobs or
commands, it will also provide method injection from the dependency
container. In our actions, we only want the constructor to have dependency
injection capabilities. Again we'll look closely into the reasons behind this later
in this chapter.

So `handle` is also out. When we started using actions, we actually gave this naming conundrum quite a lot of thought. In the end we settled on `execute`. Keep in mind though that you're free to come up with your own naming conventions because the point here is more about the pattern of using actions than it is about their names.

Into practice

With all of the terminology out of the way, let's talk about why actions are useful, and how to actually use them.

First let's talk about re-usability. The trick when using actions is to split them in small enough pieces so that some things are reusable, while keeping them large enough to not end up with an overload of classes. Take our invoice example: generating a PDF from an invoice is something that is likely to happen from within several contexts in our application. Sure, there's the PDF that's generated when an invoice is actually created, but an admin might also want to see a preview or draft before sending it.

These two user stories: *"creating an invoice"* and *"previewing an invoice"* obviously require two entry points, and therefore two controllers. On the other hand though, generating the PDF based on the invoice is something that's done in both cases.

When you start spending time thinking about what the application actually will do, you'll notice that there are lots of actions which can be reused. Of course, we also need to be careful not to over-abstract our code. It's often better to copy-paste a little code than to make premature abstractions.

A good rule of thumb when making abstractions is to think about the functionality instead of the technical properties of code. When two actions might do similar things, but are used in completely different contexts, you should be careful not to start abstracting them too early.

On the other hand, there are cases where abstractions can be helpful. Take again our invoice PDF example: chances are you need to generate more PDFs than just for invoices — that's the case in our projects, at least. It might make sense to have a general `GeneratePdfAction` accepting an interface `toPDF`, which `Invoice` then implements. This means that the functionality to generate PDFs can now be reused throughout the project.

But, let's be honest; chances are the majority of our actions will be rather specific to their user stories, and won't be re-usable. You might think that actions, in these cases, are unnecessary overhead. Hang on though, because re-usability is not the only reason to use them. Actually, the most important reason has nothing to do with technical benefits at all: actions allow the programmer to think in ways that are closer to the real world, instead of the code.

Say you need to make changes to the way invoices are created. A typical Laravel application will probably have this invoice creation logic spread across a controller and a model, maybe a job which generates the PDF, and finally an event listener to send the invoice mail. That's a lot of places you need to know of. Once again our code is spread across the codebase, grouped by its technical properties, rather than its meaning.

Actions reduce the cognitive load that's introduced by such a system. If you need to work on how invoices are created, you can simply go to the action class, and start from there.

Don't be mistaken: actions may very well work together with, for example, asynchronous jobs and event listeners; though these jobs and listeners merely provide the infrastructure for actions to work, and not the business logic itself. This is a good example of why we need to split the domain and application layers: each has their own purpose.

So we now have re-usability and a reduction of cognitive load, but there's even more!

Because actions are small pieces of software that live almost on their own, it's very easy to unit test them. In your tests you don't have to worry about sending fake HTTP requests, making facades, etc. You can simply make a new action, maybe provide some mock dependencies, pass it the required input data and make assertions on its output.

For example, let's take the `CreateInvoiceLineAction`. It will:

- take data about which product will be invoiced
- receive an amount plus a period
- calculate the total price and then,
- calculate prices with and without VAT

These are things you can write robust, yet simple, unit tests for.

If all your actions are properly unit tested, you can be very confident that the bulk of the functionality that needs to be provided by the application actually works as intended. Now it's only a matter of using these actions in ways that make sense for the end user, and write some integration tests for those pieces.

Composing actions

One important characteristic of actions that I already mentioned before briefly, is how they use dependency injection. Since we're using the constructor to pass in data from the container, and the `execute` method to pass in context-related data, we're free to compose actions out of actions out of actions out of...

You get the idea. Let's be clear though that a deep dependency chain is something you want to avoid — it makes the code complex and highly dependent on each other — yet there are several cases where having DI is very beneficial.

Take again the example of the `CreateInvoiceLineAction` which has to calculate VAT prices. Now depending on the context, an invoice line might have a price including or excluding VAT. Calculating VAT prices is something trivial, yet we don't want our `CreateInvoiceLineAction` to be concerned with the details of it.

So imagine we have a simple `VatCalculator` class — which is something that might live in the `\Support` namespace — it could be injected like so:

```php
class CreateInvoiceLineAction
{
    private VatCalculator $vatCalculator;

    public function __construct(VatCalculator $vatCalculator)
    {
        $this->vatCalculator = $vatCalculator;
    }

    public function execute(
        InvoiceLineData $invoiceLineData
    ): InvoiceLine {
        // …
    }
}
```

And you'd use it like this:

```php
public function execute(
    InvoiceLineData $invoiceLineData
): InvoiceLine {
    $item = $invoiceLineData->item;

    if ($item->vatIncluded()) {
        [$priceIncVat, $priceExclVat] =
            $this->vatCalculator->vatIncluded(
                $item->getPrice(),
                $item->getVatPercentage()
            );
    } else {
        [$priceIncVat, $priceExclVat] =
            $this->vatCalculator->vatExcluded(
                $item->getPrice(),
                $item->getVatPercentage()
            );
    }

    $amount = $invoiceLineData->item_amount;

    return new InvoiceLine([
        'item_price' => $item->getPrice(),
        'total_price' => $amount * $priceIncVat,
        'total_price_excluding_vat' => $amount * $priceExclVat,
    ]);
}
```

The `CreateInvoiceLineAction` in turn would be injected into `CreateInvoiceAction`. And this one again has other dependencies: the `CreatePdfAction` and `SendMailAction`, for example.

You can see how composition can help you keep individual actions small, yet allow for complex business functionality to be coded in a clear and maintainable way.

Alternatives to actions

There are two paradigms I need to mention at this point that wouldn't need a concept like actions.

The first one will be known to people who are familiar with DDD: commands and handlers. Actions are a simplified version of them. Where commands and handlers make a distinction between what needs to happen and how it needs to happen, actions combine these two responsibilities into one. It's true, however, that the command bus offers more flexibility than actions. On the other hand it also requires you to write more code.

For the scope of our projects, splitting actions into commands and handlers was taking it a step too far. We would almost never need the added flexibility, yet it would take a lot longer to write the code.

The second alternative worth mentioning is event driven systems. If you ever worked in an event driven system, you might think that actions are too directly coupled to the places where they are actually used. Again the same argument applies: event driven systems offer more flexibility, yet for our projects it would have been overkill to use them. Furthermore, event driven systems add a layer of indirectness that makes the code more complex to reason about. While this indirectness does offer benefits, they wouldn't outweigh the cost of maintenance for us.

I hope it's clear that I'm not suggesting we've got it all figured out and have the perfect solution for all Laravel projects. We don't. When you continue to read through this book, it's important that you keep an eye on the specific needs of your project. While you might be able to use some concepts proposed here, you might also need some other solutions to solve certain aspects.

For us, actions are the right choice because they offer the right amount of flexibility, re-usability and significantly reduce cognitive load. They encapsulate the essence of the application. They can in fact be thought of, together with DTOs and models, as the true core of the project.

That brings us to the next chapter, the last piece of the core: models.

Models

In the previous chapters, I've talked about two of the three core building blocks of every application: DTOs and actions — data and functionality. In this chapter we will look at the last piece that I consider part of this core: exposing data that's persisted in a data store; in other words: models.

Now, models are a tricky subject. Laravel provides a lot of functionality in its Eloquent model classes, which means that they not only represent the data in a data store, they also allow you to build queries, load and save data, have a built-in event system, and more.

In this chapter, I will not tell you to ditch all the model functionality that's provided by Laravel — it's quite useful indeed. However I will name a few pitfalls that you need to be aware of, and solutions for them, so that even in large projects, models won't be the cause of difficult maintainership.

My point of view is that we should embrace the framework instead of trying to fight it. Though we should embrace it in such a way that large projects stay maintainable.

Models ≠ business logic

The first pitfall that many developers fall into, is that they think of models as
the place to be when it comes to business logic. I already listed a few
responsibilities of models which are built-into Laravel, and I would argue to be
careful not to add any more.

It sounds very appealing at first, to be able to do something like
`$invoiceLine->price_including_vat` or `$invoice->total_price`; and it sure
does. I actually do believe that invoices and invoice lines should have these
methods. There's one important distinction to make though: these methods
shouldn't calculate anything. Let's take a look at what not to do:

Here's a `total_price` accessor on our invoice model, looping over all invoice
lines and making the sum of their total price.

```php
class Invoice extends Model
{
    public function getTotalPriceAttribute(): int
    {
        return $this->invoiceLines
            ->reduce(
                fn (int $totalPrice, InvoiceLine $invoiceLine) =>
                    $totalPrice + $invoiceLine->total_price,
                0
            );
    }
}
```

And here is how the total price per line is calculated:

```php
class InvoiceLine extends Model
{
    public function getTotalPriceAttribute(): int
    {
        $vatCalculator = app(VatCalculator::class);

        $price = $this->item_amount * $this->item_price;

        if ($this->price_excluding_vat) {
            $price = $vatCalculator->totalPrice(
                $price,
                $this->vat_percentage
            );
        }

        return $price;
    }
}
```

Since you read the previous chapter on actions, you might guess what I would do instead: calculating the total price of an invoice is a user story that should be represented by an action.

The `Invoice` and `InvoiceLine` models could have the simple `total_price` and `price_including_vat` properties, but they are calculated by actions first, and then stored in the database. When using `$invoice->total_price`, you're simply reading data that's already been calculated before.

There are a few advantages to this approach. First the obvious one: performance. You are only doing the calculations once, not every time when in need of the data. Second: you can query the calculated data directly. And third: you don't have to worry about side effects.

Now, we could start a purist debate about how single responsibility helps keep your classes small, better maintainable and easily testable and how dependency injection is superior to service location, but I rather state the obvious instead of having long theoretical debates where I know there's simply two sides that won't agree.

So, the obvious: even though you might like to be able to do `$invoice->send()` or `$invoice->toPdf()`, the model code is growing and growing. This is something that happens over time and it doesn't seem to be a big deal at first. In fact, `$invoice->toPdf()` might actually only be one or two lines of code to begin with.

From experience though, these one or two lines add up. One or two lines isn't the problem, but a hundred times one or two lines is. The reality is that model classes grow over time and can grow quite large.

Even if you don't agree with me on the advantages that single responsibility and dependency injection brings, there's little to disagree about this: a model class with hundreds of lines of code will not stay maintainable.

To summarise, think of models and their purpose as being to only provide data for you and let something else be concerned with making sure that data is calculated properly.

Scaling down models

If our goal is to keep model classes reasonably small — small enough to be able to understand them by simply opening their file — we need to move some more things around. Ideally, we only want to keep the data read from the

database, simple accessors for stuff we can't calculate beforehand, casts, and relations.

Other responsibilities should be moved to other classes. One example is query scopes. We could easily move them to dedicated query builder classes.

Believe it or not, query builder classes are actually the normal way of using Eloquent; scopes are simply syntactic sugar on top of them. This is what a query builder class might look like.

```php
namespace Domain\Invoices\QueryBuilders;

use Domain\Invoices\States\Paid;
use Illuminate\Database\Eloquent\Builder;

class InvoiceQueryBuilder extends Builder
{
    public function wherePaid(): self
    {
        return $this->whereState('status', Paid::class);
    }
}
```

Next up, we override the `newEloquentBuilder` method in our model and return our custom class. Laravel will use it from now on.

```php
namespace Domain\Invoices\Models;

use Domain\Invoices\QueryBuilders\InvoiceQueryBuilder;

class Invoice extends Model
{
    public function newEloquentBuilder($query): InvoiceQueryBuilder
    {
        return new InvoiceQueryBuilder($query);
    }
}
```

This is what I meant by embracing the framework: you don't need to introduce new patterns like repositories per se; you can build upon what Laravel provides. With some thought, we strike the perfect balance between using the commodities provided by the framework and preventing our code from growing too large in specific places.

Using this mindset, we can also provide custom collection classes for relations. Laravel has great collection support, though you often end up with long chains of collection functions either in the model or in the application layer. This again isn't ideal, and luckily Laravel provides us with the needed hooks to bundle collection logic into a dedicated class.

Here's an example of a custom collection class, and note that it's entirely possible to combine several methods into new ones, avoiding long function chains in other places.

```php
namespace Domain\Invoices\Collections;

use Domain\Invoices\Models\InvoiceLines;
use Illuminate\Database\Eloquent\Collection;

class InvoiceLineCollection extends Collection
{
    public function creditLines(): self
    {
        return $this->filter(fn (InvoiceLine $invoiceLine) =>
            $invoiceLine->isCreditLine()
        );
    }
}
```

This is how you link a collection class to a model — `InvoiceLine` — in this case:

```php
namespace Domain\Invoices\Models;

use Domain\Invoices\Collection\InvoiceLineCollection;

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

    public function isCreditLine(): bool
    {
        return $this->price < 0.0;
    }
}
```

Every model having a `HasMany` relation to `InvoiceLine`, will now use our collection class instead.

```php
$invoice
    ->invoiceLines
    ->creditLines()
    ->map(function (InvoiceLine $invoiceLine) {
        // …
    });
```

Try to keep your models clean and data-oriented, instead of having them provide business logic. There are better places to handle it.

Event driven models

In chapter 3, I already mentioned event-driven systems, and how they offer more flexibility, at the cost of complexity.

You might prefer an event driven approach, so I feel like there's an important note to make about model-related events. Laravel will emit generic model events by default and it expects you to use a model observer or configure listeners on the model itself.

There's also another approach, one that's more flexible and robust. You can remap generic model events to specific event classes, on a model basis, and use dedicated event subscribers for them.

In a model, it would look like this:

```php
class Invoice
{
    protected $dispatchesEvents = [
        'saving' => InvoiceSavingEvent::class,
        'deleting' => InvoiceDeletingEvent::class,
    ];
}
```

Such a specific event class in turn, would be similar to:

```php
class InvoiceSavingEvent
{
    public Invoice $invoice;

    public function __construct(Invoice $invoice)
    {
        $this->invoice = $invoice;
    }
}
```

And finally, the subscriber could be implemented as:

```php
use Illuminate\Events\Dispatcher;

class InvoiceSubscriber
{
    private CalculateTotalPriceAction $calculateTotalPriceAction;

    public function __construct(
        CalculateTotalPriceAction $calculateTotalPriceAction
    ) {/* … */}

    public function saving(InvoiceSavingEvent $event): void
    {
        $invoice = $event->invoice;

        $invoice->total_price =
            ($this->calculateTotalPriceAction)($invoice);
    }

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

Which would be registered in the `EventServiceProvider`, like this:

```php
class EventServiceProvider extends ServiceProvider
{
    protected $subscribe = [
        InvoiceSubscriber::class,
    ];
}
```

This approach also gives you the flexibility to hook into your own custom model events, and handle them in the same way eloquent events are handled. And once again, subscriber classes allow our models to stay small and maintainable.

Empty bags of nothingness

Martin Fowler once wrote about how we need to avoid objects becoming nothing more than empty bags of data, something he calls an anti-pattern. You might be thinking we're doing the same with our model classes.

My answer to those concerns is twofold. First of all: I don't think of models as empty bags with plain old data. Using accessors and casts, they provide a rich layer between the plain data in the database and data a developer wants to use. In this chapter I argued to move several other responsibilities to separate classes, that's true, yet I believe that models in their "trimmed" down state still offer a lot more value than simple bags of data, thanks to all the functionality Laravel provides.

Secondly, I think it's worth mentioning Alan Kay's vision on this topic (he's the one who came up with the term OOP). He once said that he regretted calling

the paradigm *"object oriented"* and not *"process oriented"*. Alan argues that he's actually a proponent of splitting process and data.

Whether you agree with that point of view or not is up to you. I do admit to have been influenced by some of Alan's insights, and you might notice that throughout this book. Like I said before: don't think of this book as the holy grail of software design. My goal is to challenge the current way you're writing code, making you think whether there are more optimal ways to solve some of your problems.

States

The state pattern is one of the best ways to add state-specific behaviour to models, while still keeping them clean.

This chapter will talk about the state pattern, and specifically how to apply it to models. You can think of this chapter as an extension to chapter 4, where I wrote about how we aim to keep our model classes manageable by preventing them from handling business logic.

Moving business logic away from models poses a problem with a very common use case: what to do with model states?

An invoice can be pending or paid, a payment can be failed or succeeded. Depending on the state, a model must behave differently; how do we bridge this gap between models and business logic?

States and transitions between them, are a frequent use case in large projects and in fact are so frequent that they deserve a chapter on their own.

The state pattern

At its core, the state pattern is a simple pattern, yet it allows for very powerful functionality. Let's take the example of invoices again: they can be pending or paid. Let's start with a simplified example because I want you to understand how the state pattern allows us lots of flexibility.

Say the invoice overview should show a badge representing the state of that invoice; while pending, the badge is coloured orange and green if paid.

A naive fat model approach would do something like this:

```php
class Invoice extends Model
{
    // …

    public function getStateColour(): string
    {
        if ($this->state->equals(InvoiceState::PENDING())) {
            return 'orange';
        }

        if ($this->state->equals(InvoiceState::PAID())) {
            return 'green';
        }

        return 'grey';
    }
}
```

Since we're using some kind of enum class to represent the state value, we could use that enum to encapsulate the corresponding colour:

```php
/**
 * @method static self PENDING()
 * @method static self PAID()
 */
class InvoiceState extends Enum
{
    private const PENDING = 'pending';
    private const PAID = 'paid';

    public function getColour(): string
    {
        if ($this->value === self::PENDING) {
            return 'orange';
        }

        if ($this->value === self::PAID) {
            return 'green';
        }

        return 'grey';
    }
}
```

It would be used like so:

```php
class Invoice extends Model
{
    // …

    public function getStateColour(): string
    {
        return $this->state->getColour();
    }
}
```

You could write the `InvoiceState::getColour` even shorter, by using arrays:

```php
class InvoiceState extends Enum
{
    public function getColour(): string
    {
        return [
            self::PENDING => 'orange',
            self::PAID => 'green',
        ][$this->value] ?? 'grey';
    }
}
```

Or by using `match` in PHP 8:

```php
class InvoiceState extends Enum
{
    public function getColour(): string
    {
        return match($this->value) {
            self::PENDING => 'orange',
            self::PAID => 'green',
            default => 'grey',
        };
    }
}
```

Whatever approach you prefer, in essence you're listing all available options, checking if one of them matches the current one, and doing something based on the outcome. It's a big if/else statement, whichever syntactic sugar you prefer.

Using this approach, we add a responsibility, either to the model or the enum class, which is that *it* has to know what a specific state should do and *it* has to know how a state works. The state pattern turns this the other way around: it

treats *"a state"* as a first-class citizen of our codebase. Every state is represented by a separate class, and each of these classes *acts* upon a subject.

Is that difficult to grasp? Let's take it step by step.

We start with an abstract class `InvoiceState`, this class describes all functionality that concrete invoice states can provide. In our case we only want them to provide a colour:

```
abstract class InvoiceState
{
    abstract public function colour(): string;
}
```

Next, we make two classes, each representing a concrete state:

```
class PendingInvoiceState extends InvoiceState
{
    public function colour(): string
    {
        return 'orange';
    }
}

class PaidInvoiceState extends InvoiceState
{
    public function colour(): string
    {
        return 'green';
    }
}
```

The first thing to notice is that each of these classes can easily be unit tested on their own. Take a look at this example:

```php
class InvoiceStateTest extends TestCase
{
    /** @test */
    public function the_colour_of_pending_is_orange
    {
        $state = new PendingInvoiceState();

        $this->assertEquals('orange', $state->colour());
    }
}
```

Second, you should note that *"colours"* is a naive example used to explain the pattern. You might as well have more complex business logic encapsulated by a state. Take this one: should an invoice be paid? This of course depends on the state and whether it was already paid or not, but might also depend on the type of invoice we're dealing with. Say our system supports credit notes which don't have to be paid, or it allows for invoices with a price of 0. This business logic can be encapsulated by these state classes.

There's one thing missing to make those complex rules work though: we need to be able to look at the model from within our state class, if we're going to decide whether or not that invoice must be paid. In other words, we'll need to inject our `Invoice` model into the state class.

We could handle that boilerplate setup in our abstract `InvoiceState` parent class:

```php
abstract class InvoiceState
{
    /** @var Invoice */
    protected $invoice;

    public function __construct(Invoice $invoice) { /* … */ }

    abstract public function mustBePaid(): bool;

    // …
}
```

And next implement `mustBePaid` in each concrete state:

```php
class PendingInvoiceState extends InvoiceState
{
    public function mustBePaid(): bool
    {
        return $this->invoice->total_price > 0
            && $this->invoice->type->equals(InvoiceType::DEBIT());
    }

    // …
}

class PaidInvoiceState extends InvoiceState
{
    public function mustBePaid(): bool
    {
        return false;
    }

    // …
}
```

Again we can write simple unit tests for each state, and our `Invoice` model can simply do something like this:

```php
class Invoice extends Model
{
    public function getStateAttribute(): InvoiceState
    {
        return new $this->state_class($this);
    }

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

In the database we can save the concrete model state class in the `state_class` field and we're done. Obviously, doing this mapping manually (saving and loading from and to the database) gets tedious very quickly. That's why I wrote the `spatie/laravel-model-states` package which takes care of all the grunt work for you.

State-specific behaviour, in other words *"the state pattern"*, is only half of the solution though; we still need to handle transitioning the invoice state from one to another, and ensuring only specific states may transition to others. So let's look at state transitions.

Transitions

Remember how I talked about moving business logic away from models, and only allowing them to provide data in a workable way from the database? The same thinking can be applied to states and transitions.

We should avoid side effects when using states: things like making changes in the database, sending mails, etc. States should be used to read or provide data. Transitions on the other hand don't provide anything. Rather, they make sure our model state is correctly transitioned from one to another leading to acceptable side effects.

Splitting these two concerns in separate classes gives us the same advantages I wrote about again and again: better testability and reduced cognitive load. Allowing a class to only have one responsibility makes it easier to split a complex problem into several easy-to-grasp bits.

So, transitions: classes which will take a model, an invoice in our case, and change that invoice's state when allowed to another one. In some cases there might be small side effects like writing a log message or sending a notification about the state transition.

A naive implementation might look something like this:

```php
class PendingToPaidTransition
{
    public function __invoke(Invoice $invoice): Invoice
    {
        if (! $invoice->mustBePaid()) {
            throw new InvalidTransitionException(self::class, $invoice);
        }

        $invoice->status_class = PaidInvoiceState::class;
        $invoice->save();

        History::log($invoice, "Pending to Paid");
    }
}
```

Again, there are many things you can do with this basic pattern:

- Define all allowed transitions on the model
- Transition a state directly to another one, by using a transition class under the hood
- Automatically determine what state to transition to based on a set of parameters

The package I mentioned before adds support for transitions, as well as basic transition management. If you want complex state machines though, you might want to look at other packages; I will discuss an example in the footnotes at the end of this book.

States without transitions

When we think of *"state"*, we often think they cannot exist without transitions. However, that's not true: an object can have a state that never changes and transitions aren't required to apply the state pattern. Why is this important? Well, take a look again at our `PendingInvoiceState::mustBePaid` implementation:

```
class PendingInvoiceState extends InvoiceState
{
    public function mustBePaid(): bool
    {
        return $this->invoice->total_price > 0
            && $this->invoice->type->equals(InvoiceType::DEBIT());
    }
}
```

Since we want to use the state pattern to reduce brittle if/else blocks in our code, can you guess where I'm going with this? Have you considered that `$this->invoice->type->equals(InvoiceType::DEBIT())` is in fact an if statement in disguise?

`InvoiceType` in fact could very well also apply the state pattern! It's simply a state that likely will never change for a given object. Take a look at that:

```php
abstract class InvoiceType
{
    protected Invoice $invoice;

    // …

    abstract public function mustBePaid(): bool;
}

class CreditInvoiceType extends InvoiceType
{
    public function mustBePaid(): bool
    {
        return false;
    }
}

class DebitInvoiceType extends InvoiceType
{
    public function mustBePaid(): bool
    {
        return true;
    }
}
```

Now we can refactor our `PendingInvoiceState::mustBePaid` like so:

```php
class PendingInvoiceState extends InvoiceState
{
    public function mustBePaid(): bool
    {
        return $this->invoice->total_price > 0
            && $this->invoice->type->mustBePaid();
    }
}
```

Reducing `if/else` statements in our code allows that code to be more linear, which in turn is easier to reason about.

The state pattern is, in my opinion, awesome. You're never stuck again writing huge if/else statements — in real life there are often more than two invoice states — and it allows for clean and testable code.

It's a pattern that you can incrementally introduce in your existing code bases, and I'm sure it will be a huge help keeping the project maintainable in the long run.

Enums

Enums or states?

Talking about states in the previous chapter, you might be thinking about enums right now. What's the difference between them and states? How and when to use enums?

It's a tricky subject, especially since both enums and the state pattern can be used to achieve the same result. Remember one of the examples in the previous chapter? We could model a state with one enum class, and determine value-specific behaviour with conditionals, like so:

```php
class InvoiceState extends Enum
{
    public function getColour(): string
    {
        return match($this->value) {
            self::PENDING => 'orange',
            self::PAID => 'green',
            default => 'grey',
        };
    }
}
```

Note that I'm using PHP 8's `match` expression here, you could also replace it

with a `switch` or `if` statement.

Anyway, the difference between the enum and state implementations is that the state pattern removes those conditions, by providing a dedicated class for each possible value. In effect, it works the other way around.

The state pattern's goal is to get rid of all those conditionals, and instead rely on the power of polymorphism to determine the program flow. Therefore, we could argue that's one way to choose which to implement: use the state pattern to get rid of conditional flows in your code, and enums for everything else.

But what is *"everything else"*? If we're using an enum, there will be places in our code where we do checks on them, for example `if InvoiceType is X, than do Y`. In essence, a condition which could be modeled with the state pattern. So are there any valid use cases for enums?

I think a little pragmatism is in place here. Sure we could use the state pattern to model all conditional flows in our application, but we should be aware that the state pattern comes with significant overhead: you need to create classes for each state, configure transitions between them, and you need to maintain them.

In other words: there *is* room for enums in our codebase.

If you need a collection of related values and there are little places where the application flow is actually determined by those values, then yes: feel free to use simple enums. But if you find yourself attaching more and more value-related functionality to them, I would say it's time to start looking at the state pattern instead.

There's no clear rule I can give you to sort this out so you will need to decide which solution to use on a case-by-case basis. Be careful not to overcomplicate your code by *always* applying the state pattern, but also be careful to keep that code maintainable, by using the state pattern when relevant.

Enums!

So with that question clarified, I'd like to discuss a few ways to add enums in PHP, because there's no native implementation for them.

As a little recap: an enumeration type, *"enum"* for short, is a data type to categorise named values. Enums can be used instead of hard coded strings to represent, for example, the status of a blog post in a structured and typed way.

PHP offers a very basic SPL implementation, but this really doesn't cut the mustard. On the other hand, there's a popular package written by Matthieu Napoli called `myclabs/php-enum` which is a package that many people, myself included, have been using in countless projects. It's really awesome.

Now, I do want to explore some of the difficulties you might encounter when trying to solve enums in userland, and tell you my preferred solution.

So, back to our invoice example. Imagine we would write something like this:

```php
class Invoice
{
    public function setType(InvoiceType $type): void
    {
        $this->type = $type;
    }
}
```

And be sure that the value of `Invoice::$type` is always one of two possibilities: `credit` or `debit`.

Say we'd save this `Invoice` in a database, its status would automatically be represented as a string.

The `myclabs/php-enum` package allows us to write this:

```php
use MyCLabs\Enum\Enum;

class InvoiceType extends Enum
{
    const CREDIT = 'credit';
    const DEBIT = 'debit';
}
```

Now, the previous example of type hinting `InvoiceType` obviously doesn't work: we can't assure it's one of these two values. So we could use the constant values directly:

```php
class Invoice
{
    public function setType(string $type): void
    {
        $this->type = $type;
    }
}

// …

$invoice->setType(InvoiceType::DEBIT);
```

But this prevents us from doing proper type checking, as every string could be passed to `Invoice::setType`:

```php
$invoice->setType('Whatever you want');
```

A better approach is to use a little magic introduced by the library. We make the constant values private, so that they aren't accessible directly, and we can use magic methods to resolve their value:

```php
use MyCLabs\Enum\Enum;

class InvoiceType extends Enum
{
    private const CREDIT = 'credit';
    private const DEBIT = 'debit';
}

$invoice->setType(InvoiceType::CREDIT());
```

Using the magic method `__callStatic()` underneath, an object of the class `InvoiceType` is constructed, with the `'credit'` value in it.

Now we can type check for `InvoiceType` and ensure the input is one of the two values defined by the enum.

Here's the problem with the `myclabs/php-enum` package though: by relying on `__callStatic()`, we lose static analysis benefits like auto-completion and refactoring. Your IDE or static analysis tool won't know there's a `InvoiceType::CREDIT` or `InvoiceType::DEBIT` method available:

```
$invoice->setType(InvoiceType::CREDIT());
```

Luckily, this problem is solvable with DocBlocks:

```php
use MyCLabs\Enum\Enum;

/**
 * @method static self CREDIT()
 * @method static self DEBIT()
 */
class InvoiceType extends Enum
{
    private const CREDIT = 'credit';
    private const DEBIT = 'debit';
}

$invoice->setType(InvoiceType::CREDIT());
```

But now we're maintaining duplicate code: there's the constant values, and the doc blocks.

At this point it's time to stop and think. In an ideal world, we'd have built-in enums in PHP:

```
enum InvoiceType {
    DEBIT, CREDIT;
}
```

Since that's not the case right now, we're stuck with userland implementations. Extending PHP's type system in userland most likely means two things: magic and reflection.

If we're already relying on these two elements, why not go all out and make our lives as simple as possible?

Here's how I write enums today:

```
use Spatie\Enum\Enum;

/**
 * @method static self CREDIT()
 * @method static self DEBIT()
 */
class InvoiceType extends Enum
{
}
```

This approach simply skips the need for duplicating code, and will determine enum values based on the DocBlocks - DocBlocks that we need to write anyway - to ensure proper IDE and static analysis support.

Opinionated, right? It's less code to maintain, with more benefits. Again I wrote a package for this, it's called `spatie/enum`, and there's also a Laravel

implementation called `spatie/laravel-enum` which adds automatic casting for model properties and things like that.

Now, you don't have to use any of those packages if you don't want to. I simply wanted to explain how you can have enum-like behaviour, while still keeping the benefits of a strongly typed and statically analysed system, without native PHP support.

I *know* this is far from an ideal situation. It would be amazing to see built-in support for enums in PHP one day. But until then, this has to do.

Managing domains

In the previous chapters we looked at the core building blocks of our domains: DTOs, actions models, and several related concepts. In this chapter we'll shift our focus from the technical side to the philosophical side: how do you start using domains, how to identify them, and how to manage them in the long run?

Teamwork

Back in chapter 1, I claimed that all paradigms and principles I wrote about would serve a purpose: to help teams of developers keep their larger-than-average Laravel applications maintainable over the years.

Writing this book and the preceding blog posts, some people voiced their concern: wouldn't a new directory structure and the use of complex principles make it difficult for new developers to join right away?

If you're a developer acquainted with default Laravel projects and with how they are taught to beginners, then it's true that you'll need to spend some time learning about how these projects are handled. However, this is not as big a deal as some people might want you to believe.

Imagine a project with around 100 models, 300 actions, almost 500 routes. The main difficulty in these projects is not how the code is technically structured; rather it's about the massive amount of business knowledge there is to understand. You can't expect new developers to understand all of the problems this project is solving, just in an instant. It takes time to get to know the code, but more importantly: it takes time to know the business. The less magic and indirections there are, the less room there is for any confusion.

It's essential to understand the goal of the architecture I'm unfolding in this book. It's not about writing the smallest number of characters or about the elegance of code. It's about making large codebases easier to navigate, to allow as little room as possible for confusion, and to keep the project healthy for a long time.

I have experience with this process in practice. My colleague Ruben joined as a new backend developer in a team of three developers on one of our projects.

The architecture was new to him, even if he had experience with Laravel before. So we took the time to guide him through. After only a few hours of briefing and pair programming, he was able to work independently in this project. It definitely took several weeks to get a thorough understanding of all the functionality the project was providing, but fortunately, the architecture didn't stand in his way. On the contrary: it helped Ruben to focus on the business logic instead.

If you made it until this point in this book, I hope that you understand that this architecture is not meant to be the silver bullet for every project. There are many cases where a more straightforward approach could work better, and in some cases, where a more complex approach is required.

Identifying domains

With the knowledge we now have about the basic domain building blocks, the question arises as to how exactly we start writing actual code. There are lots of methodologies you can use to better understand what you're about to build, though I feel that there are two key points:

- Even though you're a developer, your primary goal is to understand the business problem and translate that into code. The code itself is merely a means to an end; always keep your focus on the problem you're solving.
- Make sure you've got face-to-face time with your client. It will take time to extract the knowledge that you require to write a working program.

I actually came to think of my job description more as "a translator between real world problems, and technical solutions", instead of "a programmer who writes code". I firmly believe that this mindset is key if you're going to work on a long-running project. You don't just have to write the code - you need to understand the real-world problems you're trying to solve.

Depending on the size of your team, you might not need face-to-face interaction between *all* developers and the client, but nonetheless, all developers will need to understand the problems they are solving with code.

These team dynamics are such a complex topic that they deserve their own book. For now I'll keep it at this, because from here on out we can talk about how we translate these problems into domains.

In chapter 1, I wrote that one of the goals of this architecture is to group code that belongs together, based on their meaning in the real world instead of their technical properties. If you've got an open communication with your client, you'll note that it takes time — lots of time — to get a good idea about

their business. Often your client might not know it exactly themselves, and it's only by sitting down that they start thoroughly thinking about it.

That's why you shouldn't fear domain groups that change over time. You might start with an `Invoice` domain but notice half a year later that it has grown too large for you and your team to grasp fully. Maybe invoice generation and payments are two complex systems on their own, and so they can be split into two domain groups down the line.

My point of view is that it's healthy to keep iterating over your domain structure to keep refactoring it. Given the right tools it's not difficult at all to change, split, and refactor domains.

In summary: don't be afraid to start using domains because you can always refactor them later.

So that's the approach I would take if I'd want to start using this domain-oriented architecture: try to identify subsystems within the project, realizing they can — and will — change over time. You can sit down with your client, and you can ask them to write things down, you could even do event storming sessions with them. Together you form an image of what the project should be, and that image might very well be refined and also changed down the road.

And because our domain code has minimal dependencies, it's very flexible and doesn't cost much to move stuff around or refactor it.

Testing domains

Up until this point, I've explained the building blocks of domain code and how to manage it throughout time. There's one crucial aspect missing still: how we're going to test all of this. I've already shown a few tests here and there and told you how our modular architecture enables for straightforward unit testing, but in this chapter, I will show you the ins and outs of testing domain code.

But before doing so, there's one more pattern I want to explain to you because it will make our tests ten times easier. Let's look at the factory pattern.

Test factories

You probably already know of test factories in Laravel, they look something like this:

```
factory(Invoice::class)->create();
```

It's a very clean and to-the-point approach, but it actually lacks some
essential functionality. In fact, Laravel 8 brings a new type of test factory that
will much closer represent the pattern I'll describe today.

But before looking at that pattern, let's discuss where Laravel's factories are
lacking, before Laravel 8. Take the example of factory states, a powerful
pattern, yet poorly implemented in Laravel.

```php
$factory->state(Invoice::class, 'pending', [
    'status' => PaidInvoiceState::class,
]);
```

First of all, your IDE has no clue what kind of object `$factory` actually is. It
magically exists in factory files, though there's no autocompletion on it. A
quick fix is to add this DocBlock, though that's cumbersome.

```php
/** @var \Illuminate\Database\Eloquent\Factory $factory */
$factory->state(/* … */);
```

Second, states are defined as strings, making them a black box when using a
factory in tests.

```php
public function test_case()
{
    $invoice = factory(Invoice::class)
        ->states(/* what states are actually available here? */)
        ->create();
}
```

Third, there's no type hinting on the result of a factory. Your IDE doesn't know that `$invoice` actually is an `Invoice` model; again: a black box.

```
public function test_case()
{
    $invoice = factory(Invoice::class)->create();

    $invoice->/* … ?? */;
}
```

Next, given a large enough domain, you might need more than just a few states in your test suite, which become challenging to manage over time.

And finally, these factories only build models for us. What if we want to use DTOs or requests in our tests?

So we'll look at the factory pattern, and build it from the ground up, made for our tests. They will allow much more flexibility and improve the developer experience significantly. The actual goal of these factory classes is to help you write integration tests, without having to spend too much time setting up the system for it.

Note that I say *"integration tests"* and not *"unit tests"*: when we're testing our domain code, we're testing the core business logic. More often than not, testing this business logic means we won't be testing an isolated piece of a class, but rather a complex and intricate business rule that requires data to be present in the database.

As I've mentioned before: we're talking about large and complex systems in this book; it's important to keep that in mind. In particular, that's why I decided to call these tests integration tests in this chapter; I want to avoid discussions about what unit tests are and what they aren't.

So let's look at the factory pattern.

A basic factory

A test factory is nothing more than a simple class. There's no package to
require, no interfaces to implement or abstract classes to extend. The power
of a factory is not the complexity of the code, but rather one or two patterns
properly applied.

Here's what such a class looks like, simplified:

```
class InvoiceFactory
{
    public static function new(): self
    {
        return new self();
    }

    public function create(array $extra = []): Invoice
    {
        return Invoice::create(array_merge(
            [
                'number' => 'I-1',
                'status' => PendingInvoiceState::class,
                // …
            ],
            $extra
        ));
    }
}
```

Let's discuss a few design decisions.

First of all, the static constructor `new`. You might be confused as to why we need it, as we could simply make the `create` method static. I'll answer that question in depth later in this chapter, but now, you should know that we want this factory to be highly configurable before actually creating an invoice. So rest assured, it will become clear soon.

Secondly, why the name `new` for the static constructor? The answer is a practical one: within the context of factories, `make` and `create` are often associated with a factory actually producing a result. `new` helps us avoid unnecessary confusion.

Finally, the `create` method: it takes an optional array of extra data to ensure we can always make some last-minute changes in our tests.

With our simple example, we can now create invoices like so:

```php
public function test_case()
{
    $invoice = InvoiceFactory::new()->create();
}
```

Before looking at configurability, let's address a little improvement we can make right away: invoice numbers should be unique, so if we create two invoices in one test case, it will break. We don't want to worry about keeping

track of invoice numbers in most cases, so let's have the factory take care of those:

```
class InvoiceFactory
{
    private static int $number = 0;

    public function create(array $extra = []): Invoice
    {
        self::$number += 1;

        return Invoice::create(array_merge(
            [
                'number' => 'I-' . self::$number,
                // …
            ],
            $extra
        ));
    }
}
```

Factories in factories

In the original example with Laravel's factories, I showed that we might want to create a paid invoice using factory states. I was a little naive previously when I assumed this simply meant changing the status field on the invoice model. We also need an actual payment to be saved in the database!

Laravel's default factories can handle this with callbacks, which trigger after a model was created, though imagine what happens if you're managing several, maybe even tens of states, each with their side effects. A simple `$factory->afterCreating` hook just isn't robust enough to manage all this in a sane way.

So, let's turn things around. Let's properly configure our invoice factory, before creating the actual invoice.

```php
class InvoiceFactory
{
    private ?string $status = null;

    public function create(array $extra = []): Invoice
    {
        $invoice = Invoice::create(array_merge(
            [
                'status' => $this->status ?? PendingInvoiceState::class
            ],
            $extra
        ));

        if ($invoice->status->isPaid()) {
            PaymentFactory::new()->forInvoice($invoice)->create();
        }

        return $invoice;
    }

    public function paid(): self
    {
        $clone = clone $this;

        $clone->status = PaidInvoiceState::class;

        return $clone;
    }
}
```

If you're wondering about that `clone` by the way, we'll look at it later.

The thing we've made configurable is the invoice status, just like factory states in Laravel would do, but in our case, there's the advantage that our IDE knows what we're dealing with:

```php
public function test_case()
{
    $invoice = InvoiceFactory::new()
        ->paid()
        ->create();
}
```

Still, there's room for improvement. Have you seen that check we do after the invoice is created?

```php
if ($invoice->status->isPaid()) {
    PaymentFactory::new()->forInvoice($invoice)->create();
}
```

This can be made more flexible still. We're using a `PaymentFactory` underneath, but what if we want more fine-grained control about how that payment was made? You can, for example, imagine there are some business rules about paid invoices that behave differently depending on the type of payment.

We also want to avoid passing too much configuration directly into the `InvoiceFactory`, because it will become a mess very quickly. So how do we solve this?

Here's the answer: we allow the developer to optionally pass a `PaymentFactory` to `InvoiceFactory` so that this factory can be configured

however the developer wants from outside our `InvoiceFactory`. Here's how that looks:

```php
public function paid(PaymentFactory $paymentFactory = null): self
{
    $clone = clone $this;

    $clone->status = PaidInvoiceState::class;
    $clone->paymentFactory = $paymentFactory ?? PaymentFactory::new();

    return $clone;
}
```

And here's how it's used in the `create` method:

```php
if ($this->paymentFactory) {
    $this->paymentFactory->forInvoice($invoice)->create();
}
```

By doing so, a lot of possibilities arise. In this example, we're making an invoice that's paid, specifically with a Visa payment.

```php
public function test_case()
{
    $invoice = InvoiceFactory::new()
        ->paid(
            PaymentFactory::new()->type(VisaPaymentType::class)
        )
        ->create();
}
```

Here is another example: we want to test how an invoice is handled when it has been paid too late, after the invoice expiration date:

```php
public function test_case()
{
    $invoice = InvoiceFactory::new()
        ->expiresAt('2020-01-01')
        ->paid(
            PaymentFactory::new()->paidAt('2020-05-20')
        )
        ->create();
}
```

With just a few lines of code, we get a lot more flexibility.

Immutable factories

Now, what about that cloning earlier? Why is it essential to make factories immutable? See, sometimes you want to make several models with the same test case, but with small differences. Instead of creating a new factory object for each model, you could reuse the original factory object and only change the things you need.

If you're not using immutable factories there's a chance that you'll end up with data you didn't want. Take the example of the invoice payments: say we need two invoices on the same date, one paid and one pending.

```php
$invoiceFactory = InvoiceFactory::new()
    ->expiresAt(Carbon::make('2020-01-01'));

$invoiceA = $invoiceFactory->paid()->create();
$invoiceB = $invoiceFactory->create();
```

If our `paid` method weren't immutable, it would mean that `$invoiceB` would also be a paid invoice! Sure, we could micro-manage every model creation, but that takes away this pattern's flexibility. That's why immutable functions are great: you can set up a base factory, and reuse it throughout your tests, without worrying about unintended side effects!

With these two principles: configuring factories within factories and making them immutable, many possibilities arise. Sure, it takes some time to write these factories, but they also *save* lots throughout development. In my experience, they are well worth the overhead, as there's much more to gain than their cost.

Ever since using this pattern, I have never looked back at Laravel's built-in factories. There's just too much to gain from this approach.

One downside I can come up with is that you'll need a little more extra code to create several models at once. However if you want to, you can easily add a small piece of code in a base factory class such as this:

```php
abstract class Factory
{
    // Concrete factory classes should add a return type
    abstract public function create(array $extra = []);

    public function times(int $times, array $extra = []): Collection
    {
        return collect()
            ->times($times)
            ->map(fn() => $this->create($extra));
    }
}
```

Also, keep in mind that you can use these factories for other stuff too, not just models. You can use them extensively to set up DTOs, and sometimes even request classes.

So let's look at testing in practice now, where you'll see our factories shine.

Testing DTOs

I'm going to show examples on how to test all the patterns we've discussed in the previous chapters, and I'll start with DTOs, because they are the simplest: you don't test them at all.

So, let me clarify that: the beauty of DTOs is that they are strongly typed, or at least, as strong as PHP allows. That's their only purpose: representing data. And I've already explained back in chapter 2 that a strong type system combined with static analysis prevents a range of otherwise runtime errors.

So as long as our DTOs only do exactly that - representing data in a strongly typed way - then there's nothing to test.

There's, of course, the mapping between DTOs and input data: you manually map data from, for example, a request to the DTO. As you might remember, that's also something we discussed back in chapter 2. Now, if you decide to go with the built-in static constructors on the DTO itself, then yes, there are some tests to be written, specifically to test whether the mapping between input data and DTOs is done correctly.

Here's what such a test would look like:

```php
/** @test */
public function form_booking_store_request()
{
    $unit = UnitFactory::new()->create();

    $dto = BookingData::fromStoreRequest(new BookingStoreRequest([
        'name' => 'test',
        'unit_id' => $unit->id,
        'date_start' => '2020-12-01',
        'date_end' => '2020-12-05',
    ]));

    $this->assertInstanceOf(BookingData::class, $dto);
}
```

First we assert that the happy path, or several happy paths, work as we'd expect — you can see we use a `UnitFactory` to help set up some data and keep the test clean.

So we make a DTO, and then assert whether it's actually an object of the `BookingData` class. And there's really no other assertions we should do, type checks cover everything else.

Next we can test for exceptions, for example when a `unit_id` wasn't passed:
a `ModelNotFoundException` should be thrown:

```
/** @test */
public function from_booking_store_request_without_unit_fails()
{
    $this->expectException(ModelNotFoundException::class);

    BookingData::fromStoreRequest(new BookingStoreRequest([
        'name' => 'test',
        'date_start' => '2020-12-01',
        'date_end' => '2020-12-05',
    ]));
}
```

So really, DTO tests are very simple, and there isn't much more to say about
them. Let's move on to the next test.

Testing actions

As you know from chapter 3, actions can be composed out of other actions. For example, our `CreateInvoiceAction` uses the `CreateInvoiceLineAction` internally:

```
class CreateInvoiceAction
{
    private CreateInvoiceLineAction $createInvoiceLineAction;

    public function __construct(
        CreateInvoiceLineAction $createInvoiceLineAction
    ) { /* … */ }

    public function execute(
        InvoiceData $invoiceData
    ): Invoice { /* … */ }
}
```

So, in essence, there are two things to test about actions: whether they do what they are supposed to do, and wether they use their underlying actions in the right way.

In practice, for the `CreateInvoiceAction`, it means that we'll test whether an invoice is correctly saved and created — its properties are set, etc. — but also that the related invoice lines are added to this invoice.

However, we're not going to test whether `CreateInvoiceLineAction` does internally what it should do — saving a new invoice line with all its properties — that functionality will be tested in the `CreateInvoiceLineActionTest`, and not here. Here we will assume that

`CreateInvoiceLineAction` works as expected, and we'll only test that it's
actually used within our action:

```php
/** @test */
public function invoice_is_saved_in_the_database()
{
    $invoiceData = InvoiceDataFactory::new()
        ->addInvoiceLineDataFactory(
            InvoiceLineDataFactory::new()
                ->withDescription('Line A')
                ->withItemAmount(1)
                ->withItemPrice(10_00)
        )
        ->addInvoiceLineDataFactory(
            InvoiceLineDataFactory::new()
                ->withDescription('Line B')
                ->withItemAmount(3)
                ->withItemPrice(33_00)
        )
        ->create();

    $action = app(CreateInvoiceAction::class);

    $invoice = $action->execute($invoiceData);

    $this->assertDatabaseHas($invoice->getTable(), [
        'id' => $invoice->id,
    ]);

    $this->assertNotNull($invoice->number);

    $expectedTotalPrice = 1 * 10_00 + 3 * 33_00;

    $this->assertEquals(
        $expectedTotalPrice,
        $invoice->total_price
    );

    $this->assertCount(2, $invoice->invoiceLines);
}
```

Ok, there are lots of things happening here, so let's break them down:

First there's a setup phase to our test, where we create some DTOs (with our factories), and resolve the action from the container:

```
$invoiceData = InvoiceDataFactory::new()
    ->addInvoiceLineDataFactory(
        InvoiceLineDataFactory::new()
            ->withDescription('Line A')
            ->withItemAmount(1)
            ->withItemPrice(10_00)
    )
    ->addInvoiceLineDataFactory(
        InvoiceLineDataFactory::new()
            ->withDescription('Line B')
            ->withItemAmount(3)
            ->withItemPrice(33_00)
    )
    ->create();

$action = app(CreateInvoiceAction::class);
```

Next there's executing the action (which is only one line), and this is what we're actually going to test:

```
$invoice = $action->execute($invoiceData);
```

And, finally, we're doing a bunch of assertions: whether the invoice is actually saved in the database, whether a number was generated, whether the total

price was calculated and saved, and whether all invoice lines were linked to
our invoice:

```
$this->assertDatabaseHas($invoice->getTable(), [
    'id' => $invoice->id,
]);

$this->assertNotNull($invoice->number);

$expectedTotalPrice = 1 * 10_00 + 3 * 33_00;

$this->assertEquals(
    $expectedTotalPrice,
    $invoice->total_price
);

$this->assertCount(2, $invoice->invoiceLines);
```

If you want to, you can split all these assertions to separate test methods, and
move the setup part to the test class `setUp` function, I would even encourage
that. But I also wanted to show you everything combined into one, so that you
can see that with little setup, and executing one line of code, we're actually
testing lots of business functionality.

It's a pattern that will always be repeated with actions: setup, execute, assert.
Three steps that return, again and again.

By the way, there's an interesting sidenote to make here about invoice
generation side effects. You could imagine that creating an invoice would also
result in a generated PDF, probably triggered from within our
`CreateInvoiceAction`. Generating PDFs, or sending emails, or anything IO
related usually takes lots of time, and we don't want this to happen every time
we're testing our actions.

Sure we will need one or two tests to check whether a PDF is correctly generated, but for example, in this test, we're not concerned about it.

That's where composing actions out of other actions offer lots of flexibility: there will be a `GeneratePdfAction`, which is injected via the constructor in our `CreateInvoiceAction`:

```
class CreateInvoiceAction
{
    private CreateInvoiceLineAction $createInvoiceLineAction;

    private GeneratePdfAction $generatePdfAction;

    public function __construct(
        CreateInvoiceLineAction $createInvoiceLineAction,
        GeneratePdfAction $generatePdfAction
    ) { /* … */ }

    public function execute(
        InvoiceData $invoiceData
    ): Invoice { /* … */ }
}
```

Now if we want to replace that `GeneratePdfAction` with a mock, we can
simply register it in the container:

```php
namespace Tests\Mocks\Actions;

use Domain\Pdf\Actions\GeneratePdfAction;

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

    public function execute(ToPdf $toPdf): void
    {
        // Simply don't generate the PDF

        return;
    }
}
```

Now, anytime we don't want PDFs to be generated, we can call
`MockGeneratePdfAction::setUp` in, for example, the `setUp` function of our test
class, and we don't have to worry about it anymore!

Testing models

Chapter 4 showed several possible ways to refactor model code to separate
classes to keep your model classes manageable. We already tested calculated
properties in our actions, but there are still collections, query builder and

subscriber tests. Because we're abstracting functionality to smaller classes, these tests are also pretty straightforward.

Take a look at this `InvoiceLineCollectionTest`:

```php
/** @test */
public function only_negative_lines()
{
    $factory = InvoiceLineFactory::new();

    $negativeLine = $factory->withItemPrice(-1_00)->create();

    $collection = new InvoiceLineCollection([
        $negativeLine,
        $factory->withItemPrice(1_00)->create(),
    ]);

    $this->assertCount(1, $collection->onlyNegatives());
    $this->assertTrue(
        $negativeLine->is($collection->onlyNegatives()->first())
    );
}
```

We make two invoice lines: one with a negative price, and one with a positive price, and we'll test whether `InvoiceLineCollection::onlyNegatives` returns the right result.

Again the same pattern emerges setup, execution, and assertion. Again, these tests are so simple, because we can use our test factories. So we don't have to worry about what's going on behind the scenes.

Here's a query builder test, checking whether only active units are found when `UnitQueryBuilder::whereActive` is used:

```php
/** @test */
public function where_active()
{
    $factory = UnitFactory::new();

    $activeUnit = $factory->active()->create();

    $inactiveUnit = $factory->inactive()->create();

    $this->assertEquals(
        1,
        Unit::query()
            ->whereActive()
            ->whereKey($activeUnit->id)
            ->count()
    );

    $this->assertEquals(
        0,
        Unit::query()
            ->whereActive()
            ->whereKey($inactiveUnit->id)
            ->count()
    );
}
```

And finally, testing event subscribers. Of course, we need to test whether it listens to the correct events, but also whether the listeners do what we expect them to do.

Did you know you don't have to use a model to test that last point? You can simply test the subscriber itself. It will keep your tests even smaller since you

can simply call the listener method directly and give it a self-made event object:

```php
/** @test */
public function test_saving()
{
    $subscriber = app(InvoiceSubscriber::class);

    $event = new InvoiceSavingEvent(InvoiceFactory::new()->create());

    $subscriber->saving($event);

    $invoice = $event->invoice;

    $this->assertNotNull($invoice->total_price);
}
```

As objects are always passed by reference in PHP, you can simply do assertions on the $invoice object itself.

Entering the application layer

Back in chapter 1, I wrote that one of the characteristics of domain-oriented Laravel projects is the following:

> […] it's very important is that you start thinking in groups of related business concepts, rather than in groups of code with the same technical properties.

In other words: group your code based on what it resembles in the real world, instead of its purpose in the code base.

I also wrote that domain and application code are two separate things. Moreover, applications are allowed to use several groups of the domain at once if they need to expose that domain functionality to the end user.

But what exactly belongs in this application layer? How do we group code over there? These questions will be answered in this chapter.

We're entering the application layer.

Several applications

The first important thing to understand is that one project can have several applications. In fact, every Laravel project already has two by default: the HTTP and console apps. Still, there are several others parts of a project that can be thought of as standalone apps: every third-party integration, a REST API, a front-facing client portal, an admin back office, and what-not.

All of these can be thought of as separate applications, exposing and presenting the domain for their own unique use cases. In fact, I tend to think of the artisan console as just another one in this list: it's an application used by developers to work with and manipulate the project.

Though, since we're in web development, our main focus will probably be on HTTP-related apps. So what's included in them? Let's have a look:

- Controllers
- Requests
- Request-specific validation rules
- Middleware
- Resources
- ViewModels
- HTTP QueryBuilders (the ones that parse URL queries)

I would even argue that Blade views, JavaScript and CSS files belong within an application, and shouldn't be put aside in a `resources` folder. I realise this is a step too far for many people, but I wanted to mention it, at least.

In general, an application's goal is to get some kind of input, transform it for use in the domain layer, and represent the output to the user or store it somewhere. By now it probably isn't a surprise that most application code is

merely structural, often even boring code, simply passing data from one point
to another. After all: all complex logic is handled by our domain code.

Nevertheless there's lots to tell about several of the concepts mentioned
above, so we'll explore the application layer in the next chapters. But first let's
focus on the general architecture of the application layer.

Structuring HTTP applications

There's one very important point we need to discuss before moving on: how
will an HTTP application generally be structured? Should we follow Laravel's
conventions, or do we need to give it some more thought?

Since I'm dedicating a section of a chapter to this question, you can probably
already guess the answer. So let's look at what Laravel would recommend you
do in a classic HTTP application by default.

```
App/Admin
├── Http
│   ├── Controllers
│   ├── Kernel.php
│   └── Middleware
├── Requests
├── Resources
├── Rules
└── ViewModels
```

This structure is fine in small projects, but to be honest, it doesn't scale well.
To clarify what I mean by this, I'll show you the structure of an admin
application in one of our client projects. Obviously I can't reveal too much
information about this project, so I blacked out most of the class names. Note:
since we've been using invoicing as an example throughout this series, I

highlighted some invoice related classes within the admin application. Have a
look, and enjoy the page-turning!

```
App/Admin
├── Controllers
│   ├── ████████
│   │   ├── ████████████████████.php
│   │   ├── ████████████████████████████.php
│   │   ├── ██████████████████████.php
│   │   ├── █████████████████████████████.php
│   │   ├── ██████████████████████.php
│   │   ├── ████████████████████████.php
│   │   ├── ████████████████████████.php
│   │   ├── █████████████████████████████.php
│   │   ├── ██████████████████.php
│   │   └── ████████████████.php
│   ├── ████████████
│   │   ├── █████████████████████████████.php
│   │   ├── ████████████████
│   │   │   ├── ███████████████████████████████.php
│   │   │   ├── ████████████████████████████████████████.php
│   │   │   ├── ██████████████████████████████.php
│   │   │   ├── ██████████████████████████████.php
│   │   │   ├── ██████████████████████████████.php
│   │   │   └── ██████████████████████████████.php
│   │   ├── ███████████████████████████.php
│   │   ├── ███████████████████████████.php
│   │   ├── ███████████████████████████.php
│   │   ├── █████████████████████████.php
│   │   ├── █████████████████████████.php
│   │   ├── ██████████████████████████████████████.php
│   │   ├── ████████████████████████████████████.php
│   │   ├── ███████████████████████████.php
│   │   ├── ██████████████████████████████████████.php
│   │   ├── ███████████████████████████.php
│   │   ├── ██████████████
│   │   │   ├── ████████████████████████████.php
│   │   │   ├── ██████████████████████████.php
│   │   │   ├── ████████████████████████████.php
│   │   │   ├── ██████████████████████████████████████████.php
│   │   │   ├── ███████████████████████████████████████.php
```

```
│   │   │   ├── ███████████████████████████.php
│   │   │   └── ███████████████████████████.php
│   │   ├── █████████████████████████.php
│   │   └── ███████████████████.php
│   ├── ██
│   │   ├── ████████████████.php
│   │   ├── ██████████████████.php
│   │   ├── ██████████████████.php
│   │   ├── █████████.php
│   │   ├── ██████████████████.php
│   │   ├── ██████████████.php
│   │   ├── ██████████████.php
│   │   ├── █████████████████.php
│   │   ├── █████████████████.php
│   │   ├── █████████████.php
│   │   ├── █████████████.php
│   │   ├── █████████████.php
│   │   ├── ████████████████████.php
│   │   ├── ████████████████.php
│   │   ├── ██████████████.php
│   │   └── ████████████.php
│   ├── █████████████.php
│   ├── █████████████.php
│   ├── ██████
│   │   ├── ███████████████████.php
│   │   ├── ██████████████.php
│   │   ├── ███████████████.php
│   │   ├── ███████████████.php
│   │   ├── ████████████████
│   │   │   ├── ██████████████████████████.php
│   │   │   ├── ██████████████████████████.php
│   │   │   ├── ████████████████████████.php
│   │   │   ├── ███████████████████████.php
│   │   │   ├── ███████████████████████.php
│   │   │   ├── ██████████████████████████.php
│   │   │   ├── ███████████████████████.php
│   │   │   ├── ██████████████████████████.php
│   │   │   └── █████████████████████████.php
│   │   ├── Invoices
│   │   │   ├── ████████████████████████.php
│   │   │   ├── █████████████████.php
│   │   │   ├── IgnoreMissedInvoicesController.php
```

```
│   │   │   ├── ████████████████████.php
│   │   │   ├── ██████████████████.php
│   │   │   ├── InvoiceStatusController.php
│   │   │   ├── InvoicesController.php
│   │   │   ├── MissedInvoicesController.php
│   │   │   ├── █████████████████████.php
│   │   │   └── RefreshMissedInvoicesController.php
│   │   ├── ███████
│   │   │   └── █████████████████████.php
│   │   ├── ████████████████.php
│   │   └── ████████████████.php
│   ├── █████████████
│   │   ├── ████████████████.php
│   │   ├── ██████████████████.php
│   │   ├── ██████████████.php
│   │   ├── ██████████████.php
│   │   ├── ██████████████.php
│   │   ├── ████████████████.php
│   │   ├── █████████████████.php
│   │   ├── █████████████████████.php
│   │   ├── █████████████████████.php
│   │   ├── ████████████████.php
│   │   ├── ██████████████████.php
│   │   ├── ███████████████████.php
│   │   ├── ██████████████.php
│   │   ├── ██████████████.php
│   │   ├── █████████████.php
│   │   ├── ████████████████.php
│   │   ├── ██████████████████.php
│   │   ├── ████████████████.php
│   │   ├── ██████████████.php
│   │   ├── █████████████.php
│   │   ├── █████████████████.php
│   │   ├── █████████████.php
│   │   ├── ██████████████████.php
│   │   ├── ████████████.php
│   │   ├── ██████████████████.php
│   │   ├── █████████████.php
│   │   ├── ████████████.php
│   │   ├── ██████████████████.php
│   │   ├── ██████████████.php
│   │   ├── ████████████████.php
```

```
│   │   ├── ██████████████████.php
│   │   ├── ███████████████████.php
│   │   ├── ██████████████.php
│   │   ├── ███████████.php
│   │   ├── ██████████.php
│   │   ├── ███████████.php
│   │   ├── █████████████████.php
│   │   ├── █████████████.php
│   │   ├── █████████.php
│   │   ├── █████████████.php
│   │   ├── █████████.php
│   │   ├── ██████████.php
│   │   ├── █████████.php
│   │   ├── █████████.php
│   │   ├── ███████████████.php
│   │   ├── ██████████████.php
│   │   ├── ████████████.php
│   │   ├── █████████████.php
│   │   ├── ███████████.php
│   │   ├── ███████.php
│   │   └── █████████.php
│   ├── █████
│   │   └── █████████.php
│   └── ███████.php
├── Filters
│   ├── ███████████████.php
│   ├── ████████████████.php
│   ├── █████████████████.php
│   ├── ██████████████████████.php
│   ├── ████████████.php
│   ├── ██████████████████.php
│   ├── █████████████████.php
│   ├── ████████████.php
│   ├── ███████.php
│   ├── ███████.php
│   ├── ██████████.php
│   ├── InvoiceMonthFilter.php
│   ├── InvoiceOfferFilter.php
│   ├── InvoiceStatusFilter.php
│   ├── InvoiceYearFilter.php
│   ├── ██████████████.php
│   ├── ██████████.php
```

```
|       └── ████████████████.php
├── Middleware
|       ├── ███████████████████████.php
|       ├── ███████████████.php
|       ├── ████████████████████████.php
|       ├── ██████████████████████████.php
|       ├── EnsureValidHabitantInvoiceCollectionSettingsMiddleware.php
|       ├── EnsureValidInvoiceDraftSettingsMiddleware.php
|       ├── ████████████████████████.php
|       ├── EnsureValidOwnerInvoiceCollectionSettingsMiddleware.php
|       ├── ██████████████████.php
|       ├── ██████████████████.php
|       ├── █████████████████████████.php
|       ├── ███████████████████████████.php
|       ├── ██████████████████.php
|       ├── ██████████████████.php
|       └── ███████████████.php
├── Queries
|       ├── ███████████████.php
|       ├── ████████████████████████.php
|       ├── ███████████████.php
|       ├── ██████████████████████.php
|       ├── ████████████████████.php
|       ├── ███████████████.php
|       ├── ████████████████████.php
|       ├── ███████████████.php
|       ├── ██████████████████████.php
|       ├── ██████████████████████████.php
|       ├── ██████████████████████.php
|       ├── ██████████████████.php
|       ├── ████████████████████.php
|       ├── ██████████████████.php
|       ├── ██████████████████.php
|       ├── ██████████████████████.php
|       ├── ████████████████████.php
|       ├── ████████████████████.php
|       ├── ███████████████.php
|       ├── ██████████████████████████.php
|       ├── ████████████████████████.php
|       ├── ████████████████████.php
|       ├── InvoiceCollectionIndexQuery.php
|       ├── InvoiceIndexQuery.php
```

```
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓.php
|   └── ▓▓▓▓▓▓▓▓▓▓.php
├── Requests
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── InvoiceRequest.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓.php
|   ├── ▓▓▓▓▓▓▓▓▓.php
```

```
│   └── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
├── Resources
│   ├── ▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
│   ├── ▓▓▓▓▓▓▓▓▓▓▓▓▓.php
```

```
│   ├── ████████████████████.php
│   ├── █████████████████████.php
│   ├── █████████████.php
│   ├── ███████████████████.php
│   ├── ███████████████████.php
│   ├── ███████████████.php
│   ├── Invoices
│   │   ├── InvoiceCollectionDataResource.php
│   │   ├── InvoiceCollectionResource.php
│   │   ├── InvoiceDataResource.php
│   │   ├── InvoiceDraftResource.php
│   │   ├── InvoiceLineDataResource.php
│   │   ├── InvoiceLineResource.php
│   │   ├── InvoiceResource.php
│   │   ├── ███████████████.php
│   │   ├── ███████████████.php
│   │   └── ████████████.php
│   ├── InvoiceIndexResource.php
│   ├── InvoiceLabelResource.php
│   ├── InvoiceMainOverviewResource.php
│   ├── InvoiceeResource.php
│   ├── ████████████████████.php
│   ├── ██████████.php
│   ├── █████████████.php
│   ├── ██████████████████████.php
│   ├── ████████████████.php
│   ├── ███████████████.php
│   ├── █████████████.php
│   ├── ████████████████████████.php
│   ├── ███████████████████████.php
│   ├── ████████████████.php
│   ├── ██████████████████████.php
│   ├── ████████████.php
│   ├── ████████████████████.php
│   ├── █████████████████████.php
│   ├── ██████████████.php
│   ├── ██████████████.php
│   ├── ██████████████.php
│   ├── ████████████████████.php
│   ├── ████████████.php
│   ├── ████████████████.php
│   ├── █████████████████.php
```

```
│   ├── ████████████████████████.php
│   ├── ███████████.php
│   ├── ██████████████████████████.php
│   ├── █████████████████.php
│   ├── ██████████████████.php
│   ├── █████████████.php
│   ├── █████████████████.php
│   ├── ████████████.php
│   ├── ██████████████████████.php
│   ├── ███████████████.php
│   ├── ████████████████.php
│   ├── ████████████████████████.php
│   ├── ████████████████.php
│   ├── ██████████████████████.php
│   ├── ███████████████.php
│   ├── ████████████████.php
│   ├── █████████████████.php
│   ├── █████████████.php
│   ├── ████████████████.php
│   ├── █████████████.php
│   └── █████████████████.php
└── ViewModels
    ├── ██████████████████.php
    ├── ███████████████.php
    ├── █████████████████████.php
    ├── ███████████████.php
    ├── █████████████████.php
    ├── ███████████████████████████.php
    ├── ████████████████████████.php
    ├── ████████████████████.php
    ├── █████████████████████████.php
    ├── █████████████████████.php
    ├── █████████████.php
    ├── █████████████████████████.php
    ├── ████████████████████.php
    ├── █████████████████.php
    ├── ███████████.php
    ├── ██████████████.php
    ├── ███████████████.php
    ├── █████████████████.php
    ├── █████████████████.php
    ├── ████████████████████.php
```

```
├── ████████████████.php
├── ████████████.php
├── █████████████.php
├── █████████████.php
├── █████████████.php
├── █████████████████████.php
├── ███████████████████████.php
├── ██████████████████.php
├── ███████████████.php
├── ███████████████████.php
├── ████████████.php
├── ███████████████████.php
├── ████████████████
│   ├── █████████████████.php
│   ├── ████████████████.php
│   ├── ████████████████.php
│   └── ██████████████████.php
├── ██████████████████.php
├── ████████████████████████.php
├── ████████████████████████.php
├── ██████████████████.php
├── ██████████████.php
├── InvoiceCollectionHabitantContractPreviewViewModel.php
├── InvoiceCollectionOwnerContractPreviewViewModel.php
├── InvoiceCollectionPreviewViewModel.php
├── InvoiceDraftViewModel.php
├── InvoiceIndexViewModel.php
├── InvoiceLabelsViewModel.php
├── InvoiceStatusViewModel.php
├── ██████████████.php
├── ███████████████.php
├── ███████████████.php
├── ██████████.php
├── █████████████.php
├── ██████████████.php
├── ███████████.php
├── ██████████████.php
├── █████████████████.php
├── ██████████████████████.php
├── ████████████.php
├── ████████████.php
├── ███████████████.php
```

Hi again!

That was quite a lot, right? I'm not kidding though - this is what one of our projects actually looked like after a year and a half of development. And keep in mind that this is only the admin application code, it doesn't include anything domain related.

So what is the core problem here? It's actually the same as with our domain code back in chapter 1: we're grouping our code based on technical properties, instead of their real-world meaning; controllers with controllers, requests with requests, view models with view models, and so on.

Once again a concept like invoices is spread out across multiple directories, and mixed with dozens of other classes. Even with the best IDE support, it's very difficult to wrap your head around the application as a whole and there's no way to get a general overview of what's happening.

The solution — no surprises here, I hope — is the same as we did with domains: group together code that belongs together. In this example, invoices:

```
Admin
└── Invoices
    ├── Controllers
    │   ├── IgnoreMissedInvoicesController.php
    │   ├── InvoiceStatusController.php
    │   ├── InvoicesController.php
    │   ├── MissedInvoicesController.php
    │   └── RefreshMissedInvoicesController.php
    ├── Filters
    │   ├── InvoiceMonthFilter.php
    │   ├── InvoiceOfferFilter.php
    │   ├── InvoiceStatusFilter.php
    │   └── InvoiceYearFilter.php
    ├── Middleware
    │   ├── EnsureValidHabitantInvoiceCollectionSettingsMiddleware.php
    │   ├── EnsureValidInvoiceDraftSettingsMiddleware.php
    │   └── EnsureValidOwnerInvoiceCollectionSettingsMiddleware.php
    ├── Queries
    │   ├── InvoiceCollectionIndexQuery.php
    │   └── InvoiceIndexQuery.php
    ├── Requests
    │   └── InvoiceRequest.php
    ├── Resources
    │   ├── InvoiceCollectionDataResource.php
    │   ├── InvoiceCollectionResource.php
    │   ├── InvoiceDataResource.php
    │   ├── InvoiceDraftResource.php
    │   ├── InvoiceIndexResource.php
    │   ├── InvoiceLabelResource.php
    │   ├── InvoiceLineDataResource.php
    │   ├── InvoiceLineResource.php
    │   ├── InvoiceMainOverviewResource.php
    │   ├── InvoiceResource.php
    │   └── InvoiceeResource.php
    └── ViewModels
        ├── InvoiceCollectionHabitantContractPreviewViewModel.php
```

```
├── InvoiceCollectionOwnerContractPreviewViewModel.php
├── InvoiceCollectionPreviewViewModel.php
├── InvoiceDraftViewModel.php
├── InvoiceIndexViewModel.php
├── InvoiceLabelsViewModel.php
└── InvoiceStatusViewModel.php
```

How about that? When you're working on invoices, you've got one place to go to know what code is available to you. I tend to call these groups "application modules", or "modules" for short; and I can tell you from experience that they make life a lot easier when you're working in projects of this scale.
Does this mean modules should be one-to-one mapped on the domain? Definitely not! Mind you, there could be some overlap, but it's not required. For example: we've got a settings module within the admin application, which touches several domain groups at once. When we're working on settings, it's one feature on its own and its code should be grouped together, not spread out across several modules just to be in sync with the domain.

Another question that might arise looking at this structure, is: what should we do with general purpose classes? Stuff like a base request class, middleware that's used everywhere, etc. Remember the `Support` namespace back in chapter 1? That's what it is for! `Support` holds all code that should be globally accessible and could just as well have been part of the framework or a standalone package.

Now that you have a general overview of how to structure applications, and why it's useful to think about it, it's time to look at some of the patterns we use over there to make our lives easier.

View models

It's time to take a first deep dive into the application layer. One of the core ideas in this book is to keep code clean, concise and manageable. This chapter won't be any different, as we'll look at how to keep controllers clean and to-the-point.

The pattern we'll use to help us is called the view model pattern. As its name suggests, these classes are models to your view files; they are responsible for providing data to a view, which would otherwise come directly from the controller or the domain model. In addition, they allow a better separation of concerns, and provide more flexibility for the developer.

In essence, view models are simple classes that take data and transform it into something usable for the view. In this chapter I'll show you the basic principles of the pattern, we'll take a look at how they integrate in Laravel projects, and finally I'll show you how we use the pattern in one of our projects.

Let's start with a simplified example. Say you have a form which creates a blog post with a category. You'll need a way to fill the select box in the view with category options and the controller has to provide those.

```php
public function create()
{
    return view('blog.form', [
        'categories' => Category::all(),
    ]);
}
```

The above example works for the create method, but let's not forget we should also be able to edit existing posts.

```php
public function edit(Post $post)
{
    return view('blog.form', [
        'post' => $post,
        'categories' => Category::all(),
    ]);
}
```

Next there's a new business requirement: users should be restricted as to which categories they are allowed to post in. In other words, the category selection should be restricted based on the user.

```php
return view('blog.form', [
    'categories' => Category::allowedForUser(
        current_user()
    )->get(),
]);
```

Unfortunately, however, this approach doesn't scale because to begin with, you'll have to change code both in the `create` and `edit` method. Then, can

you imagine what happens when you need to add tags to a post? Or if there's another special admin form for creating and editing posts?

The next solution is to have the post model itself provide the categories, like so:

```
class Post extends Model
{
    public static function allowedCategories(): Collection
    {
        return Category::query()
            ->allowedForUser(current_user())
            ->get();
    }
}
```

There are numerous reasons why this is a bad idea, though I've seen this approach used often in Laravel projects. Let's focus on the most relevant problem for our case: it still allows for duplication.

Say there's another model `News` which also needs the same category selection. This causes duplication, but on the model level instead of in the controllers.

Another option is to put the `allowedCategories` method on the `User` model. This makes the most sense, but also makes maintenance harder. For example, imagine we're using tags as mentioned before; they don't rely on the user but now we need to get the categories from the user model, and tags from somewhere else.

I hope it's clear that using models as data providers for views isn't a scalable solution. Wherever you try to get the categories from, there always seems to

be some code duplication which makes it harder to maintain and reason about
the code.

This is where view models come into play because they encapsulate all this
logic so that it can be reused in different places. Critically, they have one
responsibility and one responsibility only: providing the view with the correct
data.

```php
class PostFormViewModel
{
    // …

    public function __construct(User $user, Post $post = null)
    {
        $this->user = $user;
        $this->post = $post;
    }

    public function post(): Post
    {
        return $this->post ?? new Post();
    }

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

Let's name a few key features of such a class:

- All dependencies are injected which gives the most flexibility to the outside
  context.
- The view model exposes some methods that can be used by the view.
- There will either be a new or existing post provided by the `post` method,
  depending on whether you are creating or editing a post.

This is what the controller looks like:

```php
class PostsController
{
    public function create()
    {
        $viewModel = new PostFormViewModel(
            current_user()
        );

        return view('blog.form', compact('viewModel'));
    }

    public function edit(Post $post)
    {
        $viewModel = new PostFormViewModel(
            current_user(),
            $post
        );

        return view('blog.form', compact('viewModel'));
    }
}
```

And finally, it can be used in the view like so:

```blade
<input value="{{ $viewModel->post()->title }}" />
<input value="{{ $viewModel->post()->body }}" />

<select>
    @foreach($viewModel->categories() as $category)
        <option value="{{ $category->id }}">
            {{ $category->name }}
        </option>
    @endforeach
</select>
```

View models in Laravel

The previous example showed a simple class as our view model. This is sufficient to use the pattern, but within Laravel projects there are a few more niceties we can add.

For example, you can pass a view model directly to the `view` function if the view model implements `Arrayable`.

```php
public function create()
{
    $viewModel = new PostFormViewModel(
        current_user()
    );

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

The view can now directly use the view model's properties like `$post` and `$categories`. The previous example now looks like this:

```html
<input value="{{ $post->title }}" />
<input value="{{ $post->body }}" />

<select>
    @foreach($categories as $category)
        <option value="{{ $category->id }}">
            {{ $category->name }}
        </option>
    @endforeach
</select>
```

You can also return the view model itself as JSON data, by implementing
`Responsable`. This can be useful when you're saving the form via an AJAX call
and want to repopulate it with up-to-date data after the call is done.

```
public function update(Request $request, Post $post)
{
    // Update the post…

    return new PostFormViewModel(
        current_user(),
        $post
    );
}
```

You might see a similarity between view models and Laravel resources.
Remember that resources map one-to-one on a model, whereas view models
may provide whatever data they want.

In some places, we're actually using resources and view models combined:

```
class PostViewModel
{
    // …

    public function values(): array
    {
        return PostResource::make(
            $this->post ?? new Post()
        )->resolve();
    }
}
```

View composers

You might be thinking there's some overlap with Laravel's view composers, but don't be mistaken. The Laravel documentation explains view composers like so:

> View composers are callbacks or class methods that are called when a view is rendered. If you have data that you want to be bound to a view each time that view is rendered, a view composer can help you organise that logic into a single location.

View composers are registered like this:

```php
class ViewComposerServiceProvider extends ServiceProvider
{
    public function boot()
    {
        View::composer('profile', ProfileComposer::class);

        View::composer('dashboard', function ($view) {
            // …
        });
    }

    // …
}
```

As you can see, you can both use a class and a closure which you can use to add variables to a view.

Here's how view composers are used in controllers:

```
class ProfileController
{
    public function index()
    {
        return view('profile');
    }
}
```

Can you see them?

No, of course not: view composers are registered somewhere in the global state, and you don't know which variables are available to the view, without that implicit knowledge.

Now I *know* that this isn't a problem in small projects. When you're the only developer and have 20 controllers and maybe 20 view composers, it'll all fit in your head.

But what about the kind of projects we're writing about in this book? When you're working with several developers, in a codebase that counts thousands upon thousands of lines of code, it won't all fit in your head anymore — certainly not on that scale. What's more, we haven't even considered your colleagues and the difficulties they will face, individually and as a team!

That's why the view model pattern is, for these kinds of projects, the better approach. It makes clear from the controller itself what variables are available to the view, and on top of that, you can re-use the same view model for multiple contexts.

One last benefit — one you might not have thought about —

is that we can pass data into the view model explicitly. If you want to use a route argument or bound model to determine data passed to the view, it is done explicitly.

In conclusion: managing global state is a pain in large applications, especially when you're working with multiple developers on the same project. Also remember that just because two solutions have the same end result, it doesn't mean that they are the same!

HTTP queries

In the previous chapter, we covered a very common use case in our applications and provided a pattern to handle it: view models will make sure all data needed by a view is provided in the right form.

There's another everyday use case in large applications: overviews showing collections of data that need to be filtered, paginated, sorted, and searched — also known as data tables.

Whether you're doing full-page refreshes or use AJAX, at scale, you'll always need to ask the server for an updated set of data, based on given conditions. A common practice is to send these conditions via query parameters, read them, and adjust the SQL query accordingly.

I realise there's ambiguity here: there are the HTTP query parameters, and there's the SQL query. Since both are called query it can cause quite a bit of confusion. For example, this chapter will focus on query builders. That is, not the Eloquent model query builders we talked about earlier, but rather classes that will handle the mapping between query parameters and the actual SQL query.

You can see the use of "query" can become unclear rather fast, so I'll try my best to make it as transparent as possible about which kind of "query" I'm referring to.

So, let me tell you upfront that we have a package that takes care of the boring stuff, it's called `spatie/laravel-query-builder`. The mapping between HTTP query parameter values and the model query conditions is done behind the scenes. As you can imagine, there's no rocket science going on over there - we're mapping values from one side to the other.

What's interesting though, is how our package can be used in large projects. Imagine an admin page that shows an overview of all invoices with a few sorts and filters. The default approach with the query builder package would be to do something like this:

```php
class InvoicesController
{
    public function index()
    {
        $invoices = QueryBuilder::for(Invoice::class)
            ->allowedFilters(
                'number',
                'client',
            )
            ->allowedSorts(
                'number',
            )
            ->get();

        // …
    }
}
```

Here we've configured the query builder to only look for specific HTTP query parameters: `filter[number]=`, `filter[client]=` and `sort=number`. All other

query parameters will be ignored. This approach works fine in smaller projects, but in large projects, you're generally dealing with more complex model queries. These queries often have to be shared (albeit with small differences) between several controllers.

It sounds like a good case for extracting these kinds of query builders to their own classes, instead of configuring them on the fly in the controller.

For example, we could make a class called `InvoiceIndexQuery`, and it would look something like this:

```php
namespace App\Admin\Invoices\Queries;

use Spatie\QueryBuilder\QueryBuilder;

class InvoiceIndexQuery extends QueryBuilder
{
    public function __construct(Request $request)
    {
        $query = Invoice::query()
            ->with([
                'invoicee.contact',
                'invoiceLines.article',
            ]);

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

        $this
            ->allowedFilters(
                'number',
                'client',
            )
            ->allowedSorts(
                'number',
            );
    }
}
```

Because we're extending `Spatie\QueryBuilder\QueryBuilder`, we'll need to call the parent constructor. It requires two things: the request, to read the HTTP query parameters from, and the actual model query we're going to modify.

Two interesting things are going on: first of all, we're injecting the request. Since it's already registered in the container by default, we can rely on autowiring to resolve it, but you'll see that later. Second: we're modifying our base query to load some relations eagerly.

It's interesting because you can do whatever you want with the base query before passing it to the parent constructor. You could also, for example, join other tables to allow for more complex filtering:

```php
// …

use Spatie\QueryBuilder\AllowedFilter;
use Support\QueryBuilder\FuzzyFilter;

class InvoiceIndexQuery extends QueryBuilder
{
    public function __construct(Request $request)
    {
        $query = Invoice::query()
            ->join('invoicees', 'invoicees.invoice_id', '=', 'invoices.id')
            ->join('contacts', 'invoicees.contact_id', '=', 'contacts.id';

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

        $this
            ->allowedFilters(
                // …
                AllowedFilter::custom(
                    'search',
                    new FuzzyFilter(
                        'contacts.number',
                    )
                );
    }
}
```

Most importantly, using this approach gets you proper static analysis on the base query, which is, as you know, a valuable asset in large code bases.

However, there's another thing I like about this approach, besides proper static analysis: separated query classes keep the controllers clean! Because

we're relying on the request to be injected, we can inject the query builder
class directly in our controller methods, like so:

```php
class InvoicesController
{
    public function index(InvoiceIndexQuery $invoiceQuery)
    {
        $invoices = $invoiceQuery->get();

        // …
    }
}
```

Of course, you could also inject the query builder in the view model, and
handle everything from there.

Once more, using these dedicated query builder classes keeps our controllers
clean and allows for reusability.

Also note that every eloquent method you're used to is available in our
custom query class. That means you could add some last-minute, controller-
specific changes to your query builder, on the fly. For example, take a
`PaidInvoicesController`, which should only show the paid invoices.

```php
class PaidInvoicesController
{
    public function index(InvoiceIndexQuery $invoiceQuery)
    {
        $invoices = $invoiceQuery
            ->whereStatus(InvoiceStatus::PAID())
            ->get();

        // …
    }
}
```

You can imagine the flexibility: you don't have to specify the filters, sorting, eager loads, etc. over and over again, yet you can still make very specific changes to the query builder for specific cases, in the controller.

There's a lot more to tell about query builders, and you can read all about them in their docs. You can find the relevant links in the footnotes chapter if you want to check them out!

Jobs

There's one important concept of the application layer that I haven't mentioned yet: jobs. Why do I consider them to be part of the application layer and not domain code? After all, the Laravel docs clearly describes jobs as the place to be when you're trying to execute asynchronous business logic.

I want to answer that question in this chapter.

See, the way I think of jobs and their responsibility is to execute code on a bus, a pipeline where jobs come in, and are handled in sequence. Now, that doesn't have to be an asynchronous pipeline per se, but in practice and production, this will most likely be the case.

And let me just say as a sidenote how thankful we should be with the infrastructure Laravel provides out of the box and how we're completely free to manage those jobs: it's as easy as dispatching a class, and Horizon is a great piece of software that scales really well.

Anyways, back to why jobs belong in the application layer. I think of their responsibility as managing that pipeline workflow. They are classes that are responsible for: configuring whether they are queueable, how many of them

can be run simultaneously, if their execution should be delayed, if they should be chained, etc.

In a way, jobs have lots of similarities to controllers:

- They receive input: controllers from the request and jobs specified by the developer (most of the time serialised).
- They are dispatched to process that input: controllers during an HTTP request, jobs mostly on an asynchronous queue.
- They both support middleware to add functionality before and after the actual processing.
- And finally their end result is some kind of output: controllers via HTTP responses, jobs by writing something to the database, sending mails, and of course, much more.

And just like controllers, we want to avoid adding too much functionality in them. This whole book has been about moving code to small classes, each with their own responsibility and all of them easy to comprehend and testable.

So let's not make jobs the exception: let them have their responsibility of managing that queue, and handle the actual business logic just like we did with controllers, by using domain actions. Jobs are just another way of exposing business functionality to the outside world.

And that's why jobs themselves, just like controllers, belong in the application layer.

Simple action jobs

There are cases, probably even many, where a job's task is simply to execute an action. These jobs often look very simple and similar, something like this:

```php
class SendInvoiceMailJob implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    private Invoice $invoice;

    public function __construct(Invoice $invoice)
    {
        $this->invoice = $invoice;
    }

    public function handle(
        SendInvoiceMailAction $sendInvoiceMailAction
    ): void {
        $sendInvoiceMailAction->execute($this->invoice);
    }
}
```

And I briefly want to mention that "sending an invoice mail" could very well be important business functionality. There's more to it than just sending the mail: you also want to keep track of which user received which mails; you might need to send extra attachments with it, depending on the type of client; and you can probably think of a few other use cases. That's just to say that indeed, "sending a mail" is domain code.

Anyway, you often end up with these kinds of jobs: one line, executing the action. Moreover, that often feels like an overhead because this is rather boring code, and writing such a class for tens, if not hundreds of actions, doesn't feel all that great.

We felt this pain ourselves when we started using this approach, so naturally we wanted a solution. What we came up with was something that probably isn't the biggest surprise: one job that's able to handle all actions. We added a little syntactical sugar on top of it so that dispatching such a job could be done via the action itself, whilst maintaining static analysis of course.

Here's what the original way of dispatching the above job would look like:

```
dispatch(new SendInvoiceMailJob($invoice));
```

And here's what our solution looks like:

```
$sendInvoiceMailAction
    ->onQueue()
    ->execute($invoice);
```

Calling `onQueue` on the action will make an action job under the hood, and dispatch it when calling `execute`. Keep in mind that your IDE can still properly autocomplete the `execute`. There's a little bit of magic involved, and if you're interested you can check it out since this functionality is a package on its own: it's called `spatie/laravel-queueable-action`.

So in summary: jobs do belong in the application layer, since business functionality shouldn't be directly handled by them. This doesn't mean we need a dedicated job class for simple actions though. With the right abstractions, we can skip all that boilerplate code.

In closing

"When will the book be complete?" — I kept asking myself this question these past few months. As I wrote, more topics popped into my head, and so the book grew. However, at some point I'd have to consider it finished, otherwise you would have never been able to read it.

I think we've arrived at that point.

I think I've written about all the things I wanted to write about, though if you'd ask me about any of these topics in person, we could probably still fill hours of conversation.

I kept the main goal of this book in mind though: it's not about learning patterns or concrete solutions, it's more about learning a mindset. And by doing so I have given you lots of patterns and solutions, but my hope is that, above all, you've learned to critically look at the architecture of your projects, and can start thinking about ways to improve them over time.

I hope that I was able to share my colleagues' and my own thought process with you: I wanted to include you in all the conversations and discussions we had over the past few years and explain why we came to the solutions we use

today. And I hope that you can go through the same process with your colleagues, to find the best solutions for your problems.

In a way, I've been working on this book for years now: first at my previous job where I discovered my passion for code architecture and began to blog about it; at Spatie where I was able to apply all that knowledge, and learn from my colleagues while doing so; then came the blog posts before finally writing this book.

And so it has come to an end, at least for now, because honestly there's lots of stuff I still want to write about. For example, I've been venturing into event sourced applications and micro services the past few months; I also feel that there's lots to write about modern PHP as well. Ideas for the future? Who knows?

So in closing, I'd like to thank you for reading this book, and I truly hope I was able to share my thought process with you, and that it might help you in your future projects. I'd also like to thank Spatie for giving me the opportunity to work on this book, as well as my wife who has supported me during this whole process.

Finally there's one more chapter after this one: a collection of resources that inspired and guided me throughout the years, and influenced the vision I shared in this book tremendously. I hope the knowledge of all those great programmers will also help you, the same way it did for me.

Thanks.

Footnotes

In this chapter I'd like to list all the sources of inspiration that shaped my vision over the years on code architecture. These resources influenced the writing of this book.

As you know, most ideas I've written about aren't my own. I built upon the great work of several developers and years of knowledge. Now that you've finished this book, there's still lots of great material that you can read, watch or learn.

Gary Bernhardt's thoughts on type systems

https://www.destroyallsoftware.com/talks/ideology

Not everyone shares my preference for strong type systems these days. Gary Bernhardt has a fantastic talk about the differences between these two groups of people.

Matthias Noback on the difference between value objects and DTOs

https://github.com/spatie/data-transfer-object/issues/17

Originally, I called DTOs *"Value Objects"*, which have a slightly different meaning. Matthias was so friendly to jump into a GitHub thread to discuss the differences.

Alan Kay's vision of OOP

https://www.youtube.com/watch?v=oKg1hTOQXoY

Alan Kay, who invented the term OOP, explains his original vision of *"object-oriented"*. This talk highly influenced how I think about programming in an OO language.

Refactoring to actions by Freek Van der Herten

https://freek.dev/1371-refactoring-to-actions

Over the years, we applied the principles described in this book in the projects of Spatie; Freek explores how to refactor old projects to a domain-oriented design.

Martin Fowler on transaction scripts

https://martinfowler.com/eaaCatalog/transactionScript.html

Transaction scripts, described by Martin Fowler, are the basis of actions.

Martin Fowler on anemic domain models

https://martinfowler.com/bliki/AnemicDomainModel.html

Martin Fowler also writes about anemic models, and how it's an anti-pattern. My vision leans closer to Alan Kay's: represent processes and data separately, and have them work together.

Custom eloquent collections by Tim MacDonald

https://timacdonald.me/giving-collections-a-voice/

Tim MacDonald gave the inspiration about using custom eloquent collections; a pattern we still enjoy to this day.

Dedicated query builders by Tim MacDonald

https://timacdonald.me/dedicated-eloquent-model-query-builders

Tim also wrote more in-depth about dedicated query builders.

Christopher Okhravi explains state machines in depth

https://www.youtube.com/watch?v=N12L5D78MAA

If you're still unsure about the state pattern, Christopher Okhravi explains it thoroughly in this video.

Symfony's workflow package to build complex state machines

https://symfony.com/doc/current/workflow/workflow-and-state-machine.html

Symfony's workflow package is an excellent alternative to the simplified state package I wrote.

Sandi Metz's vision of truly OO code

https://www.youtube.com/watch?v=29MAL8pJlmQ

Another eye-opener when it comes to object-oriented programming: Sandi Metz shows how to get rid of all if statements, and what it has to do with OO.

Freek gets started with domain-oriented Laravel

https://freek.dev/1486-getting-started-with-domain-oriented-laravel

Another practical introduction to domain-oriented Laravel by Freek.

Tighten's approach to model factories

https://tighten.co/blog/tidy-up-your-tests-with-class-based-model-factories

Both Spatie and Tighten came up with a similar approach to improved test factories at the same time. It's interesting to read about their approach as well.

Spatie packages

https://spatie.be/open-source

Over the past years, numerous packages have been written at our office to improve our domain-oriented Laravel workflow.

- `spatie/data-transfer-object`
  https://github.com/spatie/data-transfer-object
- `spatie/laravel-query-builder`
  https://github.com/spatie/laravel-query-builder
- `spatie/laravel-view-models`
  https://github.com/spatie/laravel-view-models
- `spatie/laravel-model-states`
  https://github.com/spatie/laravel-model-states
- `spatie/enum`
  https://github.com/spatie/enum
- `spatie/laravel-enum`
  https://github.com/spatie/laravel-enum
- `spatie/laravel-queueable-action`
  https://github.com/spatie/laravel-queueable-action