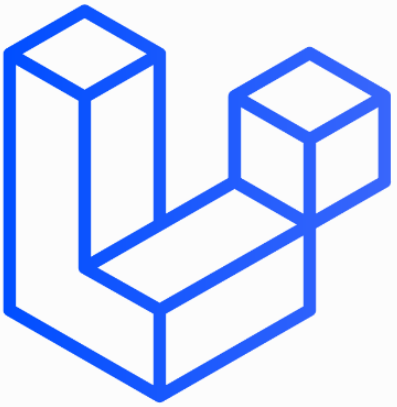
The Clean Coder's Guide to Laravel

Ash Allen Design

ashallendesign.co.uk



Contents

Contents

1

Chapter 1

Cleaning Up Laravel Controllers

Cleaning Up Your Controllers

Introduction

Controllers play a huge role in any MVC (model view controller) based project. They're effectively the "glue" that takes a user's request, performs some type of logic, and then returns a response. If you've ever worked on any fairly large projects, you'll notice that you'll have a lot of controllers and that they can start to get messy quite quickly without you realising. In this chapter, we're going to look at how we can clean up a bloated controller in Laravel.

Intended Audience

This chapter is intended for anyone who is new to Laravel development that understands the basics of controllers. The concepts covered are relatively simple to get started with.

The Problem with Bloated Controllers

Bloated controllers can cause several problems for developers. They can:

1. **Make it hard to track down a particular piece of code or functionality.** If you're looking to work on a particular piece of code that's in a bloated controller, you might need to spend a while tracking down which controller the method is actually in. When using clean controllers that are logically separated this is much easier.
2. **Make it difficult to spot the exact location of a bug.** As we'll see in our code examples later on, if we're handling authorization, validation, business logic, and response building all in one place, it can be difficult to pinpoint the exact location of a bug.
3. **Make it harder to write tests for more complex requests.** It can sometimes be difficult to write tests for complex controller methods

that have a lot of lines and are doing many different things. Cleaning up the code makes testing much easier. We'll cover testing later on in chapter 6.

The Bloated Controller

For this chapter, we're going to use an example UserController:

```
class UserController extends Controller
{
    public function store(Request $request): RedirectResponse
    {
        $this->authorize('create', User::class);

        $request->validate([
            'name'     => 'string|required|max:50',
            'email'    => 'email|required|unique:users',
            'password' => 'string|required|confirmed',
        ]);

        $user = User::create([
            'name'     => $request->name,
            'email'    => $request->email,
            'password' => $request->password,
        ]);

        $user->generateAvatar();
        $this->dispatch(RegisterUserToNewsletter::class);

        return redirect(route('users.index'));
    }

    public function unsubscribe(User $user): RedirectResponse
    {
        $user->unsubscribeFromNewsletter();

        return redirect(route('users.index'));
    }
}
```

To keep things nice and simple to read, I haven't included the `index()`, `create()`, `edit()`, `update()` and `delete()` methods in the controller. But we'll make the assumption that they are there and that we're also using

the below techniques to clean up those methods too. For the majority of the chapter, we'll be focusing on optimizing the `store()` method.

Lift Validation and Authorization into Form Requests

One of the first things that we can do with the controller is to lift any validation and authorization out of the controller and into a form request class. So, let's take a look at how we could do this for the controller's `store()` method.

We'll use the following Artisan command to create a new form request:

```
php artisan make:request StoreUserRequest
```

The above command will have created a new `app/Http/Requests/StoreUserRequest.php` class that looks like this:

```php
class StoreUserRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return false;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            //
        ];
    }
}
```

We can use the `authorize()` method to determine if the user should be allowed to carry out the request. The method should return true if they can and false if they cannot. We can also use the `rules()` method to specify any validation rules that should be run on the request body. Both of these methods will run automatically before we manage to run any code inside our controller methods without us needing to manually call either of them.

So, let's move our authorization from the top of our controller's `store()` method and into the `authorize()` method. After we've done this, we can move the validation rules from the controller and into the `rules()` method. We should now have a form request that looks like this:

```
class StoreUserRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize(): bool
    {
        return Gate::allows('create', User::class);
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules(): array
    {
        return [
            'name'     => 'string|required|max:50',
            'email'    => 'email|required|unique:users',
            'password' => 'string|required|confirmed',
        ];
    }
}
```

Our controller should now also look like this:

```php
class UserController extends Controller
{
    public function store(StoreUserRequest $request): RedirectResponse
    {
        $user = User::create([
            'name'     => $request->name,
            'email'    => $request->email,
            'password' => $request->password,
        ]);

        $user->generateAvatar();
        $this->dispatch(RegisterUserToNewsletter::class);

        return redirect(route('users.index'));
    }

    public function unsubscribe(User $user): RedirectResponse
    {
        $user->unsubscribeFromNewsletter();

        return redirect(route('users.index'));
    }
}
```

Notice how in our controller, we've changed the first argument of the `store()` method from a `\Illuminate\Http\Request` to our new `\App\Http\Requests\StoreUserRequest`. We've also managed to reduce some of the bloat for the controller method by extracting it out into the request class.

Note: For this to work automatically, you'll need to make sure that your controller is using the `\Illuminate\Foundation\Auth\Access\AuthorizesRequests` and `\Illuminate\Foundation\Validation\ValidatesRequests` traits. These come automatically included in the controller that Laravel provides you in a fresh install. So, if you're extending that controller, you're all set to go. If not, make sure to include these traits in your controller.

Move Common Logic into Actions or Services

Another step that we could take to clean up the `store()` method could be to move out our "business logic" into a separate action or service class.

In this particular use case, we can see that the main functionality of the `store()` method is to create a user, generate their avatar and then dispatch a queued job that registers the user to the newsletter. In my personal opinion, an action would be more suitable for this example rather than a service. I prefer to use actions for small tasks that do only one particular thing. Whereas for larger amounts of code that could potentially be hundreds of lines long and do multiple things, it would be more suited to a service.

So, let's create our action by creating a new Actions folder inside our app folder and then creating a new class inside this folder called `StoreUserAction.php`. We can then move the code into the action like this:

```php
class StoreUserAction
{
    public function execute(Request $request): void
    {
        $user = User::create([
            'name'     => $request->name,
            'email'    => $request->email,
            'password' => $request->password,
        ]);

        $user->generateAvatar();
        $this->dispatch(RegisterUserToNewsletter::class);
    }
}
```

Now we can update our controller to use the action:

```
class UserController extends Controller
{
    public function store(
      StoreUserRequest $request,
      StoreUserAction $storeUserAction
    ): RedirectResponse {
        $storeUserAction->execute($request);

        return redirect(route('users.index'));
    }

    public function unsubscribe(User $user): RedirectResponse
    {
        $user->unsubscribeFromNewsletter();

        return redirect(route('users.index'));
    }
}
```

As you can see, we've now been able to lift the business logic out of the controller method and into the action. This is useful because, as I mentioned before, controllers are essentially the "glue" for our requests and responses. So, we've managed to reduce the cognitive load for understanding what a method does by keeping the code logically separated. For example, if we want to check the authorization or validation, we know to check the form request. If we want to check what's being done with the request data, we can check the action.

Another huge benefit to abstracting the code out into these separate classes is that it can make testing a lot easier and faster.

Using DTOs with Actions

Another great benefit of extracting your business logic into services and classes is that you can now use that logic in different places without needing to duplicate your code. For example, let's assume that we have a `UserController` that handles traditional web requests and an

`Api\UserController` that handles API requests. For the sake of argument, we can make the assumption that the general structure of the `store()` methods for those controllers will be the same. But, what would we do if our API request doesn't use an email field, but instead uses an `email_address` field? We wouldn't be able to pass our request object to the `StoreUserAction` class because it would be expecting a request object that has an email field.

To solve this issue, we can use DTOs (data transfer objects). These are a really useful way of decoupling data and passing it around your system's code without it being tightly coupled to anything (in this case, the request). To add DTOs to our project, we'll use Spatie's `spatie/data-transfer-object` package and install it using the following Artisan command:

```
composer require spatie/data-transfer-object
```

Now that we have the package installed, let's create a new DataTransferObjects folder inside our App folder and create a new `StoreUserDTO.php` class. We'll then need to make sure that our DTO extends `Spatie\DataTransferObject\DataTransferObject`. We can then define our three fields like so:

```
class StoreUserDTO extends DataTransferObject
{
    public string $name;

    public string $email;

    public string $password;
}
```

Now that we've done this, we can add a new method to our `StoreUserRequest` from before to create and return a `StoreUserDTO` class like so:

```php
class StoreUserRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize(): bool
    {
        return Gate::allows('create', User::class);
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules(): array
    {
        return [
            'name'     => 'string|required|max:50',
            'email'    => 'email|required|unique:users',
            'password' => 'string|required|confirmed',
        ];
    }

    /**
     * Build and return a DTO.
     *
     * @return StoreUserDTO
     */
    public function toDTO(): StoreUserDTO
    {
        return new StoreUserDTO(
            name: $this->name,
            email: $this->email,
            password: $this->password,
        );
    }
}
```

We can now update our controller to pass the DTO to the action class:

```php
class UserController extends Controller
{
    public function store(
      StoreUserRequest $request,
      StoreUserAction $storeUserAction
    ): RedirectResponse {
        $storeUserAction->execute($request->toDTO());

        return redirect(route('users.index'));
    }

    public function unsubscribe(User $user): RedirectResponse
    {
        $user->unsubscribeFromNewsletter();

        return redirect(route('users.index'));
    }
}
```

Finally, we'll need to update our action's method to accept a DTO as an argument rather than the request object:

```php
class StoreUserAction
{
    public function execute(StoreUserDTO $storeUserDTO): void
    {
        $user = User::create([
            'name'     => $storeUserDTO->name,
            'email'    => $storeUserDTO->email,
            'password' => $storeUserDTO->password,
        ]);

        $user->generateAvatar();
        $this->dispatch(RegisterUserToNewsletter::class);
    }
}
```

As a result of doing all of this, we have now completely decoupled our action from the request object. This means that we can reuse this action in multiple places across the system without being tied to a specific request structure. We would now also be able to use this approach in a CLI environment or queued job that isn't tied to a web request. As an example,

if our application had the functionality to import users from a CSV file, we would be able to build the DTOs from the CSV data and pass it into the action.

To go back to our original problem of having an API request that used `email_address` rather than `email` , we would now be able to solve it by simply building the DTO and assigning the DTO's email field the request's `email_address` field. Let's imagine that the API request had its own separate form request class. It could look like this as an example:

```php
class StoreUserAPIRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize(): bool
    {
        return Gate::allows('create', User::class);
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules(): array
    {
        return [
            'name'          => 'string|required|max:50',
            'email_address' => 'email|required|unique:users',
            'password'      => 'string|required|confirmed',
        ];
    }

    /**
     * Build and return a DTO.
     *
     * @return StoreUserDTO
     */
    public function toDTO(): StoreUserDTO
    {
        return new StoreUserDTO(
            name: $this->name,
            email: $this->email_address,
            password: $this->password,
        );
    }
}
```

Use Resource or Single-use Controllers

A great way of keeping controllers clean is to ensure that they are either "resource controllers" or "single-use controllers". Before we go any further

and try to update our example controller, let's take a look at what both of these terms mean.

A resource controller is a controller that provides functionality based on a particular resource. So, in our case, our resource is the User model and we want to be able to perform all CRUD (create, update, update, delete) operations on this model. A resource controller typically contains `index()`, `create()`, `store()`, `show()`, `edit()`, `update()` and `destroy()` methods. It doesn't necessarily have to include all of these methods, but it wouldn't have any methods that weren't in this list. By using these types of controllers, we can make our routing RESTful. For more information about REST and RESTful routing, check out this article here.

A single-use controller is a controller that only has one public `__invoke()` method. These are really useful if you have a controller that doesn't really fit into one of the RESTful methods that we have in our resource controllers.

Based on the above information, we can see that our `UserController` could probably be improved by moving the unsubscribe method to its own single-use controller. By doing this, we'd be able to make the UserController a resource controller that only includes resource methods.

So let's create a new controller using the following Artisan command:

```
php artisan make:controller UnsubscribeUserController -i
```

Notice how we passed `-i` to the command so that the new controller will be an invokable, single-use controller. We should now have a controller that looks like this:

```
class UnsubscribeUserController extends Controller
{
    public function __invoke(Request $request)
    {
        //
    }
}
```

We can now move our method's code over and delete the unsubscribe method from our old controller:

```
class UnsubscribeUserController extends Controller
{
    public function __invoke(Request $request): RedirectResponse
    {
        $user->unsubscribeFromNewsletter();

        return redirect(route('users.index'));
    }
}
```

Make sure that you remember to switch over your route in your `routes/web.php` file to use the `UnsubscribeController` controller rather than the `UserController` for this method.

Conclusion

This chapter should have given you an insight into the different types of things you can do to clean up your controllers in your Laravel projects. Please remember though that the techniques I've used here are only my personal opinion. I'm sure that there are other developers that would use a totally different approach to building their controllers. The most important part is being consistent and using an approach that fits in with your (and your team's) workflow.

2

Chapter 2

Quick & Easy Tips to Speed Up Your Application

Quick & Easy Tips to Speed Up Your Application

Introduction

It's estimated that 40% of people will leave a website if it takes longer than 3 seconds to load! So, it's incredibly important from a business standpoint to make sure that you stay under that 3-second threshold.

Therefore, whenever I write any code for my new Laravel projects, I try to make sure to optimise the code as much as I can within my given time and cost constraints. If I ever work on any existing projects, I also try to use these techniques to update any slow-running code to improve the overall experience for the users.

In this chapter, we will look at some of the techniques that I use (or suggest to other developers) to get some quick performance improvements for my clients' and my own Laravel websites and applications.

Intended Audience

This chapter is intended for Laravel developers at any stage that is looking for some quick tips that are easy to add to their application.

Only Fetch the Fields You Need in Your Database Queries

One easy way of speeding up your Laravel site is by reducing the amount of data transferred between your app and the database. A way that you can do this is by specifying only the columns that you need in your queries using a select clause.

As an example, say you have a User model that contains 20 different fields. Now, imagine that you have 10000 users in your system and you're trying to do some form of processing on each of them. Your code might look something like this:

```
$users = User::all();

foreach ($users as $user) {
    // Do something here
}
```

The above query would be responsible for retrieving 200,000 fields worth of data. But, imagine that when you're processing each user, you only ever actually use the `id`, `first_name`, and `last_name` fields. So, this means that out of the 20 fields that you're retrieving, 17 of them are more or less redundant for this particular piece of code. So, what we can do is explicitly define the fields that are returned in the query. In this case, your code may look something like this:

```
$users = User::select(['id', 'first_name', 'last_name'])->get();

foreach ($users as $user) {
    // Do something here
}
```

By doing this, we will have reduced the number of fields returned in the query from 200,000 to 30,000. Although this probably wouldn't have much effect on the database's IO load, it would reduce the network traffic between your app and the database. This is because there would (presumably) be less data to serialise, send, and then deserialise than if you were to fetch all of the available fields. By reducing the network traffic and the amount of data that needs to be processed, this would help to speed up your Laravel site.

Please note, in the above example you might not ever actually do something like this and you would probably use chunks or pagination depending on the situation. The example is just to show a possible, easy-to-implement solution.

This solution might not gain you large improvements on a smaller site or application. However, it is something that can definitely assist in reducing load times on applications where performance is an important must-have. You might also see better improvements if you're querying a table that has BLOB or TEXT fields. These fields can potentially hold megabytes of data and so can potentially increase the query time. So, if your model's table contains either of these fields, consider explicitly defining the fields that you need in your query to reduce the load time.

Use Eager Loading Wherever Possible

When you are fetching any models from the database and then doing any type of processing on the model's relations, it's important that you use eager loading. Eager loading is super simple using Laravel and basically prevents you from encountering the N+1 problem with your data. This problem is caused by making N+1 queries to the database, where N is the number of items being fetched from the database. To explain this better and give it some context, let's check out the example below.

Imagine that you have two models (`Comment` and `Author`) with a one-to-one relationship between them. Now imagine that you have 100 comments and you want to loop through each one of them and output the author's name.

Without eager loading, your code might look like this:

```php
$comments = Comment::all();

foreach ($comments as $comment ) {
    print_r($comment->author->name);
}
```

The code above would result in 101 database queries! The first query would be to fetch all of the comments. The other one hundred queries would come from getting the author's name in each iteration of the loop. Obviously, this can cause performance issues and slow down your application. So, how would we improve this?

By using eager loading, we could change the code to say:

```php
$comments = Comment::with('authors')->get();

foreach ($comments as $comment) {
    print_r($comment->author->name);
}
```

As you can see, this code looks almost the same and is still readable. By adding the `::with('authors')` this will fetch all of the comments and then make another query to fetch the authors at once. So, this means that we will have cut down the query from **101** to **2**!

For more information, check out the Laravel documentation on eager loading.

How to Force Laravel to Use Eager Loading

In Laravel, you can actually prevent lazy loading. This feature is incredibly useful because it can help to ensure that the relationships are eager loaded. As a result of this, it can improve performance and reduce the number of queries that are made to the database as shown in the example above.

It's super simple to prevent lazy loading. All we need to do is add the following line to the `boot()` method of our `AppServiceProvider`:

```
Model::preventLazyLoading();
```

So, in our `AppServiceProvider`, it would look a bit like this:

```
namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    public function boot(): void
    {
        // ...
        Model::preventLazyLoading();
        // ...
    }
}
```

Allowing Eager Loading in Production Environments

It's possible that you might only want to enable this feature when in your local development environment. By doing that, it can alert you to places in your code that are using lazy loading while building new features, but not completely crash your production website. For this very reason, the `preventLazyLoading()` method accepts a boolean as an argument, so we could use the following line:

```
Model::preventLazyLoading(! app()->isProduction());
```

So, in our `AppServiceProvider`, it could look like this:

```php
namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    public function boot(): void
    {
        // ...
        Model::preventLazyLoading(! app()->isProduction());
        // ...
    }
}
```

By doing this, the feature will be disabled if your `APP_ENV` is production so that any lazy loading queries that slipped through don't cause exceptions to be thrown on your site.

What Happens If We Try to Lazy Load?

If we have the feature enabled in our service provider and we try to lazy load a relationship on a model, an `Illuminate\Database\LazyLoadingViolationException` exception will be thrown.

To give this a little bit of context, let's use our `Comment` and `Author` model examples from above. Let's say that we have the feature enabled.

The following snippet would throw an exception:

```php
$comments = Comment::all();

foreach ($comments as $comment ) {
    print_r($comment->author->name);
}
```

However, the following snippet would not throw an exception:

```php
$comments = Comment::with('authors')->get();

foreach ($comments as $comment ) {
    print_r($comment->author->name);
}
```

Get Rid of Any Unneeded or Unwanted Packages

Open up your `composer.json` file and look through each of your dependencies. For each of your dependencies, ask yourself *"do I really need this package?"*. Your answer is mostly going to be yes, but for some of them, it might not be.

Each time you include a new Composer library into your project, you are potentially adding extra code that might be run unnecessarily. Laravel packages typically contain service providers that are run on each request that register services and run code. So, say if you add 20 Laravel packages to your application, that's probably a minimum of 20 classes being instantiated and run on each request. Although this isn't going to have a huge impact on performance for sites or applications with small amounts of traffic, you'll definitely be able to notice the difference on larger applications.

The solution to this is to determine whether you actually need all of the packages. Maybe you're using a package that provides a range of features but you're only actually using one small feature out of it. Ask yourself *"could I write this code myself and remove this entire package"*? Of course, due to time constraints, it's not always feasible to write the code yourself because you'll have to write it, test it and then maintain it. At least with using the package, you're making use of the open-source community to do those things for you. But, if a package is simple and quick to replace with your own code, then I'd consider removing it.

Cache, Cache, Cache!

Laravel comes with plenty of caching methods out of the box. These can make it really easy to speed up your website or application while it's live without needing to make any code changes.

Route Caching

Because of the way that Laravel runs, it boots up the framework and parses the routes file on each request that is made. This requires reading the file, parsing its contents, and then holding it in a way that your application can use and understand. So, Laravel provides a command that you can use which creates a single routes file that can be parsed much faster:

```
php artisan route:cache
```

Please note though that if you use this command and change your routes, you'll need to make sure to run:

```
php artisan route:clear
```

This will remove the cached routes file so that your newer routes can be registered. It might be worthwhile to add these two commands to your deploy script if you don't already have them. If you don't use a deploy script, you might find my package Laravel Executor useful to help you when running your deployments.

Config Caching

Similar to the route caching, each time that a request is made, Laravel is booted up and each of the config files in your project are read and parsed.

So, to prevent each of the files from needed to be handled, you can run the following command which will create one cached config file:

```
php artisan config:cache
```

Just like the route caching above though, you'll need to remember to run the following command each time you update your .env file or config files:

```
php artisan config:clear
```

Event and View Caching

Laravel also provides two other commands that you can use to cache your views and events so that they are precompiled and ready when a request is made to your application. To cache the events and views, you can use the following commands:

```
php artisan event:cache
```

```
php artisan view:cache
```

Like all the other caching commands, though, you'll need to remember to bust these caches whenever you make any changes to your code by running the following commands:

```
php artisan event:clear
```

```
php artisan view:clear
```

In the past, I've seen a lot of developers cache their config in their local development environment and then spend ages trying to figure out why their .env file changes aren't showing up. So, I'd probably recommend only caching your config and routes on live systems so that you don't end up in the same situation.

Caching Queries and Values

Within your Laravel app's code, you can cache items to improve the website's performance. As an example, imagine you have the following query:

```
$users = DB::table('users')->get();
```

To make use of caching with this query, you could update the code to the following:

```
$users = Cache::remember('users', 120, function () {
    return DB::table('users')->get();
});
```

The code above uses the `remember()` method. What this basically does is it checks if the cache contains any items with the key users. If it does, it returns the cached value. If it doesn't exist in the cache, the result of the `DB::table('users')->get()` query will be returned and also cached. In this particular example, the item would be cached for 120 seconds.

Caching data and query results like this can be a really effective way of reducing database calls, reducing runtime, and improving performance. However, it's important to remember that you might sometimes need to remove the item from the cache if it's no longer valid.

Using the example above, imagine that we have the user's query cached. Now imagine that a new user has been created, updated, or deleted. That cached query result is no longer going to be valid and up to date. To fix this issue, we could make use of Laravel model observers to remove this item from the cache. This means that next time we try and grab the `$users` variable, a new database query will be run that will give us the up-to-date result.

Use the Latest Version of PHP

With each new version of PHP that comes out, performance and speed are improved. Kinsta ran a lot of tests across multiple PHP versions and different platforms (e.g. - Laravel, WordPress, Drupal, Joomla) and found that PHP 8.0 gave the best performance increase.

This particular tip might be a bit more difficult to implement compared to the other tips above because you'll need to audit your code to make sure that you can safely update to the latest version of PHP. As a side note, having an automated test suite might help give you the confidence to do this upgrade!

Make Use of the Queues

This tip might take a little bit longer than some of the other code-based tips above to implement. Despite this, this tip will probably be one of the most rewarding in terms of user experience.

One way that you can cut down the performance time is to make use of the Laravel queues. If there's any code that runs in your controller or classes in a request that isn't particularly needed for the web browser response, we can usually queue it.

To make it easier to understand, check out this example:

```
class ContactController extends Controller
{
    /**
     * Store a new contact.
     *
     * @param  Request  $request
     * @return JsonResponse
     */
    public function store(ContactFormRequest $request)
    {
        $request->storeContactFormDetails();

        Mail::to('mail@ashallendesign.co.uk')->send(
          new ContactFormSubmission()
        );

        return response()->json(['success' => true]);
    }
}
```

In the above code, when the `store()` method is invoked it stores the contact form details in the database, sends an email to an address to inform them of a new contact form submission, and returns a JSON response. The issue with this code is that the user will have to wait until the email has been sent before they receive their response on the web browser. Although this might only be several seconds, it can potentially cause visitors to leave.

To make use of the queue system, we could update the code to the following instead:

```php
class ContactController extends Controller
{
    /**
     * Store a new contact.
     *
     * @param  Request  $request
     * @return JsonResponse
     */
    public function store(ContactFormRequest $request)
    {
        $request->storeContactFormDetails();

        dispatch(function () {
            Mail::to('mail@ashallendesign.co.uk')->send(
              new ContactFormSubmission()
            );
         })->afterResponse();

        return response()->json(['success' => true]);
    }
}
```

The code above in the `store()` method will now store the contact form details in the database, queue the mail for sending and then return the response. Once the response has been sent back to the user's web browser, the email will be added to the queue so that it can be processed. By doing this, it means that we don't need to wait for the email to be sent before we return the response.

Check out the Laravel docs for more information on how to set up the queues for your Laravel website or application.

Conclusion

This chapter should have given you an overview of several quick ways that you can speed up your Laravel project without needing to totally refactor your code. Of course, there are more things that you can do but these are the ones that I typically like to use myself when in need of a quick performance gain.

3

Chapter 3

How to Create Your Own Helper Functions

How to Create Your Own Helper Functions

Introduction

Helper functions can be extremely useful in your Laravel projects. They can help to simplify the code in your projects in a clean and easy way. Laravel comes with many helper functions out-of-the-box such as `dd()`, `abort()` and `session()`. But as your projects start to grow, you'll likely find that you'll want to add your own.

In this chapter, we're going to look at how we can create and use our own custom PHP helper functions in our Laravel projects.

Intended Audience

This chapter is aimed at developers who have a basic understanding of using Laravel and that are looking to tidy up their code using functions that can be accessed system-wide.

Creating the Helper Function

To add our own helper functions, we need to start by creating a new PHP file to place them in. So, let's create a `helpers.php` file inside our `app` folder. As a side note, this location is down to personal preference really. Another common place I've seen the file placed is inside an `app/Helpers/helpers.php` file. So, it's down to you to choose a directory that you feel works best for you.

Now that we've got our file, we can add our helper function to it. As a basic example for this chapter, we're going to be creating a super-simple function that converts minutes to hours.

Let's add the function to our `helpers.php` file like so:

```php
<?php

if (! function_exists('seconds_to_hours')) {
    function seconds_to_hours(int $seconds): float
    {
        return $seconds / 3600;
    }
}
```

As you can see in the example above, the function itself is really simple. However, one thing that you might notice is that the function name is written in snake case (e.g. `seconds_to_hours`) rather than camel case (e.g. `secondsToHours`) like you'd usually see with method names on a class. You aren't necessarily fixed to using snake case for defining the function name, but you'll find that all of the Laravel helper functions are written this way. So, I'd probably advise using the format just so that you can follow the typical standard and format that's expected. This is totally up to you though.

Another thing that you might have noticed is that we have wrapped the function name inside an 'if' statement. This is to stop us from accidentally redeclaring any helpers with the same name that have already been registered. For example, let's say that we were using a package that already had a `seconds_to_hours()` function registered, this would prevent us from registering our own function with the same name. To get around this, we could just simply change the name of our own function to avoid any clashes.

It's also really important to remember when creating helper functions that they are only supposed to be used as helpers. They aren't really meant to be used to perform any business logic, but rather to help you tidy up your code. Of course, you can add complex logic to them, but if this is something you're looking to do, I'd probably advise thinking about if the

code would be a better fit in another place such as a service class, action class, or trait.

Registering the Helper Function

Now that we've created our helper file, we need to register it so that we can use our new function. To do this, we can update our `composer.json` file so that our file is loaded at runtime on each request and is available for using. This is possible because Laravel includes the Composer class loader in the `public/index.php` file.

In your `composer.json` file, you should have a section that looks like this:

```
"autoload": {
    "psr-4": {
        "App\\": "app/",
        "Database\\Factories\\": "database/factories/",
        "Database\\Seeders\\": "database/seeders/"
    }
},
```

In this section, we just need to add the following lines to let Composer know that you want to load your file:

```
"files": [
    "app/helpers.php"
],
```

The autoload section of your composer.json file should now look like this:

```
"autoload": {
    "files": [
        "app/helpers.php"
    ],
    "psr-4": {
        "App\\": "app/",
        "Database\\Factories\\": "database/factories/",
        "Database\\Seeders\\": "database/seeders/"
    }
},
```

Now that we've manually updated the `composer.json` file, we'll need to run the following command to dump our autoload file and create a new one:

```
composer dump-autoload
```

Using the Helper Function

Congratulations! Your helper function should now be all set up and ready to use. To use it, you can simply use:

```
seconds_to_hours(331);
```

Because it's been registered as a global function, this means you can use it in a variety of places such as your controllers, service classes, and even other helper functions. The part that I love about helper functions most though is being able to use them in your Blade views. For example, let's imagine that we had a `TimeServiceClass` that contained a `secondsToHours()` method that did the same as our new function. If we

were to use the service class in our Blade view, we might have to do something like this:

```
{{ \App\Services\TimeService::secondsToHours(331) }}
```

As you can imagine, if this was used in multiple places across a page, it could probably start to make your view a bit messy.

Using Multiple Helper Files

Now that we've seen how we can register and use the helpers, we will look at how we can take it one step further. Over time, as your Laravel projects grow, you might find that you have a large number of helpers that are all in one file. As you can imagine, the file might start to look a bit unorganised. So, we might want to think about splitting our functions out into separate files.

As an example, let's imagine that we have many helpers in our `app/helpers.php` file; some related to money, some related to time, and some related to user settings. We could start by splitting those functions out into separate files such as: `app/Helpers/money.php`, `app/Helpers/time.php`, and `app/Helpers/settings.php`. This means that we can now delete our `app/helpers.php` file because we don't need it anymore.

After that, we can update our `composer.json` file in a similar way to before so that it nows loads our 3 new files:

```
"autoload": {
    "files": [
        "app/Helpers/money.php",
        "app/Helpers/settings.php",
        "app/Helpers/time.php",
    ],
    "psr-4": {
        "App\\": "app/",
        "Database\\Factories\\": "database/factories/",
        "Database\\Seeders\\": "database/seeders/"
    }
},
```

We'll need to remember to dump the Composer autoload file again by running the following command:

```
composer dump-autoload
```

You should now be able to continue using your functions and have the benefit of them being split into logically separated files.

Conclusion

This chapter should have shown you how to create and register your own PHP helper functions for your Laravel projects. Remember not to use them to perform complex business logic and to see them more as a way of tidying up little bits of code.

4

Chapter 4

Using Interfaces to Write Better PHP Code

Using Interfaces to Write Better PHP Code

Introduction

In programming, it's important to make sure that your code is readable, maintainable, extendable and easily testable. One of the ways that we can improve all of these factors in our code is by using interfaces.

Intended Audience

In comparison to the previous chapters, the content in this chapter can seem a bit daunting at first. It is aimed at developers who have a basic understanding of OOP (object-oriented programming) concepts and using inheritance in PHP. If you know how to use inheritance in your PHP code, this chapter should hopefully be understandable.

What Are Interfaces?

In basic terms, interfaces are just descriptions of what a class should do. They can be used to ensure that any class implementing the interface will include each public method that is defined inside it.

Interfaces **can** be:

- Used to define public methods for a class.
- Used to define constants for a class.

Interfaces **cannot** be:

- Instantiated on their own.
- Used to define private or protected methods for a class.
- Used to define properties for a class.

Interfaces are used to define the public methods that a class should include. It's important to remember that only the method signatures are defined and that they don't include the method body (like you would typically see in a method in a class). This is because the interfaces are only used to define the communication between objects, rather than defining the communication and behaviour like in a class. To give this a bit of context, this example shows an example interface that defines several public methods:

```
interface DownloadableReport
{
    public function getName(): string;

    public function getHeaders(): array;

    public function getData(): array;
}
```

According to php.net, interfaces serve two main purposes:

1. To allow developers to create objects of different classes that may be used interchangeably because they implement the same interface or interfaces. A common example is multiple database access services, multiple payment gateways, or different caching strategies. Different implementations may be swapped out without requiring any changes to the code that uses them.
2. To allow a function or method to accept and operate on a parameter that conforms to an interface, while not caring what else the object may do or how it is implemented. These interfaces are often named like Iterable, Cacheable, Renderable, or so on to describe the significance of the behavior.

Using Interfaces in PHP

Interfaces can be an invaluable part of OOP (object-oriented programming) codebases. They allow us to decouple our code and improve extendability. To give an example of this, let's take a look at this class below:

```
class BlogReport
{
    public function getName(): string
    {
        return 'Blog report';
    }
}
```

As you can see, we have defined a class with a method that returns a string. By doing this, we have defined the behaviour of the method so we can see how `getName()` is building up the string that is returned. However, let's say that we call this method in our code inside another class. The other class won't care how the string was built up, it will just care that it was returned. For example, let's look at how we could call this method in another class:

```
class ReportDownloadService
{
    public function downloadPDF(BlogReport $report)
    {
        $name = $report->getName();

        // Download the file here...
    }
}
```

Although the code above works, let's imagine that we now wanted to add the functionality to download a users report that's wrapped inside a `UsersReport` class. Of course, we can't use the existing method in our

`ReportDownloadService` because we have enforced that only a `BlogReport` class can be passed. So, we'll have to rename the existing method and then add a new method, like below:

```
class ReportDownloadService
{
    public function downloadBlogReportPDF(BlogReport $report)
    {
        $name = $report->getName();

        // Download the file here...
    }

    public function downloadUsersReportPDF(UsersReport $report)
    {
        $name = $report->getName();

        // Download the file here...
    }
}
```

Although you can't actually see it, let's assume that the rest of the methods in the class above use identical code to build the download. We could lift the shared code into methods but we will still likely have some shared code. As well as this, we're going to have multiple points of entry into the class that runs near-identical code. This can potentially lead to extra work in the future when trying to extend the code or add tests.

For example, let's imagine that we create a new `AnalyticsReport`; we'd now need to add a new `downloadAnalyticsReportPDF()` method to the class. You can likely see how this file could start growing quickly. This could be a perfect place to use an interface!

Let's start by creating one; we'll call it `DownloadableReport` and define it like so:

```php
interface DownloadableReport
{
    public function getName(): string;

    public function getHeaders(): array;

    public function getData(): array;
}
```

We can now update the `BlogReport` and `UsersReport` to implement the `DownloadableReport` interface as seen in the example below. But please note, I have purposely written the code for the `UsersReport` wrong so that I can demonstrate something!

```php
class BlogReport implements DownloadableReport
{
    public function getName(): string
    {
        return 'Blog report';
    }

    public function getHeaders(): array
    {
        return ['The headers go here'];
    }

    public function getData(): array
    {
        return ['The data for the report is here.'];
    }
}
```

```php
class UsersReport implements DownloadableReport
{
    public function getName()
    {
        return ['Users Report'];
    }

    public function getData(): string
    {
        return 'The data for the report is here.';
    }
}
```

If we were to try and run our code, we would get errors for the following reasons:

1. The `getHeaders()` method is missing.
2. The `getName( )` method doesn't include the return type that is defined in the interface's method signature.
3. The `getData()` method defines a return type, but it isn't the same as the one defined in the interface's method signature.

So, to update the `UsersReport` so that it correctly implements the `DownloadableReport` interface, we could change it to the following:

```php
class UsersReport implements DownloadableReport
{
    public function getName(): string
    {
        return 'Users Report';
    }

    public function getHeaders(): array
    {
       return [];
    }

    public function getData(): array
    {
        return ['The data for the report is here.'];
    }
}
```

Now that we have both of our report classes implementing the same interface, we can update our `ReportDownloadService` like so:

```php
class ReportDownloadService
{
    public function downloadReportPDF(DownloadableReport $report)
    {
        $name = $report->getName();

        // Download the file here...
    }
}
```

We could now pass in a `UsersReport` or `BlogReport` object into the `downloadReportPDF()` method without any errors. This is because we now know that the necessary methods needed on the report classes exist and return data in the type that we expect.

As a result of passing in an interface to the method rather than a class, this has allowed us to loosely couple the `ReportDownloadService` and the report classes based on **what** the methods do, rather than **how** they do it.

If we wanted to create a new `AnalyticsReport`, we could make it implement the same interface and then this would allow us to pass the report object into the same `downloadReportPDF()` method without needing to add any new methods. This can be particularly useful if you are building your own package or framework and want to give the developer the ability to create their own class. You can simply tell them which interface to implement and they can then create their own new class. For example, in Laravel, you can create your own custom cache driver class by implementing the `Illuminate\Contracts\Cache\Store` interface.

As well as using interfaces to improve the actual code, I tend to like interfaces because they act as code-as-documentation. For example, if I'm trying to figure out what a class can and can't do, I tend to look at the interface first before a class that is using it. It tells you all of the methods that can be called without me needing to care too much about how the methods are running under the hood.

It's worth noting for any of my Laravel developer readers that you'll quite often see the terms "contract" and "interface" used interchangeably. According to the Laravel documentation, "Laravel's contracts are a set of interfaces that define the core services provided by the framework". So, it's important to remember that a contract is an interface, but an interface isn't necessarily a contract. Usually, a contract is just an interface that is provided by the framework. For more information on using the contracts, I'd recommend giving the documentation a read as I think it does a good job of breaking down what they are, how to use them, and when to use them.

Conclusion

This chapter should have given you a brief overview of what interfaces are, how they can be used in PHP, and the benefits of using them.

5

Chapter 5

Using the Strategy Pattern

Using the Strategy Pattern

Introduction

In software and web development, it's always important to write code that is maintainable and extendable. The solution that you first create will likely change over time. So, you need to make sure you write your code in a way that doesn't require a whole rewrite or refactor in the future.

The **strategy pattern** can be used to improve the extendability of your code and also improve the maintainability over time.

Intended Audience

This chapter is probably the most complex in this guide. It's written for Laravel developers who have an understanding of how interfaces work and how to use them to decouple your code. If you've read the previous chapter, you should be able to understand the overall concept. In my personal opinion, the best way to understand the concepts covered in this chapter is to give them a try yourself while (or after you've finished) reading it.

It's also strongly advised that you have an understanding of dependency injection and how the Laravel service container works.

What Is the Strategy Pattern?

Refactoring Guru defines the strategy pattern as a "behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.". This might sound a bit scary at first, but I promise that it's not as bad as you think. If you want to read more into design patterns, I'd highly recommend checking out

Refactoring Guru. They do a great job of explaining the strategy pattern in-depth as well as other structural patterns.

The strategy pattern is basically a pattern that helps us to decouple our code and make it super extendable.

Using the Strategy Pattern in Laravel

Now that we have a basic idea of what the strategy pattern is, let's look at how we can use it ourselves in our own Laravel application.

Let's imagine that we have a Laravel application that users can use for getting exchange rates and currency conversions. Now, let's say that our app uses an external API (exchangeratesapi.io) for getting the latest currency conversions.

We could create this class for interacting with the API:

```
class ExchangeRatesApiIO
{
    public function getRate(string $from, string $to): float
    {
        // Make a call to the exchangeratesapi.io API here.

        return $rate;
    }
}
```

Now, let's use this class in a controller method so that we can return the exchange rate for a given currency. We're going to use dependency injection to resolve the class from the container:

```php
class RateController extends Controller
{
    public function __invoke(
      ExchangeRatesApiIO $exchangeRatesApiIO
    ): JsonResponse {
        $rate = $exchangeRatesApiIO->getRate(
            request()->from,
            request()->to,
        );

        return response()->json(['rate' => $rate]);
    }
}
```

This code will work as expected, but we've tightly coupled the `ExchangeRatesApiIO` class to the controller method. This means that if we decide to migrate over to using a different API, such as Fixer, in the future, we'll need to replace everywhere in the codebase that uses the `ExchangeRatesApiIO` class with our new class. As you can imagine, in large projects, this can be a slow and tedious task sometimes. So, to avoid this issue, instead of trying to instantiate a class in the controller method, we can use the strategy pattern to bind and resolve an interface instead.

Let's start by creating a new `ExchangeRatesService` interface:

```php
interface ExchangeRatesService
{
    public function getRate(string $from, string $to): float;
}
```

We can now update our `ExchangeRatesApiIO` class to implement this interface:

```
class ExchangeRatesApiIO implements ExchangeRatesService
{
    public function getRate(string $from, string $to): float
    {
        // Make a call to the exchangeratesapi.io API here.

        return $rate;
    }
}
```

Now that we've done that, we can update our controller method to inject the interface rather than the class:

```
class RateController extends Controller
{
    public function __invoke(
      ExchangeRatesService $exchangeRatesService
    ): JsonResponse {
        $rate = $exchangeRatesService->getRate(
            request()->from,
            request()->to,
        );

        return response()->json(['rate' => $rate]);
    }
}
```

Of course, we can't instantiate an interface; we want to instantiate the `ExchangeRatesApiIO` class. So, we need to tell Laravel what to do whenever we try and resolve the interface from the container. We can do this by using a service provider. Some people prefer to keep things like this inside their `AppServiceProvider` and keep all of their bindings in one place. However, I prefer to create a separate provider for each binding that I want to create. It's purely down to personal preference and whatever you feel fits

your workflow more. For this example, we're going to create our own service provider.

Let's create a new service provider using the Artisan command:

```
php artisan make:provider ExchangeRatesServiceProvider
```

We'll then need to remember to register this service provider inside the `app/config.php` like below:

```
return [

    'providers' => [
        // ...
        \App\Providers\ExchangeRatesServiceProvider::class,
        // ...
    ],

];
```

Now, we can add our code to the service provider to bind the interfaces and class:

```
class ExchangeRatesServiceProvider extends ServiceProvider
{
    public function register(): void
    {
        $this->app->bind(
          ExchangeRatesService::class,
          ExchangeRatesApiIO::class
        );
    }
}
```

Now that we've done all of this, when we dependency inject the `ExchangeRatesService` interface in our controller method, we'll receive an `ExchangeRatesApiIO` class that we can use.

Binding Multiple Classes to Interfaces

Now that we know how to bind an interface to a class, let's take things a bit further. Let's imagine that we want to be able to decide whether to use the ExchangeRatesAPI.io or the Fixer.io API whenever we'd like just by updating a config field.

We don't have a class yet for dealing with the Fixer.io API yet, so let's create one and make sure that it implements the `ExchangeRatesService` interface:

```
class FixerIO implements ExchangeRatesService
{
    public function getRate(string $from, string $to): float
    {
        // Make a call to the Fixer API here and fetch the exchange rate.

        return $rate;
    }
}
```

We'll now create a new field in our `config/services.php` file:

```
return [

    //...

    'exchange-rates-driver' => env('EXCHANGE_RATES_DRIVER'),

];
```

We can now update our service provider to change which class will be returned whenever we resolve the interface from the container:

```php
class ExchangeRatesServiceProvider extends ServiceProvider
{
    public function register(): void
    {
        $this->app->bind(ExchangeRatesService::class, function ($app) {
            $driver = config('services.exchange-rates-driver');

            if ($driver === 'exchangeratesapiio') {
                return new ExchangeRatesApiIO();
            }

            if ($driver === 'fixerio') {
                return new FixerIO();
            }

            throw new Exception('The exchange rates driver is invalid.');
        });
    }
}
```

Now if we set our exchanges rates driver in our `.env` to `EXCHANGE_RATES_DRIVER=exchangeratesapiio` and try to resolve the `ExchangeRatesService` from the container, we will receive an `ExchangeRatesApiIO` class. If we set our exchanges rates driver in our `.env` to `EXCHANGE_RATES_DRIVER=fixerio` and try to resolve the `ExchangeRatesService` from the container, we will receive a `FixerIO` class. If we set the driver to anything else accidentally, an exception will be thrown to let us know that it's incorrect.

Due to the fact that both of the classes implement the same interface, we can seamlessly change the `EXCHANGE_RATES_DRIVER` field in the `.env` file and not need to change any other code anywhere.

Conclusion

Is your brain fried yet? If it is, don't worry! Personally, I found this topic pretty difficult to understand when I first learned about it. I don't think I started to really understand it until I put it into practice and used it myself. So, I'd advise spending a little bit of time experimenting with this yourself. Once

you get comfortable with using it, I guarantee that you'll start using it in your own projects.

This chapter has given you an overview of what the strategy pattern is and how you can use it in Laravel to improve the extendability and maintainability of your code.

6

Chapter 6

Making Your Laravel Application More Testable

Making Your Laravel Application More Testable

Introduction

Testing is an integral part of web and software development. It helps to give you the confidence that any code that you have written meets acceptance criteria and also reduces the chances of your code having bugs. In fact, TDD (test driven development), a popular development approach, actually focuses on tests being written before any code is added to the actual app codebase.

Intended Audience

This chapter is aimed at developers who are fairly new to the Laravel world but have a basic understanding of tests. We won't cover how to write basic tests, but it will show you how you can approach your code in a slightly different way to improve your code quality and test quality.

Why Should I Write Tests?

Tests are often thought of as being an afterthought and a "*nice to have*" for any code that is written. This is seen especially in organisations where business goals and time constraints are putting pressure on the development team. And in all fairness, if you're only trying to get an MVP (minimum viable product) or a prototype built together quickly, maybe the tests can take a bit of a backseat. But, the reality is that writing tests before the code is released into production is always the best option!

When you write tests, you are doing multiple things:

- **Spotting bugs early** - Be honest, how many times have you written code, ran it once or twice, and then committed it to your version control system. I'll hold my hand up, I've done it myself. You think to yourself "it looks right and it seems to run, I'm sure it'll be fine". Every single time I did this, I ended up with either my pull requests on GitHub being rejected or bugs being released into production. So by writing tests, you can spot bugs before you commit your work and have a bit more confidence whenever you release them to production.
- **Making future work and refactoring easier** - Imagine that you need to refactor one of the core classes in your application. Or, that you maybe need to add some new code to that class to extend the functionality. Without tests, how are you going to know for certain that changing or adding any code isn't going to break the existing functionality? Without a lot of manual testing, there's not much way of quickly checking. So, by writing tests when you write the first version of the code, you can treat them as regression tests. That means that every time you update any code, you can run the tests to make sure everything is still working. You can also keep adding tests every time you add new code so that you can be sure your additions are also working.
- **Changing the way you approach writing code** - When I first learned about testing and started writing my first tests (for a Laravel app using PHPUnit), I quickly realised that my code was pretty difficult to write tests for. It was hard to do things like mocking classes, preventing third-party API calls, and making some assertions. To be able to write code is a way that can be tested, you have to look at the structure of your classes and methods from a slightly different angle than before.

Writing Controller Tests

To explain how we can make your code more testable, we'll use a simple example. Of course, there are different ways that you could write the code and this might be that simple that it doesn't matter. But, hopefully, it should help explain the overall concept.

Let's take this example controller method:

```
use App\Services\NewsletterSubscriptionService;
use Illuminate\Http\JsonResponse;
use Illuminate\Http\Request;

class NewsletterSubscriptionController extends Controller
{
    /**
     * Store a new newsletter subscriber.
     *
     * @param  Request  $request
     * @return JsonResponse
     */
    public function store(Request $request): JsonResponse
    {
        $service = NewsletterSubscriptionService();
        $service->handle($request->email);

        return response()->json(['success' => true]);
    }
}
```

The above method, which we'll assume is invoked if you make a POST request to `/newsletter/subscriptions`, accepts an `email` parameter which is then passed to a service. We can then assume that the service handles all of the different processes that need to be carried out to complete a user's subscription to the newsletter.

To test the above controller method, we could create the following test:

```php
class NewsletterSubscriptionControllerTest extends TestCase
{
    /** @test */
    public function success_response_is_returned()
    {
        $this->postJson('/newsletter/subscriptions', [
            'email' => 'mail@ashallendesign.co.uk',
        ])->assertExactJson([
            'success' => true,
        ]);
    }
}
```

There's just one problem that you might have noticed in our test. It doesn't actually check to see if the service class' `handle()` method was called! So, if by accident we were to delete or comment out that line in the controller, we wouldn't actually know.

Writing Better Controller Tests

What's the Problem?

One of the problems that we have here is that without adding extra code to flag or log that the service class has been called, it's pretty difficult for us to check that it's been written.

Sure, we could add more assertions in this controller test to test the service class's code is all being run. But that can lead to an overlap in your tests. For argument's sake, let's imagine that our Laravel app allows users to register and that whenever they register they are automatically signed up to the newsletter. Now, if we were to write tests for this controller as well that checked that all of the service class was run correctly, we'd have 2 near duplicates of test code. This would mean that if we were to update the way that the service class runs internally, we'd also need to update all of these tests as well.

In all fairness, sometimes you might actually want to do that. If you're writing a feature test and run assertions against the whole end-to-end process, this would be suitable. However, if you're trying to write unit tests and only want to check the controller, this approach won't quite work.

How Can We Fix the Problem?

In order to improve the test that we've got, we can make use of mocking, the service container, and dependency injection. I won't go too much into too much depth about what the service container is, but, I'd definitely recommend reading into it because it can be incredibly helpful and is a core part of Laravel.

In short (and very basic terms), the service container manages class dependencies and allows us to use classes that Laravel has already set up for us. To understand what I mean by this, let's take a look at the following examples below.

To make our code example more testable, we can instantiate the `NewsletterSubscriptionService` by using dependency injection to resolve it from the service container, like this:

```php
use App\Services\NewsletterSubscriptionService;
use Illuminate\Http\JsonResponse;
use Illuminate\Http\Request;

class NewsletterSubscriptionController extends Controller
{
    /**
     * Store a new newsletter subscriber.
     *
     * @param  Request  $request
     * @param  NewsletterSubscriptionService $service
     * @return JsonResponse
     */
    public function store(
      Request $request,
      NewsletterSubscriptionService $service
    ): JsonResponse {
        $service->handle($request->email);

        return response()->json(['success' => true]);
    }
}
```

What we've done above is we've added the `NewsletterSubscriptionService` class as an argument to the `store()` method because Laravel allows dependency injection in controllers. What this basically does is it tells Laravel when it's calling this method is "Hey, I also want you to pass me a `NewsletterSubscriptionService` !". Laravel then replies and says "Okay, I'll grab one now for you from the service container".

In this case, our service class doesn't have any constructor arguments, so it's nice and simple. However, if we had to pass in constructor arguments, we'd potentially have to create a service provider that handles what data is passed into the class when we first instantiate it.

Because we're now resolving from the container, we can update our test like so:

```php
class NewsletterSubscriptionControllerTest extends TestCase
{
    /** @test */
    public function success_response_is_returned()
    {
        // Create the mock of the service class.
        $mock = Mockery::mock(NewsletterSubscriptionService::class)
            ->makePartial();

        // Set the mocked class' expectations.
        $mock->shouldReceive('handle')
            ->once()
            ->withArgs(['mail@ashallendesign'])
            ->andReturnNull();

        // Add this mock to the service container to take
        // the service class' place.
        app()->instance(NewsletterSubscriptionService::class, $mock);

        $this->postJson('/newsletter/subscriptions', [
            'email' => 'mail@ashallendesign.co.uk',
        ])->assertExactJson([
            'success' => true,
        ]);
    }
}
```

Now, in the above test, we start off by using Mockery to create a mock of the service class. We then tell the service class that by the time the test finishes running, we expect that the `handle()` method will have been called once and have mail@ashallendesign.co.uk as the only parameter. After doing that, we then tell Laravel "Hey Laravel, if you need to resolve a `NewsletterSubscriptionService` at any point, here's one for you to return".

This means now that in our controller, the second parameter isn't actually the service class itself, but instead a mocked version of this class.

So, when we run the test now, we'll see that the `handle()` method is actually called. As a result of this, if we were to ever delete where that code is called or add any logic which might prevent it from being called, the test

would fail because Mockery would detect that the method was not invoked.

Bonus Testing Tip

There might be times when you're inside a class of code and it turns out that without some major refactoring you won't be able to inject your class (that you want to mock) by passing it as an extra method argument. In these cases, you can make use of the `resolve()` helper method that comes with Laravel.

The `resolve()` method simply returns a class from the service container. As a small example, let's look at how we could have updated our example controller method to still be testable with Mockery but without adding an extra argument:

```
use App\Services\NewsletterSubscriptionService;
use Illuminate\Http\JsonResponse;
use Illuminate\Http\Request;

class NewsletterSubscriptionController extends Controller
{
    /**
     * Store a new newsletter subscriber.
     *
     * @param  Request  $request
     * @return JsonResponse
     */
    public function store(Request $request): JsonResponse
    {
        $service = resolve(NewsletterSubscriptionService::class);
        $service->handle($request->email);

        return response()->json(['success' => true]);
    }
}
```

Conclusion

This chapter should have given you a little bit of an insight into how you can make your Laravel app more testable by making use of the service container, mocking, and dependency injection.

Remember that tests are your friends and can save you an unbelievably huge amount of time, stress, and pressure if they're added when code is first written. And as an added bonus, higher-quality tests and increased test coverage usually (but not always) lead to fewer bugs, which means fewer support tickets and happier clients!

Ash Allen

Laravel Web Developer

Hey, I'm Ash Allen! 👋

I'm a **freelance Laravel web developer** based in the UK that is passionate about writing clean code and blogging about it for others to read about.

I hope that you found this guide useful and that you've learned something new that you can start using in your own projects.

If you did find it useful, feel free to **share this PDF** with other web developers.

If you didn't find it useful, or if you spotted any mistakes, feel free to **drop me a message** so that I can get it updated.

Keep on building awesome stuff! 🚀

github.com/ash-jc-allen

linkedin.com/in/ashleyjcallen

ashallendesign.co.uk

mail@ashallendesign.co.uk