Ash Allen

Battle Ready Laravel

A guide to auditing, testing, fixing, and improving your Laravel applications

Table of Contents

Introduction

Hey there!

My name is Ash Allen and I'm a web developer based in the United Kingdom. Over the past five years, I've worked as a freelancer on Laravel projects and have had the chance to work on so many cool and different projects, from simple websites to complex SaaS (software as a service) platforms.

While freelancing, I've realised that I have a passion for writing clean, maintainable, and extensible code. So on almost every project I've worked on, I've always ended up becoming "the test guy" or the "he'll fix it for you guy". This means that on pretty much every project I work on, it has been my responsibility to try to improve the overall quality of the codebase; whether through refactoring or building entire test suites from scratch. In fact, since I started freelancing, I've written just over 10,000 tests!

So I've honed in on this small niche of testing and refactoring and typically use it when getting new clients. It's not very often that I work on greenfield projects anymore; I usually work on existing projects. I'm usually brought on board to work with clients to maintain their existing system, fix bugs, remove technical debt, build test suites, and add new features.

In this book, I'm going to take you through the same process that I would typically use when being brought on board to fix Laravel applications. I'll show you my usual auditing process that I use to find places where the code can be improved. I'll then cover how I build out a testing strategy that can be used to incrementally improve the test coverage of the code. After that, I'll cover the different ways that I approach fixing bugs in Laravel applications. Finally, we'll look at different ways that we can improve our codebase.

I won't be covering every single possible auditing tool, code issue, and improvement approach. If I did that, this book would likely be thousands of pages long and it would never get released. But I'll be covering the most common issues that I've found after working on many projects, and the ways that I would typically find and then remove them.

Of course, depending on the types of projects that you're working on, you might not be able to implement all the ideas covered in this book. If you're a developer who's looking for a few quick fixes for projects you work on, you might only be able to implement some of the ideas from individual sections. If you're a developer who has some extra time to dedicate to removing technical debt and improving the codebase, you might be able to implement all ideas. Either way, there should be something in here to cover your situation.

By the end of the book, you should feel more confident as a Laravel web developer. I hope that I'll be able to show you how you can effectively audit, test, fix, and improve your applications to make them more maintainable, testable, and performant.

Section 1: Auditing

What's Covered?

In this section, we're going to cover steps you can take to audit your Laravel applications.

We'll be stepping through some automated tools (such as Enlightn, Larastan, PHP Insights, and StyleCI) that you can use to quickly get an understanding of your code's quality and spot any obvious bugs.

After that we'll take a look at the steps that I would typically take when manually looking through the code. I'll show the common errors I usually come across when auditing projects and different ways you can spot them.

By the end of this section, you should hopefully have a better understanding of the different things to look out for in your code and ways of spotting them.

Remember that this section isn't a substitute for a penetration test. You'll still need one; but this can help you find some common security-related problems.

Planning Your Auditing Strategy

Before we start auditing our application, it's important to make sure that you have a clear strategy in place. This is particularly important if you're working as part of a team and you have multiple team members auditing the codebase at once.

Getting Into the Right Mindset

First off, I like to try to be relaxed before starting any type of audit. It's almost inevitable that as you're looking through the code, you're going to find some pieces of code that will make you think "What on Earth were they thinking when they wrote this?". So it's important to enter into the audit with a calm and open mind.

As you likely know, there can be a lot of pressure on developers to hit deadlines and budgets, and sometimes features will get rushed out to keep stakeholders or managers happy. So when the developers were writing these features they might not have written them in a way that was suitable for long term use, and may have rushed the code. Of course, in an ideal world, this would never happen and the features should always be fully thought out before being implemented. But unfortunately, this isn't always the case.

It's also important to take into account how old the project is. If the project was originally built with an older version of the language (like PHP 5.4), there might be some places where the code isn't as "clean" as it could be if it was written now. If you do come across places where the code doesn't look very clean, it could be due to constraints that the original developers had when the project was built.

You need to remember to look at the code with a critical mindset but without taking personal hits at anyone. So make sure that you don't nitpick at the smallest things, because this could lead to tension between you and the other team members. Just because you would have written a particular feature differently, doesn't necessarily mean that it's wrong. When we're looking through the code, we're looking for code that:

- Has bugs
- Is vulnerable to security exploits
- Makes testing difficult
- Makes it difficult to extend in the future.

If you come across a piece of code where you think "This works, but I could have written it better", that doesn't necessarily mean that we should flag it (unless it's causing issues or might make future development more difficult, of course).

Recording Your Findings

As you're working through your audit, you'll need to find a place to record your findings so that you can tackle the issues later on. This is because you typically won't want to start changing any code until you've got an overview of the entire system (or section of the system) and the changes that should be made. If you want to start making improvements as soon as you spot issues, this is entirely your choice. However, I typically wait until the end of my audit so that I can be more confident that I have a better understanding of the project as a whole.

Where you record the findings is all down to how you work as a team and what fits your workflow the best. In the past, I've recorded my findings in spreadsheets or taken a more Kanban-like approach and put them into a Trello board. The benefit of doing this is that if you're working in a team, the other team members can see what issues have been found and what needs to be done to solve each one. When I've audited projects that I've been working on by myself, I've used `TODO`s directly in the code. For example, I might leave a comment like this in my code:

```
// TODO Add typehint.
// TODO Add return type.
public function getUser($id)
{
    // ...
}
```

Doing this means that I get to couple my actions to the codebase itself. If you're using an IDE (integrated development environment) such as PhpStorm, you'll be able to see a list of all the places you've left `TODO`s. However, please remember that this approach might not work very well if you're working as part of a team because you won't be able to actively see who's working on each issue. It might also prevent project managers from getting a clear view on the progress you're making.

If you come across any potential vulnerabilities that could have been exploited in the past, it's really important to make sure that these findings are investigated. For example, if you find any routes that don't have the correct authentication or authorisation checks, you'll want to investigate and find out if there's any sign of a malicious person having visited this route. Any security-related issues that you find will likely need to be prioritised after you've finished your audit. Leaving vulnerable code in your project can be dangerous and costly if they are exploited (depending on the severity of the vulnerability).

Automated Tooling

There are many automated tools we can use to get started with auditing our code and looking for obvious improvements. We're going to step through the tools I would typically use and what I would do with the findings.

It's worth noting that some of these tools overlap in the features that they offer and the analysis that they'll provide. However, each of them have their own unique checks that can provide a great base for your audit.

PHP Insights

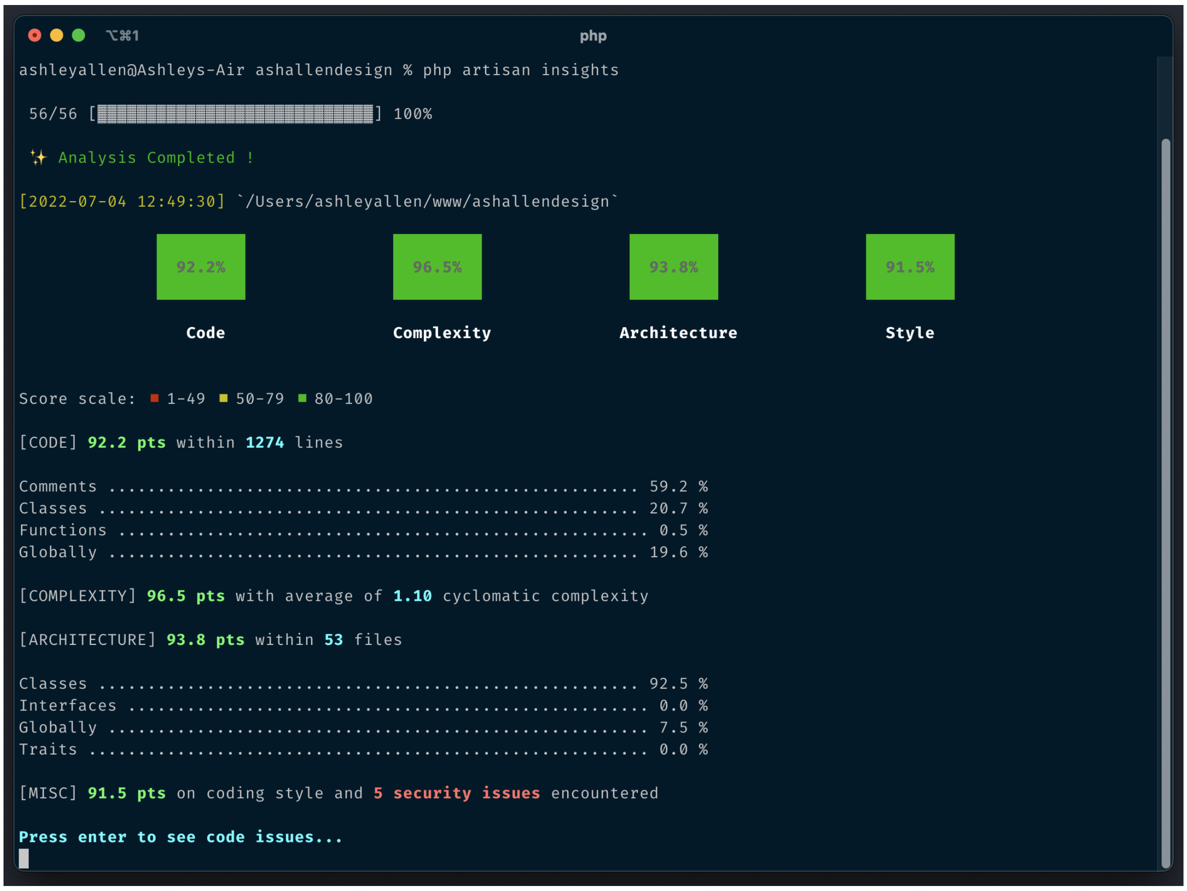
The first tool that we'll use to get an overview of our application is PHP Insights.

PHP Insights is a tool that has built-in checks that you can use to help make your code reliable, loosely coupled, simple, and clean. Out of the box, it works with Laravel, Symfony, WordPress, and more. So this tool can be useful in more than just your Laravel projects.

At the time of writing, PHP Insights gives you insights into 4 distinct categories:

- Code (105 insights)
- Architecture (20 insights)
- Complexity (1 insight)
- Style (84 insights)

The results that PHP Insights generates are also presented in a nice, easy-to-read format in your terminal after running. After the command has run, you'll be presented with an overview of the results like so:


```
ashleyallen@Ashleys-Air ashallendesign % php artisan insights

 56/56 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] 100%

 ✨ Analysis Completed !

[2022-07-04 12:49:30] `/Users/ashleyallen/www/ashallendesign`

              92.2%                 96.5%                  93.8%                 91.5%

              Code               Complexity           Architecture              Style

Score scale: ■ 1-49 ■ 50-79 ■ 80-100

[CODE] 92.2 pts within 1274 lines

Comments ..................................................... 59.2 %
Classes ...................................................... 20.7 %
Functions ..................................................... 0.5 %
Globally ..................................................... 19.6 %

[COMPLEXITY] 96.5 pts with average of 1.10 cyclomatic complexity

[ARCHITECTURE] 93.8 pts within 53 files

Classes ...................................................... 92.5 %
Interfaces .................................................... 0.0 %
Globally ...................................................... 7.5 %
Traits ........................................................ 0.0 %

[MISC] 91.5 pts on coding style and 5 security issues encountered

Press enter to see code issues...
```

PHP Insights provides a huge amount of customisation, so if you're interested in tailoring the analysis to fit your project needs more, you can check out the documentation at https://phpinsights.com/.

Installation

To get started, we'll first need to install PHP Insights using the following command:

```
composer require nunomaduro/phpinsights --dev
```

After that, you can publish the package's config file using the following command:

```
php artisan vendor:publish --
provider="NunoMaduro\PhpInsights\Application\Adapters\Laravel\Insi
ghtsServiceProvider"
```

Analysing Your Code with PHP Insights

That's it! You should now be ready to run PHP Insights using the following command:

```
php artisan insights
```

Running the command above will analyse your entire codebase. However, if you'd prefer to only analyse a specific file or directory, you can pass a path to the `insights` Artisan command. For example, if we only wanted to analyse the code in our `app/Http/Controllers` directory, we could run the following command:

```
php artisan insights app/Http/Controllers
```

Or for example, if we wanted to analyse a specific controller (such as `app/Http/Controllers/UserController.php`), we could run the following command:

```
php artisan insights app/Http/Controllers/UserController.php
```

Automatically Fixing Issues

By default, PHP Insights will only analyse your code and won't change anything about it. However, some checks that PHP Insights ships with have support for automatically fixing issues for you by using the `--fix` option in the command.

To automatically fix the issues, you can run the following command:

```
php artisan insights --fix
```

Although this can be really useful to run, as mentioned previously, it can sometimes be helpful to start making changes to your codebase after you've completed your entire audit. After you've completed your audit, you may wish to come back to this tool and run it to get some easy wins and improvements.

Example - Finding Unused Method Parameters

When you're running PHP Insights, you'll be able to step through the different insights and see what checks failed. To give a brief look into a few of the checks that it offers, let's take a look at a few examples.

In the example project that I ran the command in, PHP Insights detected that we had an unused parameter that could possibly be deleted:

```
• [Code] Unused parameter:
  app/Nova/User.php:83: Unused parameter $request.
  app/Nova/User.php:94: Unused parameter $request.
  app/Nova/User.php:105: Unused parameter $request.
  +32 issues omitted
```

It's important to remember that this does not always mean that the parameter is unused and not needed. However, it can usually give a good indication of the obvious fields that aren't being used. For example, let's take a look at where this error is being thrown from:

```
/**
 * Get the cards available for the request.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function cards(Request $request)
{
    return [];
}
```

In this case, the unused `$request` field is being passed because it is required so that the method matches its parent that it's extending. But the method is also an exact copy of its parent and doesn't provide any different behaviour. So this could potentially suggest to us that we can actually delete this method entirely from our `app/Nova/User.php` class.

As you can see, PHP Insights can provide a good indication of places where code can be improved. But with an extra bit of manual investigation, you can take the improvement further.

Example - Finding Vulnerable Dependencies

Laravel projects use packages and dependencies from developers all around the world. This means that the code running under the hood of your application (whether it be part of the Laravel framework or a package you've installed) is maintained by a third party, and you don't always have control over any changes.

There may be times when you're using a package within your project that contains a vulnerability that can be exploited. So it's important that you periodically check that your dependencies are up to date and aren't vulnerable.

There are some automated tools like Dependabot you can use that automatically check to see if you're using any dependencies with known security issues or CVEs (Common Vulnerabilities and Exposures). If you're using a platform like GitHub to host your code, this tool can usually be enabled for free.

In the example project, PHP Insights detected that it was using a dependency with a known security vulnerability. In this case, it was `guzzlehttp/guzzle@7.4.2`.

```
• [Security] Security issues found on dependencies:
  guzzlehttp/guzzle@7.4.2 Cross-domain cookie leakage - https://github.com/guzzle/guzzle/security/advisories/GHSA-c
wmx-hcrq-mhc3
  guzzlehttp/guzzle@7.4.2 Failure to strip the Cookie header on change in host or HTTP downgrade - https://github.c
om/guzzle/guzzle/security/advisories/GHSA-f2wf-25xc-69c9
  guzzlehttp/guzzle@7.4.2 Fix failure to strip Authorization header on HTTP downgrade - https://github.com/guzzle/g
uzzle/security/advisories/GHSA-w248-ffj2-4v5q
  +2 issues omitted
```

By flagging the dependency issues to us, we can prioritise updating the packages and removing the vulnerabilities. If you find any vulnerable dependencies in your project, it's important to investigate its severity and whether it may have been exploited in your application. Some vulnerabilities can lead to user data being leaked or malicious users getting access to parts of the system they aren't supposed to be able to.

Example - Finding Unused Imports

In the example project, PHP Insights also detected that we had some unused `use` statements in some of our classes.

```
• [Code] Unused parameter:
  app/Nova/User.php:83: Unused parameter $request.
  app/Nova/User.php:94: Unused parameter $request.
  app/Nova/User.php:105: Unused parameter $request.
  +32 issues omitted
```

By detecting that classes' imports aren't being used, we can remove them from our classes to reduce the amount of unnecessary code that is present in the codebase.

Enlightn

Another great tool that we can use to get insight into our project is Enlightn.

Enlightn is a CLI (command-line interface) application that you can run to get recommendations about how to improve the performance and security of your Laravel projects. Not only does it provide some static analysis tooling, it also performs dynamic analysis using tooling that was built specifically to analyse Laravel projects. So the results that it generates can be incredibly useful.

At the time of writing, Enlightn offers a free version and paid versions. The free version offers 64 checks, whereas the paid versions offer 128 checks.

One useful thing about Enlightn is that you can install it on your production servers (which is recommended by the official documentation) as well as your development environment. It doesn't incur any overhead on your application, so it shouldn't affect your project's performance and can provide additional analysis into your server's configuration.

For the purpose of this guide, we'll be using the free, open-source version and installing it in our local development environment to analyse our code. However, if you want to run it on your production server and get a more in-depth analysis of your project, you can follow the same steps and check out the full documentation at: https://www.laravel-enlightn.com.

Installation

To start using Enlightn, we'll need to install it using the following command:

```
composer require enlightn/enlightn
```

You'll then be able to publish the package's config file using the following command:

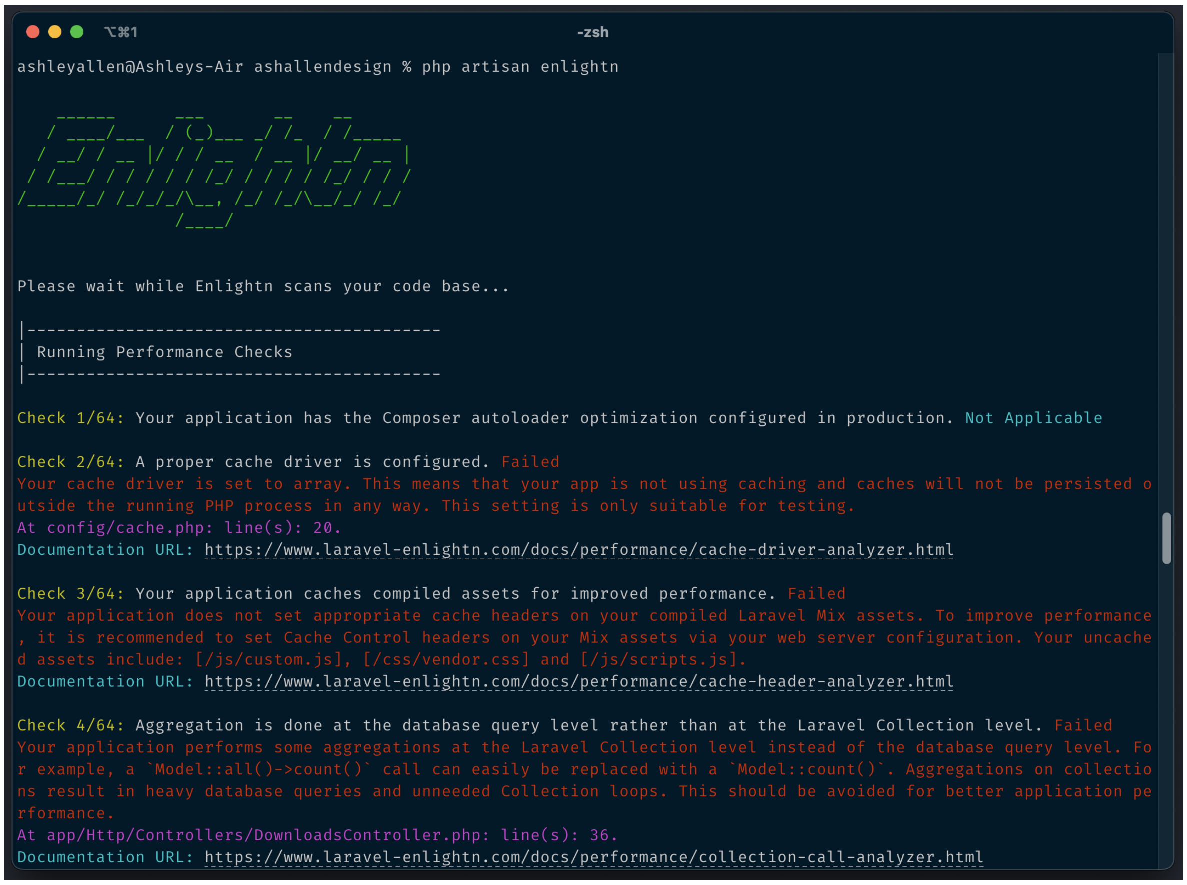
```
php artisan vendor:publish --tag=enlightn
```

Running Enlightn

To run Enlightn and analyse your Laravel application, you can then run the following command:

```
php artisan enlightn
```

By running this command, you'll be presented with an easy-to-read overview displaying the results of the checks that were run, like so:



```
ashleyallen@Ashleys-Air ashallendesign % php artisan enlightn


    ______      ___       __    __
   / ____/___  / (_)___ _/ /_  / /_____
  / __/ / __ |/ / / __  / __ |/ __/ __ |
 / /___/ / / / / / /_/ / / / / /_/ / / /
/_____/_/ /_/_/_/\__, /_/ /_/\__/_/ /_/
                /____/


Please wait while Enlightn scans your code base...

|------------------------------------------
| Running Performance Checks
|------------------------------------------

Check 1/64: Your application has the Composer autoloader optimization configured in production. Not Applicable

Check 2/64: A proper cache driver is configured. Failed
Your cache driver is set to array. This means that your app is not using caching and caches will not be persisted o
utside the running PHP process in any way. This setting is only suitable for testing.
At config/cache.php: line(s): 20.
Documentation URL: https://www.laravel-enlightn.com/docs/performance/cache-driver-analyzer.html

Check 3/64: Your application caches compiled assets for improved performance. Failed
Your application does not set appropriate cache headers on your compiled Laravel Mix assets. To improve performance
, it is recommended to set Cache Control headers on your Mix assets via your web server configuration. Your uncache
d assets include: [/js/custom.js], [/css/vendor.css] and [/js/scripts.js].
Documentation URL: https://www.laravel-enlightn.com/docs/performance/cache-header-analyzer.html

Check 4/64: Aggregation is done at the database query level rather than at the Laravel Collection level. Failed
Your application performs some aggregations at the Laravel Collection level instead of the database query level. Fo
r example, a `Model::all()->count()` call can easily be replaced with a `Model::count()`. Aggregations on collectio
ns result in heavy database queries and unneeded Collection loops. This should be avoided for better application pe
rformance.
At app/Http/Controllers/DownloadsController.php: line(s): 36.
Documentation URL: https://www.laravel-enlightn.com/docs/performance/collection-call-analyzer.html
```

It's important to remember that some checks will likely fail because they're set up to test against a production environment. This is completely normal and not anything that you should be worried about. So make sure to just record the errors that are related to the codebase rather than the infrastructure.

Notice: Enlightn also offers a "Web UI" you can use to view your results. You'll need to create an account at https://www.laravel-enlightn.com/register and follow the documentation to set this up using the API token that they provide. This can be useful if you manually run (or schedule) Enlightn in your CI/CD pipeline or on your production servers.

Example - Unnecessary Collection Calls

Enlightn is very useful at detecting method calls on Collections that could have been called against the underlying database query. For example, let's take the following block of code:

```php
use App\Models\User;

$userCount = User::all()->count();
```

The code above does its job and assigns `userCount` with the amount of `User` models that exist in the database. However, to achieve this, it's actually fetching all the users from the database, with all of their properties, hydrating the models (assigning the fields returned from the query to the `User` models) and then creating a Collection of the models. It's then counting how many models exist in the Collection.

As you can imagine, this approach isn't very optimised and depending on how complex the query is and where it's used, it could potentially increase the execution time of your code (and frustrate your users).

For our particular example above, Enlightn would flag the query and let us know that we can optimise it by changing it to the following:

```php
use App\Models\User;

$userCount = User::count();
```

Changing the query to the above version removes the steps where we were fetching all the data from the table and then counting it in a Collection. Instead, the SQL query used will be changed so that it only returns the count from the database as an integer.

To give an example of the types of speed improvements you could see by making this improvement, here are some execution times (in seconds) from a local machine when running locally with different amounts of rows in the `users` database table:

	User::all()->count()	User::count()
1,000	0.014s	0.001s
10,000	0.064s	0.003s
100,000	0.862s	0.006s
1,000,000	12.934s	0.150s

As you can see from the table above, there's a clear performance improvement; especially as the number of rows in the database grows.

Please note that the execution times above are based on running an example project locally using PHP 8.1 and MySQL 8 and are solely to be used as an indication of speed differences, rather than being statistics generated in a fully-controlled environment.

Example - Mass Assignment Analysis

Another useful analysis that Enlightn performs is the "Mass Assignment Analyzer". It scans through your application's code to find potential mass assignment vulnerabilities.

Mass assignment vulnerabilities can be exploited by malicious users to change the state of data in your database that isn't meant to be changed. To understand this issue, let's take a quick look at a potential vulnerability that I have come across in projects in the past.

Assume that we have a `User` model that has several fields: `id`, `name`, `email`, `password`, `is_admin`, `created_at`, and `updated_at`.

Imagine our project's user interface has a form a user can use to update the user. The form only has two fields: `name` and `password`. The controller method to handle this form and update the user might like something like the following:

```
class UserController extends Controller
{
    public function update(Request $request, User $user)
    {
        $user->update($request->all());

        return redirect()->route('users.edit', $user);
    }
}
```

The code above would *work*, however it would be susceptible to being exploited. For example, if a malicious user tried passing an `is_admin` field in the request body, they would be able to change a field that wasn't supposed to be able to be changed. In this particular case, this could lead to a permission escalation vulnerability that would allow a non-admin to make themselves an admin. As you can imagine, this could cause data protection issues and could lead to user data being leaked.

The Enlightn documentation provides several examples of the types of mass assignment usages it can detect:

```
$user->forceFill($request->all())->save();
User::update($request->all());
User::firstOrCreate($request->all());
User::upsert($request->all(), []);
User::where('user_id', 1)->update($request->all());
```

It's important to remember that this becomes less of an issue if you have the `$fillable` field defined on your models. For example, if we wanted to state that only the `name` and `email` fields could be updated when mass assigning values, we could update the user model to look like so:

```php
namespace App;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    protected $fillable = [
        'name',
        'email',
    ];
}
```

This would mean that if a malicious user was to pass the `is_admin` field in the request, it would be ignored when trying to assign the value to the `User` model.

However, I would recommend completely avoiding using the `all` method when mass assigning and only ever use the `validated`, `safe`, or `only` methods on the `Request`. By doing this, you can always have confidence that you explicitly defined the fields that you're assigning. For example, if we wanted to update our controller to use `only`, it may look something like this:

```php
class UserController extends Controller
{
    public function update(Request $request, User $user)
    {
        $user->update($request->only(['name', 'email']));

        return redirect()->route('users.edit', $user);
    }
}
```

If we want to update our code to use the `validated` or `safe` methods, we'll first want to create a form request class. In this example, we'll create a `UpdateUserRequest`:

```php
use Illuminate\Foundation\Http\FormRequest;

class UpdateUserRequest extends FormRequest
{
    // ...

    public function rules(): array
    {
        return [
            'email' => 'required|email|max:254',
            'name' => 'required|string|max:200',
        ];
    }
```

We can change our controller method to use the form request by type hinting it in the `update` method's signature and use the `validated` method:

```php
use App\Http\Requests\UpdateUserRequest;

class UserController extends Controller
{
    public function update(
        UpdateUserRequest $request,
        User $user
    ) {
        $user->update($request->validated());

        return redirect()->route('users.edit', $user);
    }
}
```

Alternatively, we can also use the `safe` method like so:

```php
use App\Http\Requests\UpdateUserRequest;

class UserController extends Controller
{
    public function update(
        UpdateUserRequest $request,
        User $user
    ) {
        $user->update($request->safe());

        return redirect()->route('users.edit', $user);
    }
}
```

Example - Dead Code Analysis

Another analyzer I find useful in Enlightn is the "Dead Code Analyzer" which scans through your application's code to find unreachable or dead code.

The Enlightn documentation provides an example block of code that would be flagged by this analyzer:

```php
function exampleFn(array $a) {
    if (count($a) === 0) {
        foreach ($a as $val) {
            echo $a;
        }
    }
}
```

At a first glance, it's not immediately obvious that there's anything wrong with the code above. However, you may notice that the first line in the method is checking whether the `$a` array has no values. If it has no values, it then tries iterating through the array. But this means that the code within the `foreach` will never actually be run because the `$a` array has no items to iterate over.

I find that this analyzer is particularly useful for detecting bugs (similar to the example) that might have been introduced during a refactor that has been missed accidentally.

Larastan

Another useful tool you can use for a quick analysis of your project is Larastan.

Larastan is a package built on top of PHPStan (a static analysis tool) that can be used to perform code analysis. Typically, static analysis tools analyse your code without actually running it. However, according to the Larastan documentation "Larastan boots your application's container, so it can resolve types that are only possible to compute at runtime. That's why we use the term 'code analysis' instead of 'static analysis'".

Larastan can be used for spotting potential bugs in existing code, and can also be used in your CI (continuous integration) pipeline to reduce the chance of bugs being added in the future (we'll be covering how to do this in a later section).

Installation

To start using Larastan, install it via Composer using the following command:

```
composer require nunomaduro/larastan --dev
```

Next, create a `phpstan.neon` file in your Laravel project's root. The package's documentation provides the following as an example file:

```
includes:
    - ./vendor/nunomaduro/larastan/extension.neon

parameters:


    paths:
        - app

    # The level 9 is the highest level
    level: 5

    ignoreErrors:
        - '#PHPDoc tag @var#'

    excludePaths:
        - ./*/*/FileToBeExcluded.php

    checkMissingIterableValueType: false
```

That's it. Larastan is now ready to go! To run Larastan, all you need to do is run the following command:

```
./vendor/bin/phpstan analyse
```

The command should then output a list of the errors that it has detected to your terminal.

False Positives

If it's your first time running Larastan (or any other code analysis tool) on your codebase, you'll likely have a lot of errors when you run it. This isn't always something to worry about as some errors may be false positives. For example, imagine we have the following method:

```
use App\Models\Admin;

class StatsService
{
    public function generateStatsForAdmin(Admin $admin): void
    {
        // ...
    }
}
```

Assume our application provides a way for admin users to sign in and that they are stored in an `admins` database table and are authenticated using an `admin` guard. In a controller, we may want to access the authenticated admin using `auth('admin')->user()` and pass it into the `generateStatsForAdmin` method like so:

```
app(NewsletterService::class)->generateStatsForAdmin(
    auth('admin')->user()
);
```

This is perfectly valid and would run. However, due to the fact that the `auth()->user()` method's docblock specifies that the method will either return an instance of `Illuminate\Contracts\Auth\Authenticatable` or `null`, Larastan would flag the following error:

```
Parameter #1 $user of method
App\Services\StatService::generateStatsForAdmin() expects
App\AdminUser, Illuminate\Contracts\Auth\Authenticatable|null
given.
```

I come across this scenario a lot in projects. Usually, to get to a certain point in a controller, it's likely that you'll have passed through the `auth` middleware and will be confident (especially if you have tests) that `auth()->user()` will always return an object. So if you're confident this is the case and want to silence the error without changing any code, you can use either of the following approaches.

You could either add `// @phpstan-ignore-next-line` above the method call like so:

```
// @phpstan-ignore-next-line
app(NewsletterService::class)->generateStatsForAdmin(auth('admin')
->user());
```

Or you could add `// @phpstan-ignore-line` on the same line as the method call like so:

```
app(NewsletterService::class)->generateStatsForAdmin(auth('admin')
->user()); // @phpstan-ignore-line
```

As you can probably guess from the examples, adding either of these two comments will make Larastan ignore the lines and not flag any errors with them. It's very important to remember that you should use this approach sparingly and only if you're confident that there isn't an issue. Adding these comments can potentially hide bugs and reduce the effectiveness of running a code analysis tool like Larastan.

Incrementally Increasing the Level

To better understand the types of errors in your application and draw up a plan to fix them, you may want to start by setting the `level` field in the `phpstan.neon` file to `1` (the least strict level). By doing this, you'll be able to get an idea of the simplest fixes you can make to the codebase. This means that instead of tackling a huge number of errors at once, you can take small, incremental steps to improve the quality of your code. As a result, less code is changed at once, reducing the likelihood of introducing bugs during refactoring.

One approach I like is to tackle all the errors at level 1 and then slowly work my way up (usually to the highest level, which is 9). Depending on the size and complexity of the project, the process of fixing all the issues can take days, weeks, and sometimes even months. During this process, you'll find places where your code can be improved.

Let's look at an example of code that Larastan would flag an issue for. Imagine that we have a blogging application with a `Post` model. Assume this model has a naive `calculateReadTime` method which estimates how long it will take to read the blog post in seconds (assuming a reader reads 265 words per minute):

```php
class Post extends Model
{
    // ...

    /**
     * @return int
     */
    public function calculateReadTime()
    {
        return ceil(
            (Str::wordCount($this->content) / 265) * 60
        );
    }

    // ...
}
```

The method is taking the word count of the model's `content` field, dividing it by 265, multiplying it by 60, then rounding it up to the nearest whole number. It appears that there aren't any problems with this code and that it should work without issues. We could even write some small tests for this method:

```php
use App\Post;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

class CalculateReadTimeTest extends TestCase
{
    use RefreshDatabase;
    use WithFaker;

    /**
     * @dataProvider readTimeDataProvider
     */
    public function test_correct_read_time_is_returned(
        int $wordCount,
        int $expectedResult
    ): void {
        $content = $this->faker->words($wordCount, true);

        $post = Post::factory()->make(['content' => $content]);

        self::assertEquals(
            $expectedResult,
            $post->calculateReadTime(),
        );
    }

    public function readTimeDataProvider(): array
    {
        return [
            [3, 1],
            [265, 60],
            [530, 120],
            [500, 114],
            [2000, 453],
        ];
    }
}
```

We are testing that the method returns the correct read time when we're creating `Post` models with different lengths of `content`. These tests would all pass, and could give you extra confidence that the code is bug-free.

However, the `calculateReadTime` method actually has a small bug (that Larastan could detect for us) that could potentially cause further issues if the result of the method is used in other parts of the codebase. In the method, the read time value is being rounded up using the `ceil` function and returning a `float` but the method actually expects an `int` to be returned. This means we are working with data in subsequent code that is in a different format than expected. I often find pieces of code like this that have gone unnoticed during reviews and then caused bugs in a production environment.

Larastan would present this bug to us in the command's output like so:

```
Method App\Post::calculateReadTime() should return int but returns
float.
```

To fix a bug like this, we could start by improving the test suite to use the `assertSame` assertion rather than the `assertEquals` assertion. When we use `assertEquals`, only a loose-comparison is performed. For example, if our method returned a float with a value of `1.0` and we were asserting that it was equal to an integer of value `1`, the assertion would pass. However, by using `assertSame`, a strict comparison is performed, and so it asserts that both the values and the types are the same. After using the stricter assertion, the tests will start to fail and indicate that there's an issue.

If we were confident that the method should always in fact return an `int`, we could add a return type to the `calculateReadTime` method like so:

```
public function calculateReadTime(): int
{
    return ceil(
        (Str::wordCount($this->content) / 265) * 60
    );
}
```

This would provide a solution to the problem by casting the result of the `ceil` function to an `int` when returning. But it's important to remember that this change should only be made if you are confident that you are expecting an `int` as defined in the method docblock. There may be times when the docblock is incorrect or outdated and that it should not exist. For example, the method may have returned an `int` in the past but was refactored to purposely return a `float`, but the docblock wasn't updated.

Although this would now allow the tests and Larastan analysis to pass at a lower level, this code would be flagged when using a stricter and higher Larastan level. This is because we are still returning a `float` and relying on the return type for the result to be cast to an `int`. If we were to enable "strict types" in the file, the code wouldn't actually run. For example, we could enable strict types by adding `declare(strict_types=1);` to the top of our `Post` class like so:

```
declare(strict_types=1);

class Post extends Model
{
    // ...

    public function calculateReadTime(): int
    {
        return ceil(
            (Str::wordCount($this->content) / 265) * 60
        );
    }
}
```

Now, when this method is called, a `TypeError` with the following message will be thrown:

```
TypeError : App\Models\Post::calculateReadTime(): Return value
must be of type int, float returned
```

To allow the code to run with strict types enabled, and to pass the stricter Larastan checks, we could explicitly type cast the result before we return it using `(int)`, like so:

```php
public function calculateReadTime(): int
{
    return (int) ceil(
        (Str::wordCount($this->content) / 265) * 60
    );
}
```

Using Larastan, we can highlight potential causes of bugs like the one in our simple `calculateReadTime` method so we can fix them.

Setting a Baseline

As discussed, when you first run Larastan, you'll likely have a large amount of errors that are difficult to tackle. Larastan provides the ability to set a "baseline", which effectively ignores the existing code and only detects issues in new or changed code.

To generate the baseline, run the following command:

```
./vendor/bin/phpstan analyse --generate-baseline
```

After I've tackled all the low-hanging fruit and solved as many issues as possible, I like to use this command. It means that I can still use Larastan in my future code and can use it in my CI pipeline using GitHub Actions so that I can catch new bugs that I may have introduced without having all the noise in command output from issues in the older code. Of course, it's important to remember that you should use this approach carefully because you can hide existing errors that would have been detected otherwise.

Style CI

StyleCI is a fully-managed continuous integration tool you can use to apply styling to your code automatically. You can connect Style CI to GitHub, GitLab, and Bitbucket accounts so that it can automatically fix your code style whenever you make a pull request or push to a project's repository.

I like using StyleCI as both an auditing tool and as a continuous integration tool for future code. It supports styling for PHP, JavaScript, CSS, Vue.js, and Python, so it should cover a majority of the code in your projects. This means that you'll be able to have a codebase that is consistently styled (whether to match a standard such as PSR-12 or to match your company's style guide). By offloading the responsibility of dealing with code style to an automated tool, it means you don't need to bother as much with perfecting the styling when writing your code, because you can be sure that it will be fixed before being merged into your `master` or `main` branch.

In my opinion, having code that is well formatted is important because it can make it much easier for everyone in your team to read and understand by making the structure and styling of the code predictable.

There are other tools that are available to use (and potentially cost no money) such as PHP CS Fixer, but I like the fact that StyleCI can fix code style issues in multiple languages and that I don't need to configure and set up multiple tools in my CI pipeline to achieve the same result.

Another huge benefit of using StyleCI is that it runs very quickly and gives almost immediate feedback on the styling of your code. Graham Campbell, the creator of StyleCI, reported that during 2021, the median analysis run time (including code-fetching from the version control providers) was 1.6 seconds. This means that you don't have to wait for any GitHub Actions to boot up and run and can continue building new features quickly.

To get started with StyleCI, you'll first need to head over to https://styleci.io to register, then connect your accounts to your version control provider.

When you run your first analysis with StyleCI, it's likely you'll have a lot of potential changes to make. But this is completely fine and is to be expected. No matter how meticulous you are at manually formatting your code before committing it, there will always be inconsistencies. Usually I wouldn't merge the initial analysis into my project's codebase until the entire audit is complete. As already discussed, this is because I like to complete an entire audit before I begin making changes. However, if you have looked through all the proposed changes and are confident that they won't affect any code behaviour in your application, you should *usually* be safe to accept them. By beautifying the code this way, you may find that it could also make the manual auditing process easier because the code will be in a more predictable standard.

When setting up StyleCI, I recommend configuring it so that it runs on each push (or pull request) so that all your future code is styled correctly. The proposed style changes can be set to automatically merge into a branch, or can be proposed using a pull request so that you can check the changes before accepting them.

Configuration and Risky Behaviour

I use the default styling rules that StyleCI provides. However, if you want to edit the rules (maybe to match your company's style guide), you can do this by creating a `.styleci.yml` file in your Laravel project's root. For example, we could create an example file like so:

```yaml
php:
  risky: false
  version: 8.1
  preset: recommended
```

It's important to note that we have explicitly defined `risky` to be `false` (which would also be the same behaviour if the field wasn't present). This is because some fixers can change code behaviour, but can potentially introduce bugs. So by disabling the "risky" fixers, we can be sure that only the code style is affected and not the code behaviour. As shown in the StyleCI documentation, an example of a risky fixer is the `array_push` fixer. By enabling this, it will convert any simple instances of `array_push($x, $y)` to `$x[] = $y`. Although it's unlikely that this would cause any issues and would improve the codebase, it shouldn't be blindly allowed and must be checked (either through manual or automated testing) to make sure it does not introduce any bugs to the code.

Code Coverage

If you're working on a project that already has a test suite setup, your audit process can benefit from running the tests with a code coverage tool. By doing this, it can help to give you an idea of which parts of the project's code have some tests running over them and which bits aren't currently covered.

Running the tests and seeing the code coverage can also be useful for when you come to deciding on your testing strategy later. For example, it can help you determine which parts of the project should be prioritised for writing tests.

Generating the Code Coverage Report

If you're running your Laravel project's tests using the `php artisan test` command, you can make use of the built-in code coverage report feature.

To start with using the code coverage tool, you'll need to make sure that you have Xdebug or PCOV configured in PHP. Then you just pass the `--coverage` option to the command like so:

```
php artisan test --coverage
```

This will run your application's tests then generate the coverage report (the results of both will be output in the command line). To get an idea of what the output looks like, let's take a look at the code coverage output for an example project:

```
Actions/ConfirmSubscription  ........................ 100.0 %
Actions/SendSubscriptionConfirmation  ............... 100.0 %
...
Listeners/ReportDetectedSpam .......................... 0.0 %
...
Notifications/Admin/AttemptedSubscription 45..55 ..... 22.2 %
Notifications/Admin/ConfirmedSubscription 32..38 ..... 33.3 %
Notifications/NewsletterDailyStats .................... 0.0 %
Notifications/Visitor/AttemptedSubscription 40..44 ... 28.6 %
Notifications/Visitor/ConfirmedSubscription 33..37 ... 16.7 %
...
Post 81, 197..214 .................................... 71.8 %
Providers/AppServiceProvider  ....................... 100.0 %
...
User  ............................................... 100.0 %

Total Coverage ....................................... 45.1 %
```

Note: The length of the report was reduced for the sake of this example, so some lines have been removed and replaced with `...`.

In our example report above, we can see that files such as `Actions/ConfirmSubscription` and `Actions/SendSubscriptionConfirmation` have a code coverage of 100%. We can also see that `Listeners/ReportDetectedSpam` has a code coverage of 0%, meaning that the tests never ran any lines of code within that file.

The report also shows us which lines of code in a given file weren't run if the file was partially run. For example, we can see that only 22.2% of the lines in `Notifications/Admin/AttemptedSubscription` were run. The report also tells us that lines 45-55 in that file were not run; so we can get a clear understanding of what code hasn't been run during the testing.

It's also worth noting that you can also pass a `--min` option to the command to specify the minimum coverage that the application needs to have to pass. If the coverage is lower than the given minimum, the test suite will fail. This can sometimes be useful if you want to maintain a certain amount of code coverage and want your continuous integration pipeline to fail when the coverage falls below a given point. For example, if we want to specify that the application must have at least 70% code coverage, you could run the following command:

```
php artisan test --coverage --min=70
```

Things To Be Aware Of

When viewing your code coverage it's important to remember that it's purely a guide and is only highlighting whether a line of code has been run. So if you have some complex pieces of code that appear to have code coverage, this doesn't necessarily mean that the code is bug-free; it simply means that the code was run at least once during the tests being executed.

For example, let's take this basic controller that just updates a user:

```php
use App\Http\Requests\UpdateUserRequest;

class UserController extends Controller
{
    public function update(
        UpdateUserRequest $request,
        User $user
    ): RedirectResponse {
        $user->update($request->validated());

        return redirect()->route('users.edit', $user);
    }
}
```

We can then take a look at a simple test for this method:

```php
namespace Tests\Feature\Controllers\BlogController;

use Illuminate\Foundation\Testing\LazilyRefreshDatabase;
use Tests\TestCase;

class UpdateTest extends TestCase
{
    use LazilyRefreshDatabase;

    /** @test */
    public function user_can_be_updated(): void
    {
        $user = User::factory()->create();

        $patchData = [
            'name' => 'A new name',
            'email' => 'new-email@example.com',
        ]

        $this->patch(
            route('users.update', $user),
            $patchData
        )->assertRedirect();
    }
}
```

By running the test above we can assert that the controller method has accepted the data and has returned a redirect response. However, it hasn't asserted where the user is redirected to, or whether the user has actually been updated in the database. So this exemplifies how you can sometimes have high code coverage but that doesn't necessarily show us that there aren't any bugs.

It's also not necessarily a huge issue if some parts of your code aren't covered. As an example, if our application contained a simple helper function that is used to format a string before outputting it in the view, we'd be less concerned about that being tested than a service class that handles billing and subscriptions. So you must decide on a project-by-project basis whether you're happy with your current code coverage and if you think more tests should be added for specific parts of the codebase.

Manual Auditing

Now that we've seen some automated tools to audit a large amount of our codebase, we can step through the code manually to find issues that may have been missed.

Manually searching through your code is useful for taking a closer look at the code and logic so that you can spot some of the less obvious issues. However, it can be a long, tedious, and daunting task depending on the size and complexity of your project. So you might find it easier to do in chunks (of course depending on things like your deadlines and timescales). You might start to find that if you try and review too much at once that you'll start to get bored and lose concentration, so you'll miss things.

Choosing how to perform the manual audit is entirely up to you. I like to start at the top of my project root and then work through every folder and file in order. You might want to tackle this in a different order or only work on specific chunks of your applications. For example, you might only be interested in auditing your application's API.

As we've mentioned, it's really easy to want to open up the git history and blame a past developer whenever you find an error. But it's really important that you remember not to do this, and only search through the history to get an understanding or context around a particular feature or code block. Pinning the blame on someone won't benefit anyone, so remember to keep an open mind and be open to conversation.

Let's take a look at the different things that we'll be looking out for.

Investigating "raw" Database Queries

When auditing your project, you may come across "raw" database queries. Raw queries aren't necessarily a bad thing and can definitely be very powerful when used in the right ways. However, if they aren't written and managed properly, they can lead to "SQL injection" vulnerabilities in your code. SQL injection is a vulnerability that allows attackers to run additional code inside your SQL queries so that they can access your database. They can use these attack vectors to do several harmful things such as:

- View your entire database's contents
- Export your database's contents (including user data)
- Update content within your database
- Completely destroy your database

Thanks to Eloquent, the majority of queries made from a Laravel application aren't vulnerable to SQL injection attacks. However, using "raw" queries incorrectly can create SQL injection attack vectors.

Let's take a look at an example that I have come across in a project in the past. The project that I was auditing was a multi-tenant application and provided a search functionality for users using code similar to this:

```
User::where('tenant_id', auth()->user()->tenant_id)
    ->whereRaw(
        "CONCAT(first_name, ' ', last_name) LIKE '%"
        .$request->input('search')."%'"
    )
    ->get();
```

If we were to assume that we passed "Joe B" in as the `search` query parameter and that the authenticated user is a member of tenant `1`, this would result in the following SQL query:

```
SELECT *
FROM `users`
WHERE tenant_id = 1
  AND CONCAT(first_name, ' ', last_name) LIKE '%Joe B%'
```

The above code is concatenating the `first_name` and `last_name` fields in the database so that they can be searched against at the same time. For example, if we had a user in the database called "Joe Bloggs", we'd want them to show up in our search results if we searched for "Joe B". It's also adding a constraint to the query so that users can only see other users that are part of the same tenant as they are. However, we are directly passing the `search` parameter from the request into the query. This means that Laravel won't be escaping this field for us and is leaving us susceptible to an SQL injection attack.

For example, if we assume that this query is being used within a `/users` route, an attacker could make a request to route with the parameters:

```
/users?search='OR 1=1;-- --
```

This would result in the following SQL query:

```
SELECT *
FROM `users`
WHERE tenant_id = 1
    AND CONCAT(first_name, ' ', last_name) LIKE '%'
    OR 1=1;-- --'%'
```

As you can see, by passing `'OR 1=1;-- --`, we've managed to leave the `LIKE` clause and add some new code to the query. In this case, we have added `OR 1=1`. Thus, every user in the database (even ones from other tenants) will be returned to the user. This is extremely dangerous and can lead to user data being leaked.

Depending on how the query is written and where it is used, it may be possible for an attacker to export your entire application's database contents or even delete it completely.

To rectify this issue without making too many changes to the structure of the query, you can switch to using bindings that are provided by the "raw" methods. To fix our example query above and prevent the SQL injection, we can update the query to:

```
User::where('tenant_id', auth()->user()->tenant_id)
    ->whereRaw(
        'CONCAT(first_name, " ", last_name) LIKE ?',
        ['%'.$request->input('email').'%']
    )
    ->get();
```

We've changed the query to be a prepared statement and replaced the `'%".$request->input('search')."%'"` with a `?`. We've then defined in the second parameter of the `whereRaw` method what the `?` should be replaced with. This will prevent an attacker from breaking out of the current query (as we did earlier) and adding harmful code to the query.

To find any raw queries in my own project, I would typically do a project-wide search (in my IDE or editor) for the term "raw". Doing this makes it much easier to track down any usages of raw queries (such as in `whereRaw`, `orWhereRaw`, `selectRaw`, etc.).

If you do come across any potentially vulnerable raw queries, you might want to spend some time looking into them and checking whether they are vulnerable and if they can be exploited. If you find that any of them are vulnerable, I would highly recommend fixing the query as a priority due to the fact that it can lead to user's data being leaked or your database being edited or deleted.

Finding Incorrect Authorisation

A potentially harmful security issue that I have found in many projects that I have audited is incomplete (or incorrect) authorisation checks.

Authorisation is the process of providing access to a resource for someone depending on things such as their assigned role or permissions. For example, you may have a web application that allows users to create and generate invoices. In this project, we would want to allow the user who created the invoice to be able to view it. But we would want to make sure that the other users in the system cannot view that same invoice; they should only be able to view their own.

As you can imagine, incorrect authorisation checks can lead to users being able to access information that does not belong to them and it could lead to user data being leaked.

I would advise stepping through each of the routes in your application and carefully analysing whether the authorisation is complete and doesn't include any potential vulnerabilities. If you do come across any, depending on the route and the information that can be accessed from it, you may want to prioritise fixing the issue to prevent any potential misuse.

To understand the different types of incomplete authorisation that you may come across in your application, let's look at a few examples:

Missing Server-Side Authorisation

I have worked on a number of projects where authorisation has not been applied to routes because it's thought unnecessary since the UI never presents a way of getting to that route if the user isn't authorised.

For example, let's imagine that we have a page that contains a link for users with the `view-users` permission to view a user's details. For the sake of the example, we'll also assume that we have added a `hasPermissionTo` method that can be used to check whether a user has a specific permission assigned to them or not. The block of code to display this link in the Blade view might look something like this:

```
@if(auth()->user()->hasPermissionTo('view-users'))
    <a href="{{ route('users.show', $user) }}">
        View {{ $user->name }}
    </a>
@endif
```

With the block of code in the Blade view above, this would mean that the link will only be displayed for users that have the `view-users` permission. If they don't, then the link will not be displayed.

I have seen this in projects before as an argument that server-side authorisation is not needed. However, it is important to remember that you **always** need to provide server-side authorisation. Not displaying the button provides a user experience improvement, but it doesn't prevent any malicious users from attempting to access the route itself, likely via scripted attacks.

Let's imagine that the basic controller (that doesn't contain any authorisation) looks this:

```
use App\Models\User;
use Illuminate\Contracts\View\View;

public function UserController extends Controller
{
    // ...

    public function show(User $user): View
    {
        return view('users.show', [
            'user' => $user,
        ]);
    }
}
```

We could simply add some authorisation to it by mimicking the same permission check from the Blade view to the controller like so:

```php
use App\Models\User;
use Illuminate\Contracts\View\View;

public function UserController extends Controller
{
    // ...

    public function show(User $user): View
    {
        if (!$user->hasPermissionTo('view-users')) {
            abort(403);
        }

        return view('users.show', [
            'user' => $user,
        ]);
    }

    // ...
}
```

This would add authorisation to the controller method and protect it, but you may want to consider extracting the authorisation logic out into a policy. This means that if you're using this logic anywhere, and you need to add an extra condition to the check (such as checking that a user belongs to a specific tenant or team), you can apply it in one place rather than everywhere. To do this, you could create a `App\Policies\UserPolicy` like so:

```
class UserPolicy
{
    // ...

    /**
     * Determine if the authenticated user can view another user.
     *
     * @param  \App\Models\User  $authUser
     * @param  \App\Models\User  $user
     * @return bool
     */
    public function view(User $authUser, User $user): bool
    {
        return $authUser->hasPermissionTo('view-users');
    }

    // ...
}
```

We can then update our Blade code to use the policy instead of the raw permission check:

```
@if(auth()->user()->can('view', $user))
    <a href="{{ route('users.show', $user) }}">
        View {{ $user->name }}
    </a>
@endif
```

We can also update the controller to use the policy method by making use of the `authorize` method:

```
use App\Models\User;
use Illuminate\Contracts\View\View;

public function UserController extends Controller
{
    // ...

    public function show(User $user): View
    {
        $this->authorize('view', $user);

        return view('users.show', [
            'user' => $user,
        ]);
    }

    // ...
}
```

This now means that both the Blade view and controller are using the same logic for displaying the link and authorising any users that access it. This is beneficial because it means that any changes to the logic in the future will be applied to both places. This reduces the chances of forgetting to update one of the checks when updating authorisation logic in the future.

Not Checking the Tenant/Team/Owner

Say we have a multi-tenant blogging application that allows users to create, update, delete, and publish blog posts. Imagine that the application has two different roles:

- **Manager** - They have permission to create, update, delete, and publish blog posts.
- **Writer** - They have permission to create and update blog posts.

A team would potentially consist of many writers who write the articles, which can then be published after being checked by the manager.

If we wanted to write the authorisation check for publishing a blog post in plain English, it might look something like this:

"A blog post can only be published by a user that has permission to publish a blog post, and only if it belongs to their team".

As you can see, there are two different checks that we need to perform here. However, in several applications that I've audited, I've found that many authorisation checks miss the check to ensure that the resource belongs to the user's tenant or team. Therefore, in this scenario, it would allow any users with the "manager" role to publish blog posts that belong to other teams.

For example, if we were to assume that we are performing the authorisation checks within a `PostPolicy` policy, we might have a `publish` method that looks like so:

```
class PostPolicy
{
    // ...

    /**
     * Determine if the post can be published by the user.
     *
     * @param  \App\Models\User  $user
     * @param  \App\Models\Post  $post
     * @return bool
     */
    public function publish(User $user, Post $post): bool
    {
        return $user->hasPermissionTo('publish-post');
    }

    // ...
}
```

This means that if we were to call `auth()->user()->can('publish', $post)` for a blog post that belongs to another user, access would be granted as long as the user has the `publish-post` permission granted via the "manager" role. So in order to strengthen this check we would also need to assert that the user belongs to the team as the blog post.

This can be achieved in multiple ways, depending on how the project is structured. In a lot of projects, this check may be implicitly applied through the use of a query scope that adds a constraint to every database query to only use rows in the database that have the tenant or team ID (sometimes in a column called `tenant_id` or `team_id`). However, this bug tends to be found more in the projects that apply this check manually.

If we imagine that all the tables in our database (except for the `teams` table) have a `team_id` column, we could make use of this in the policy like so:

```php
class PostPolicy
{
    // ...

    /**
     * Determine if the post can be published by the user.
     *
     * @param  \App\Models\User  $user
     * @param  \App\Models\Post  $post
     * @return bool
     */
    public function publish(User $user, Post $post): bool
    {
        return $user->hasPermissionTo('publish-post')
            && $user->team_id === $post->team_id;
    }

    // ...
}
```

This now means that when we call `auth()->user()->can('publish', $post)`, we are also making sure that the user belongs to the same team as the post.

Not Scoping the Relationships

When auditing some applications, I have come across routes that contain relationships but that aren't properly scoped and can be used to access resources that belong to different users/teams/tenants.

Let's imagine that we have a very basic invoicing application that allows users to create projects and then create invoices for them.

To view an invoice you might have a route that looks similar to this:

```php
Route::get(
    '/projects/{project}/invoices/{invoice}',
    [InvoiceController::class, 'show']
);
```

Say we have a user (we'll call them User One) in the database that has a project (with ID: 1) and an invoice belonging to that project (with ID: 10). For the user to access this invoice, they might navigate to the following route in their browser: `/projects/1/invoices/10`.

We'll also say that we have another user (we'll call them User Two) in the database that has a project (with ID: 2) and an invoice belonging to that project (with ID: 20). For this user to access the invoice, they might navigate to the following route in their browser: `/projects/2/invoices/20`.

So far, this seems very normal and nothing out of the ordinary. However, I have come across routes that only check authorisation against one of the resources (in this case, the project or invoice) but without checking that they actually belong to each other. As a basic example, I have seen this approach used quite often in controller methods:

```php
use App\Models\Invoice;
use App\Models\Project;
use Illuminate\Contracts\View\View;

public function InvoiceController extends Controller
{
    // ...

    public function show(
        Project $project,
        Invoice $invoice
    ): View {
        // Check that the project belongs to the user.
        if ($project->user_id !== auth()->user()->id) {
            abort(403);
        }

        return view('invoices.show', [
            'invoice' => $invoice,
        ]);
    }

    // ...
}
```

As we can see, we are only confirming that the project belongs to the user, and are assuming that users will only ever try to view invoices that belong to their project. However, due to this, it means that as long as the user passes a project ID that they belong to in the URL, they can access any invoice that they want. This means that User One can access User Two's invoice by going to: `/projects/1/invoices/20`. This can be dangerous and, depending on where this type of bug exists, can be used to compromise users' data.

If we wanted to prevent this within the controller, we could add a new condition to our controller method like so:

```php
use App\Models\Invoice;
use App\Models\Project;
use Illuminate\Contracts\View\View;

public function InvoiceController extends Controller
{
    // ...

    public function show(
        Project $project,
        Invoice $invoice
    ): View {
        if ($invoice->project_id !== $project->id) {
            abort(404);
        }

        if ($project->user_id !== auth()->user()->id) {
            abort(403);
        }

        return view('invoices.show', [
            'invoice' => $invoice,
        ]);
    }

    // ...
}
```

This means that if a user was to navigate to this route using an invoice that doesn't belong to the given project, an HTTP 404 response would be returned.

However, as of Laravel 9, you can use the `scopeBindings` feature to automatically apply this logic when creating a route. To update the route to automatically check that the invoice belongs to the projects via a relationship, it could be changed to:

```
Route::get(
    '/projects/{project}/invoices/{invoice}',
    [InvoiceController::class, 'show']
)->scopeBindings();
```

This removes the need for the `$invoice->project_id !== $project->id` check in the controller method.

When building routes in the future, you may also want to consider using shallow nesting routes that remove the concept of parent and child identifiers, and only use one at once. In our example, the invoice ID is already a unique value, so we don't necessarily need the project ID as well (but this might not be the case depending on your project). So we could update the route like so:

```
Route::get(
    '/invoices/{invoice}',
    [InvoiceController::class, 'show']
);
```

This would remove the need to check whether an invoice belongs to a project because there wouldn't be a project in the request. This can help simplify authorisation logic.

If you're using resource controllers, you can also automatically apply this logic using the `shallow` method when generating the route. For example, we could generate a set of resourceful routes like so:

```
Route::resource(
    'projects.invoices',
    InvoiceController::class
)->shallow();
```

This would generate the following routes:

Verb	URI	Action	Route Name
GET	`/projects/{project}/invoices`	index	projects.invoices.index
GET	`/projects/{project}/invoices/create`	create	projects.invoices.create
POST	`/projects/{project}/invoices`	store	projects.invoices.store
GET	`/invoices/{invoice}`	show	invoices.show
GET	`/invoices/{invoice}/edit`	edit	invoices.edit
PUT/PATCH	`/invoices/{invoice}`	update	invoices.update
DELETE	`/invoices/{invoice}`	destroy	invoices.destroy

Checking Validation

When auditing your application, you'll want to check through each of your controller methods to ensure that the request data is correctly validated. As we've already covered in the Mass Assignment Analysis example that Enlightn provides, we know that all data that is used in our controller should be validated and that we should never trust data provided by a user.

Whenever you're checking a controller, you must ask yourself "Am I sure that this request data has been validated and is safe to store?". As we briefly covered earlier, it's important to remember that client-side validation isn't a substitute for server-side validation; both should be used together. For example, in past projects I have seen no server-side validation for "date" fields because the form provides a date picker, so the original developer thought that this would be enough to deter users from sending any other data than a date. As a result, this meant that different types of data could be passed to this field that could potentially be stored in the database (either by mistake or maliciously).

Applying the Basic Rules

When validating a field, I try to apply these four types of rules as a bare minimum:

- **Is the field required?** - Are we expecting this field to always be present in the request? If so, we can apply the `required` rule. If not, we can use the `nullable` or `sometimes` rule.
- **What data type is the field?** - Are we expecting an email, string, integer, boolean,

or file? If so, we can apply `email`, `string`, `integer`, `boolean` or `files` rules.
- **Is there a minimum value (or length) this field can be?** - For example, if we were adding a price filter for a product listing page, we wouldn't want to allow a user to set the price range to -1 and would want it to go no lower than 0.
- **Is there a maximum field (or length) this field can be?** - For example, if we have a "create" form for a product page, we might not want the titles to be any longer than 100 characters.

So you can think of your validation rules as being written using the following format:

```
REQUIRED|DATATYPE|MIN|MAX|OTHERS
```

By applying these rules, not only can it add some basic standard for your security measures, it can also improve the readability of the code and the quality of submitted data.

For example, let's assume that we have these fields in a request:

```
'name',
'publish_at',
'description',
'canonical_url',
```

Although you might be able to guess what these fields are and their data types, you probably wouldn't be able to answer the 4 questions above for each one. However, if we were to apply the four questions to these fields and apply the necessary rules, the fields might look like so in our request:

```
'name' => 'required|string|max:200',
'publish_at' => 'required|date|after:now',
'description' => 'required|string|min:50|max:250',
'canonical_url' => 'nullable|url',
```

Now that we've added these rules, we have more information about four fields in our request and what to expect when working with them in our controllers. This can make development much easier for users that work on the code after you because they can have a clearer understanding of what the request contains.

Applying these four questions to all of your request fields can be extremely valuable. If you find any requests that don't already have this validation, or if you find any fields in a request missing any of these rules, I'd recommend adding them. But it's worth noting that these are only a bare minimum and that in a majority of cases you'll want to use extra rules to ensure more safety.

Checking for Empty Validation

An issue I've found quite often with projects that I have audited is the use of empty rules for validating a field. To understand this a little better, let's take a look at a basic example of a controller that is storing a user in the database:

```php
use App\Models\User;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    public function store(Request $request): RedirectResponse
    {
        $validated = $request->validate([
            'name' => 'required|string|max:255',
            'email' => 'required|email',
        ]);

        $user = User::create($this->validated());

        return redirect()->route('users.show', $user);
    }
}
```

The `store` method in the `UserController` is taking the validated data (in this case, the `name` and `email`), creating a user with them, then returning a redirect response to the users 'show' page. There isn't any problem with this so far.

However, let's say that this was originally written several months ago and that we now want to add a new `twitter_handle` field. In some projects that I have audited, I have come across fields added to the validation but without any rules applied like so:

```php
use App\Models\User;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    public function store(Request $request): RedirectResponse
    {
        $validated = $request->validate([
            'name' => 'required|string|max:255',
            'email' => 'required|email',
            'twitter_handle' => '',
        ]);

        $user = User::create($this->validated());

        return redirect()->route('users.show', $user);
    }
}
```

This means that the `twitter_handle` field can now be passed in the request and stored. This can be a dangerous move because it circumvents the purpose of validating the data before it is stored.

It is possible that the developer did this so that they could build the feature quickly and then forgot to add the rules before committing their code. It may also be possible that the developer didn't think it was necessary. However, as we've covered, all data should be validated on the server-side so that we're always sure the data we're working with is acceptable.

If you come across empty validation rule sets like this, you will likely want to add the rules to make sure the field is covered. You may also want to check the database to ensure that no invalid data has been stored.

Finding "Fake Facades"

One issue I've seen with a lot of projects that I've worked on is the use of what I refer to as "fake facades". In a nutshell, "fake facades" are just classes with static methods on them that are misused and are purely there to make the code look more "Laravel-y".

When I've asked developers why they've written the code this way, they tend to explain that it's because they've seen that facades in Laravel use this same approach. So they get slightly confused and assume that facades are just static methods on an object. That is an easy mistake to make, and one that I made myself when starting out with PHP and Laravel.

It's important to identify these blocks of code because they can make code more complex for no reason. For example, making a method static removes the ability to encapsulate and manage state inside an object. This means that if the method is static, the state is stored against the class rather than a specific object. This can affect subsequent calls of the same method.

This does not mean that static methods are bad. They're far from it. They can be incredibly useful for creating utility methods (such as those in the `Str` and `Arr` classes provided by Laravel). It's just important that we use them in the correct way to keep our code understandable.

To get a bit of context around fake facades and possible ways to remove them, let's take this small example of a fake facade:

```php
class PricingService
{
    private static bool $withTax = false;

    public static function calculatePrice(Product $product)
    {
        return static::$withTax
            ? $product->price * 1.2
            : $product->price;
    }

    public static function withTax(bool $withTax = true): void
    {
        static::$withTax = $withTax;
    }
}
```

At first glance, there doesn't seem to be anything wrong with the example code. If we wanted to call it, we could use something like:

```php
PricingService::withTax();
$price = PricingService::calculatePrice($product);
```

This would result in the price being calculated for the `$product` with tax.

We can see from the `PricingService` code example that the `withTax` field defaults to `false`. At first, you may think that if you were to just call the `calculatePrice` method that the price would be calculated without tax. However, if we want to calculate the price for another product without tax in the same request, this would not be the case.

For example, let's take these two separate blocks of code that are called in different parts of the codebase:

Block 1:

```php
PricingService::withTax();
$priceWithTax = PricingService::calculatePrice($productOne);
```

Block 2:

```php
$priceWithoutTax = PricingService::calculatePrice($productTwo);
```

Although it appears that the `priceWithoutTax` variable should have had the price calculated without the `tax`, it will have been included because we called the `withTax` method beforehand.

As you can imagine, this approach of using "fake facades" can cause hard-to-find bugs in code. The bugs can become more difficult to track down when the individual static method calls are in unrelated parts of the codebase.

In this particular example, we can solve these issues by making the most of OOP (object-oriented programming) and encapsulating the state inside an object. To do this, we can update the class to remove the `static` keywords like so:

```php
class PricingService
{
    private bool $withTax = false;

    public function calculatePrice(Product $product)
    {
        return $this->withTax
            ? $product->price * 1.2
            : $product->price;
    }

    public function withTax(bool $withTax = true): void
    {
        $this->withTax = $withTax;
    }
}
```

Now that we've done this, we'll need to invoke the classes before we use them. Taking our example from above, we can do that like so:

Block 1:

```php
$pricingService = new PricingService();

$pricingService->withTax();
$priceWithTax = $pricingService->calculatePrice($productOne);
```

Block 2:

```php
$pricingService = new PricingService();

$priceWithoutTax = $pricingService->calculatePrice($productTwo);
```

In our example above, we've updated the code to use separate instances when we need to calculate the price. This means that any changes made in the `PricingService` from block 1 (such as including tax in the calculation) won't affect the calculations in block 2.

To further improve this code, we could update the classes to resolve from the service container like so:

```
$pricingService = app(PricingService::class);

$pricingService->withTax();
$priceWithTax = $pricingService->calculatePrice($productOne);
```

By doing this, when we create the new `PricingService` object, it will be resolved from Laravel's service container, which improves things such as extensibility, mocking, and future flexibility of the code.

When auditing the code, if you come across any classes that use a lot of static methods or static properties, you or your team will need to decide if the class is a "fake facade". If it is, you can assess whether you think the class should be updated and look for any bugs caused by having it in the codebase.

Finding Business Logic in Helpers

Another issue I've come across a lot is when projects encapsulate complex or large amounts of business logic inside helper functions. As with the fake facades we've just discussed, helper functions can be extremely powerful in our code when used in the right way. They can improve the readability of our code and can provide useful functionality.

For example, you can have handy little helper functions like this one:

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

As you can see, the code is very simple, and we could easily write a test for it. It also would not matter if the code we were testing was using this helper function, because the output is easily predictable.

However, as is the case with static methods, you can't typically mock helpers in your tests. So this makes it difficult when writing tests to mock the input and output of a specific helper function that you might be using. Additionally, by placing code inside helper functions, it removes the ability to easily make use of class-based features such as properties, and splitting code into smaller methods.

Let's take a look at an example of a helper function that contains too much business logic:

```
use App\DataTransferObjects\UserData;
use App\Jobs\AlertNewUser;
use App\Models\Role;
use App\Models\User;
use Illuminate\Support\Facades\DB;

if (! function_exists('store_user')) {
    function store_user(UserData $userData): User
    {
        return DB::transaction(function () use ($userData) {
            $user = User::create([
                'email' => $userData->email,
            ]);

            $user->roles()->attach(
                Role::where('name', 'general')->first()
            );

            AlertNewUser::dispatch($user);

            return $user;
        });
    }
}
```

This function is doing too many things! It is using a database transaction, storing a user in the database, assigning them a role, and dispatching a job to alert them that they've been created.

This means that any time we test where this function is being used, we'll need to allow all of that code to run. We could mock parts of it (like the `AlertNewUser` job being dispatched), but it could get messy depending on the complexity of the helper function. Instead, it would be much nicer if we were able to mock the actual function or create a test double so that it's never actually invoked (unless we want it to be).

One of the easiest ways of solving this issue is to move the functions into classes. This would provide us with the ability to resolve the class from the service container so that we can mock it during our tests or replace it with a test double.

Finding N+1 Queries

It's inevitable that a project's code will have some N+1 queries. Almost every single project that I've worked on has had at least one N+1 query.

What Are N+1 Queries?

N+1 queries are queries in which you make N+1 calls to the database, where N is the number of items being fetched from the database. To explain this a bit better, let's look at an example:

Imagine that you have two models (`Comment` and `Author`) with a many-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:

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

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

In the above code, author names will be "lazy loaded". This would result in 101 database queries! The first query would be to fetch all 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?

To use "eager loading", we would change the code to:

```
$comments = Comment::with('author')->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 `::with('author')` this will fetch all the comments and then make another query to fetch all the authors at once. This means that we will have cut down the number of queries from 101 to 2!

How to Disable N+1 Queries

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

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

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

```php
namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    public function boot(): void
    {
        // ...

        Model::preventLazyLoading();

        // ...
    }
}
```

You might only want to enable this feature when in your local development environment. Doing that can alert you to places in your code that are using lazy loading while building new features, but not completely crash your production application and cause issues for your users. For this very reason, the preventLazyLoading() method accepts a boolean as an argument, so we could use the following line:

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

In our `AppServiceProvider`, it could look like this:

```php
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.

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 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:

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

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

How to Find N+1 Queries

We know what N+1 queries are, the effects that they can have on our application, and how to disable them. We can now start looking for them.

There are different ways to search for the occurrences, but unfortunately, the majority of them include manual, tedious work.

To get started, you'll want to disable lazy loading in our service provider like we did above. After you've done this, if you have an existing test suite, you should run your tests. You'll likely get some errors that are related to the setup of your tests. Generally, you can ignore them because, for the time being, we're only looking for N+1 queries in our app code. Depending on the size of your test suite, you'll likely spot some lazy loading occurrences in your code.

After you've run your test suite, we can start the tedious part of the process: going to each individual page in our browser. To do this, I'd step through every single page in your application and check that the page loads correctly. If you want to be extremely thorough, I'd also recommend submitting any forms, when possible.

Although this method might seem long-winded (especially if you have a large application), it can be really helpful for spotting N+1 queries in places you might not have covered in your tests. Don't worry if you don't have the time (or patience) to step through each and every page and form. Remember — auditing and improving your codebase isn't something that will happen overnight. If you find it easier, you might want to chunk the sections that you're checking. For example, check the billing section, then the user profile section, etc.

A top tip for performing this manual audit would be to ensure that you have some data in your application (either factory-generated data or real data). The lazy loading check and exception will only be thrown if you are actually calling data in the relationship. So if you don't have any data in the relationship, you might miss an occurrence. For example, let's look at our example from earlier:

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

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

If we don't have any comments stored in the database, it means we'll never enter the foreach loop. This means that the `$comment->author` relationship won't be invoked and so the exception won't be thrown even though there is lazy loading in that code.

Finding Controllers That Use Other Controllers

A common issue I've come across when auditing projects has been controller methods calling other controller methods. This usually happens when the controllers share similar functionality and indicates that the shared code should be extracted into something like a service class.

To get a bit of context about this issue, let's take a look at an example. For the purpose of this guide, the example is rather simple but it should highlight the general issue. I once audited a project that contained API routes that returned JSON data with:

- All the non-admin users
- All the admin users
- All the non-admin and admin users

The controller method for returning the non-admin users looked similar to this:

```
class UserController extends Controller
{
    public function index()
    {
        return User::where('is_admin', false)
            ->get()
            ->toArray();
    }
}
```

The controller method for returning the admin users looked similar to this:

```
class AdminUserController extends Controller
{
    public function index()
    {
        return User::where('is_admin', true)
            ->get()
            ->toArray();
    }
}
```

The controller method for returning both the admin and non-admin users looked similar to this:

```
class AllUserController extends Controller
{
    public function index()
    {
        $nonAdminUsers = (new UserController())->index();
        $adminUsers = (new AdminUserController())->index();

        return [
            'users' => $nonAdminUsers,
            'admin_users' => $adminUsers,
        ];
    }
}
```

As we can see in the `AllUserController`, the controller method is calling different controller methods to form the response. This is a bad practice and should be avoided as much as possible because controllers are only ever meant to handle HTTP requests and not be invoked manually.

When a controller method is manually called by ourselves, any necessary dependencies required for the controller method to work might not have been instantiated. Therefore, it's best to only ever allow controller methods to be called by the framework when a request is sent.

To rectify this particular issue without making too many changes to the structure of the code or query, we can lift the queries out into a service method. We could create a new `UserService` method that looks similar to this:

```
use Illuminate\Database\Eloquent\Collection;

class UserService
{
    public function users(bool $isAdmin): Collection
    {
        return User::where('is_admin', $isAdmin)
            ->get();
    }
}
```

We've created a new `users` method in the `UserService` class that accepts an `isAdmin` parameter. After doing this, we can update our `UserController` like so:

```
use App\Services\UserService;
use Illuminate\Http\JsonResponse;

class UserController extends Controller
{
    public function index(UserService $userService): JsonResponse
{
        return response()->json(
            $userService->users(isAdmin: false)
        );
    }
}
```

We've made several changes to the controller method such as:

- Type hinting a `UserService` instance so that it is resolved from the container for us to use.
- Adding a `JsonResponse` return type to explicitly define that we are returning a JSON response rather than an array. This can help dissuade future developers from trying to use the method's result again.

- Resolving the users that should be returned from the service class rather than inside the controller.

We can also mimic similar changes over to the `AdminUserController`:

```php
use App\Services\UserService;
use Illuminate\Http\JsonResponse;

class AdminUserController extends Controller
{
    public function index(
        UserService $userService
    ): JsonResponse {
        return response()->json(
            $userService->users(isAdmin: true)
        );
    }
}
```

Now that we've done this, we can also update our `AllUserController` to make use of the new service class. This would result in our controller looking something like this:

```php
use App\Services\UserService;
use Illuminate\Http\JsonResponse;

class AllUserController extends Controller
{
    public function index(
        UserService $userService
    ): JsonResponse {
        $nonAdminUsers = $userService->users(isAdmin: false);
        $adminUsers = $userService->users(isAdmin: true);

        return response()->json([
            'users' => $nonAdminUsers,
            'admin_users' => $adminUsers,
        ]);
    }
}
```

By extracting the code out into the service class, we've managed to stop the controller from relying on the other controllers.

Finding Logic and Queries in Blade Views

Another common issue that I've come across when auditing projects is finding business logic or database queries performed in Blade views. The logic is usually added using the `@php` Blade directive, or using PHP opening tags such as `<?php`, `<?=`, and `<?`.

When asking developers why they used this approach, responses often indicated a lack of experience with the Laravel ecosystem or general MVC (Model View Controller) concepts. Most of the developers mention that they use this approach because they are used to placing PHP at the top of their view files.

For example, imagine that you have a `users/index.blade.php` file that displays a list of users and the count of each of their published blog posts in the application. Instead of fetching and formatting the users in the controller method and then passing them to the view, the Blade file may look something like this:

```
@php
    $users = User::where('is_admin', false)
        ->select()
        ->withCount(['posts' => function ($query) {
            $query->where('published_at', '<=', now());
        }])
        ->get();
@endphp

<table>
    <tr>
        <th>Name</th>
        <th>Posts</th>
    </tr>

    @foreach($users as $user)
        <tr>
            <td>$user->name</td>
            <td>$user->posts_count</td>
        </tr>
    @endforeach
</table>
```

As we can see, the users are being fetched from the database by a query within the Blade view rather than being passed to the view from the controller method. This violates MVC principles and breaks the separation of concerns due to the fact that the Blade file is only intended for rendering data that has already been fetched.

A drawback of keeping the query within the Blade view is that it can make it difficult to track down logic and queries when auditing or refactoring code.

This issue is usually easy to solve. Depending on the complexity of the code, it just requires the logic or query to be extracted into the controller invoking the view. If we were to assume that we have a `UserController`, it may look something like this:

```php
use App\Models\User;
use Illuminate\Contracts\View;

class UserController extends Controller
{
    public function index(): View
    {
        $users = User::where('is_admin', false)
            ->select()
            ->withCount(['posts' => function ($query) {
                $query->where('published_at', '<=', now());
            }])
            ->get();

        return view('users.index', [
            'users' => $users,
        ])
    }
}
```

We've added the query to fetch the users into the controller method and then passed the result to the view as a `users` field. This means that we can then remove our `@php` block from the top of our Blade file so that it now looks like this:

```
<table>
    <tr>
        <th>Name</th>
        <th>Posts</th>
    </tr>

    @foreach($users as $user)
        <tr>
            <td>$user->name</td>
            <td>$user->posts_count</td>
        </tr>
    @endforeach
</table>
```

This results in a much cleaner looking view, and we can have confidence that the data needed to render the view is ready for us before returning it.

It's important to remember that this does not always mean that using `@php` is a bad thing. It has some valid use cases (such as keeping track of counters or toggles when iterating through items and displaying them).

Finding Hard-Coded Credentials

When you're manually auditing your code, it's important to look for any hard-coded credentials in your application. Hard coded credentials can include API keys, email addresses, usernames, and passwords.

Depending on the types of credentials and where they are used, keeping hard-coded credentials in your code can lead to serious security risks.

Hard-Coded Emails

Let's look at a basic, but very common example. A lot of applications I've worked on have Laravel Telescope installed for use as a debugging tool. To directly quote the Laravel documentation, Telescope "provides insights into the requests coming into your application, exceptions, log entries, database queries, queued jobs, mail, notifications, cache operations, scheduled tasks, variable dumps, and more". Ask yourself this: would you feel comfortable giving any of this data to a malicious attacker or a disgruntled ex-employee?

When you're setting up Telescope, you have the option to define some extra authorisation so that you (and your team) can access Telescope in non-local environments, such as "staging" and "production". The quickest and most common way that I've seen the authorisation set up is like so:

```php
/**
 * Register the Telescope gate.
 *
 * This gate determines who can access Telescope in
 * non-local environments.
 *
 * @return void
 */
protected function gate()
{
    Gate::define('viewTelescope', function ($user) {
        return in_array($user->email, [
            'mail@example.com',
            'joe.bloggs@example.com',
        ]);
    });
}
```

This code is defining a new "gate" that allows users to access Telescope in a non-local environment if their email address is either "mail@example.com" or "joe.bloggs@example.com". There are a few problems with this approach.

First of all, imagine that you have an employee on your team (in this case, "joe.bloggs@example.com") and that you have to let them go from your company so they'll no longer be employed by you. If they feel disgruntled by the fact that they're no longer employed by you, they still have access to Telescope and could use this maliciously. This would give them access to your users' data and other metrics from your application that could be used to compromise your system or cause issues. Additionally, they could also pass on their credentials to another malicious user (potentially for money) who could do something similar.

Hard-coding a list of users like this also means that each time employees leave your team, you'll need to release a new code update that removes their emails from the gate. Although this could be straightforward, this might not always be easy depending on the type of system that you're managing.

The easiest way to resolve this issue would be to lift the emails out into a config variable. This way they can be changed without needing to do a code release and it can be easily changed within minutes.

To do this, we could create a new field in our `.env` like so:

```
TELESCOPE_ALLOWED_EMAILS="mail@example.com,joe.bloggs@example.com"
```

And then we could have a new config field in our `config/telescope.php` file:

```
return [

    // …

    'allowed_emails' => env('TELESCOPE_ALLOWED_EMAILS', ''),

    // …

];
```

And then in our service provider, we could update the code to look something like this:

```php
/**
 * Register the Telescope gate.
 *
 * This gate determines who can access Telescope
 * in non-local environments.
 *
 * @return void
 */
protected function gate()
{
    Gate::define('viewTelescope', function ($user) {
        return in_array(
            $user->email,
            explode(',', config('telescope.allowed_emails')),
        );
    });
}
```

Now that we've done that, we've removed the hard-coded email from our code and can provide and revoke access just by changing the app's config.

Hard-Coded API Keys

Another common issue that I see is hard-coded API keys, whether they be in app code, as config defaults, or in an `.env.example` file.

For example, let's imagine that your Laravel app interacts with the Mailgun API in one of your classes. To make the requests to the API, you'll need to have a Mailgun API key. On some projects I've worked on, this key was hard-coded as a const or property at the top of the code using it.

As is the issue with hard-coded emails, if we ever needed to change that key, we would need to release a new version of our code. This could be problematic if you found that the API key had been compromised and needed to be changed instantly. It could also cause issues if you wanted to use a new API key with different abilities/scopes than the key that you already have.

Another major downside to including API keys in your code is that it likely means that the keys are committed in your version control. This adds an attack vector that a malicious attacker could use to compromise your system. Rather than needing to get access to your server (and maybe server management system, such as Laravel Forge) to get access to your API keys, they could also try to compromise your version control accounts (like GitHub, Gitlab, Bitbucket, etc).

Say you have five developers on your team with access to the project on GitHub. That's five new potential attack vectors that someone could use to access your keys. Of course, strong passwords and two-factor authentication can reduce the chance of this happening. But how confident are you that your team members are all following these security practices? It's important that you remove any hard-coded keys from your code.

The easiest resolution to such problems is very similar to the fix for the hard-coded emails. The best way to fix this would be to extract the fields out into config fields in your `.env` file. If you do find any keys that have been hard-coded, you might want to rotate them and create new keys when you extract them to your `.env`. Your old keys will still be present in your project's git history, so this can give you confidence that if the version control was ever compromised, you wouldn't still be as vulnerable.

When you're creating new API keys in the future, always try to follow the "principle of least privilege" — if the service allows it. The principle of least privilege states that a subject should be given only those privileges needed for it to complete its task. In the context of API keys, this means creating API keys with the lowest possible scopes/abilities needed. For example, if you're interacting with an API and only ever need to read data, then create an API key that only has the ability to read data. Don't create an API key that has the ability to write data. This adds extra security and means that if the API key is ever compromised, it will reduce the amount of things an attacker can do with it. I know it can sometimes be tempting to allow all abilities when creating an API key because it can make future development easier, but make sure not to do it. If you need the extra abilities in the future, you can create a new API key and drop it into your `.env` file easily.

Check Open Package Routes

A mistake I've seen in several projects are routes that are registered by packages and that are unknowingly left "open".

As mentioned earlier, packages like Telescope can register routes that can potentially be accessed on live systems. By default, this package only allows access if your `APP_ENV` is set to `local`. However, it's possible that you might have other packages installed in your project that don't follow this same approach.

Of course, some of the packages that are registering routes might not create any potential vulnerability. For example, the package's route might provide something as simple as a redirect. On the other hand, the package might be registering a route that creates a possible vulnerability for a malicious user to exploit, so it's important to be aware of the different routes that are currently registered.

In the past, I have been onboarded onto projects that accidentally left a package's route wide open on production systems to access the health checks for a server. Although this route didn't necessarily give away any credentials, it gave an overview of the application's infrastructure and the third-party services that were being used. This information could be used to curate an attack against a system.

One approach you could take to find the routes registered by packages would be to look at the repositories and documentation online. However, this can be extremely time-consuming, and you might miss a part in the documentation that states that a particular route is registered. As well, it's also likely that if any routes have been registered by the package's dependencies, they won't be documented.

In my opinion, the best way to check for any registered routes is to run the `php artisan route:list` command. Depending on the size of your project, this list could be quite large, but it will be able to provide you with a list of routes that Laravel is currently aware of and has registered.

If your project is running at least Laravel 9.15.0, you can use the `--only-vendor` option like so: `php artisan route:list --only-vendor`. This will result in only the routes being displayed that have been registered by packages included in your project. This can make it even easier to spot any routes you might not be aware of.

After you've found the routes, you can check each one to see whether they are "open" and need some sort of protection added to them. If you do spot any, make sure to note them and do an investigation into this. Ask yourself questions such as:

- Does this route need any authentication or authorisation added to it?
- Could any data have been compromised if anyone accessed this route in our live systems?
- Is there a possible attack vector against our system from this route?
- Can we remove this route from our application?

Reviewing Project Documentation

One step that you can take in your manual audit process is to review your project's documentation. Although this isn't necessarily related directly to the code, this can be a very important part of a project.

Up-to-date documentation can make it much easier for new developers to be onboarded onto a project. For example, say a developer has joined your team. If you were to hand them access to the project's repository on GitHub, would they be able to set up the project locally without any help?

There will likely be questions when setting up projects for the first time but by reducing ambiguity in the process, a developer can get set up faster and can start contributing sooner. This documentation is also likely to be useful for existing developers.

To review this documentation, I would start by checking over the existing documentation to see if it includes the sections listed further below. There isn't really a gold standard for writing your documentation, so I would advise writing it in a way that works well for your team, and using a system that is easy to keep current.

Some companies choose to write their project's documentation within the `README.md` file for the repository. This means that whenever you're working on the project, you also have a local copy of the documentation. This is quite handy because it can reduce the need for you to leave your IDE. I've also seen some other projects that use Notion, GitHub wiki, or Gitbook to write their documentation. In my opinion, the location doesn't matter as long as it is informative, up-to-date, and secure.

Your documentation could contain things like:

Local development instructions

What environment are they expected to use? Docker? Valet? Homestead? Telling them what environment they're using reduces the chances of one developer having an environment different from what the other developers use. We'll cover this topic later. These instructions are also useful for telling them about any API keys that they might need to configure, or any commands that they need to run.

Information about the production environment

What sort of infrastructure is being used to host the project? Is it running on an AWS EC2? Is it a serverless-based application that's using Laravel Vapor? Is Laravel Octane being used? What type of database are you using? In some cases the specifics of this don't matter much (and could compromise security if seen by the wrong people). However, being aware of the infrastructure being used can help avoid bugs. For example, if a developer has been using PostgreSQL-specific syntax to write some of their queries locally, this could cause issues if the production environment uses MySQL. We'll further cover this topic later when we discuss how to set up a sensible development environment.

Code style

What code style should the developers use? You might be using a common standard, such as PSR-12, or you might have your own company style guide. Making this obvious from the start can reduce any friction when it comes to contributing to the project. Although this process can be fully automated using a tool like StyleCI (which we have discussed previously), having awareness of the style can be useful.

Code standards

You may also want to specify any standards that you expect the developers to follow. As a simple example, you may want to enforce that any usages of `in_array` should use `true` as the third parameter to enable strict type-checking. Letting developers know about this early on can also help reduce friction and allow them to contribute to the project faster without needing to make changes after code reviewing.

Frequently asked questions

As your project and team grow, you might find questions commonly asked about the project by new developers. Having a small, bullet-pointed "FAQs" section addressing some of these questions can be beneficial. It might explain a specific feature or part of the system. Or point out some common "gotchas". By providing a knowledge base, it can make it easier for developers to get an understanding of the system without needing to trawl through huge amounts of code or ask questions. Please remember — this is not a replacement for answering questions from your team. I'm a firm believer that, as developers, we should always be asking questions and learning from each other. A FAQs section merely allows developers to solve some of their potential problems without asking someone else.

Section 2: Testing

What's Covered?

In this section, we're going to learn how to approach adding tests to your Laravel application. We'll take a look at the benefits of writing tests for your code, different ways you can structure tests, a common strategy called "Arrange-Act-Assert", and reducing duplicated test code with data providers.

Following this, we'll cover how I determine which parts of the system to prioritise writing tests for, and how to avoid "testing fatigue". We'll then explore how to use Laravel Dusk to write tests for your user interface.

After that, we'll look at how we can build a CI (continuous integration) workflow using GitHub Actions to improve the quality of our code in the future using:

- Feature and unit tests
- Dusk tests
- Larastan
- StyleCI

By the end of this section, you should have a better understanding of testing and how to ensure the quality of future code.

Planning Your Testing Strategy

Before you start writing tests, it's important to have a clear strategy. This will make it easier to build your new test suite, and will also make it easier to cooperate with your peers if working as part of a team.

As we'll discuss later, when I start writing tests, I like to prioritise writing tests for the mission-critical parts of the system. After writing those tests, I like to write tests for the rest of the codebase. If I still have available time, I also like to add Dusk tests to ensure the user interface works as expected.

When you start writing tests, it's possible that you'll come across bugs in the code that you might have missed during your audit. If this happens, you need to have a plan for how to tackle bugs. More than likely, you'll probably want to fix a bug as soon as you have a test that proves the bug exists. In the "Fixing" section, we'll explore this concept and look at how we can use test-driven development to fix bugs.

When you have finished writing a test and fixing a bug, make sure to use "atomic commits" when committing these changes to your version control. "Atomic commits" refers to the idea of commits only being related to a single change or feature. For example, say you spend an afternoon writing tests and fixing bugs in your project. You'll want to make sure to avoid bundling all of those bug fixes into a single "Fixed bugs" commit at the end of the day. Instead, break the changes down into smaller commits such as "Fixed a bug that was using the incorrect auth guard". Splitting your commits out into more granular commits provides more context to what was actually changed. It also makes it easier to look through the version history when trying to track down causes of bugs in the future. By following this, you can make future auditing processes easier for yourself and other team members.

The Benefits of Writing Tests

Tests are often thought of as 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 quickly, maybe the tests can take a back seat. But the reality is that writing tests before the code is released into production is almost 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? I've certainly 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 being rejected or bugs being released into production. By writing tests, you can spot bugs before you commit your work and have more confidence when you release code to a live server.

Making Future Work and Refactoring Easier

Imagine you need to refactor one of the classes that provide a core feature in your Laravel app. Or you need to add some new code to that class to extend the functionality. Without tests, how will you know that changing or adding code isn't going to break the existing functionality? A lot of manual testing would be required, and there's no easy way to check quickly, easily, and consistently. When you write tests alongside 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 correctly. You can also keep adding tests every time you add new code so that you can be sure your additions run as expected.

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 difficult to write tests for. It was hard to do things like mock classes, use test fakes, prevent third-party API calls, and make some assertions. To write code in 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. I genuinely think that learning to write your code in a testable way can greatly improve your confidence in your projects.

Tests-As-Documentation

Your tests can act as a form of documentation and give hints that can help a developer understand what a method does. I often check the tests for methods when I need to see an example of the data structure that is returned when a method is provided with different inputs. For example, I might want to check what JSON is returned from an API route if I pass a specific parameter in the request body.

Prove That Bugs Exist

In the event that a bug is reported by a user, you can write tests to mimic the actions of the user. If the test is written properly, you should be able to reproduce the same error with your test. This means that when you implement a fix for the bug, you'll be able to use your test to assert that the fix solves the issue.

Structuring Your Tests

When writing your tests, it's important that you define a structure before starting. This can ensure consistency across your codebase and make it easier to write new tests and maintain the existing ones when needed.

Directory Structure

There are different directory structures you can use for writing tests, and every project that I've worked on has always been different. However, I've found that using a directory structure that matches the application code is a useful approach. For example, imagine we want to write tests for the following controller methods:

- `app/Http/Controllers/UserController@index`
- `app/Http/Controllers/UserController@store`
- `app/Http/Controllers/UserController@update`

I would place tests for these methods within the following files:

- `tests/Feature/Http/Controllers/UserController/IndexTest.php`
- `tests/Feature/Http/Controllers/UserController/StoreTest.php`
- `tests/Feature/Http/Controllers/UserController/UpdateTest.php`

This approach can make it easy to track down where the tests for a specific method are found. For example, I know that if I make changes to the `app/Http/Controllers/UserController@store` method and want to write a new test, I can check in the `tests/Feature/Http/Controllers/UserController/StoreTest.php` file (or create one if it doesn't already exist).

Of course, you don't need to follow this same structure, and you can choose whatever works best for you and your team. But I do think it's important that you and your team decide on a suitable structure that is consistent.

Choosing What To Test

If you're new to writing tests, you may find that one of the most difficult parts of the process is figuring out what to actually test.

When I'm writing a test, I try to break the method's code down and identify specific pieces that I can assert against. For example, we might want to assert against:

- The returned output of a method based on the input.
- Whether a user was authenticated or logged out.
- Whether any exceptions were thrown.
- Whether any rows in the database changed.
- Whether any jobs were added to the queue.
- Whether any events were dispatched.
- Whether any mail or notifications were sent (or queued).
- Whether any external HTTP calls were made.

To put this into context, take this basic example method:

```php
use App\DataTransferObjects\UserData;
use App\Models\User;
use App\Services\MyThirdPartyService;

public function store(UserData $userData): User
{
    return DB::transaction(function () use ($user, $userData) {
        $user = User::create([
            'email' => $userData->email,
        ]);

        $this->assignRole('admin');

        $service = app(MyThirdPartyService::class);

        if (! $service->createUser($user)) {
            throw new \Exception('User could not be created');
        }

        return $user;
    });
}
```

In the example above, we are creating a user and assigning them a role. We're then making a request (via a service class) to a third-party API to create the user in an external system. From this method we can identify 5 things that we can test:

- Test that the user is stored in the database.
- Test that the user is assigned the 'admin' role.
- Test that the user can be created in the third-party system.
- Test that if the user can't be created in the third-party system that an exception is thrown.
- Test that the method returns the `User` if everything is successful.

Although we have 5 things to test, we wouldn't necessarily need to write 5 tests. Instead, our test cases might look like so:

```
/** @test */
public function user_can_be_created()
{
    // 1. Assert that the user is created in the database.
    // 2. Assert that the user is assigned the 'admin' role.
    // 3. Assert that the user is created in the third-party
    //    system.
    // 4. Assert that the method returns the created model.
}

/** @test */
public function error_is_thrown_if_user_cannot_be_created()
{
    // 1. Assert that an exception is thrown if the user
    //    cannot be created in the third-party system.
}
```

This means we can write two tests that cover multiple assertions.

Test Structure

When writing tests, it helps to follow a structure that is consistent and understandable. For this reason, I like to follow the "Arrange-Act-Assert" pattern for all of my tests. The Arrange-Act-Assert pattern encourages you to split your test into 3 distinct sections:

- **Arrange** - Make all the preparations for your test to start. This might mean using a model factory to add data to your database. It might include you creating any mocks, test doubles, or dummy files.
- **Act** - Act on the target behaviour. This might mean making an HTTP call to a controller or calling a method in a class.
- **Assert** - Assert against the results of the target and the expected outcomes. For example, assert that the correct data was returned from the tested method or that a new row was created or updated in the database.

Following a pattern like this can make your tests predictable and help other members of your team understand your tests more quickly. I also believe that it encourages a testing mindset when you're writing your application code. For myself, it usually makes me think when I'm adding a new feature "What would I need to 'arrange' to test this? What would I want to 'assert' against?". If those questions are difficult to answer, it's sometimes a hint that your approach may be too complex and could be simplified or split up into smaller blocks of code.

Let's look at a basic example and see how we can use Arrange-Act-Assert to structure our test. Imagine we have this controller method that returns a view along with published blog posts:

```php
use App\Models\Post;

class BlogController extends Controller
{
    public function index()
    {
        $posts = Post::query()
            ->where('published_at', '<=', now())
            ->orderByDesc('published_at')
            ->get();

        return view('blog.index', $posts);
    }
}
```

If we wanted to write a test for this method, to "arrange" the test we'd first need to:

- Create some published blog posts in our database.
- Create some unpublished blog posts in our database.

We'd then want to "act" on the test by:

- Making an HTTP call to the controller method (which we'll assume can be reached using a route with the name `blog.index`).

We can then "assert" in our test by asserting that:

- The correct view is returned.
- The published blog posts are returned in order of their `published_at` date.
- The unpublished blog posts are not returned.

As a result, our test may look something like so:

```php
namespace Tests\Feature\Controllers\BlogController;

use App\Post;
use Illuminate\Database\Eloquent\Collection;
use Illuminate\Database\Eloquent\Factories\Sequence;
use Illuminate\Foundation\Testing\LazilyRefreshDatabase;
use Tests\TestCase;

class IndexTest extends TestCase
{
    use LazilyRefreshDatabase;

    /** @test */
    public function view_is_returned(): void
    {
        // ARRANGE...

        // Create a post that has not been published yet.
        // This should not be included in the response.
         Post::factory()
            ->create(['published_at' => now()->addDay()]);

        // Create 3 posts that have been published.
        // These should be included in the response.
        $posts = Post::factory()
            ->count(3)
            ->state(new Sequence(
                ['published_at' => now()->subYear()],
                ['published_at' => now()->subMinute()],
                ['published_at' => now()->subSecond()],
            ))
            ->create();

        // ACT...
        $response = $this->get(route('blog.index'));

        // ASSERT...
        $response->assertOk()
            ->assertViewIs('blog.index')
            ->assertViewHas(
                'posts',
                function (Collection $viewPosts) use ($posts) {
                    // Assert that only the published posts were
                    // returned and that they were in the
                    // correct order.
                    $expectedPosts = [
                        $posts[2]->id,
                        $posts[1]->id,
                        $posts[0]->id,
                    ];

                    return $viewPosts->pluck('id')->toArray()
                        === $expectedPosts;
                }
            );
    }
}
```

It's worth noting that in the example above the "act" and "assert" sections could have been linked together because the `get` method is chainable. However, for the purpose of the example and to distinguish between the two sections, they were split out.

Data Providers

You may find when writing tests that you want to test the same method but with many different inputs or scenarios. This can lead to many tests that are mostly identical; the maintenance of these tests can be cumbersome and tedious.

To reduce this repetitive code, you can use "data providers". Data providers allow you to use different data and input in your tests without needing to duplicate the test code itself.

Let's look at an example of a function that we could test using data providers. The following code is a simple helper function that can calculate the time (in seconds) it takes for a person to read a string passed as an argument:

```
function calculate_read_time(string $words): int
{
    return (int) ceil(
        (Str::wordCount($words) / 265) * 60
    );
}
```

To write a test for this method using a data provider, our test may look like this:

```php
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

class CalculateReadTimeTest extends TestCase
{
    use WithFaker;

    /**
     * @test
     * @dataProvider readTimeDataProvider
     */
    public function correct_read_time_is_returned(
        int $wordCount,
        int $expectedResult
    ): void {
        $words = $this->faker->words($wordCount, true);

        self::assertEquals(
            $expectedResult,
            calculate_read_time($words),
        );
    }

    public function readTimeDataProvider(): array
    {
        return [
            [3, 1],
            [265, 60],
            [530, 120],
            [500, 114],
            [2000, 453],
        ];
    }
}
```

Let's break down what's happening in this test.

We created a method called `readTimeDataProvider` that returns an array of arrays. Each item within the array will have the individual items passed as arguments to our test. For example, on the first run, the test will be passed `3` as the first argument and `1` and the second argument. On the second run, the test will be passed `265` as the first argument and `60` as the second argument, and so on.

In this test, we want the first argument to be `$wordCount` (the number of words in the string that we are passing to the function). We also intend the second argument to be the `$expectedResult` (the read time result returned from the function).

We specified in our test's docblock (using `@dataProvider readTimeDataProvider`) that the test will use the `readTimeDataProvider`.

Due to the data provider having five elements in its array, when we run this test it will be run five times — each time with the different data from the provider's next element. This is a great way of reducing duplicate tests and testing code under different conditions without needing to write a separate test case for each input and output pair.

In tests such as the preceding example, you can simplify the code by removing the data provider method and providing the data in the test's docblock using `@testWith`. If we want to update our test to use this approach, our test file could look like so:

```php
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

class CalculateReadTimeTest extends TestCase
{
    use WithFaker;

    /**
     * @test
     * @testWith [3, 1]
     *           [265, 60]
     *           [530, 120]
     *           [500, 114]
     *           [2000, 453]
     */
    public function correct_read_time_is_returned(
        int $wordCount,
        int $expectedResult
    ): void {
        $words = $this->faker->words($wordCount, true);

        self::assertEquals(
            $expectedResult,
            calculate_read_time($words),
        );
    }
}
```

As you can see, the test file looks cleaner and easier to understand. However, it's important that you use the `@testWith` approach wisely and only if:

- You aren't using the same data in other tests. This will increase duplication and test maintenance. In this case, it would be easier to use a data provider method instead.
- You are only providing simple data for your test. In our test above, we're only passing

two integers per test. But if we were wanting to pass more arguments or more complex data (such as classes), this approach would hinder maintenance. In that case, it would be easier to extract the data into a data provider instead.

Writing the Tests

Prioritising Mission-Critical Tests First

To start testing your project, you'll first want to identify the mission-critical parts of your code. These pieces of code would typically be for features that provide the most benefit to your users and majorly affect your users, or cause a loss of revenue if the feature was broken. For example, on an e-commerce site, you might identify the "basket" and "checkout" features as mission-critical parts of your code. This is because your users wouldn't be able to perform their intended task of buying a product if the feature was broken or unavailable. Thus, you would want to prioritise adding test coverage for this feature rather than writing tests to ensure that the user's avatar is displayed properly on the page.

During your audit process, you may have also identified vulnerabilities in your code that could be exploited. Although these vulnerabilities may not be part of mission-critical features, I'd usually bundle them in with the mission-critical feature tests as being a priority.

Writing the Rest of the Tests

After you've started building up a test suite that covers the mission-critical parts of your project, you can write tests for other parts of the system.

To do this, I like to systematically work through the codebase and add tests that would be quick to write and provide extra confidence. I usually find that the following types of code are "easy wins" and that they are typically easy to write tests for:

- **`index`, `show`, and `edit` methods in controllers** - These typically are easy because they don't usually alter data and the tests can assert that correct views and responses are returned.
- **Helper functions** - If your project includes helper functions (usually found in a file such as `app/helpers.php`), these usually contain simple code where the output of the method can be checked easily.
- **Model methods** - A lot of projects have models that include helper methods to make the code more expressive. For example, if we had a blog project, there might be `publish` and `unpublish` methods on a `BlogPost` model. These methods are usually simple, only updating a row in the database, and are easy to write tests for.

Benefits of Writing the Easy Tests First

A key benefit of writing the easiest tests first is that you can write them faster than tests for more difficult methods. As a result, you can write more tests in a shorter space of time and grow your test suite faster.

It's also a great way of practising writing tests and getting more comfortable with the process. This will make the more complex tests easier to write because you'll have a better understanding of writing tests and the different assertions that are available.

However, it's still important to prioritise your tests. You might have specific parts of your system that you would rather have test coverage for first. So you'll need to decide on a project-by-project basis how you want to tackle adding tests to provide the most confidence in the quickest way.

Preventing Test Fatigue

Writing tests can be a hugely rewarding and fulfilling activity. I find it extremely satisfying when you start writing tests for a project that doesn't have a test suite and then have hundreds (or thousands) of passing tests weeks later. But it can be very tedious if you tackle too much at once.

In some applications I've worked on, it was my sole job to add tests for projects that didn't have any tests at all. On one of the projects, I spent a whole month (8 hours a day, 5 days a week) and added just over 2,000 tests! Although the outcome of doing this was beneficial to me and the rest of the team, I experienced some fatigue. Writing many tests in one go can become tedious, dull, and repetitive. This can lead to you cutting corners and writing bad tests that don't provide any value purely so that you can say you have test coverage for a particular method or feature. Thus, it's important to pace yourself and split this task into smaller chunks of work if you're starting to feel test fatigue.

To avoid this, an approach I use is to write a test if I have a few spare minutes in the day. For example, let's say that I intend to finish working at 5 pm one day, but I come to a natural end working on a feature at 4:30 pm. In this situation, I would spend the final 30 minutes (between 4:30 pm and 5 pm) writing tests for an untested piece of code. Alternatively, I might actually schedule some time at the end of each day for this. By doing this, you can write a small number of tests which, over time, will become a larger test suite. This follows the strategy of "aggregation of marginal gains" — small yet significant improvements can add up to monumental results.

Although this process might sound slow, remember that it's better to have some tests than no tests. As you write more tests, you'll feel more comfortable writing them and find you can write them more quickly. This means that when you're writing new code in the future, you'll already be in the habit of adding or updating tests to cover changes.

It's also important to remember that your tests don't need to be perfect at first. For example, if you have an API route that returns a complex JSON response, you might just want to write a quick test to assert that you get an HTTP 200 response. Having this test would instantly provide value. This test could then be improved in the future, as you increase your knowledge of testing, to make more assertions such as on the structure of the JSON or the actual values that are returned.

Testing Your UI with Laravel Dusk

When building a web application, you might benefit from writing UI (user interface) tests. UI tests can simulate user actions on your site or application. For example, if you have an application that has a checkout process for buying products, you might want to write a test that asserts a user can successfully pay for a product.

Although most of the billing process will be testable through traditional PHPUnit tests, a UI test suite can help add confidence that the UI is also working as expected. As an example, you might want to check that some JavaScript on the page correctly updates the view, or that some validation error messages are displayed correctly.

In past projects, I've found that writing UI tests can sometimes be a slow, tedious task; particularly because the tests are usually slow to run. After speaking to a number of developers, they stated that this was their main reason for usually avoiding writing them. This can become particularly troublesome if you're working on a relatively new project that has a UI that is still changing at a fast pace. Thus, UI tests work well for projects that are "stable" and unlikely to change their UI much.

It's worth noting that you don't need UI tests for every part of your UI. It's best to focus on the mission-critical parts of your system. For example, in my opinion, it would be more beneficial to write a test to assert that a user can buy a product and successfully pay for it, rather than a test to assert that the user's avatar is displayed properly on their profile page. Of course, both of these tests would provide value and confidence, but it's important to prioritise your tests where possible to gain confidence in your code quickly, especially when you have time and cost constraints.

To write UI tests for your Laravel application, you can use tools like Cypress or Laravel Dusk. Both of these tools offer similar functionality, but use different languages to write the tests; Cypress uses JavaScript, and Laravel Dusk uses PHP.

For the rest of this chapter, we're going to step through how you can write UI tests using Laravel Dusk.

Installation

To get started with using Laravel Dusk, we'll install it via Composer using the following command:

```
composer require --dev laravel/dusk
```

After doing this, we'll run the Artisan command to prepare our project to use Dusk:

```
php artisan dusk:install
```

The above command should have created a `tests/Browser` folder in your project with some additional folders and example files. That's it. Dusk should now be ready to use.

Testing a Simple Form

Dusk provides an easy-to-use API for interacting with views. Let's look at how to test that a user can log in to the application correctly.

Imagine we have the following Blade view that is accessed via the `/login` route in our project:

```
<form action="{{ route('login') }}" method="post">
    @csrf

    <label for="email">Email</label>
    <input type="email" name="email" id="email">

    <label for="password">Password</label>
    <input type="password" name="password" id="password">

    <button type="submit">
        Login
    </button>
</form>
```

We can create a new Dusk test class using the `dusk:make` Artisan command like so:

```
php artisan dusk:make LoginTest
```

Running the preceding command would create a new `tests/Browser/LoginTest.php` file. We can then create a new `user_can_login` test inside this class similar to the following:

```
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;
use Tests\DuskTestCase;

class LoginTest extends DuskTestCase
{
    use DatabaseMigrations;

    /** @test */
    public function user_can_login()
    {
        $user = User::factory()->create([
            'email' => 'mail@ashallendesign.co.uk',
            'password' => Hash::make('password'),
        ]);

        $this->browse(function (Browser $browser) use ($user) {
            $browser->visit('/login')
                ->type('email', $user->email)
                ->type('password', 'password')
                ->press('Login')
                ->assertPathIs('/dashboard');
        });
    }
}
```

To understand this test, let's look at what is happening.

The test is:

1. Creating a new `User` model using a factory.
2. Navigating to the `/login` route.
3. Finding a field on the page with the ID of `email` and typing the user's email into it.
4. Finding a field on the page with the ID of `password` and typing the user's password into it.
5. Pressing the `Login` button.
6. Asserting that we are redirected to a new page with the URL of `/dashboard`.

To run this test, we could then run the following command:

```
php artisan dusk
```

The above command will also output the test results to the command line so that you can see which tests passed and which failed.

Dusk Pages and Selectors

The Dusk test we've shown is only a simple test that interacts with a single form on a page. However, you might find that you want to start adding tests for more complex pages and features.

Using the same approach shown before, the test suite could become messy and complicated. The tests could also become fragile and break when there is any change in the user interface.

This is because you select and interact with the views in the test via DOM elements that you locate using IDs, classes, and data attributes. So when writing tests, it can sometimes be tempting to select an element on a page just using its CSS classes. But this can mean that a change in the page's style can break the tests.

For example, using selectors such as `:nth-child(1)`, `:first-child`, or `:last-child` can cause errors if the order of DOM elements on the page changes. Similarly, if you're using a CSS framework like Tailwind, the CSS selectors might not provide a lot of context to what a DOM element is purely through using its applied CSS classes. For example, we might have the following HTML that creates a button and styles it using Tailwind:

```
<button type="button"
        class="inline-flex items-center px-2.5 py-1.5 border
border-transparent text-xs font-medium rounded shadow-sm text-
white bg-indigo-600 hover:bg-indigo-700 focus:outline-none
focus:ring-2 focus:ring-offset-2 focus:ring-indigo-500"
>
    Save
</button>
```

In our Dusk test, we might not want to use `button` as the selector because there could be multiple buttons on the page. We also might not want to use the `Save` button text as a selector because there may be multiple buttons with the same text on the page. We also wouldn't want to use the applied CSS class because if any of the style changes (either by a class being added or removed), Dusk wouldn't be able to locate the element.

To solve this issue and make our tests more robust, we can use Dusk "pages" and "selectors".

Dusk pages allow us to group common actions and selectors into a centralised class in our test suite. I think of them as a middleman between my actual test-suite code and the user interfaces they're interacting with.

To understand the importance of pages and selectors and how to use them in our own tests, let's look at an example. In the Blade example below, we are displaying a table listing users in the system. Above the table is a button that can be used to export the users' details to a CSV file. We'll also make the assumption that when the 'Export Users' button is pressed, a modal appears on the screen to confirm that the export has been triggered. We'll also assume the `id` and `class` attributes added to the `div` and `button` elements have been placed there purely as a way of targeting the DOM elements in the tests and that they have no other purpose throughout the application.

Our Blade code might look something like this:

```
<div id="users-table-container">
    <div>
        <button type="button" class="export-users">
            Export Users
        </button>
    </div>

    <table>
        <tr>
            <th>Name</th>
            <th>Email</th>
            <th>Action</th>
        </tr>

        @foreach($users as $user)
            <tr>
                <td>{{ $user->name }}</td>
                <td>{{ $user->email }}</td>
                <td>
                    <a href="{{ route('users.show', $user) }}">
                        View
                    </a>
                    <button type="button">
                        Export Details
                    </button>
                </td>
            </tr>
        @endforeach
    </table>
</div>
```

A test without using pages or selectors might look something like so:

```php
/** @test */
public function users_export_can_be_triggered()
{
    $authUser = User::factory()->create();

    User::factory()
        ->count(2)
        ->create();

    $this->browse(function (Browser $browser) use ($authUser) {
        $browser->loginAs($authUser)
            ->visitRoute('users.index')
            ->click('#users-table-container .export-users')
            ->waitForText('Export started')
            ->assertSee('Export started');
    });
}
```

As you can see, in the test we're logging in as a user, navigating to the users index page, clicking the "Export Users" button and then waiting for the confirmation text. In this particular test, we've made use of the `id` and `class` attributes of the DOM elements to locate the button and press it. Although this is valid and would work, we've introduced an issue into our Blade view just in order to get our tests running. We've added `id` and `class` attributes to our HTML purely for selecting elements in Dusk. If a developer found this HTML, they might assume those identifiers are used to apply styles or interactivity using CSS or JavaScript. This means we've introduced some ambiguity and that the selectors' purposes aren't clear. To rectify this, we can replace the `id` and `class` attributes with `dusk` attributes like so: `[dusk="users-table-container"]`.

We can start by replacing the unclear `id` and `class` attributes with `dusk` attributes that signify to the reader they are being used for Dusk testing:

```
<div dusk="users-table-container">
    <div>
        <button type="button" dusk="export-button">
            Export Users
        </button>
    </div>

    <table>
        <tr>
            <th>Name</th>
            <th>Email</th>
            <th>Action</th>
        </tr>

        @foreach($users as $user)
            <tr>
                <td>{{ $user->name }}</td>
                <td>{{ $user->email }}</td>
                <td>
                    <a href="{{ route('users.show', $user) }}">
                        View
                    </a>
                    <button type="button">
                        Export Details
                    </button>
                </td>
            </tr>
        @endforeach
    </table>
</div>
```

Now we can create a Dusk "page" to interact with our users page. I like to keep the structure of the pages in the test suite the same as the structure in the app code. We'll assume that our Blade view above is located at `resources/view/users/index.blade.php` and so we want our Dusk page class to be located at `tests/Browser/Pages/Users/Index.php`. To create this file, we can run the following command:

```
php artisan dusk:page Users/Index
```

This should create our new `tests/Browser/Pages/Users/Index.php` file.

Within this file, we must set the correct URL in the `url` method to be `/users` so Dusk knows where to navigate when we use this page. We'll also be able to set elements in our class to select the DOM elements during the test. Our finished class may look something like so:

```php
namespace Tests\Browser\Pages\Users;

use Laravel\Dusk\Browser;
use Laravel\Dusk\Page;

class Index extends Page
{
    /**
     * Get the URL for the page.
     *
     * @return string
     */
    public function url()
    {
        return '/users';
    }

    /**
     * Assert that the browser is on the page.
     *
     * @param  Browser  $browser
     * @return void
     */
    public function assert(Browser $browser)
    {
        $browser->assertPathIs($this->url());
    }

    /**
     * Get the element shortcuts for the page.
     *
     * @return array
     */
    public function elements()
    {
        return [
            '@usersTableContainer' => '[dusk="users-table-
container"]',
            '@exportButton' => '[dusk="export-button"]'
        ];
    }
}
```

As you can see, we've created two new elements in our `elements` method; the first being `@usersTableContainer` and the second being `@exportButton`. These are mapped to the DOM elements we updated in our Blade view to have the `dusk` attribute so we can interact with them.

We can now update our previous test to use the Dusk page selectors like so:

```php
use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;
use Tests\Browser\Pages\Users\Index;
use Tests\DuskTestCase;

/** @test */
public function users_export_can_be_triggered()
{
    $authUser = User::factory()->create();

    User::factory()
        ->count(2)
        ->create();

    $this->browse(function (Browser $browser) use ($authUser) {
        $browser->loginAs($authUser)
            ->visit(new Index())
            ->within(
                '@usersTableContainer',
                function (Browser $browser) {
                    $browser->click('@exportButton');
                }
            )
            ->waitForText('Export started')
            ->assertSee('Export started');
    });
}
```

In the above test, we've replaced the `visitRoute` method with the `visit` method and passed in a new instance of our `Users/Index` page that we created. We've then specified that we want to press the `@exportButton` (or `[dusk="export-button"]`) inside the `@usersTableContainer` (or `[dusk="users-table-container"]`) DOM element. Using the `within` method reduces the chance of us pressing another instance of `[dusk="export-button"]` if one gets added in a future feature somewhere else on the page.

We can further improve our test by making the code more expressive and making the purpose of the `within` and `click` method calls more obvious. To do this, we can move the button-clicking code to a new method we'll call `clickUserExportButton` in our `Users/Index` page class like so:

```php
use Laravel\Dusk\Browser;
use Laravel\Dusk\Page;

class Index extends Page
{
    // ...

    public function clickUserExportButton(Browser $browser): void
    {
        $browser->within(
            '@usersTableContainer',
            function (Browser $browser) {
                $browser->click('@exportButton');
            }
        );
    }
}
```

Now that we've done this, we can replace our `within` and `click` code with the new `clickUserExportButton` in our test class:

```php
/** @test */
public function users_export_can_be_triggered()
{
    $authUser = User::factory()->create();

    User::factory()
        ->count(2)
        ->create();

    $this->browse(function (Browser $browser) use ($authUser) {
        $browser->loginAs($authUser)
            ->visit(new Index())
            ->clickUserExportButton()
            ->waitForText('Export started')
            ->assertSee('Export started');
    });
}
```

As you can see, the steps the test takes are now much more obvious. This can be beneficial when working as part of a team because other developers can understand the logic behind tests more quickly. As a result, developers working on the project (including yourself) will feel more inclined to write new tests and keep existing ones up to date.

Although the test we've created above isn't overly complex, it should highlight how you can use Dusk pages and selectors. At first, you may feel they add unnecessary complexity to your test suite. However, I believe you reap the rewards of using them as your test suite grows and as you write longer, more complex tests. Using them early, no matter how simple a test may seem, is a good habit to get into and can save time in the future.

By using pages and selectors, you can create reusable classes that can be used in multiple tests. For example, if you want to write more tests for the `users.index` page, you can use the `Users/Index` page class and have access to all the existing selectors and methods. This can make future tests much faster to write. It also means that if you make changes to your user interface requiring test changes, you can make the test updates in a centralised area rather than updating each individual test.

Running Failed Tests and Groups

If you run your entire test suite and have failures, you might want to only rerun the failed tests after you've made changes.

To do this, you can use the `dusk:fails` Artisan command like so:

```
php artisan dusk:fails
```

There may also be times when you only want to run a specific group of tests; such as only tests associated with your system's authentication. This can be useful when you're working on code related to a specific feature and want to test that your changes haven't broken the user interface — without needing to run the entire test suite.

For example, we could create an `authentication` group by adding `@group authentication` to the docblock for the tests we want to include in the group, like so:

```
class LoginTest extends DuskTestCase
{
    use DatabaseMigrations;

    /**
     * @group authentication
     * @test
     */
    public function user_can_login_without_2fa()
    {
        // ...
    }

    /**
     * @group authentication
     * @test
     */
    public function user_can_login_with_2fa()
    {
        // ...
    }
}
```

You can then pass a group name to the `--group` argument of the `dusk` Artisan command to only run the tests in the given group. To run the tests in the `authentication` group, we could run the following Artisan command:

```
php artisan dusk --group=authentication
```

Creating a CI Workflow Using GitHub Actions

Once you have a test suite set up and working, you'll want to set up a CI (continuous integration) workflow that can be used to run checks against all our future code. For example, we could add a workflow that runs the following each time a new commit is pushed to our version control provider, or when a new pull request is made:

- **Test suite** - Run our PHPUnit feature and unit tests.
- **Dusk tests** - Run our Dusk tests to check our user interface.
- **Larastan** - Run Larastan to detect potential bugs.
- **StyleCI** - Run StyleCI to detect code style issues.

Running these can encourage developers to only write high-quality code that passes all the checks. Although it's impossible to guarantee that your code will never have bugs, having these checks set up can drastically reduce the likelihood of bugs being introduced in your production system if used correctly.

There are several providers you can use to run CI workflows, such as Travis CI, Circle CI, and Chipper CI (a CI platform built specifically for Laravel). Some version control providers, such as GitHub and Gitlab, also provide their own CI solutions. In this guide, we're going to be using GitHub's solution: GitHub Actions.

Due to the fact that StyleCI is a managed service and runs automatically (assuming that you have configured it correctly in your StyleCI dashboard), you won't need to create a GitHub Action to run StyleCI.

Using an `.env.ci` File

When running your tests within a CI workflow, you might want to set specific environment variables that you wouldn't typically set when running your tests locally. To handle this, you can create a new `.env.ci` file in your project's root directory. You can place all of your environment variables here, and when we start our GitHub Action, we'll copy our `.env.ci` file so it becomes our `.env` file while the tests are running.

Running the Test Suite

To create a CI workflow with GitHub Actions, we must first create a workflow that runs our PHPUnit test suite. To do this, we create a YAML file in a `.github/workflows` directory that GitHub Actions will automatically detect. We'll call this file `tests.yml`.

We'll then want our `.github/workflows/test.yml` file to look like so:

```yaml
name: Test Suite

on: [pull_request, push]

jobs:
  phpunit:
    runs-on: ubuntu-22.04
    container:
      image: kirschbaumdevelopment/laravel-test-runner:8.1

    steps:
      - uses: actions/checkout@v3
        with:
          fetch-depth: 1

      - name: Install composer dependencies
        run: |
          composer install --no-scripts

      - name: Prepare Laravel Application
        run: |
          cp .env.ci .env
          php artisan key:generate

      - name: Prepare Front End Assets
        run: |
          npm install
          npm run dev

      - name: Run Test Suite
        run: php artisan test
```

Although this file might look complicated, it's not often you need to change these files after you get them set up. So I wouldn't worry much about spending a long time understanding the syntax.

However, let's take a high-level look at what our `.github/workflows/tests.yml` file is doing.

We're first defining the name of the workflow and calling it "Test Suite". We then specified that we want the workflow to run every time someone pushes to the repository on GitHub or makes a pull request.

Following this, we've defined that our workflow contains one job called "phpunit" and that it should be run on an instance of Ubuntu 22.04 (presuming that this is the same operating system our production server runs). The job is also instructed to use a Docker container named `kirschbaumdevelopment/laravel-test-runner:8.1` which is specifically built for running tests for Laravel projects using PHP 8.1. Then an action named `actions/checkout` checks out our changes so that we can run the tests.

We then:

1. Install our PHP dependencies via Composer
2. Copy our `.env.ci` file to be our `.env` file
3. Generate a new APP_KEY for our `.env` file
4. Install any NPM dependencies
5. Build our front-end assets

After all of this is complete, we can run our tests.

For the purpose of this workflow, our tests would have been run using an in-memory SQLite database. Therefore, we would have had the following database environment variables set in our `.env.ci` file:

```
DB_CONNECTION=sqlite
DB_DATABASE=:memory:
```

Using MySQL

If we wanted to run our tests using a MySQL 8.0 database, we might want to set the following fields in our `.env.ci`:

```
DB_CONNECTION=mysql
DB_HOST=mysql
DB_PORT=3306
DB_DATABASE=testing
DB_USERNAME=root
DB_PASSWORD=password
```

We could then update our `tests.yml` file to use the `mysql` service by adding the following after the `container` but before the `steps`:

```
services:
  mysql:
    image: mysql:8.0
      env:
        MYSQL_ROOT_PASSWORD: password
        MYSQL_DATABASE: testing
      ports:
        - 33306:3306
      options: --health-cmd="mysqladmin ping" --health-
interval=10s --health-timeout=5s --health-retries=3
```

This will result in our `tests.yml` file looking like so:

```yaml
name: Test Suite

on: [pull_request, push]

jobs:
  phpunit:
    runs-on: ubuntu-22.04
    container:
      image: kirschbaumdevelopment/laravel-test-runner:8.1

    services:
      mysql:
        image: mysql:8.0
        env:
          MYSQL_ROOT_PASSWORD: password
          MYSQL_DATABASE: testing
        ports:
          - 33306:3306
        options: --health-cmd="mysqladmin ping" --health-
interval=10s --health-timeout=5s --health-retries=3

    steps:
      - uses: actions/checkout@v3
        with:
          fetch-depth: 1

      - name: Install composer dependencies
        run: |
          composer install --no-scripts

      - name: Prepare Laravel Application
        run: |
          cp .env.ci .env
          php artisan key:generate
      - name: Prepare Front End Assets
        run: |
          npm install
          npm run dev

      - name: Run Test Suite
        run: php artisan test
```

PostgreSQL

If we wanted to run our tests using a PostgreSQL 14.5 database, we could update our database environment variables in our `.env.ci` to be:

```
DB_CONNECTION=pgsql
DB_HOST=postgres
DB_PORT=5432
DB_DATABASE=testing
DB_USERNAME=postgres
DB_PASSWORD=password
```

Then we could update our `tests.yml` file to use the `postgres` service by adding the following after the `container` but before the `steps`:

```
services:
  postgres:
    image: postgres:14.5
    env:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: password
      POSTGRES_DB: testing
    ports:
      - 5432:5432
    options: --health-cmd pg_isready --health-interval 10s --
health-timeout 5s --health-retries 5
```

This will result in our `tests.yml` file looking like so:

```
name: Test Suite

on: [pull_request, push]

jobs:
  phpunit:
    runs-on: ubuntu-22.04
    container:
      image: kirschbaumdevelopment/laravel-test-runner:8.1

    services:
      postgres:
        image: postgres:14.5
        env:
          POSTGRES_USER: postgres
          POSTGRES_PASSWORD: password
          POSTGRES_DB: testing
        ports:
          - 5432:5432
        options: --health-cmd pg_isready --health-interval 10s --
health-timeout 5s --health-retries 5

    steps:
      - uses: actions/checkout@v3
        with:
          fetch-depth: 1

      - name: Install composer dependencies
```

```
        run: |
          composer install --no-scripts

      - name: Prepare Laravel Application
        run: |
          cp .env.ci .env
          php artisan key:generate

      - name: Prepare Front End Assets
        run: |
          npm install
          npm run dev

      - name: Run Test Suite
        run: php artisan test
```

Larastan

If you have installed Larastan (as shown previously in the Automated Tooling chapter), you might want to set up a workflow to run it against each pull request and push. To do this, we can create a `.github/workflows/larastan.yml` file like so:

```
name: Static Analysis

on: [pull_request, push]

jobs:
  larastan:
    runs-on: ubuntu-22.04
    steps:
      - name: Checkout
        uses: actions/checkout@v3
```

```yaml
      - name: Install dependencies
        run: composer install --no-scripts

      - name: Run Larastan
        run: vendor/bin/phpstan analyse
```

You might have noticed that this workflow is much simpler than the workflow for running the PHPUnit test suite. That's because we don't need to specify a Docker container (using the `container` field) because we don't need our workflow to do much with Laravel (such as connecting to databases, caches, filesystems, and such).

Instead, we can run Larastan on the default GitHub Actions runner machine using the `ubuntu-22.04` image that already has PHP 8.1.9, Composer 2.3.10, and PHPUnit 8.5.28 preinstalled (at the time of writing).

As a result, this workflow can run more quickly than your test suite workflow because it just needs to check out your changes, install the Composer dependencies, then run Larastan.

Laravel Dusk

If your test suite contains Laravel Dusk tests, we can create a workflow that will run the tests using GitHub Actions. To do this, we can create a new `.gtihub/workflows/dusk.yml` file like so:

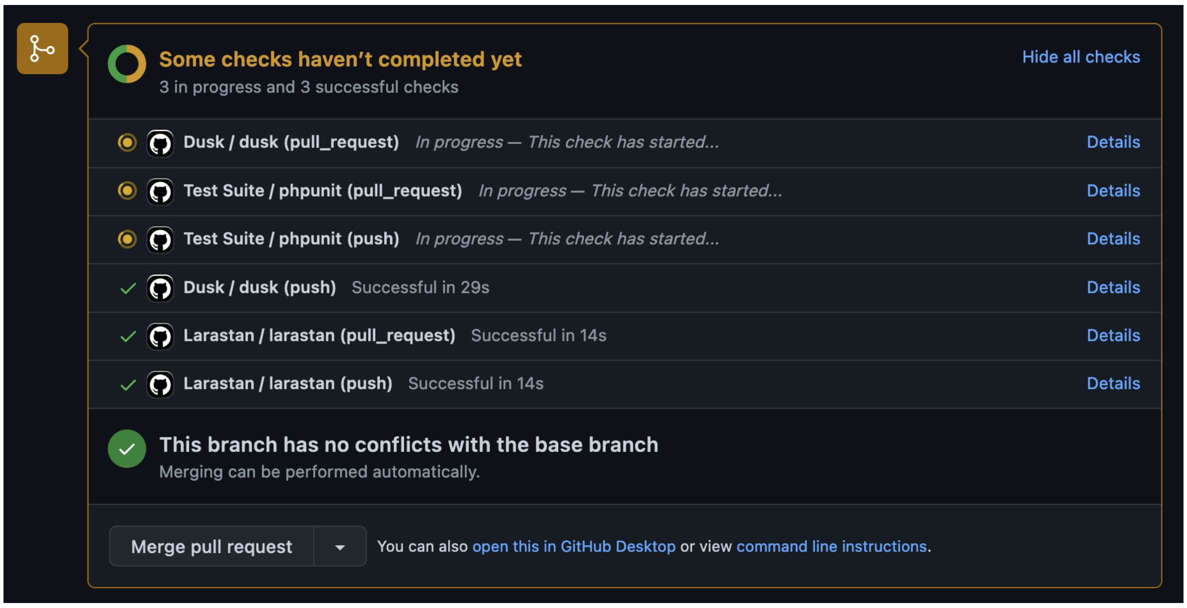
```yaml
name: Dusk

on: [pull_request, push]

jobs:

  dusk:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v3
      - name: Create Database
        run: |
          sudo systemctl start mysql
          mysql --user="root" --password="root" -e "CREATE
DATABASE testing character set UTF8mb4 collate utf8mb4_bin;"

      - name: Install Composer Dependencies
        run: composer install --no-scripts

      - name: Prepare Laravel Application
        run: |
          cp .env.ci .env
          php artisan key:generate

      - name: Upgrade Chrome Driver
        run: php artisan dusk:chrome-driver
`/opt/google/chrome/chrome --version | cut -d " " -f3 | cut -d "."
-f1`

      - name: Start Chrome Driver
        run: ./vendor/laravel/dusk/bin/chromedriver-linux &

      - name: Run Laravel Server
        run: php artisan serve --no-reload &

      - name: Run Dusk Tests
        env:
          APP_URL: "http://127.0.0.1:8000"
        run: php artisan dusk

      - name: Upload Screenshots
        if: failure()
        uses: actions/upload-artifact@v2
        with:
          name: screenshots
          path: tests/Browser/screenshots

      - name: Upload Console Logs
        if: failure()
        uses: actions/upload-artifact@v2
        with:
          name: console
          path: tests/Browser/console
```

The workflow above runs your Dusk tests using Chrome as its web browser. If the tests fail, the console logs and screenshots of the pages with failures will be stored on GitHub alongside the Action in ZIP files ready for you to download. This is extremely useful in situations when you need to debug failing tests and need a visual indication of what was happening on the page when the test failed.

Output

Now that we've set up our GitHub Actions, whenever a pull request is made on GitHub, all the workflows will run and give a visual indication of the quality of the code.

When the workflows are running, you may see the following on the page:



Some checks haven't completed yet
3 in progress and 3 successful checks

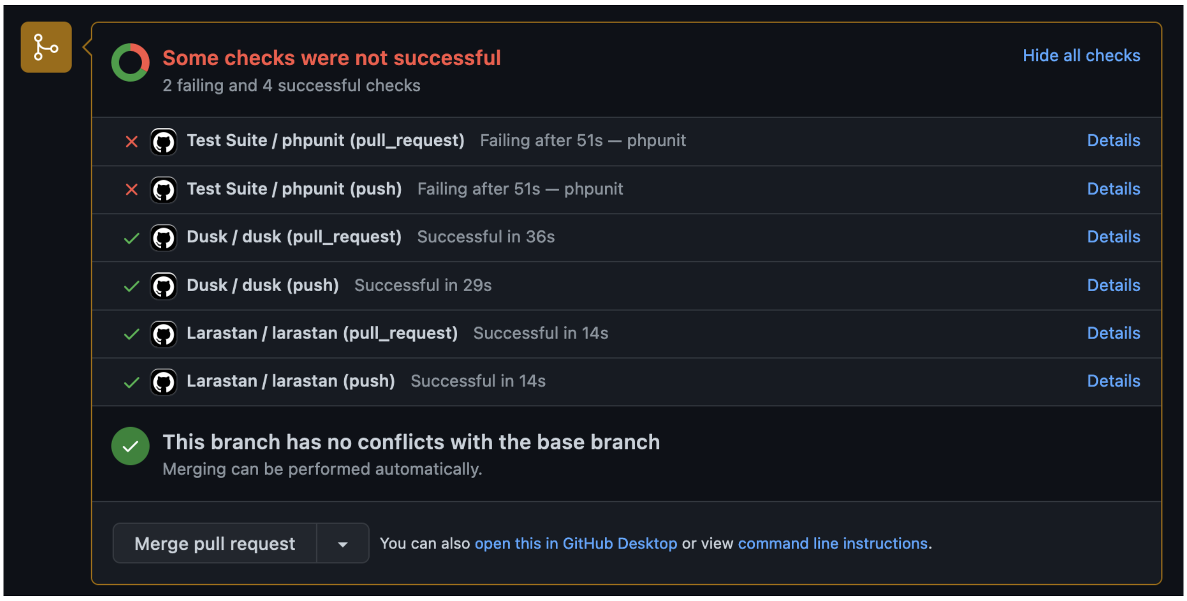
Hide all checks

Dusk / dusk (pull_request) *In progress — This check has started...* Details
Test Suite / phpunit (pull_request) *In progress — This check has started...* Details
Test Suite / phpunit (push) *In progress — This check has started...* Details
Dusk / dusk (push) Successful in 29s Details
Larastan / larastan (pull_request) Successful in 14s Details
Larastan / larastan (push) Successful in 14s Details

This branch has no conflicts with the base branch
Merging can be performed automatically.

You can also open this in GitHub Desktop or view command line instructions.

After the workflows finish running, if any of them failed, you would also get a visual indication like so:



Some checks were not successful
2 failing and 4 successful checks
Hide all checks

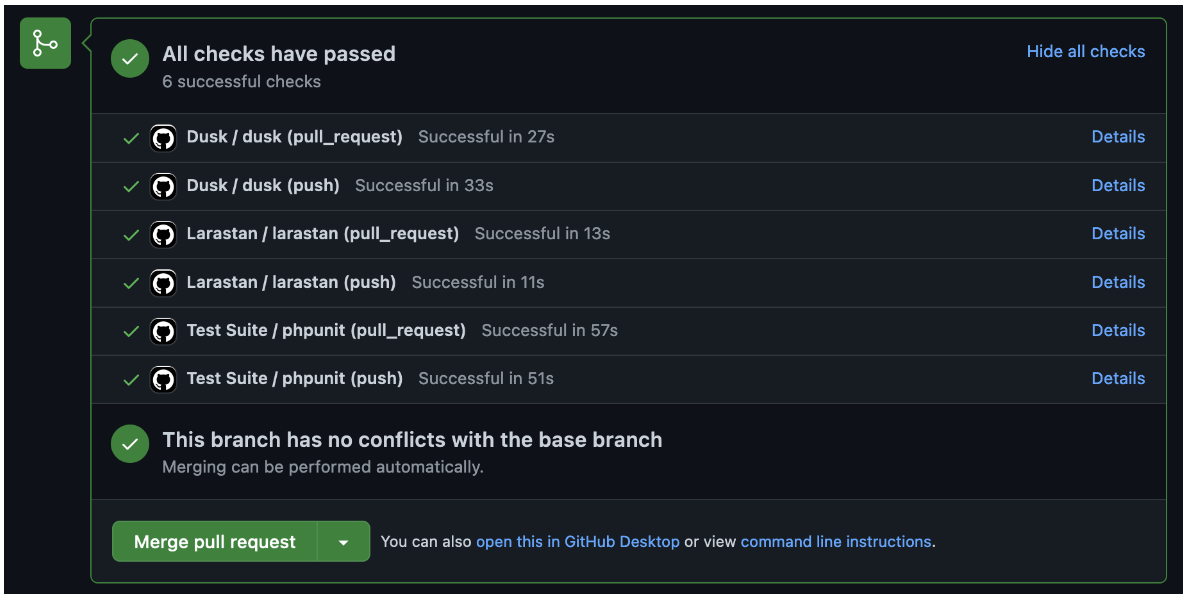
✗ Test Suite / phpunit (pull_request)	Failing after 51s — phpunit	Details
✗ Test Suite / phpunit (push)	Failing after 51s — phpunit	Details
✓ Dusk / dusk (pull_request)	Successful in 36s	Details
✓ Dusk / dusk (push)	Successful in 29s	Details
✓ Larastan / larastan (pull_request)	Successful in 14s	Details
✓ Larastan / larastan (push)	Successful in 14s	Details

This branch has no conflicts with the base branch
Merging can be performed automatically.

You can also open this in GitHub Desktop or view command line instructions.

In the image above, we can see that Dusk and Larastan both ran successfully, but that our test suite failed.

If all the workflows successfully run and all your checks pass, you may see the following:



All checks have passed
6 successful checks
Hide all checks

✓ Dusk / dusk (pull_request)	Successful in 27s	Details
✓ Dusk / dusk (push)	Successful in 33s	Details
✓ Larastan / larastan (pull_request)	Successful in 13s	Details
✓ Larastan / larastan (push)	Successful in 11s	Details
✓ Test Suite / phpunit (pull_request)	Successful in 57s	Details
✓ Test Suite / phpunit (push)	Successful in 51s	Details

This branch has no conflicts with the base branch
Merging can be performed automatically.

You can also open this in GitHub Desktop or view command line instructions.

The images above only show the GitHub Actions we created. However, if you also set up StyleCI and configured it to automatically analyse pull requests, you would also see extra checks to indicate whether the code met your code style requirements.

Section 3: Fixing

What's Covered?

In this section, we're going to discuss different ways you can fix your Laravel application and prepare it for future improvement.

We'll start by looking at the importance of using an automated error reporting system and then how to use Flare to catch bugs and errors in our Laravel applications.

Then we'll look at uptime monitoring, queue monitoring, and scheduler monitoring using Oh Dear and see how it can help give us confidence that our application is up and running.

Following this, we'll briefly discuss the approach you can take towards upgrading PHP, Laravel, and your project's dependencies in a safe, sensible manner that reduces the likelihood of introducing bugs.

Next, we'll quickly cover the importance of having a good local development environment that is similar to what your team members use and that closely resembles your production infrastructure, and how this helps reduce errors.

We'll then look at test-driven development, its advantages and disadvantages, how to use "red, green, refactor" in your development lifecycle, and look at a real-life example of how I used it to fix a bug in a project.

Finally, we'll take a quick look at different approaches you can use to remove dead code with confidence, and reduce the chance of causing errors when removing it.

Planning Your Fixing Strategy

When deciding how to start fixing your project, it's important that you have a clearly defined strategy. This will help you understand how much work is possible in the available time and what must be fixed first.

Setting up a suitable local development environment will be quick compared to the other tasks and provides a strong foundation for future work, so you may want to do this sooner than later. The initial process of transitioning from one environment to another may be finicky and time-consuming at first, but will be beneficial in the long run.

I believe it is imperative for any project to use error reporting and uptime monitoring systems. Not only can they provide a solid foundation for spotting errors in your existing system, they can also help identify bugs when you release new versions of your code to your production server or make infrastructure changes.

I understand that sending data to third-party systems is a sensitive topic that needs to be discussed and thought through carefully. It's important that, if you choose to set up either of these systems, you don't break any terms specified in contracts or terms of service with your clients or users.

When planning how to tackle upgrading PHP and Laravel versions, I think this will need to be decided on a project-by-project basis. You may be working on a project that is only one version of Laravel behind and will be fast to upgrade. On the other hand, you may be working on a project that is four versions of Laravel behind and on an older version of PHP. In this case, you may want to take longer to upgrade the project to reduce the chance of introducing bugs and errors.

Using an Error Reporting System

Errors and bugs are an unfortunate, yet inevitable part of developing every application. No matter how many processes we put in place, the human element involved in writing code means it will likely have flaws.

Types of Errors

In development, errors can generally be split into one of three categories:

1. Runtime errors
2. Syntax errors
3. Logic errors

Runtime errors occur while the code runs. For example, a runtime error may be thrown if you try to access a nonexistent item in an array.

Syntax errors occur when the code has errors and does not properly follow the syntax rules of the programming language being used. For example, a syntax error would be thrown if you did not add a `;` to the end of a statement in PHP.

Logic errors are a result of the code running, but not doing what was expected. For example, a logic error may occur if you used `<` in your code when `<=` would have been correct.

Logic errors can be more difficult to spot because they don't necessarily throw errors. Unfortunately, if logic errors aren't picked up by the test suite or during a code review, it's unlikely that they will be detected by an automated error reporting system. However, due to the nature of the errors, runtime errors and syntax errors can usually be detected and recorded with error reporting systems because they stop the execution of code.

The Benefits of an Automated Error Reporting System

If a bug in your code makes it to your production site and causes an error, there are typically two ways that you find out about it: by a human or by an automated error reporting system such as Flare, Honeybadger, or Bugsnag.

Humans aren't very good at reporting bugs and shouldn't be relied upon heavily. If a user encounters a bug, they might not realise that there actually is an issue so it would not be reported.

If the user did notice the bug, they might choose not to report it. This could be because of frustration with spending time filling out a support ticket to report the bug, or because they are too busy completing the task they were already doing. Because of the psychological phenomenon known as "diffusion of responsibility", they also might not submit the report because they assume that someone else already submitted it.

If the user does notice the bug and submits a ticket to flag the issue, they might not provide enough information to allow you to reproduce the error. This can lead to tedious email chains trying to discover information about the bug to solve it. Not only can this be mentally draining, it can be costly and slow down team progress because you are spending less time working on new features, and more time tracking down the source of an error.

Despite this, it is necessary to allow users to report bugs because they can pick up logic errors that go undetected by automated error reporting.

An automated error reporting system is an extremely useful tool that can provide many benefits to both the web developers and the end users. These systems usually have packages that can be installed into your Laravel projects and record valuable information (such as the user's details, HTTP request details, and code stack traces) to help you track down the cause of issues. The more information you have available when investigating bugs, the faster you can find their causes.

Automated error monitoring can also play a part in providing reassuring customer service for your users. For example, imagine that when a user tries to view an invoice in their system, an exception is thrown and the user is presented with an error page. By using an error monitoring system, you could be notified about it instantly and start investigating the cause of the issue before the user has the chance to submit a support ticket.

Error Reporting Using Flare

A popular automated error monitoring service we can use in Laravel projects is Flare.

Flare is an online tool built by Spatie specifically for tracking errors in Laravel. Because it's built for Laravel, it can intelligently suggest potential solutions and documentation depending on the types of errors that it detects.

Flare also feels familiar because it couples with and has a similar user interface to Ignition which provides the local error page you see when debugging Laravel applications.

You can be notified of any errors detected by Flare via email, SMS, webhooks, Slack, Telegram, Discord, or Microsoft Teams.

Let's take a look at how we can set up Flare for our project so we can start detecting errors. You may wish to set this up before you start making any improvements to your code so that you can catch bugs you introduce, or that already exist.

Installation

To start using Flare, you first need to head to the Flare website at https://flareapp.io and register for an account (if you don't have one).

You can then create a new project to track the errors for your application. If your project is running Laravel 7 or below, you can follow the documentation provided by Flare to complete the setup.

If your project is running Laravel 8 or above, you'll need to start by removing the `facade/ignition` dependency that you might have installed by running the following command in your project root:

```
composer remove facade/ignition
```

We then want to move the `spatie/laravel-ignition` dependency (that is likely already installed in your project) from the `require-dev` to the `require` section of our `composer.json` file. This allows us to use the package in production. We do this by running the following command:

```
composer require spatie/laravel-ignition
```

You will likely be presented with the following prompt when running this command:

```
spatie/laravel-ignition is currently present in the require-dev
key and you ran the command without the --dev flag, which would
move it to the require key.

Do you want to move this requirement? [no]?
```

In most cases, you can just type `yes` and continue.

We can then register the Flare logging channel in our `config/logging.php` file like so:

```
return [

    // ...

    'channels' => [
        'flare' => [
            'driver' => 'flare',
        ],
    ],

    // ...

];
```

We can then configure our logging stack to use our new `flare` channel when reporting errors. For example, if we are using the `stack` channel, we can update our `config/logging.php` file like so:

```
return [

    // ...

    'channels' => [
        'stack' => [
            'driver' => 'stack',
            'channels' => ['single', 'flare'],
            'ignore_exceptions' => false,
        ],

        'flare' => [
            'driver' => 'flare',
        ],
    ],

    // ...

];
```

You can then copy the key that Flare provides to your `.env` file like so:

```
FLARE_KEY=key-goes-here
```

Flare should now be ready to use. To confirm this, run the following command to send a test error:

```
php artisan flare:test
```

If everything is set up correctly, you should see a new error in Flare.

Available Information

Flare can provide a lot of information that is useful for debugging errors. Let's look at some of the key information it provides:

Type

This can be `web` or `cli` and can give an indication of where the error came from. `web` errors are typically errors that occurred during an HTTP request. `cli` errors are usually errors that occurred when running an Artisan command or when running a job on the queue.

Occurrence

Not only does Flare provide statistics that show how many times an error occurred, it can show how many unique users encountered the error. This is useful when trying to pinpoint whether an error is related to a specific user or not.

Stack trace

Flare provides a full stack trace so you can see exactly where an error occurred and what code ran before it happened.

Request information

If the error is a `web` error, Flare provides information about the HTTP request that was being handled when the error occurred. This information includes the URL, HTTP method, headers, query string, request body, session fields, and cookies.

User information

Flare also provides information about the user that made the request for `web` errors. It allows you to view information such as the IP address (if sending these is enabled), user agent, operating system, device, and browser. If the user is authenticated, it can also provide some user details so that you can track down the error more quickly.

Context

Flare provides information about the current Git branch (or tag) that is being used and that triggered the error. This can be useful if you need to track down a recent change in code to see what might have caused the error. You can also see the PHP and Laravel versions used. Flare also allows you to define extra "context" fields in your Laravel application so you can provide extra information each time an error is thrown. For example, if you are running a command for a specific tenant in a multi-tenant system, you could provide the tenant's name to provide context about errors related to it.

SQL queries

Flare records all the SQL queries that were executed up until the error is thrown. This can serve as a useful way of debugging whether the error was caused by an incorrect database query.

Guests and Sharing Errors

Flare provides features that allow you to collaborate with people outside your usual team.

You can invite "guests" to specific projects that can view errors and configure notifications. This is a useful feature if you want to invite a freelancer to work on your project and want to allow them to see errors being thrown on the production system without giving them access to your other projects.

Additionally, Flare also allows you to share your errors publicly. This is useful if you are struggling to fix a particular bug and want to share the bug details to get suggestions from other developers for possible solutions. When sharing an error, you can choose which information you want to share. For example, you may decide that you don't want to share the user's data, but that you want to share all the other information.

Although this feature is incredibly useful, be careful that you don't break any terms in your contracts by sharing sensitive information with people who should not have access to it. If you do use guests or public shares, remember to deactivate them as soon as they are no longer needed. Otherwise, you may risk malicious users accessing the information.

Redacting User Information

When reporting bugs to Flare, you may want to prevent sensitive information from being sent. Likewise, you may want to only send the minimum amount of user data (such as IDs or email addresses).

Although this may be enforced as part of the terms of service or a contract, I'd recommend doing this regardless to ensure your users' privacy and security.

By default, Flare will redact values of the `password` and `password_confirmation` fields in a request body and the `API-KEY` header. However, you may have other fields you want to redact. For example, you may collect a user's two-factor authentication code but don't want it reported in Flare. To do this, you must publish Flare's config file using the following Artisan command:

```
php artisan vendor:publish --
provider="Spatie\LaravelIgnition\IgnitionServiceProvider" --
tag="flare-config"
```

The above command creates a `config/flare.php` file that you can edit to change how Flare interacts with your application.

To change the request body fields that are redacted, you can add them to the `censor_fields` key that belongs to the `CensorRequestBodyFields::class` key in the `flare_middleware` field. For example, to add a `2fa_token` field, the config value may look something like so:

```
return [

    // ...

    'flare_middleware' => [

        // ...

        CensorRequestBodyFields::class => [
            'censor_fields' => [
                'password',
                'password_confirmation',
                '2fa_token',
            ],
        ],

        // ...

    ],

    // ...

];
```

To redact an `API-SECRET` header, we can follow a similar approach and add it to the `headers` field under the `CensorRequestHeaders::class` config item like so:

```
return [

    // ...

    'flare_middleware' => [

        // ...

        CensorRequestHeaders::class => [
            'headers' => [
                'API-KEY',
                'API-SECRET',
            ],
        ],

        // ...

    ],

    // ...

];
```

In addition to redacting request data reported in Flare, you can also restrict user data that is sent if an error occurs when a user is authenticated.

If you are using a `User` model for your application's users, by default, any fields added to the model's `hidden` field will automatically be removed before submitting the error report to Flare. If you want to manually specify the data that can be sent, you can add a `toFlare` method to your model that returns an array of the data that should be submitted.

For example, if you only wanted to submit the user's name and email address, you could update your model like so:

```
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Model
{
    // ...

    public function toFlare(): array
    {
        return [
            'name' => $this->name,
            'email' => $this->email
        ];
    }
}
```

For more information about Flare and to see other features for error tracking in your Laravel projects, check out their website at https://flareapp.io.

Uptime Checking, Queue Monitoring, and Scheduler Monitoring

In modern web applications, there can be a lot of moving parts. For example, your application may run on multiple application servers, using a load balancer, a separate Redis server, a separate database server, and a separate queue server. With so many components, there are many places a failure could occur.

Imagine you have a Laravel application that uses your queue to send emails to your users. If the queues stopped working, how long would it take for you to realise that the emails weren't being sent anymore? Or worse, how long would it take for you to notice that your application was completely down and couldn't be reached at all? It's likely that you'd only realise after a user submits a support ticket or bug report. By using an uptime monitoring service, these sorts of issues could be spotted before your users notice, allowing you to react and fix issues quickly.

If your application or website goes down, it can result in:

- Unhappy clients and users that want to access the system but can't.
- Lost revenue for clients if the application is part of a business process.
- Reputational damage that encourages visitors to use another website or application instead.
- Reduced ranking in Google search results.

Using an uptime monitoring service to ensure your application is up and running, you can avoid some of these issues. Not only can it give you confidence that your application's code is running as expected, it can also give you confidence that your infrastructure is running as expected.

Uptime monitoring services typically work by making HTTP requests to your application or site periodically (maybe every minute) and alerting you if your site isn't available. Some of them provide additional features for checking things like your queue workers and schedulers.

Many services are available for uptime monitoring, such as Oh Dear, Honeybadger, and Pingdom.

We're going to look at how we can set up uptime monitoring, scheduler monitoring, and queue monitoring using Oh Dear.

Oh Dear is an online service that can monitor your application's uptime, SSL certificates, broken links, scheduled tasks, application health, DNS, and domain expiry.

In the event that Oh Dear detects any errors with your application, you can be notified via email, Slack, Telegram, webhooks, SMS, Discord, Microsoft Teams, Pushover, and HipChat.

Uptime Monitoring with Oh Dear

To get started with uptime monitoring, you first need an account on Oh Dear at: https://ohdear.app.

You can then create a new site and input the URL of the site you want to monitor (e.g. `battle-ready-laravel.com`). You'll also want to enable the "Uptime" check.

Oh Dear will then be configured to send HTTP requests every minute to your specified URL to check that the site is running and available.

To change how Oh Dear determines your website's uptime, you can adjust the following settings:

- **HTTP method** - By default, Oh Dear sends `GET` HTTP requests. These can be changed to `POST`, `PATCH`, or `PUT`.
- **Expected response status codes** - By default, Oh Dear will consider a site available if the request receives a response with any status beginning with `2` (e.g. `200`). You may want to change this to a specific status code. If your application or website is password protected, you might want to update this to `401`.
- **Timeout** - By default, Oh Dear considers the site down if it takes longer than five seconds to receive a response from it. You may want to change this to be more strict or more relaxed if your application has issues causing it to run slowly.
- **Maximum redirects** - By default, Oh Dear allows your server to return five redirects before marking the site as down. You may want to change this to be more strict and allow fewer redirects if you always expect your application to return a successful response such as `200`.

Scheduler Monitoring with Oh Dear

You may want to set up scheduler monitoring for your application to check that your cron jobs are running as expected.

To use this feature, enable the "Scheduled tasks" feature for your application. You can then create a new scheduled task.

Let's look at how we can set up a scheduled task check to ensure your Laravel app's scheduler is running.

To start, we'll create a new scheduled task with the following fields:

- **Name** - Laravel Scheduler
- **Type** - Simple
- **Frequency (how often to expect the check to be run in minutes)** - 1
- **Grace time (how long to wait for the check to be run in minutes)** - 5
- **Description** - A simple check to ensure that our scheduler in Laravel is running

You will then be given a URL you can use to send HTTP requests and confirm that your scheduler is still running.

We can take this URL and add it to our `.env` as a new `OH_DEAR_PING_URL` like so:

```
OH_DEAR_PING_URL=https://ping.ohdear.app/123-abc
```

So we can access this URL within our code, we then add it to our `config/services.php` file:

```
return [

    // ...

    'oh_dear' => [
        'ping_url' => env('OH_DEAR_PING_URL'),
    ],

    // ...

];
```

If we then wanted to assert that the scheduler was run, we could update our `app/Console/Kernel.php` file:

```php
use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;
use Illuminate\Support\Facades\Http;

class Kernel extends ConsoleKernel
{
    protected function schedule(Schedule $schedule)
    {
        $pingUrl = config('services.oh_dear.ping_url');

        $schedule->call(
            fn () => Http::get($pingUrl)
        )->everyMinute();
    }
}
```

In the code above, we've specified that every minute a closure should be called that sends a `GET` request to our Oh Dear URL. As a result, if the scheduler isn't running correctly and the request isn't made, Oh Dear will detect this and alert us.

To assert that a specific command was run rather than asserting that the scheduler was run in general, we could use the `thenPing` method provided by Laravel.

For example, if we had a `reports:generate` command that runs daily and we wanted to ping Oh Dear every time it was run, the `app/Console/Kernel.php` file may look like so:

```
use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;
use Illuminate\Support\Facades\Http;

class Kernel extends ConsoleKernel
{
    protected function schedule(Schedule $schedule)
    {
        $schedule->command('reports:generate')
            ->daily()
            ->thenPing(config('services.oh_dear.ping_url'));
    }
}
```

If you choose to use this approach, you may want to create more scheduled tasks on Oh Dear so that you can have a single ping URL per command that needs to be run.

Queue Monitoring with Oh Dear

It's possible to use the scheduled tasks feature in Oh Dear to check that your Laravel application's queues are running as expected.

Let's look at how to do this.

Imagine we have three queues we want to monitor for our Laravel application: `default`, `email`, and `sms`.

We start by creating three new scheduled tasks (one for each queue) in Oh Dear. We can then add their URLs to our `.env` file like so:

```
DEFAULT_QUEUE_PING_URL=https://ping.ohdear.app/123-abc
EMAIL_QUEUE_PING_URL=https://ping.ohdear.app/456-def
SMS_QUEUE_PING_URL=https://ping.ohdear.app/789-ghi
```

We can then update our `config/services.php` file so that we can access the URLs in our code:

```
return [

    // ...

    'oh_dear' => [
        'default_queue_ping_url' => env('DEFAULT_QUEUE_PING_URL'),
        'email_queue_ping_url' => env('EMAIL_QUEUE_PING_URL'),
        'sms_queue_ping_url' => env('SMS_QUEUE_PING_URL'),
    ],

    // ...

];
```

We would then want to create a new job that can be run on these queues. To do this, we'll use the following Artisan command:

```
php artisan make:job SendPing
```

After doing this, we can update the newly created `app/Jobs/SendPing.php` file to send a `GET` request to a given URL like so:

```php
namespace App\Jobs;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Support\Facades\Http;

class SendPing implements ShouldQueue
{
    use Dispatchable;
    use InteractsWithQueue;
    use Queueable;

    public function __construct(private readonly string $url)
    {

    }

    public function handle(): void
    {
        Http::get($this->url);
    }
}
```

By looking at the job, we can see that the `SendPing` job simply accepts a URL and then makes a `GET` request to it when the job is run.

We can now update our scheduler in the `app/Console/Kernel.php` file to run this job:

```php
use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;
use Illuminate\Support\Facades\Http;

class Kernel extends ConsoleKernel
{
    protected function schedule(Schedule $schedule)
    {
        // Check the 'default' queue is running.
        $schedule->job(
            new SendPing(
                config('services.ohdear.default_queue_ping_url')
            ),
            'default'
        )->everyMinute();

        // Check the 'email' queue is running.
        $schedule->job(
            new SendPing(
                config('services.ohdear.email_queue_ping_url')
            ),
            'email'
        )->everyMinute();

        // Check the 'sms' queue is running.
        $schedule->job(
            new SendPing(
                config('services.ohdear.sms_queue_ping_url')
            ),
            'sms'
        )->everyMinute();
    }
}
```

We've now configured three jobs to be dispatched every minute via our scheduler. If each queue is running, their related job will ping the scheduled task URL in Oh Dear. Otherwise, if any of the queues aren't running, we'll be notified so we can resolve the issue.

Updating PHP, Laravel, and Packages

It would be ideal to keep up-to-date with the latest PHP and Laravel versions so you can use the latest and greatest features. But it's not always possible. Keeping on top of dependency updates and making sure upgrades don't break any existing functionality can be a time-consuming task.

However, it's important to keep your PHP version, Laravel version, and dependency versions up to date as much as possible to ensure you receive the latest security patches and bug fixes.

Having a good quality automated test suite can make this upgrade process much easier and help you spot errors earlier. But when upgrading major versions of PHP, Laravel, or any packages, there will likely be an element of manual checking on top of the automated testing to ensure that the upgrade went as planned.

When upgrading, you'll likely come across packages that aren't compatible with your Laravel or PHP version. If this is the case, you might want to investigate whether the package is still being actively maintained and whether support for the new PHP or Laravel version will be added. This means that your upgrade process may be slowed down by external factors. If this is the case, consider waiting for the maintainer to add support, or, if you feel experienced enough with using the package, you may even want to attempt to make a pull request and contribute to the dependency if it's open-source. In fact, by offering to help the maintainer, you may be helping out other teams around the world that also need that support added; it can be extremely fulfilling if you successfully contribute to the project.

If you do need to upgrade any dependencies or packages, I recommend reading through the dependencies' upgrade guides to ensure you make all the necessary changes to your codebase to avoid bugs.

Upgrading in Small Increments

In general, the best approach to upgrading is to do it in small pieces. It's much better to make lots of small changes rather than fewer big changes. For example, let's say you decide to upgrade your Laravel 6 application to Laravel 9 in one go and introduce a bug. You might have difficulty tracking down the source of the bug because you won't be sure what stage of the upgrade caused the bug. For this reason, you might find it easier to make changes in small increments, such as upgrading Laravel from version 6 to 7, then 7 to 8, and then 8 to 9.

Of course, making single changes in each upgrade isn't always possible, because a single upgrade might require other parts of your project (such as the PHP version or package version) to be updated at the same time. However, using atomic commits and splitting upgrades into small iterations can significantly reduce the chance of introducing bugs and can also reduce the amount of time it would take to track down the cause of a bug.

Depending on how your team and deployment process works, you may want to test all of your changes locally and on a staging application before releasing all the changes in one bulk update to the production server. However, you may find it easier to release smaller upgrades to your production server and wait between each release. For example, you may want to upgrade some packages, release the changes to your production site, and then wait for a week before pushing any new changes. This means that if issues occur on your production site, it would be easier to identify the cause of the issue and maybe even revert the changes.

Automating the Upgrade Using Laravel Shift

Although small changes can reduce the chance of introducing bugs, making lots of small changes can be tedious and errors can still occur without you noticing. To further reduce the chance of causing bugs and the amount of time an upgrade takes, you can use a service such as Laravel Shift.

Laravel Shift is an automated tool that helps you upgrade your project's Laravel version whenever a new major version is released. It works by opening a pull request for your project and completing the upgrade for you using atomic commits.

At the time of writing, Laravel Shift has completed over 70,000 upgrades on Laravel projects and is recommended by developers such as Taylor Otwell (creator of Laravel), Jeffrey Way (creator of Laracasts), Freek Van der Herten (Laravel package creator), Matt Stauffer (partner at Tighten), and Jacob Bennett (host of the Laravel News podcast). So the service is highly recommended by many Laravel experts.

Planning Upgrades Early

If you're upgrading a dependency related to a specific feature in your application, you may sometimes feel nervous or apprehensive about making the upgrade due to fear of breaking the feature. To increase your confidence around upgrading the dependency, you may want to write tests covering the code. This allows you to use the tests as regression tests to be sure that the upgrade did not break the existing functionality. Depending on the type of feature, you may benefit from writing a unit or feature test. However, you may also find that writing a test for the user interface using a tool such as Laravel Dusk may also give you extra confidence.

If your project is running on Laravel 8.65.0 or newer, you can use deprecation logging. So you may want to read through your logs to identify any deprecated code you are currently using. This means you will be able to solve the issue now in preparation for the upgrade rather than trying to fix the issue when you upgrade your versions. As a result, your version upgrades can be much faster and easier to validate, because there will be fewer changes at once.

Using a Suitable Local Development Environment

If you've ever worked on a project with other team members, you've likely heard someone in your team say the dreaded words that every developer hates to hear: "Well, it works on my machine". It can be frustrating when you're trying to track down the cause of a bug that is only replicable on your machine and then finding out that it is due to a slight inconsistency between your dependencies or the PHP version. It is also frustrating when you have an issue with your project that can only be reproduced on the production site and not locally.

To reduce occurrences of these scenarios, it's important to have a suitable local development environment that can be shared amongst your team and that matches your production environment as closely as possible. For example, you shouldn't be writing code that only works with PHP 8.1 if your project is running on PHP 8.0.

Since members of your team may be working on machines using different operating systems (such as macOS, Linux, and Windows), there will likely be many solutions being used to create the local environment such as Valet, Docker, Homestead, Laravel Sail, Laragon, XAMPP, etc. So it's important to decide on a single environment that is suitable for each of you and uses all the same dependencies. For example, you may all decide that every developer in your team must use Valet. Or maybe you'll decide to use Docker.

The closer you can make your local environment to your production environment, the better. They don't need to be exactly the same, but by reducing inconsistencies, you can reduce the likelihood of errors. As an example, you wouldn't want to be using Apache as your local environment, if your production server was using Nginx. You also wouldn't want to be using PostgreSQL locally, but using MySQL in your production environment.

In past projects, I have seen developers install packages using their own local PHP version (such as PHP 7.4) and then attempt to deploy the code to the production server running PHP 8.0. As you can imagine, this caused downtime for the application and could have been avoided if a suitable local development environment had been defined and used.

If I'm working on a project that does not already have a local environment defined, I like to use Laravel Sail. Laravel Sail is a command-line interface that you can use to interact with a Docker development environment. You can use it to get your Laravel application up and running with PHP, MySQL, Redis, Mailhog, and other services within a few minutes.

If you don't have much experience using Docker, Sail can be a great starting point for getting a development environment set up on your machine. However, you need to make sure you don't use it in production, as it's not optimised for that type of environment.

One of the things that I loved about Sail was how fast it was to go from running the installation command to having a site running in my browser. I didn't need experience using Docker and I did not have to spend much time reading through guides on how to get it all working; it just worked. This is especially great if you need to share the development environment across multiple operating systems.

A huge benefit to using Laravel Sail (and Docker) is that the local environment configuration is committed to the codebase. So this means that if any changes are made (such as changing the version of PHP being used), every developer can rebuild their application and make use of the changes right away.

Using Tests to Fix Bugs

What is Test-Driven Development?

As already discussed in this book, an automated test suite is an important part of your application's codebase. A good test suite helps give you confidence that the code you're writing works and does not break any existing functionality.

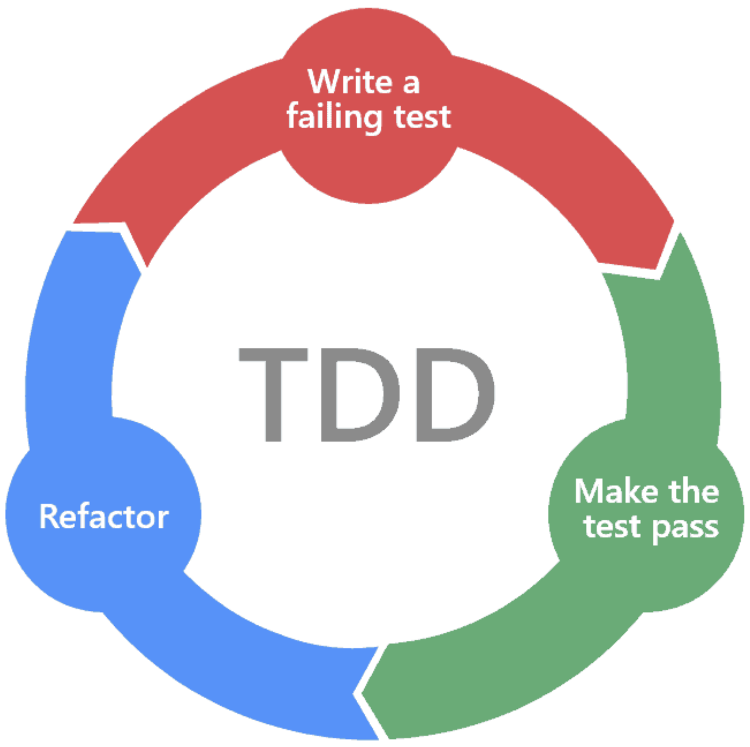
You can use tests as part of a TDD (test-driven development) workflow, which usually (but not always) involves you writing tests before writing features or changing any code. This approach encourages you to write tests as part of your development cycle to validate your code, rather than creating your tests after writing the code to check that your code works.

In a "traditional" development cycle, you might follow these steps:

- Write your application code.
- Refactor the code and clean it up.
- Add tests for the new code.

Although there's nothing wrong with this approach (I use it myself sometimes), it can mean that you won't catch bugs or errors in your architecture until you've written the tests. This can then slow down the development cycle because you'll need to spend time figuring out whether your original code worked, or if the bug was introduced during the refactoring.

TDD encourages us to follow the RGR (red, green, refactor) principle that reduces the chances of these errors occurring. The RGR approach tells us that we should start by writing a test (or updating existing tests) for a change that we're intending to make to the codebase. Due to the fact that the change won't have been made yet, this will result in our tests failing. We should then write a simple implementation to make the new change and get the tests to pass. After doing this, we can then refactor the code and clean it up, using the tests to help validate that our changes work as intended.



This changes our development cycle to follow these steps:

- Write tests for the new code.
- Write the application code.
- Check that the tests pass.
- Refactor the code and clean it up.
- Check that the tests still pass.

As you can see, this development involves tests much more and integrates them into the process, rather than treating them as something that should be added at the end.

The Advantages of Test-Driven Development

After adopting TDD as a development approach, I felt I could see a clear difference in the way that I approached solving issues and the quality of the code that was written. I found that TDD had benefits such as:

Writing code in a way that is easier to test

Because you must write the tests first, you structure your code differently than you would while using a traditional development approach. For example, it encouraged me to make more use of dependency injection, make my code more modular, and follow SOLID principles. As a result of this, it even changed the way that I approach writing code or tackling an issue outside a TDD environment.

Improving maintenance time and cost

A huge benefit of using TDD that I found is a reduction in the amount of time (and therefore, money) needed to maintain existing code. Of course, there will always be bugs — these are an inevitable part of any codebase. But I've found that projects which implemented TDD had fewer bugs than projects not using TDD. I also found that these projects were able to fix bugs more quickly because they already had a test suite covering code around the bugs to act as regression tests.

The Disadvantages of Test-Driven Development

I also found that TDD had some disadvantages such as:

It is difficult for every team member to adopt

Writing the tests before writing your application code can sometimes be a difficult thing to do, especially if you've never had to do it before. I found that being able to plan what code I was going to write before actually writing it was a skill that I had to pick up. It definitely didn't feel natural at first, but with practice, I started to see general patterns in the code that I was writing and could write my tests faster as time went on.

It is difficult to adopt when working on complex features

In an ideal world, the specification for the feature you're working on would be fully complete and predictable. But, in reality, the specification for features is usually always changing and this can mean that you can't fully write the tests before the code. This means that for complex features, or features that aren't fully thought-out in advance, TDD can be difficult to adopt.

You don't see the full benefits until the project starts to grow

As I've mentioned, TDD can be difficult to adopt and so the up-front costs and time can be much higher than using a more traditional development cycle. However, as you (and your team members) become better at it and the project grows, you start to fully reap the rewards. Whenever I've worked on projects that adopted a TDD-based approach to writing new features, I felt like the code was much more stable. But for small projects, projects with tight budgets or deadlines, or prototype projects that were likely to change soon, I felt like using a TDD approach reduced the amount of time that we had available to write application code because we spent a large majority of it writing tests. This added extra pressure to try to build the project in a shorter amount of time.

Fixing a Real Bug Using Test-Driven Development

When you find bugs in your code, you may wish to opt for using TDD and the RGR approach. When a bug is reported to me, I typically write an automated test that replicates the user's actions and that fails. This way I can keep running the tests when I'm fixing the issue and have a visual indication that the bug is fixed when the test passes. I find this usually helps me solve the issue faster than writing a possible fix for the code and then writing the test, because my first change may not correctly fix the issue.

Let's look at a real-life example of how I used TDD and RGR to fix a bug in an open-source package I maintain named Favicon Fetcher. Favicon Fetcher is a simple Laravel package that can get favicons from websites for use in your own application. It has support for multiple drivers so that you can choose which online service you use to fetch the favicons.

A bug was reported shortly after I released it stating that when using certain drivers, the favicons weren't being stored in the filesystem using the correct file extension. They were being stored with an extension that matched the website's top level domain. For example, if the favicon for `https://laravel.com` was fetched, it would be stored as `laravel.com` rather than `laravel.png`.

This was because of how I handled the favicon URLs and guessing the file extension from them. For example, when using the `google-shared-stuff` driver to fetch the `laravel.com` favicon, it would generate the following favicon URL: `https://www.google.com/s2/favicons?domain=https://laravel.com`.

I had naively written the code to simply guess the file extension based on the URL, so in this case, it resulted in `com` being the file extension. The method looked like so:

```php
protected function buildStoragePath(string $directory, string
$filename): string
{
    return Str::of($directory)
        ->append('/')
        ->append($filename)
        ->append('.')
        ->append(File::extension($this->faviconUrl));
}
```

So I added a test to prove that this bug existed:

```
/** @test */
public function
favicon_contents_be_stored_if_the_favicon_url_does_not_have_an_ima
ge_extension(): void
{
    Storage::fake();

    Http::fake([
        'https://example.com/favicon.ico' =>
Http::response('Favicon contents here'),
        'https://example.com/favicon.com' => Http::response(body:
'Favicon contents here', headers: ['content-type' =>
'image/png']),
        '*' => Http::response('Should not hit here'),
    ]);

    $favicon = new Favicon(
        url: 'https://example.com',
        faviconUrl: 'https://example.com/favicon.com',
    );

    $path = $favicon->store('favicons');

    self::assertSame(
        'Favicon contents here',
        Storage::get($path)
    );
    self::assertTrue(Str::of($path)->endsWith('.png'));
}
```

In this test, we are asserting that whenever a favicon is being stored, it uses the correct file extension (in this case `.png`). This test failed, confirming that the bug existed.

After writing the test, I was able to update the code to add a simple fix:

```php
protected function buildStoragePath(string $directory, string
$filename): string
{
    $extension = File::extension($this->faviconUrl);

    if (!Str::of($this->faviconUrl)->endsWith(['png', 'ico',
'svg'])) {
        $faviconMimetype = Http::get($this->faviconUrl)
            ->header('content-type');

        $mimeToExtensionMap = [
            'image/x-icon' => 'ico',
            'image/x-ico' => 'ico',
            'image/vnd.microsoft.icon' => 'ico',
            'image/jpeg' => 'jpeg',
            'image/pjpeg' => 'jpeg',
            'image/png' => 'png',
            'image/x-png' => 'png',
            'image/svg+xml' => 'svg',
        ];

        $extension = $mimeToExtensionMap[$faviconMimetype]
            ?? $extension;
    }

    return Str::of($directory)
        ->append('/')
        ->append($filename)
        ->append('.')
        ->append($extension);
}
```

The fix I implemented makes an additional HTTP request to fetch the favicon and then uses the `content-type` header of the response to determine which extension to use. Adding this fix resulted in the tests passing and the bug being fixed.

I then proceeded to the "refactor" stage and refactored the code by extracting the logic into separate methods:

```
protected function buildStoragePath(
    string $directory,
    string $filename
): string {
    return Str::of($directory)
        ->append('/')
        ->append($filename)
        ->append('.')
        ->append($this->guessFileExtension());
}

protected function guessFileExtension(): string
{
    $default = File::extension($this->faviconUrl);

    if (Str::of($this->faviconUrl)->endsWith(['png', 'ico',
'svg'])) {
        return $default;
    }

    return $this->guessFileExtensionFromMimeType() ?? $default;
}

protected function guessFileExtensionFromMimeType(): ?string
{
    $faviconMimetype = Http::get($this->faviconUrl)]
        ->header('content-type');

    $mimeToExtensionMap = [
        'image/x-icon' => 'ico',
        'image/x-ico' => 'ico',
        'image/vnd.microsoft.icon' => 'ico',
        'image/jpeg' => 'jpeg',
        'image/pjpeg' => 'jpeg',
        'image/png' => 'png',
        'image/x-png' => 'png',
        'image/svg+xml' => 'svg',
    ];

    return $mimeToExtensionMap[$faviconMimetype] ?? null;
}
```

After I made these changes, I was able to use my tests to confirm that I had not broken any existing functionality and that my bug fix still worked.

Safely Removing Dead Code

During your audit, you might have come across dead code that isn't being used in your project. This may be in the form of an unused method, or maybe a variable that isn't being used anymore.

Deleting dead code can keep your project's codebase lean, making future development easier. Instead of wondering whether a method is used or not, you'll be able to say with certainty that all the code in your project is used and provides meaningful functionality. This means you can write code in the future at a faster pace which is also easier to maintain and understand.

In some cases, it might be obvious that the dead code isn't being used anymore and that it can definitely be removed. However, there may be times when you find code that you can't be sure is not being used anymore. In these cases, you might want to take extra precautions to ensure that you can safely remove the code without causing bugs.

Checking the Version Control History

To start, you might check the version history of your code. By reviewing the history of a particular piece of code, you might be able to get a clue from the commits or commit messages about what the code was (or is) used for.

While looking through the history, you might find that the code was supposed to be deleted after a refactor or feature change. If this is the case, it can probably be deleted and the commit history will provide some context as to what it was used for. It's important to remember that this is only a hint and does not definitely mean that you can delete some code.

Scream Test

Another method of checking whether the code can be deleted is to perform a "scream test". This consists of you deleting (or commenting out) the code in your local or test environment and seeing if it breaks anything.

To do this, start by running your tests and checking that they all pass. It's important to remember that even if the tests pass, it doesn't show that the code can be safely removed. Your test might not cover the feature that the particular method used to (or still does) use, but it can help to give more confidence if it does.

You can also manually test the features around this particular code. For example, say you have dead code relating to updating your user's profile picture. After you've deleted the code, can you still (acting as a user) upload a new profile picture? Or has the code removal caused any issues?

Some code might appear to be unused if it's called by a dynamic method call. As a simple example, if you have a scope called "adminUsersOnly" defined on your `User` model, it would actually be defined using `scopeAdminUsersOnly`. This might make it appear in your IDE as if it isn't being used. But by going to a page that relates to admin users, you might find that the scope was, in fact, being used.

Logging or Reporting the Usage

If you're looking for extra confidence about whether a method can be deleted, you can make use of logging or reporting. For example, say we have a method called `renewSubscription` that handles renewing a user's billing subscription. As you can imagine, this method could be important and cause monetary issues if it's deleted because we thought it wasn't being used anymore.

Instead of deleting the method, we could add a `logger` or `report` call to the top of it. For example, the method could look like this:

```php
class BillingService
{
    public function renewSubscription(User $user)
    {
        report('BillingService@renewSubscription');

        // Rest of the method…
    }
}
```

Now, whenever this method is called, it will be reported for us to see, but the code will still run for the user. This is particularly useful if you're using a third-party error reporting system such as Flare, Honeybadger or Bugsnag. Depending on the importance of the code, you could wait for a set amount of time (such as a week or a month) and see if any usages are ever reported. If some usages are reported, you'll know that the code is still in use and that you can remove the `report` call. But if you aren't notified, you have more confidence that the method isn't used anymore.

Removing the Code with an Atomic Commit

Once you're confident that the code can be removed, you can simply delete the code. But it's important to remember not to make any other changes in that same commit.

As discussed earlier in this book, it's nice to keep your commits atomic (meaning only one thing happens in each one) because this makes it easier to roll back if a particular delete starts to cause any issues. Atomic commits also help when looking back at the version control history in the future to see what was done (for any future audits).

Section 4: Improving

What's Covered?

In this section, we're going to cover steps you can take to improve your code; either your existing code or code you might write in the future.

We'll start by covering how to use type hints and return types to make the most of PHP's type system to improve your project's type safety.

We'll then cover how you can reduce duplication in your code by using DRY (don't repeat yourself) principles.

Following this, we'll move on to simple ways you can improve conditions in your code by using guard clauses, replacing `if` and `elseif` with `match` expressions, and using the nullsafe operator.

We'll then cover how to improve the maintainability, stability and testability of Laravel projects by using database transactions, dependency injection, and data transfer objects.

By the end of this section, you should have a better understanding of how to improve the quality of code in your existing Laravel applications, and how to utilise these principles in new code you write.

Planning Your Improvement Strategy

Before you start making any improvements to your existing codebase, it's important to make sure that you have a clear strategy in place.

You may want to dedicate a block of time (maybe for the next week) purely to making improvements to your code. Or you may follow the "Boy Scout Rule" of "always leave the code better than you found it", meaning that you may just want to improve code whenever you find it when working on new features.

No matter which way you wish to tackle improving your codebase, it's important that you remember the following from the previous sections in this book:

- Use atomic commits to make it easier to track changes in your version control history.
- Try to make changes to code only if it has tests covering the functionality. This way you reduce the chance of introducing bugs when trying to improve the code.
- If permitted, ensure you have uptime monitoring and error reporting systems set up to report potential bugs introduced into your production application by the changes.

Making the Most of PHP's Type System

Using Type Hints and Return Types

As a freelance developer, I'm lucky to get the chance to work on a lot of exciting projects. The majority of the time, the projects I work on are existing projects and I'm usually brought on to help add extra functionality, fix bugs and help maintain systems. In a lot of projects I've worked on, I've found many bugs that could have been avoided by using return types and type hints.

In Section 1: Auditing, we briefly covered the benefits of using `declare(strict_types=1)` in your code to improve type safety. Now we'll explore the benefits of using type hinting and return types to improve code quality.

To recap and get some context into what type safety is in PHP, let's first look at how we can make our code more type safe. In PHP, we can define functions like so:

```php
/**
 * @param int $numberOne
 * @param int $numberTwo
 * @return int
 */
function addNumbers($numberOne, $numberTwo)
{
    return $numberOne + $numberTwo;
}
```

In the above code block, we created a function called `addNumbers` that adds two numbers together and returns the result. However, this method lacks type hints for the two parameters, so we could pass anything we wanted to the function, such as strings, like so:

```php
addNumbers('hello', 'goodbye');
```

If we were to run this, a `TypeError` would be thrown with the message `Unsupported` `operand types: string + string` because we can't add two strings together as if they were numbers.

To prevent these errors, we can update our function to only allow integers to be passed to the two parameters:

```
/**
 * @param int $numberOne
 * @param int $numberTwo
 * @return int
 */
function addNumbers(int $numberOne, int $numberTwo)
{
    return $numberOne + $numberTwo;
}
```

Now if we try to call this function again and pass `'hello'` and `'goodbye'`, PHP will throw `TypeError`s telling us that the parameters must be integers. As a result of doing this, we can be sure that whenever we are working inside the function that the parameters passed in are both integers.

We can additionally define return types for methods so that we enforce the returned data type. For example, we could update the `addNumbers` method to only return integers:

```
function addNumbers(int $numberOne, int $numberTwo): int
{
    return $numberOne + $numberTwo;
}
```

By adding type hints and return types, we've removed the need for the docblock that was specifying the types. Previously, we were using the docblocks to give us an indication of what type of data should be passed to the method and what would be returned. However, docblocks can be incorrect or can become outdated if they're not updated when a method signature or return value is changed. By adding the type hints and return type, we've enforced the specific types so we can rely on PHP to throw errors if incorrect types are used.

We also reduced the amount of maintenance we need to do in the future by removing the need to keep docblocks up to date which encourages us to follow the principle of "code as documentation". However, docblocks can still be useful for describing a method's functionality and, if you're using a tool like Larastan, the chances of having outdated docblocks can be reduced.

Union Types

When you're writing the signature for a method, you may find that there isn't a single type you can use. For example, you may not be sure what type of data will be returned or what type of data the method should receive. If this is the case, consider using a union type.

Let's see how we can use union types using an example from the PHP documentation. Imagine that we have a `Number` class we can use to hold an `integer` or `float`. To create this class using union types, it could look something like this:

```
class Number
{
    private int|float $number;

    public function setNumber(int|float $number): void
    {
        $this->number = $number;
    }

    public function getNumber(): int|float
    {
        return $this->number;
    }
}
```

As you can see, we managed to use union types in three different places within the class:

- The `number` property. This field can now only be set to be an `integer` or `float`. Attempting to set the property to be anything else will cause an error to be thrown.
- The type hinting for the parameter in the `setNumber` method.

- The return type of the `getNumber` method. This is also useful because it indicates to developers what types of data can be returned and also enforces that it will be one of the specified types.

Type Hints and Return Types in Closures

You can also add return types and type hints to closures in your PHP applications to improve the type safety of your code.

I like to do this because it:

- Improves the autocomplete suggestions in my IDE.
- Reduces the chances of bugs from working with the wrong types of data within closures.
- Improves the readability of the code by making the input and output of the closures more obvious.

As an example, let's take the following simple code block:

```php
/** @var array $subscribers */
$subscribers = $this->getSubscribersFromApi();

$paidMembers = collect($subscribers)
    ->filter(function ($subscriber) {
        return $subscriber['is_paid_member'];
    })
    ->map(function ($subscriber) {
        return new PaidMember(
            id: $subscriber['id'],
            email: $subscriber['email'],
            name: $subscriber['name'],
        );
    });
```

In the code above, we get an array of subscribers from an API. We then add the subscribers to a `Collection`, filtering them so that we are only left with the paid members, and then mapping through each of the subscribers so that we can convert each array into a `PaidMember` object. By the end of the code, the `paidMembers` variable should be a `Collection` containing only `PaidMember` objects.

Although this code might work, it's not immediately obvious what the `subscriber` parameters are that get passed into the two closures. However, by adding type hints and return types, we can enforce the types that the closures will accept and return to make their intent more obvious to developers reading the code.

```php
/** @var array $subscribers */
$subscribers = $this->getSubscribersFromApi();

$paidMembers = collect($subscribers)
    ->filter(function (array $subscriber): bool {
        return $subscriber['is_paid_member'];
    })
    ->map(function (array $subscriber): PaidMember {
        return new PaidMember(
            id: $subscriber['id'],
            email: $subscriber['email'],
            name: $subscriber['name'],
        );
    });
```

Just by doing this, we've made it more obvious that the closure used in the `filter` method accepts an `array` and returns a `bool`. It's also clear that the closure used in the `map` method accepts an `array` and returns a `PaidMember` object.

DRYing Up Your Code

As your application grows, you'll likely find pieces of duplicated code scattered around your codebase.

As a simple example, imagine we have a blogging application that allows users to view blog posts, comment on them, and react to them (using reactions such as "like" and "love" that you might see on social media platforms). We might want to only allow these actions to be carried out if the blog post is published; and for access to be denied if the post isn't published.

So we may have a controller that looks like so:

```php
class PostController extends Controller
{
    public function show(Post $post)
    {
        if ($post->published_at > now()) {
            abort(403);
        }

        // Show the post...
    }

    public function comment(Request $request, Post $post)
    {
        if ($post->published_at > now()) {
            abort(403);
        }

        // Comment on the post...
    }

    public function react(Post $post)
    {
        if ($post->published_at > now()) {
            abort(403);
        }

        // React to the post...
    }
}
```

As you can see, we have duplicated the exact same condition so that it is being used in three different places. Although this example is simple to understand, there is space for bugs to be introduced when we need to change the code everywhere that it's being used.

For example, imagine we want to update authorisation checks so users with the `admin` role can view, comment, and react to posts even if they're unpublished. We could update the code like so:

```php
class PostController extends Controller
{
    public function show(Post $post)
    {
        if (
            $post->published_at > now()
            && !auth()->user()->hasRole('admin')
          ) {
            abort(403);
        }

        // Show the post...
    }

    public function comment(Request $request, Post $post)
    {
        if (
            $post->published_at > now()
            && auth()->user()->hasRole('admin')
        ) {
            abort(403);
        }

        // Comment on the post...
    }

    public function react(Post $post)
    {
        if (
            $post->published_at > now()
            && !auth()->user()->hasRole('admin')
        ) {
            abort(403);
        }

        // React to the post...
    }
}
```

Did you spot the mistake? In the `comment` method, the condition was updated to use `&& auth()->user()->hasRole('admin')` rather than `&& !auth()->user()->hasRole('admin')` (missing the `!`). This means that if a user with the `admin` role were to try commenting on an unpublished post, access would be denied. And if a user without the `admin` role tried to comment on an unpublished post, access would be permitted. This isn't the behaviour we wanted to add.

A mistake like this is very easy to make, especially if there are many occurrences of the code to change. By making use of an automated test suite, these issues can usually be spotted. But by DRYing up our code, we can reduce the chances of such issues happening in the first place.

The acronym "DRY" stands for "don't repeat yourself". So "DRYing up your code" means reducing the amount of repetition and duplicated code in your project. You can reduce the amount of duplication in your code by extracting shared code into methods.

Let's see how we can use DRY code principles with our example controller.

Depending on where the code is located, we can reduce duplication by extracting logic into classes such as gates, policies, service classes, action classes, middleware, and models. In our example, the code that is duplicated is related to authorisation, so we can use a policy to extract the logic.

We can create a `PostPolicy` like so:

```php
class PostPolicy
{
    public function view(User $user, Post $post): bool
    {
        return $this->canAccessPost($user, $post);
    }

    public function comment(User $user, Post $post): bool
    {
        return $this->canAccessPost($user, $post);
    }

    public function show(User $user, Post $post): bool
    {
        return $this->canAccessPost($user, $post);
    }

    private function canAccessPost(User $user, Post $post): bool
    {
        return $post->published_at <= now()
            || auth()->user()->hasRole('admin');
    }
}
```

Our authorisation check has been extracted into a single place in our policy, so we can now use the policy methods in our controller:

```php
class PostController extends Controller
{
    public function show(Post $post)
    {
        $this->authorize('view', $post);

        // Show the post...
    }

    public function comment(Request $request, Post $post)
    {
        $this->authorize('comment', $post);

        // Comment on the post...
    }

    public function react(Post $post)
    {
        $this->authorize('react', $post);

        // React to the post...
    }
}
```

Advantages of DRYing Up Your Code

By extracting our authorisation logic into a policy, we've reduced the duplication. As a result, we have:

Improved maintainability and readability by encapsulating logic within methods

This means if we need to change the conditions used for the authorisation check, we can simply add it in one place and it will be available in every place we are using it. This reduces the chance of us forgetting to update an occurrence of the duplicated code or making a mistake like in the preceding example.

Improved testability

By moving the duplicated code into a single method, it is easier to write smaller unit tests for the newly created method.

When to DRY Up Your Code

It's important to remember that not all duplicated code is bad code! Depending on the situation, DRYing up your code may add extra complexity which actually makes the code more difficult to understand.

You may not benefit from extracting duplicated logic if it's only duplicated once. You want to think about extracting the code if:

- The code is being used in three or more different places.
- The code is being duplicated once but is very complex or many lines.

Refactoring Conditions

Reducing Indented Code

When writing code, it is easy to create nested "if statements" when you only want to run certain lines of code under multiple conditions.

I've found this exists more in older projects where developers have added extra "if statements" into existing code when adding a new feature or fixing a bug. This can quickly become messy and difficult to maintain.

In general, the more indentations (usually from nested conditionals) that code has, the more difficult a developer will find it to read and understand. Reducing the number of nested conditionals and indentations makes your code more maintainable by making it easier to work with.

One way to achieve this is with "guard clauses". This follows the principle of checking for any negative conditions before executing the "happy path" in a method.

Let's look at a basic example of some code and see how we can clean it up step-by-step:

```php
public function postToTwitter(Post $post): Tweet
{
    if ($this->hasValidApiKey()) {
        if ($post->alreadyPublishedOnTwitter()) {
            throw new Exception(
                'The post has already been tweeted.'
            );
        } else {
            $tweet = $this->tweet($post);

            if ($tweet) {
                return $tweet;
            } else {
                throw new Exception(
                    'The post could not be tweeted.'
                );
            }
        }
    } else {
        throw new Exception('No valid API key is available.');
    }
}
```

To begin with, we can remove the last `else` near the end of the function. This is because the code inside the first `if` will always either return a `Tweet` object or throw an exception, so we don't need to worry about the code in the `else` block being executed after the code in the `if` block. The resulting code will look like so:

```
public function postToTwitter(Post $post): Tweet
{
    if ($this->hasValidApiKey()) {
        if ($post->alreadyPublishedOnTwitter()) {
            throw new Exception(
                'The post has already been tweeted.'
              );
        } else {
            $tweet = $this->tweet($post);

            if ($tweet) {
                return $tweet;
            } else {
                throw new Exception(
                    'The post could not be tweeted.'
                );
            }
        }
    }

    throw new Exception('No valid API key is available.');
}
```

After doing this, we can then negate the first `if` block. This means we can move the exception to the top of the method and move our "happy path" to the bottom of the method to reduce indentation:

```php
public function postToTwitter(Post $post): Tweet
{
    if (!$this->hasValidApiKey()) {
        throw new Exception('No valid API key is available.');
    }

    if ($post->alreadyPublishedOnTwitter()) {
        throw new Exception('The post has already been tweeted.');
    } else {
        $tweet = $this->tweet($post);

        if ($tweet) {
            return $tweet;
        } else {
            throw new Exception('The post could not be tweeted.');
        }
    }
}
```

Following this, we can split the next `if` block into its own block and remove the `else` statement to further reduce the indentation:

```php
public function postToTwitter(Post $post): Tweet
{
    if (!$this->hasValidApiKey()) {
        throw new Exception('No valid API key is available.');
    }

    if ($post->alreadyPublishedOnTwitter()) {
        throw new Exception('The post has already been tweeted.');
    }

    $tweet = $this->tweet($post);

    if ($tweet) {
        return $tweet;
    } else {
        throw new Exception('The post could not be tweeted.');
    }
}
```

Next we can then remove the final `else` statement:

```php
public function postToTwitter(Post $post): Tweet
{
    if (!$this->hasValidApiKey()) {
        throw new Exception('No valid API key is available.');
    }

    if ($post->alreadyPublishedOnTwitter()) {
        throw new Exception('The post has already been tweeted.');
    }

    $tweet = $this->tweet($post);

    if ($tweet) {
        return $tweet;
    }

    throw new Exception('The post could not be tweeted.');
}
```

We can then simplify the last part of the code by merging the `return` and `throw` together like so:

```php
public function postToTwitter(Post $post): Tweet
{
    if (!$this->hasValidApiKey()) {
        throw new Exception('No valid API key is available.');
    }

    if ($post->alreadyPublishedOnTwitter()) {
        throw new Exception('The post has already been tweeted.');
    }

    return $this->tweet($post)
        ?? throw new Exception('The post could not be tweeted.');
}
```

To further improve this code, we could also extract our guard clauses into their own `ensureTweetCanBeSent` method. By doing this, we can significantly simplify the method:

```php
public function postToTwitter(Post $post): Tweet
{
    $this->ensureTweetCanBeSent($post);

    return $this->tweet($post)
        ?? throw new Exception('The post could not be tweeted.');
}

private function ensureTweetCanBeSent(Post $post): void
{
    if (!$this->hasValidApiKey()) {
        throw new Exception('No valid API key is available.');
    }

    if ($post->alreadyPublishedOnTwitter()) {
        throw new Exception('The post has already been tweeted.');
    }
}
```

As a result of using guard clauses in our method, we've managed to:

- Improve the readability of the code.
- Reduce the amount of indentation in the code.
- Improve the maintainability of the code. If we need to add any other checks, these could be added to the `ensureTweetCanBeSent` method to keep the code readable.

Replacing `if` and `elseif` with `match`

Since PHP 8.0, I've been using `match` expressions to reduce the need for `if`, `elseif`, and `else` blocks. As well as reducing indentation, I've found that using the `match` statement improves readability of a method and reduces the amount of `return`s.

Take this example method that returns an object based on the name of the driver passed to the `driver` method:

```php
use App\Drivers\BitbucketDriver;
use App\Drivers\GitHubDriver;
use App\Drivers\GitlabDriver;
use App\Exceptions\GitDriverException;
use App\Interfaces\GitDriver;

public function driver(string $driver): Driver
{
    if ($driver === 'github') {
        return new GitHubDriver();
    } elseif ($driver === 'gitlab') {
        return new GitlabDriver();
    } elseif ($driver === 'bitbucket') {
        return new BitbucketDriver();
    } else {
        throw new GitDriverException(
            $driver.' is not a valid driver'
        );
    }
}
```

We can refactor this method to use a `match` expression like so:

```
use App\Drivers\BitbucketDriver;
use App\Drivers\GitHubDriver;
use App\Drivers\GitlabDriver;
use App\Exceptions\GitDriverException;
use App\Interfaces\GitDriver;

public function driver(string $driver): Driver
{
    return match ($driver) {
        'github' => new GitHubDriver(),
        'gitlab' => new GitLabDriver(),
        'bitbucket' => new BitbucketDriver(),
        default => throw new GitDriverException(
            $driver.' is not a valid driver'
        ),
    }
}
```

You can extend this approach by passing `true` to the `match` expression and adding your conditions within the statement itself. This can be a nice way of using operators such as `instanceof` in your checks. The Laravel framework itself actually uses this approach for writing simple and clean conditionals. For example, the `render` method in the `\Illuminate\Foundation\Exceptions\Handler` class uses the following code to determine what type of response to return when handling an exception:

```
return match (true) {
    $e instanceof HttpResponseException => $e->getResponse(),
    $e instanceof AuthenticationException =>
$this->unauthenticated($request, $e),
    $e instanceof ValidationException =>
$this->convertValidationExceptionToResponse($e, $request),
    default => $this->renderExceptionResponse($request, $e),
};
```

Using the Nullsafe Operator

Another great way to improve conditionals in your code is to make use of the nullsafe operator that was added in PHP 8.0.

The nullsafe operator can be used when chaining methods or properties together where one of the parts of the chain may return `null`.

Take this basic example that attempts to find a user's Twitter handle or defaults to `null`:

```
$twitterHandle = null;

if ($user = auth()->user()) {
    if ($settings = $user->getSettings()) {
        if ($socials = $settings->socials) {
            $twitterHandle = $social->twitter_handle;
        }
    }
}
```

This code could be rewritten using the nullsafe operator like so:

```
$twitterHandle = auth()->user()
    ?->getSettings()
    ?->socials
    ?->twitter_handle;
```

Now if the `user` method returned `null`, the chain would end and `twitterHandle` would be equal to `null`. Likewise, if the `getSettings` method returned `null` or the `socials` property was equal to `null`, the chain would end and `twitterHandle` would be equal to `null`. As a result, the `twitterHandle` field would only be set if none of the fields returned `null`.

I find this is a great way of reducing the amount of indentation in the code and merging `if` statements into a single line of code.

Using Database Transactions

One way to improve your application's code is to implement database transactions.

There are a lot of technical, complicated-sounding explanations for what a database transaction is. But for a large majority of us, as web developers, we just need to know that transactions are a way of completing a unit of work in a database.

To understand what this means, let's look at a basic example that provides context. Imagine we have an application that lets users register. When a user registers, we want to create a new account for them then assign them a default role of "general".

Our code might look something like this:

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

$user->roles()->attach(
    Role::where('name', 'general')->first()
);
```

At first glance, this code seems completely fine. But when we take a closer look, we can see that there's actually something that could go wrong. It's possible that we could create the user but not assign them the role. This could be caused by several things, such as a bug in the code that assigns the roles or a problem that prevents us from reaching the database.

This would result in a user in the system that doesn't have a role. As you can imagine, this will likely cause exceptions and bugs in other places across your application because you would always be making an assumption that a user has a role (and rightly so).

To solve this issue, we can use database transactions. Using transactions would ensure that if anything goes wrong when executing the code, any changes to the database from inside that transaction would be rolled back. For example, if the user was inserted into the database but the query to assign the role failed for any reason, the transaction would be rolled back and the user record would be removed. Thus we wouldn't be able to create a user without an assigned role.

In other words, it's "all or nothing".

Adding the Database Transactions

Now that we have an idea of what transactions are and what they achieve, let's look at how to use them in Laravel.

In Laravel, it's really easy to start using transactions thanks to the `transaction` method that we can access on the `DB` facade. Continuing with our example code from earlier, let's look at how we could use a transaction when creating a user and assigning them a role:

```
use Illuminate\Support\Facades\DB;

DB::transaction(function () use ($user, $request): void {
    $user = User::create([
        'email' => $request->email,
    ]);

    $user->roles()->attach(
        Role::where('name', 'general')->first()
    );
});
```

Now that our code is wrapped in a database transaction, if an exception is thrown at any point inside it, any changes to the database will be returned to how they were before the transaction started.

Manually Using Database Transactions

There may be times when you want more granular control over your transactions. For example, imagine you're integrating with a third-party service such as Mailchimp or Xero. And we'll say that whenever you create a new user, you also want to make an HTTP request to the third party's API to create the new user in that system too.

We might want to update our code so that if we can't create the user in our own system and the third-party system, neither of them should be created. It's possible that if you're interacting with a third-party system, you might have a class you can use to make requests. Or there might be a package that you can use. Sometimes, the classes making the request might throw an exception when certain requests aren't successful. However, some of them may silence the errors and instead just return `false` from the method you called and place the errors in a field on the class.

Imagine we have the following example class that makes a call to the API:

```php
class ThirdPartyService
{
    private array $errors;

    public function createUser($userData): array|bool
    {
        $request = $this->makeRequest($userData);

        if ($request->successful()) {
            return $request->body();
        }

        $errors = $request->errors();

        return false;
    }

    public function getErrors(): array
    {
        return $this->errors;
    }
}
```

We could update our code to use the `ThirdPartyService` class:

```php
use Illuminate\Support\Facades\DB;
use App\Services\ThirdPartyService;

DB::beginTransaction();

$thirdPartyService = new ThirdPartyService();

$userData = [
    'email' => $request->email,
];

$user = User::create($userData);

$user->roles()->attach(Role::where('name', 'general')->first());

if ($thirdPartyService->createUser($userData)) {
    DB::commit();

    return;
}

DB::rollBack();

report($thirdPartyService->getErrors());
```

In the code above, we can see that we start a transaction, create the user and assign them a role, and then we make a call to the third-party service. If the user is successfully created in the external service, we can safely commit our database changes knowing that everything has been created correctly. However, if the user wasn't created in the external service, we roll back the changes in our database (remove the user and their role assignment) and then report the errors.

Tips for Interacting with Third-Party Services

As a bonus tip, I usually recommend putting any code that affects any third-party systems, file storage, or caches after your database calls.

To understand this a bit better, let's take the preceding code example. Notice how we made all our changes to our database first before making the request to the third-party service. This means that if any errors were returned from the third-party request, the user and role assignment in our own database would be rolled back.

However, if we did this the other way around and made the request before making our database changes, this would not be the case. If we had any errors when creating our user in our database, we would have created a new user in the third-party system but not our own. As you can imagine, this could potentially lead to more issues. The severity of this issue could be reduced with a clean-up method that deletes the user from the third-party system but this could cause more problems and is more code to write, maintain, and test.

I recommend putting your database calls before your API calls, but this isn't always possible. There might be times when you need to save a value in your database that's returned from a third-party request. If this is the case, this is fine as long as you ensure you have code in place to handle any failures.

Using Automatic or Manual Transactions

Note that because our original example (using the `DB::transaction()` method) rolls back transactions if an exception is thrown, we could also use that approach for making requests to our third-party service. Instead, we could update our class to do something like this:

```php
use Illuminate\Support\Facades\DB;
use App\Services\ThirdPartyService;

DB::transaction(function () use ($user, $request): void {
    $user = User::create([
        'email' => $request->email,
    ]);

    $user->roles()->attach(
        Role::where('name', 'general')->first()
    );

    if (! $thirdPartyService->createUser($userData)) {
        throw new \Exception('User could not be created');
    }
});
```

This is definitely a viable solution and would successfully roll back the transaction as expected. In fact, I prefer the way this looks to manually using the transactions. I think it is simpler to read and understand.

However, exception handling can be expensive in terms of time and performance in comparison to using an `if` statement like when we are manually committing or rolling back the transactions.

As an example, if this code was being used for something like importing a CSV file with 10,000 users' data, you might find that throwing the exception will slow down the import considerably.

However, if it was just being used in a simple web request where a user can register, you would likely be okay with throwing the exception. Which is best comes down to the size of your application, and how much performance is a key factor; so this is something that you need to decide on a case-by-case basis.

Dispatching Queued Jobs inside Database Transactions

Whenever you work with jobs inside transactions, there's a "gotcha" you need to be aware of.

Let's revisit our example from earlier. Imagine that after we created a user, we want to run a job that alerts an admin of a new sign up and sends a welcome email to the new user. We'll do this by dispatching a queued job called `AlertNewUser` like so:

```php
use Illuminate\Support\Facades\DB;
use App\Jobs\AlertNewUser;
use App\Services\ThirdPartyService;

DB::transaction(function () use ($user, $request): void {
    $user = User::create([
        'email' => $request->email,
    ]);

    $user->roles()->attach(
        Role::where('name', 'general')->first()
    );

    AlertNewUser::dispatch($user);
});
```

When you begin a transaction and make changes to any data inside it, those changes are only available to the request/process that the transaction is running in. For any other requests or processes to access the data you've changed, the transaction first has to be committed. Therefore, if we dispatch any queued jobs, event listeners, mailables, notifications, or broadcast events from inside our transaction, our data changes might not be available inside them due to a race condition.

This can happen if the queue worker starts processing the queued code before the transaction is committed. This can lead to your queued code trying to access data that doesn't exist yet and may cause errors. In our case, if the queue `AlertNewUser` job is run before the transaction is committed, our job will try accessing a user that's not stored in the database yet. As you can expect, this will cause the job to fail.

To prevent this race condition, we can make changes to our code and/or our config to ensure that the jobs are only dispatched after the transactions are successfully committed. We can update `config/queue.php` and add the `after_commit` field. Imagine we are using the `redis` queue driver. We could update our config like so:

```
return [

  // ...

  'connections' => [

         // ...

         'redis' => [
             'driver' => 'redis',
             // ...
             'after_commit' => true,
         ],

         // ...

  ],

  // ...

];
```

After this change, if we try to dispatch a job inside a transaction, the job will wait for the transaction to be committed before actually dispatching the job. If the transaction is rolled back, it will also prevent the job from being dispatched.

There may be a reason that you don't want to set this option globally in the config. If this is the case, Laravel still provides some nice helper methods that we can use on a case-by-case basis.

If we wanted to update the code in our transaction to only dispatch the job after it's committed, we could use the `afterCommit` method like so:

```php
use Illuminate\Support\Facades\DB;
use App\Jobs\AlertNewUser;
use App\Services\ThirdPartyService;

DB::transaction(function () use ($user, $request): void {
    $user = User::create([
        'email' => $request->email,
    ]);

    $user->roles()->attach(
        Role::where('name', 'general')->first()
    );

    AlertNewUser::dispatch($user)->afterCommit();
});
```

If we have set the global `after_commit => true` in our queue config but don't want to wait for a transaction to be committed, we can use Laravel's `beforeCommit` method. To do this we update our code like so:

```php
use Illuminate\Support\Facades\DB;
use App\Jobs\AlertNewUser;
use App\Services\ThirdPartyService;

DB::transaction(function () use ($user, $request): void {
    $user = User::create([
        'email' => $request->email,
    ]);

    $user->roles()->attach(
        Role::where('name', 'general')->first()
    );

    AlertNewUser::dispatch($user)->beforeCommit();
});
```

Improving the Testability of Your Code

You might find it difficult to write tests for your code. For example, you might be testing a method in a class that tries to make API requests to external services. There are definitely cases where you'd want to send the request so that you can do full end-to-end testing. But in most cases, you wouldn't want to actually send it and would just want to assert that your code attempted to send the request.

To give more context about this subject and briefly explore how we can make your code more testable, we'll use a simple example. Take this example controller method:

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

class NewsletterSubscriptionController extends Controller
{
    public function store(Request $request): JsonResponse
    {
        $service = new NewsletterSubscriptionService();
        $service->subscribe($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 assume the service handles all the different processes that need to be carried out to complete a user's subscription to the newsletter, including making an API request to the third-party newsletter platform.

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

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

There are two problems you might have noticed in our test:

- It doesn't 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 likely wouldn't know.
- We are calling the `handle` method on the service class, which means that it will attempt to send the API request to the third-party newsletter platform each time the test is run.

We could add more assertions in this controller test to be sure that the service class code is 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 for the newsletter. Now if we were also to write tests for this controller that checked that the service class ran correctly, we'd have near duplicates of test code. This means that if we updated the way that the service class runs internally, we'd also need to update all of these tests.

We could add checks in our service class method to detect whether Laravel is running in a test environment (using something like `app()->runningUnitTests()`) and exit out of the method if this is the case. But I'd advise against doing this and only using this approach as a last resort.

Instead, we can make use of mocks, fakes, Laravel's service container, and dependency injection in our tests. According to the Laravel documentation, "The Laravel service container is a powerful tool for managing class dependencies and performing dependency injection. Dependency injection is a fancy phrase that essentially means this: class dependencies are 'injected' into the class via the constructor or, in some cases, 'setter' methods.".

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
{
    public function store(
        Request $request,
        NewsletterSubscriptionService $service
    ): JsonResponse {
        $service->subscribe($request->email);

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

In the above code, we've added the `NewsletterSubscriptionService` class as an argument to the `store` method because Laravel allows dependency injection in controllers. This means that when Laravel invoked this controller method, it detected that we had type hinted the service class and retrieved it from the service container for us.

As a result of this, we can now look at ways of telling Laravel to use a different class (in our case, a test double) when we try to resolve this class. This is so we can use our existing service class when running our code in production, but use a test double when running our tests.

To do this, we need to ensure the service class and its test double both contain the same public methods. We can do this by making sure that both classes implement an interface.

In basic terms, interfaces are just descriptions of what a class should do. They can be used to ensure any class implementing the interface includes 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 define the public methods 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 in a method in a class). This is because the interfaces only define communication between objects rather than defining the communication and behaviour like in a class.

We'll create a new interface in an `app/Interfaces` folder and call it `NewsletterSubcriptionInterface`. The interface will look like so:

```
interface NewsletterSubscriptionInterface
{
    public function subscribe(string $email): void;
}
```

In our interface we specified that any classes implementing this interface must include a `subscribe` method that accepts a string parameter and doesn't return anything. We can now update our `NewsletterSubscriptionService` to implement this interface:

```php
class NewsletterSubscriptionService implements
NewsletterSubscriptionInterface
{
    public function subscribe(string $email): void
    {
        // Code goes here to subscribe the user...
    }
}
```

We can now instruct Laravel to return an instance of our `NewsletterSubscriptionService` whenever we try to resolve a `NewsletterSubscriptionInterface`. We can do this by using `$this->app->bind()` in the `register` method of the `app/Providers/AppServiceProvider.php` like so:

```php
use App\Interfaces\NewsletterSubscriptionInterface;
use App\Services\NewsletterSubscriptionService;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->bind(
            NewsletterSubscriptionInterface::class,
            NewsletterSubscriptionService::class
        );
    }

    public function boot()
    {
        //
    }
}
```

We can then update our controller method to type hint the interface rather than the class itself:

```php
use App\Interfaces\NewsletterSubscriptionInterface;
use Illuminate\Http\JsonResponse;
use Illuminate\Http\Request;

class NewsletterSubscriptionController extends Controller
{
    public function store(
        Request $request,
        NewsletterSubscriptionInterface $service
    ): JsonResponse {
        $service->subscribe($request->email);

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

Now that we've updated the controller, we can move on to the test fake. Let's create a new `NewsletterSubscriptionFake` class that implements the `NewsletterSubscriptionInterface` interface like so:

```
namespace App\Services;

use App\Interfaces\NewsletterSubscriptionInterface;
use PHPUnit\Framework\Assert as PHPUnit;

class NewsletterSubscriptionFake implements
NewsletterSubscriptionInterface
{
    private array $subscribers = [];

    public function subscribe(string $email): void
    {
        $this->subscribers[] = $email;
    }

    public function assertSubscription(string $email): void
    {
        PHPUnit::assertContains(
            $email,
            $this->subscribers,
            '['.$email.'] was not subscribed.',
        );
    }
}
```

This class contains several methods and properties. It contains the `subscribe` method that is enforced by the interface that it implements. This method simply takes the `email` field that is passed to it and stores it in a class-level array. This is so we can use this value later in our assertions in the tests.

There is also an `assertSubscription` method. This method accepts an `email` parameter then checks whether the email exists in our class-level `subscribers` array. If it does, the test will continue as expected and PHPUnit will record that as a successful assertion. Otherwise, if the email wasn't present (maybe because the `subscribe` method wasn't called, or was called with an incorrect email), PHPUnit would fail and return an error message. For example, if we passed `joe.blogs@example.com` then the failure message would read `[joe.bloggs@example.com] was not subscribed.`.

Now that we have our test double prepared, we can update the test itself.

We need to instruct Laravel that, whenever we attempt to instantiate a `NewsletterSubscriptionInterface` instance, we should use our new `NewsletterSubscriptionFake` rather than the `NewsletterSubscriptionService` that would usually be returned. We can do this by using the `swap` method in our test. This would result in our test looking something like this:

```
use App\Interfaces\NewsletterSubscriptionInterface;
use App\Services\NewsletterSubscriptionFake;

class NewsletterSubscriptionControllerTest extends TestCase
{
    /** @test  */
    public function success_response_is_returned()
    {
        $fake = new NewsletterSubscriptionFake();

        $this->swap(
            NewsletterSubscriptionInterface::class,
            $fake
        );

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

This means that when our test is run, the email address that we're passing (`mail@ashallendesign.co.uk`) should be stored in the `subscribers` field in our test fake. We can then use the `assertSubscription` method we added to our fake to assert that this is true:

```php
use App\Interfaces\NewsletterSubscriptionInterface;
use App\Services\NewsletterSubscriptionFake;

class NewsletterSubscriptionControllerTest extends TestCase
{
    /** @test  */
    public function success_response_is_returned()
    {
        $fake = new NewsletterSubscriptionFake();

        $this->swap(
            NewsletterSubscriptionInterface::class,
            $fake
        );

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

        $fake->assertSubscription('mail@ashallendesign.co.uk');
    }
}
```

That's it! You've now successfully used an interface to improve the testability of this controller method. This approach is powerful because it allows us to decouple our code from concrete implementations and use interfaces instead. It also encourages you to write unit tests to individually test the methods in your `NewsletterSubscriptionService` while having confidence that the service won't be run in your feature tests.

It's worthwhile remembering that this isn't the only approach you can take. We can also use mocks, but these can be a little cumbersome when working with feature tests and can be a maintenance burden whenever your application code changes. This is because mocks require you to write expectations for all the methods that are being called. This means that if the order of the methods being called changes in the future, it could lead to the expectations also needing to be updated. I admit that I have used **a lot** of mocks in the past when writing tests. Over time, a lot of these tests became technical debt that were difficult to maintain when a feature was changed. Looking back, I think a large number of the issues I faced would not have happened if I used more fakes in places better suited to them than mocks. However, mocks can be very handy for isolating dependencies and writing unit tests to test an individual method.

Note that your test doubles may become complicated depending on the types of assertions you want to perform. If this is the case, you may want to consider writing several unit tests to assert that your fakes actually assert as expected. Otherwise, you may risk having tests that pass when they should have failed and flagged a bug in your code.

Using Objects Over Arrays

In PHP, arrays are both a blessing and a curse. They are really flexible and you can pretty much put anything you want in them, which makes them great for moving data around your application's code.

However, misusing them can make your code more difficult to understand and can make it unclear what data they contain. Additionally, arrays can make it more difficult to benefit from features such as type hinting.

For these reasons, I like to use classes such as "data transfer objects" instead of arrays where possible. Using these allows you to benefit from PHP's type system, get better autocomplete suggestions from your IDE, and reduces the chance of bugs.

To understand the advantages of using classes instead of arrays, let's look at an example. Imagine we have a web application with an endpoint for creating a new `User`.

Assume that the endpoint accepts the following fields: `name`, `email`, and `password`. The controller may look something like so:

```php
use App\Actions\StoreUserAction;
use App\Requests\StoreUserRequest;
use Illuminate\Http\Request;

class UserController extends Controller
{
    public function store(
        Request $request,
        StoreUserAction $storeUserAction
    ) {
        $validated = $request->validate([
            'name' => 'required|string|max:255',
            'email' => 'required|email',
            'password' => 'nullable|string|min:8',
        ]);

        $storeUserAction->execute($validated);

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

In the controller method, we're validating the data that is passed in from the request and then we pass the validated data (as an array) to the `StoreUserAction` object. The `StoreUserAction` class may look like this:

```php
use App\Models\User;
use App\Jobs\SendWelcomeEmail;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;

class StoreUserAction
{
    public function execute(array $userData): User
    {
        $password = Hash::make(
            $userData['password'] ?? Str::random()
        );

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

        $user->generateAvatar();

        dispatch(new SendWelcomeEmail($user));
    }
}
```

In the action class, we create a new user, generate an avatar for them, then send them a welcome email.

Because we're passing an array to the `execute` method, we have several issues:

- It's difficult to know exactly what data is available to use in the method without checking the other parts of the codebase where this code is called.
    - We can assume the `name`, `email`, and `password` fields are all available, but we can't be 100% sure.
    - There might be other fields available in the array that we aren't aware of.

- We aren't sure of the data types of the fields.

To resolve these issues, we could update the `execute` method to accept a data transfer object instead.

We'll start by creating a `NewUserData` class in an `app/DataTransferObjects` directory. We'll then make use of PHP 8.0 and 8.1 features such as constructor property promotion, named arguments, and readonly properties to create our `NewUserData` class like so:

```php
class NewUserData
{
    public function __construct(
        public readonly string $name,
        public readonly string $email,
        public readonly ?string $password,
    ) {
        //
    }
}
```

Using constructor property promotion and readonly properties, we can avoid having to create "setter" or "getter" methods and don't need to assign the property values inside the constructor. So we can have a simple object that is immutable and doesn't need to be updated after instantiation.

We can now update our `StoreUserAction` class to use the DTO rather than the array:

```php
use App\DataTransferObjects\NewUserData;
use App\Models\User;
use App\Jobs\SendWelcomeEmail;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;

class StoreUserAction
{
    public function execute(NewUserData $userData): User
    {
        $password = Hash::make(
            $userData->password ?? Str::random()
        );

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

        $user->generateAvatar();

        dispatch(new SendWelcomeEmail($user));
    }
}
```

We then want to update the controller method to make use of the DTO rather than the array:

```php
use App\DataTransferObjects\NewUserData;
use App\Actions\StoreUserAction;
use Illuminate\Http\Request;

class UserController extends Controller
{
    public function store(
        Request $request,
        StoreUserAction $storeUserAction
    ) {
        $validated = $request->validate([
            'name' => 'required|string|max:255',
            'email' => 'required|email',
            'password' => 'nullable|string|min:8',
        ]);

        $storeUserAction->execute(new NewUserData(
            name: $validated['name'],
            email: $validated['email'],
            password: $validated['password'],
        ));

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

Making these changes has improved the readability and maintainability of our code, and has made use of PHP's type system to improve the type safety of our application.

Using the DTO also provides a more predictable method signature and decouples the method from the request data, making it much easier to reuse the action class. For example, if we were to add an API route for creating new `User` models that had slightly different names for the fields (such as `email_address` instead of `email`), we could map the data to the `NewUserData` object and be confident that the action class would still work as expected.

To take this approach one step further, we could even use a form request class and lift the DTO creation code out of the controller. We could create a new `StoreUserFormRequest` like so:

```
use App\DataTransferObjects\NewUserData;
use Illuminate\Foundation\Http\FormRequest;

class StoreUserRequest extends FormRequest
{
    public function rules(): array
    {
        return [
            'name' => 'required|string|max:255',
            'email' => 'required|email',
            'password' => 'nullable|string|min:8',
        ];
    }

    public function toDto(): NewUserData
    {
        return new NewUserData(
            name: $this->validated('name'),
            email: $this->validated('email'),
            password: $this->validated('password'),
        );
    }
}
```

We can then update our controller to use the new `StoreUserRequest` by type hinting it as a method signature like so:

```php
use App\Actions\StoreUserAction;
use App\Requests\StoreUserRequest;
use Illuminate\Http\Request;

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

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

By using the `StoreUserRequest`, we've cleaned up our controller's code, improving readability by lifting the validation and DTO creation into the request class.

Final Words

That's it. All done!

I hope you enjoyed reading Battle Ready Laravel as much as I enjoyed writing it. I really do hope you've learned some new things and that you feel more confident as a web developer.

If you have any feedback on the book, feel free to reach out to me:

- Website - ashallendesign.co.uk
- Email - mail@ashallendesign.co.uk
- GitHub - github.com/ash-jc-allen
- Twitter - twitter.com/AshAllenDesign
- LinkedIn - linkedin.com/in/ashleyjcallen

I want to give a huge thank you to Jess (jesspickup.co.uk) for creating the amazing front cover for the book!

I also want to say a huge thanks to JD Lien for proofreading the book and spotting all the mistakes I'd made.

Keep on building awesome stuff!

Discount Codes

Thanks to the generosity of the teams over at Spatie, Style CI, and Laravel Security In Depth, you have access to limited edition discount codes that you can use with Flare, Oh Dear, Style CI, and Laravel Security In Depth.

Flare

€10 EUR discount for the first 3 months of new Flare subscriptions.

URL: https://flareapp.io

Discount code: **BATTLETESTEDLARAVELFLARE**

Oh Dear

€5 EUR discount for the first 3 months of new Oh Dear subscriptions.

URL: https://ohdear.app

Discount code: **BATTLETESTEDLARAVEL**

StyleCI

20% discount for the first payment of new "Startup" or "Premium" StyleCI subscriptions.

URL: https://styleci.io

Discount code: **BATTLE20**

Laravel Security In Depth

25% discount for the first 1 year of new subscriptions.

URL: **https://larasec.substack.com/BattleReadyLaravel**