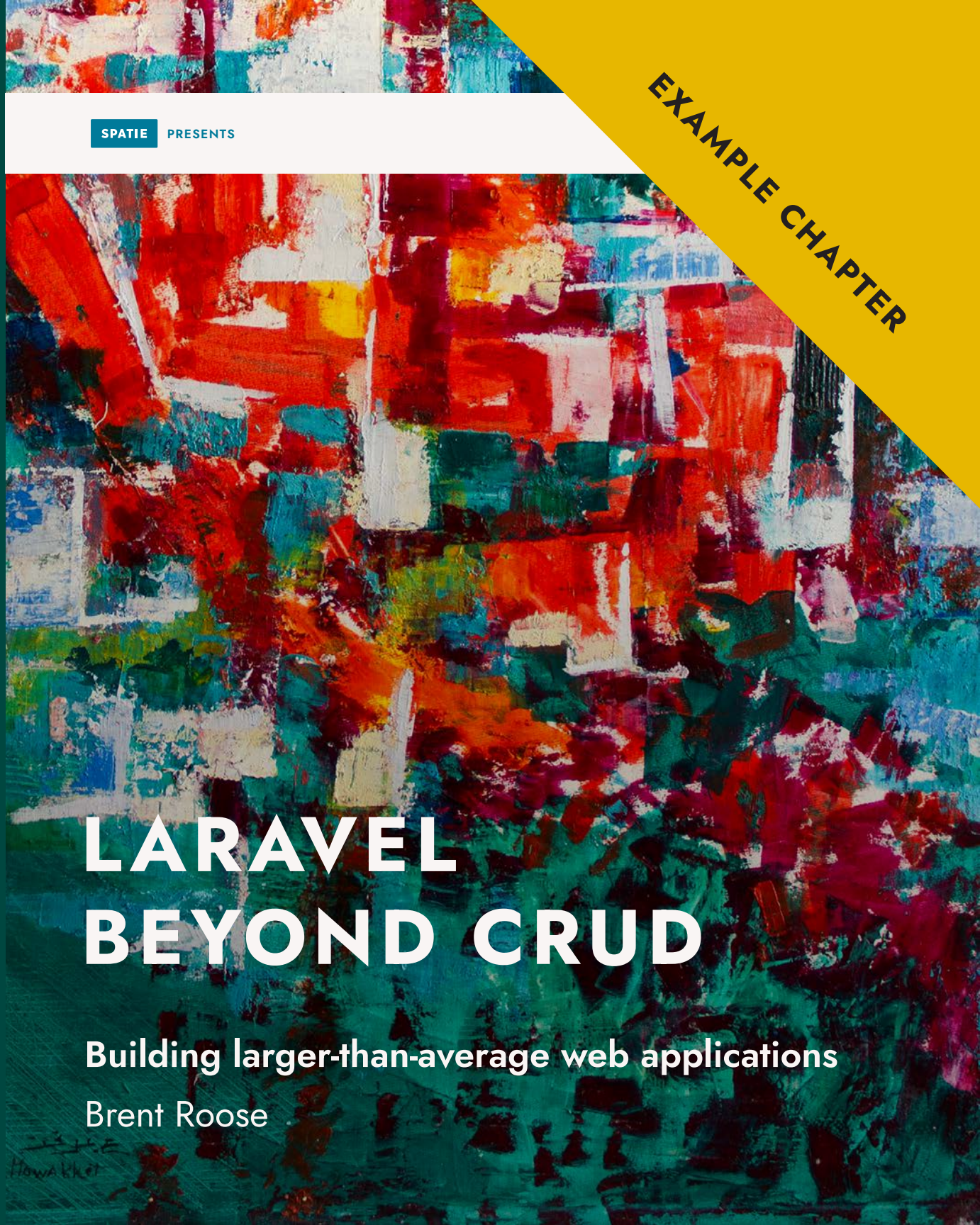
SPATIE PRESENTS
EXAMPLE CHAPTER
LARAVEL
BEYOND CRUD
Building larger-than-average web applications
Brent Roose

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.

```
$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.

```
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.

```
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:

```
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.

```
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.

```
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.

```
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.

```
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:

```
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.

```
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.

```
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.

```
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.