
BEST PRACTICES FOR LARAVEL ENTERPRISE APPLICATIONS
WENDELL ADRIEL

Table of Contents

Introduction

When embarking on the journey of building `Enterprise Applications`, the choice of the `Tech Stack` can be one of the most critical decisions you make. In the vast ecosystem of PHP frameworks, `Laravel` has climbed to the top of the ladder. But what makes `Laravel` such a compelling choice for Large-Scale, Enterprise-Level applications?

Let's start with Laravel's popularity. `Laravel` is the most starred PHP framework on `GitHub`, which is not merely a vanity metric: it's a signal of widespread adoption and industry trust. When technology professionals at Fortune 500 companies and lean startups alike turn to the same framework, you know we have something special there.

Backing this popularity is an active and vibrant community. Laravel's users are passionate and they love to share new insights and solutions. What this means for you in the enterprise context is simple: if you run into a problem, chances are someone else has already solved it, blogged about it, or built a package for it. This not only accelerates development but also dramatically lowers the risk of getting stuck on obscure bugs.

Speaking of packages, Laravel's ecosystem of packages is not just "a lot", it's an embarrassment of riches. Need robust authentication and authorization? `Laravel Breeze` and `Laravel Jetstream` have you covered. Want to add real-time features? `Laravel Echo` is there. Need to add payments? `Laravel Cashier` has your back. If `Laravel` does not have a first-party package for you, you'll probably find a Spatie package for it. This package ecosystem let the enterprise developer focus on the `Core Business`, less wheel-reinventing.

But `Laravel` is not just a collection of code. It is a mindset and a complete ecosystem. Tools like `Laravel Forge` for server management, `Laravel Cloud` and `Laravel Vapor` for serverless, `Laravel Nigthwatch` for application monitoring are just some of the products it offers and providing you with everything that you need to go from development to production.

In this book, we are going to dive deep into what it actually takes to build and maintain `Laravel` applications at scale. This is not going to be yet another compilation of abstract design principles, here you'll find real-world, hands-on examples: patterns, pitfalls, and lessons learned from working in and with enterprise teams across various industries.

After more than a decade of experience working with `Enterprise Applications`, I have accumulated a collection of case studies and practical approaches. These will form the backbone of this book, giving you not only theoretical best practices but grounded, actionable strategies you can tailor to your own projects.

So, whether you're an Architect, Developer or a Project Manager, this book aims to provide you a toolkit for success with `Laravel` in the enterprise. Let's get started.

What is an Enterprise Application

Before we can fully deep dive in the best practices for `Laravel` in the enterprise context, we first need to define what we actually mean by an **`“Enterprise Application”`**. This definition is more nuanced than simply saying “an app used by a big company.” Instead, enterprise applications share a range of characteristic traits that set them apart from smaller or more specialized solutions.

At its core, an `Enterprise Application` is a software system designed to solve complex problems, automate demanding workflows, or provide comprehensive services for organizations at scale. Let’s break down some of the essential attributes:

Scale and Complexity

Enterprise applications are built to handle large volumes of data and serve potentially thousands (or even millions) of users. They often span multiple business units or departments and must integrate with other internal and external systems. Complexity may arise from intricate business rules, layered permissions, or cross-domain workflows.

Mission-Critical Reliability

These applications are the backbone of the business. If an enterprise application fails, it can mean lost revenue, regulatory penalties, and lost trust. Uptime, redundancy, and robust error handling are table stakes.

Security and Compliance

Security cannot be an afterthought in the enterprise world. These applications frequently manage sensitive or regulated data—think financial transactions, personal information, or industry-specific records. Compliance with standards like GDPR, HIPAA, PCI DSS, or SOC 2 may also be necessary.

Integration and Extensibility

Enterprise applications usually communicate with other systems: CRMs, ERPs, legacy apps, APIs, third-party services, and more. Extensibility means the system can be adapted and integrated as business needs evolve.

Longevity and Maintainability

Contrary to many web applications that are written for short-term use and quick pivots, enterprise applications are designed for years or even decades of operation. Maintainable code, clear architecture, solid documentation, and rigorous testing become not just best practices, but critical requirements for long-term success.

Collaboration and Access Control

Last but not least, enterprise applications typically support a wide variety of users with different roles and permissions. Advanced access control and auditability are vital for both security and operational clarity.

Conclusion

As we saw above, an enterprise application is characterized by its scale, critical importance, security demands, integration capabilities, long lifespan, and complex access controls. Understanding these characteristics shapes every architectural and design decision we make—including why selecting the right framework, such as `Laravel`, is so vital for enterprise success.

Throughout this book, we'll use this definition to frame our best practices, recommendations and real-world case studies to learn how to face the unique challenges of enterprise development.

Preparing Your Application

Before you write your first line of code, preparing your `Laravel` application for enterprise development is the key to success that you cannot afford to skip. The foundation you lay at the beginning will echo throughout the project's lifecycle—impacting maintainability, security, and even your team's day-to-day velocity. In this chapter, we'll look at how to set yourself up for success from day one.

We'll start by exploring best practices for organizing your `Project Structure`: moving beyond default conventions to create a codebase that scales as your business grows. Next, we'll discuss why it's crucial to `Choose Wisely Your Dependencies`: each package you install may introduce both opportunities and risks. Finally, we will show you how to `Keep an Eye for Vulnerabilities`: ensuring your application is resilient against ever-evolving security threats from the start.

A bit of careful preparation here pays off down the road. Let's dive in.

From Cozy to Scalable: Adapting Laravel's Structure for Growth

Laravel’s default project structure is amazing: minimalist, yet elegant and powerful. Everything has it's place and with the help of the awesome commands from `Artisan` you can build everything that you need quickly. The structure offers an easy learning curve for newcomers, while supporting rapid prototyping for experienced developers.

This structure works exceptionally well for small to medium-sized projects. The conventions are clear, onboarding is straightforward, and the default organization aligns naturally with Laravel’s philosophy, allowing you to focus on delivering features without overthinking architecture.

However, once your application begins to grow into enterprise territory, you can start to see some issues. `Enterprise Applications` often involve dozens, sometimes hundreds, of Models, Controllers, Services/Actions, Events, etc. Suddenly the once-cozy `/app/Models` directory (and many others) start to grow like Laravel's popularity, and you may notice challenges such as name collisions, difficult navigation, and confusion about where specific logic should reside.

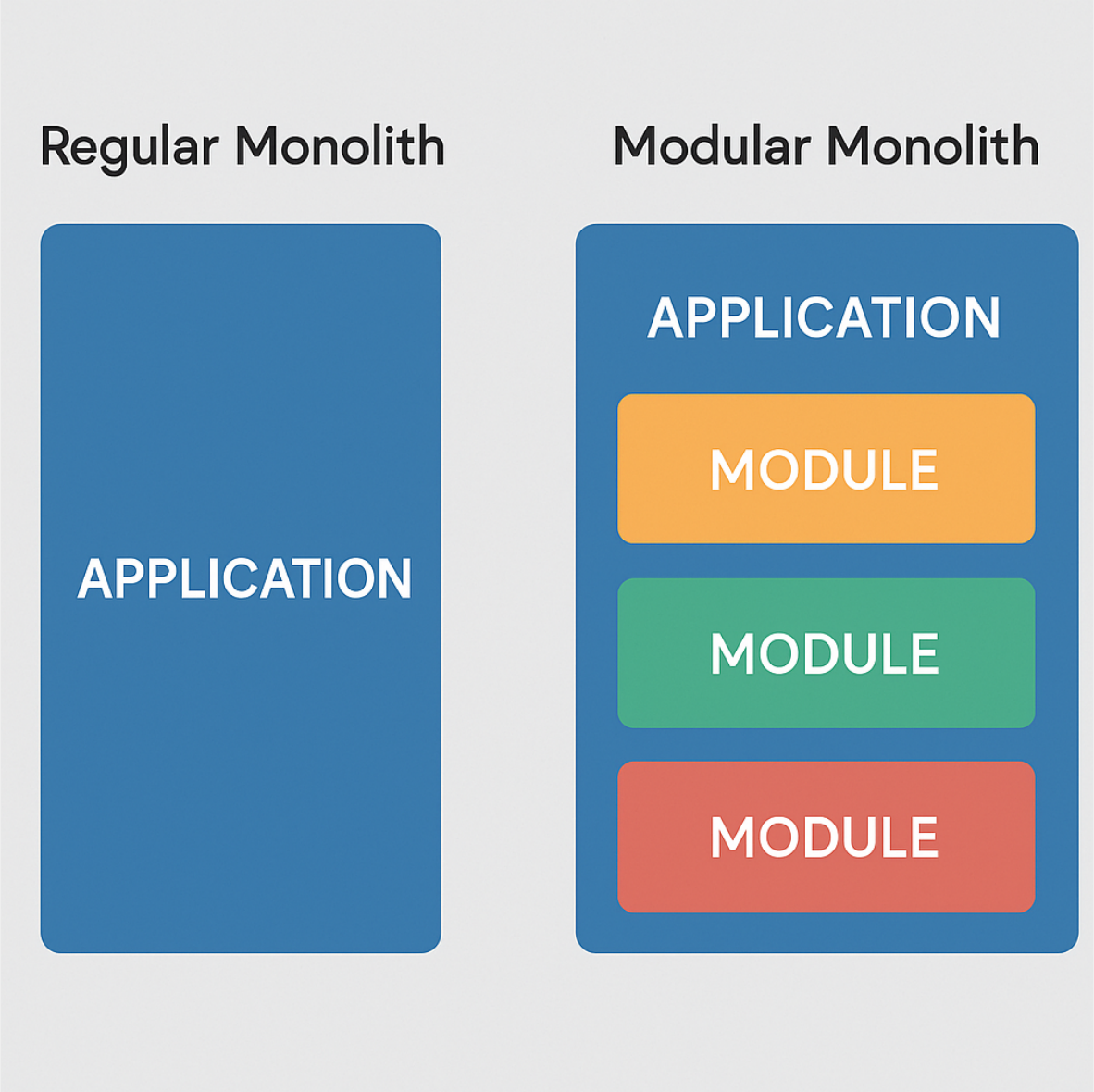
When you get to this point, it's time to think on another solution. Not to replace the default structure, but to extend it. And that's when the `Modular Monolith` enters the fight.

What is a Modular Monolith

A `Modular Monolith` is an architectural style where your entire application is deployed as a single codebase (the “Monolith”), but the internal code is structured into well-defined, independent modules. Each module encapsulates a specific business domain or a feature. For example: “Billing”, “User Management”, "Mailing", etc. Each of these modules have a structure of a normal application, having its own Models, Controllers, Actions, Events, Routes and Views.

Using this approach, you can keep the boundaries between modules clear, with a well-defined structure for each of the modules, improving a lot the DX and the maintainability of the application in the long run.

This approach also offers many of the organizational benefits of Microservices like separation of concerns and code ownership, but removes all the complexity of managing distributed systems.



Transforming Your Application Into a Modular Monolith

Now that we understood what is a `Modular Monolith`, let's understand how we can apply this approach into our applications.

There are many ways we can apply this approach, but I'm going to list the two ones I've seen more being used in `Laravel Enterprise Applications`:

Using the `app` Directory

For this approach, you would add a directory, usually named as `modules`, `domains` or `features` inside the `app` directory. With this approach you'll have a namespace like `App\Modules|Domains|Features`.

This can feel more natural for most developers that are used to have the whole source code

in the app directory, but like everything else, it also has some trade-offs. You'll need to remove all the "common" directories from the default structure like app/Http, app/Models to a new place. For things that are "common" or "core" components like the AppServiceProvider, you can either leave it on the app/Providers or you can create a Common or Core module where these things will live, if you do that, you need to remember to change from where Laravel loads the Service Providers from:

> **Notice:** For Laravel versions 12 and 11 this can be done changing the bootstrap/providers.php file. For Laravel versions 10 or below this can be done changing the config/app.php file.

A great benefit from this approach is that it needs minimal configuration, and you can use all Artisan commands out-of-the-box.

Using a Standalone Directory

For this approach, you would add a directory in the root of the repository, outside the app directory. With this approach you'll have a namespace like Modules|Domains|Features.

This can be a little weird at first for developers that never worked with a structure like this, but it also brings some great things like:

- You keep your application code totally separate from Laravel's code.
- It's much easier to upgrade versions because Laravel's files are in a separate directory.

But you know that everything has a trade-off, and when going with this approach you need to do a lot of changes, and you can't use the Artisan commands out-of-the-box. However, as you'll see below, there are packages to help with this.

The Module Structure

Now that you've seen two approaches for transforming your application into a Modular Monolith and hopefully you chose which one you want to use, you must be thinking how these modules should be structured.

Again, there are a lot of ways for doing this. For example, I created a Modular API Skeleton for Laravel called ExA without using any packages for that. You can check it here. Another common way is treating each module as a package and following the guidelines for package development from Laravel.

As said above, the approach of having the modules outside the app directory brings a lot of work that needs to be done, but if you remind about the introduction chapter, Laravel community and its package ecosystem are AMAZING, and I leave below two packages that are actively maintained and that you can use to quickly transform your Laravel application

into a `Modular Monolith`:

- Laravel Modules: my go-to choice when working with `Modular Monolith` applications in `Laravel`. Is a complete package with tons of commands that brings all the goodies from default `Artisan` commands to your `Modular Monolith` application and it's easy to set up and configure.

- Modular: another great package for `Modular Monolith` applications in `Laravel`. It's a more lightweight package, that sticks more to the `Laravel` conventions and instead of adding new commands to create files, it adds a new `--module` flag to the `make:*` commands.

REAL-WORLD CASE: Modular Monolith 1 x Microservices 0

One company that I worked on had an "ancient" legacy project built in `CodeIgniter 1`, and they wanted to build a new modern version. The person in-charge for this assembled a team and spent almost 2 years building a new version with `Microservices` with `Laravel`. I joined the company in the middle of the process and helped to implement some of these services.

The platform that before was a `Monolith` with `CodeIgniter 1` transformed in nearly `20 Microservices` with `Laravel`. Each `Microservice` on they own didn't have big issues. The issues started to show up when everything was "glued" together. Not to deviate from the topic, I'll just share the outcome of this choice:

- `Costs went up:` the costs for the infrastructure were much higher than before.

- `Performance went down:` for some actions from the user, the request had to reach 5 to 10 different Microservices, adding that the company dealed with `MILLIONS` of records per day, the result was that in general the platform was much slower than the `Monolith` version.

- `Team Velocity went down:` the team that was used to handle one single codebase and one single deploy had to start managing 20 codebases and 20 deploys. The result was that the team felt overwhelmed and even some simple debugging in local turned into a journey for being able to debug an issue across multiple Microservices.

At the end, the Product Leadership decided to shut down the project, A LOT OF MONEY was lost, imagine paying a team for 2 years just to see that things didn't work. Another person was put in-charge to create a new version, at the time I was already working with this person, and he asked me if I wanted the challenge to create a new architecture for the platform, I already had experience with `Modular Monoliths` before, I don't even think

this term was used at the time, but this gave me a huge chance to make a difference and shine at the company.

Nowadays, this `Modular Monolith` in `Laravel` is still being used in production. The application has more than 300 endpoints, dozens (or even hundreds) of background jobs run daily, and it handles millions of records for reporting, analytics and a lot of different features without any struggle.

`LESSON LEARNED:` Keeping simple may work much better than overthinking and overengineering your solutions.

Conclusion

In conclusion, the way you structure your `Laravel` application lays the foundation for long-term success, especially in the context of `Enterprise Applications`. While Laravel's default structure is a great starting point, adapting it to a `Modular Monolith` approach can bring much-needed clarity, scalability, and maintainability to your codebase while avoiding the complexity that `Microservices` would add.

Dependency Decisions: Choosing with Care and Caution

In any software project, dependencies play a crucial role. They can save time and effort by providing pre-built functionalities, but choosing them carelessly can cause more harm than good. So, let's dive into why it's important to select dependencies wisely and what factors to consider during this process.

When you choose dependencies, you're deciding to trust third-party code in your project. These pieces of code can introduce vulnerabilities, impact performance, or even become a roadblock in the future. A careful selection helps ensure that the dependencies add value and align with your project needs.

Dependencies and Security

Security should be at the top of your checklist when selecting dependencies. Since they are third-party code, you have limited control over them. Always perform a security assessment to ensure that the dependency doesn't have known vulnerabilities. Tools like `composer audit` and `npm audit` for PHP and JavaScript can be very helpful. Additionally, evaluate the track record of the maintainer, how quickly do they respond to security issues? An abandoned package can leave your project exposed.

Dependencies and Budget

When evaluating the cost of implementation, it's essential to compare the time and money spent building a feature from scratch against using a dependency. This involves a careful assessment of both explicit and implicit costs.

Start by estimating how long it would take to implement the feature yourself. Consider the complexity and specific requirements of the feature. Compare this with how quickly you can integrate an existing package that provides similar functionality. The cost-benefit analysis should include developer hours, testing, and potential revisions.

While dependencies can save time, they come with implicit costs:

- `Learning Curve:` some packages might be complex, requiring developers to invest time understanding how to use them effectively.
- `Integration Challenges:` dependencies might not fit seamlessly into your existing architecture, leading to additional time spent on integration.
- `Updating and Upgrading:` regular updates might be necessary to keep up with security patches or new features, requiring ongoing maintenance efforts.

Understanding these implicit costs helps you decide whether a dependency is worthwhile. If

a package significantly reduces the time to launch but introduces substantial maintenance overhead, it might not be the right choice.

For an `Enterprise Application`, maintaining the balance of immediate benefits with long-term implications is a MUST.

Dependencies and Maintainability

When dealing with open-source dependencies, maintainability is something to look out to. Look for packages that have active communities, regular updates, and clear documentation. This ensures that any bugs or issues are likely to be resolved and that the package will continue to evolve alongside your project. Avoid dependencies that appear abandoned, as this can lead to difficulties if your project needs updates or bug fixes down the line.

Dependencies and Version Updates

Your project will evolve, and so will the dependencies. Evaluate how often a package gets updated and how those updates are managed. Packages with frequent breaking changes can cause significant rework every time an update is needed. It's wise to choose those that follow semantic versioning, as it helps you anticipate the impact of updates. Check if the dependency provides a clear upgrade path and if it supports backward compatibility.

Conclusion

In the realm of `Enterprise Applications`, choosing dependencies is a strategic decision that goes beyond technical considerations. It's about aligning with the long-term goals and complexities of the project. By prioritizing security, assessing implementation costs, and ensuring maintainability and smooth updates, you can turn dependencies into valuable assets. A careful approach today can save countless hours and resources in the future, ensuring that your application remains robust, efficient, and scalable.

Shield Your Code: Vigilance Against Vulnerabilities

When developing **Enterprise Applications**, security is a top priority. Vulnerabilities can lead to serious issues like data breaches and unauthorized access.

For PHP projects using **Composer**, you can use the **composer audit** tool. It checks your PHP dependencies for known vulnerabilities and provides guidance on how to update or patch them. Similar to that, if you use Node.js, **npm audit** does the same.

Laravel is known for its strong emphasis on security. Right out-of-the-box, it provides several features that help protect your application. Let's go over some of these features in a quick summary:

- **CSRF Protection:** Laravel automatically protects your application from Cross-Site Request Forgery (CSRF) attacks. It does this by generating a token for each active user session. This token must match what's submitted with any form on your site, ensuring that requests are from legitimate sources.

- **SQL Injection:** both Eloquent ORM and the Query Builder protect against SQL Injection. They use parameter binding, which makes manipulating queries with malicious data much harder for attackers.

- **XSS Protection:** with Laravel's Blade templating engine, output from your views is automatically escaped. This means if any malicious scripts are injected into your forms or data, they are neutralized before being executed in the browser.

- **Encryption:** Laravel offers easy-to-use functions for encrypting data. It uses the secure OpenSSL and AES-256-CBC encryption, ensuring that sensitive data remains safe and private.

OWASP

OWASP stands for the Open Web Application Security Project. It's a nonprofit organization dedicated to improving software security. They create free tools, resources, and documentation to help developers protect their applications.

OWASP releases a list called the OWASP Top Ten. This list highlights the most critical security risks for web applications. You can use it as a guide to understand potential vulnerabilities in your code.

CVEs

CVE stands for Common Vulnerabilities and Exposures. It's a list of publicly disclosed

cybersecurity vulnerabilities. Each vulnerability is assigned a unique identifier, making it easier to track and address.

You can search for CVEs using the National Vulnerability Database. Regularly checking for new CVEs related to the software you use can help you stay informed and patch vulnerabilities quickly.

Zed Attack Proxy (ZAP)

ZAP is a free and open source Web App Scanner tool. It has a huge community, and it's actively maintained. You can use it to help you find security vulnerabilities automatically in your web applications.

Keep Everything Updated

One important best practice for keeping your any application secure and efficient is regularly updating your dependencies, the framework, and even the language itself.

Every update often comes with security patches. These fixes are crucial because attackers always look for weaknesses. If you're not updating, you're leaving the door open for potential threats. Newer versions also bring performance improvements. This can make your application faster and more efficient, which is always a good thing. Not only that, you can use the full potential of dependencies, the framework and the language when you keep up with the latest releases.

> **Notice:** When it comes to major releases, I usually wait between one to three months before updating. This gives time for any early issues and bugs to be discovered and fixed.

REAL-WORLD CASE: Docker Vulnerabilities Hell

Using containers for deploying applications has become really common, especially for features like auto-scaling. `Docker` is well-known and widely used for this purpose. However, I once worked at a company that faced many vulnerabilities due to a simple but critical issue in the `Dockerfile` configurations.

`AWS Inspector`, a tool from AWS that scans servers, container images, and other resources, found vulnerabilities in many images. These included critical ones already addressed in security patches. Why did these images still have vulnerabilities? It was because many `Dockerfile` files missed these two simple commands:

```
RUN apt-get update && apt-get upgrade -y
```

LESSON LEARNED: A small oversight can cause significant security issues. Always pay attention to the details.

Level Up Your Application

Building an `Enterprise Application` with `Laravel` involves more than just writing code. It’s about creating a high-quality, robust solution that can handle real-world challenges. In this chapter, we'll explore proven techniques that will help you take your application to the next level.

You'll learn how to implement best practices that are essential for crafting powerful, efficient, and scalable `Enterprise Applications`. We'll delve into strategies that have been successfully used in real-world projects that I worked on, giving you practical insights you can apply right away.

Get ready to enhance your skills and knowledge as we deep dive in techniques to elevate your `Laravel` applications to meet the demands of enterprise environments.

Elevate Your Code with Linters and Static Analyzers

Tools like Linters and Static Analyzers are useful for any application, but for **Enterprise Applications**, they are essential. They help maintain high-quality code, ensure standards, and catch bugs before they reach production.

In the **PHP** and **Laravel** ecosystem, there are many tools available. In this book, we'll focus on two that are great for any **Laravel** application: **Pint** and **LaraStan**.

Pint

Pint is a code style fixer built on top of the well-known PHP-CS-Fixer. It offers some updates, like configuration with a simple **JSON** file and a **Laravel** preset that follows Laravel's own standards and conventions.

In **Enterprise Applications**, where many developers work on the same codebase, having consistent code is even more vital. It makes collaboration smoother and reduces misunderstandings. With **Pint**, you get several benefits, being the main ones:

- **Consistency:** ensures uniform coding standards across your application and teams.
- **Readability:** cleaner code, making it easier to understand and work with.

The stricter your rules are defined in **Pint**, fewer errors can be made and less time can be spent in Code Reviews.

I have strong opinions on how my code should look. I have a "template" of a **Pint** configuration file that I use for many projects and adjust it as needed depending on the project. I'll share it with you below. Feel free to use it as a and starting point and tweak it to your liking.

```
{
    "preset": "laravel",
    "rules": {
        "array_push": true,
        "assign_null_coalescing_to_coalesce_equal": true,
        "combine_consecutive_issets": true,
        "combine_consecutive_unsets": true,
        "concat_space": {
            "spacing": "one"
        },
        "declare_strict_types": true,
        "explicit_indirect_variable": true,
        "explicit_string_variable": true,
```

```
        "global_namespace_import": true,
        "method_argument_space": {
            "on_multiline": "ensure_fully_multiline"
        },
        "modernize_strpos": true,
        "modernize_types_casting": true,
        "new_with_braces": true,
        "no_superfluous_elseif": true,
        "no_useless_else": true,
        "nullable_type_declaration_for_default_null_value": true,
        "ordered_imports": {
            "sort_algorithm": "alpha"
        },
        "ordered_class_elements": {
            "order": [
                "use_trait",
                "case",
                "constant",
                "constant_public",
                "constant_protected",
                "constant_private",
                "property_public",
                "property_protected",
                "property_private",
                "construct",
                "destruct",
                "magic",
                "phpunit",
                "method_abstract",
                "method_public_static",
                "method_public",
                "method_protected_static",
                "method_protected",
                "method_private_static",
                "method_private"
            ],
            "sort_algorithm": "none"
        },
        "strict_comparison": true,
        "ternary_to_null_coalescing": true,
        "use_arrow_functions": true
    }
}
```

LaraStan

LaraStan is an extension for PHPStan, a powerful Static Analysis tool for `PHP` applications. Being more precise, `LaraStan` is considered a Code Analysis tool, because it boots the application's container for being able to resolve types that are only possible to check during runtime.

For `Enterprise Applications`, which are often large and complex, finding issues early saves time and effort. `LaraStan` helps keep the application running smoothly by:

- `Catching Bugs Early:` scans and finds both obvious and tricky bugs before they reach production.
- `Better Maintenance:` makes it easier to keep code clean and bug-free.

A great feature of `PHPStan` and `LaraStan` is the ability to set a `Rule Level`. Even if you already have a large and complex codebase, it's easy to integrate it without being overwhelmed by the amount of errors on your first scan.

> **Notice:** For existing applications, I suggest starting at Level 1. Once you tackle all the issues, move to the next level. For new applications, start at least on Level 5. If your team is up for a challenge, consider starting at Level 8.

From Dynamic to Disciplined Types

PHP is a dynamically typed language, meaning the type of a variable is determined at runtime, so you don't need to specify it. However, this doesn't mean we shouldn't use types. Using types consistently can save us from headaches and hours of debugging. That's why we should use types whenever possible. In recent years, especially starting from version 7, PHP has increasingly adopted types.

Using types in your Enterprise Applications is crucial, it makes your application more robust and reliable, which is essential for enterprise-level software, here are some reasons why you should be using types everywhere:

- Improve Code Clarity: types make your code easier to understand. When you specify the type of a parameter or return value, it’s clear what kind of data is expected. This helps everyone on your team, especially in large projects.
- Reduce Bugs: types catch mistakes early. By knowing what kind of data should be handled, PHP can alert you to errors before they cause problems in production.
- Enhance Maintenance: typed code is easier to maintain. As your application grows, clearly defined types help ensure that changes don't introduce new errors. This is especially vital in enterprise settings where many developers collaborate.

> **Notice:** When you declare types in PHP, it ensures that the value is of the specified type at call time. If the value type is different, PHP will throw a TypeError.

Strict Typing

Even when using types, the default behavior of PHP is to try converting values of the wrong type into the expected scalar type when possible. For example, if you pass an int to a function expecting a string, PHP will automatically convert the int into a string.

```php
function hello(string $name): void
{
    echo "Hello, {$name}";
}

hello(3); // "Hello, 3"
```

If you want PHP to be stricter with types, you can disable this behavior by enabling its Strict Mode. You can do this by adding the following line at the beginning of your files:

```
declare(strict_types=1);
```

With the Strict Mode enabled, our last example would throw a TypeError instead:

```
declare(strict_types=1);

function hello(string $name): void
{
    echo "Hello, {$name}";
}

hello(3); // Fatal error: Uncaught TypeError: hello(): Argument #1
($name) must be of type string, int given
```

The Pint configuration shared earlier in this book includes a rule to enable the Strict Mode for all files that haven't yet enabled it: "declare_strict_types": true.

> **Warning:** Strict Mode is enabled on a per-file basis and is only defined for scalar type declarations. It applies to function calls made from within the file with Strict Mode enabled, not to the functions declared in that file. This means if a file without Strict Mode enabled calls a function defined in a file with Strict Mode, PHP will still try to convert the value.

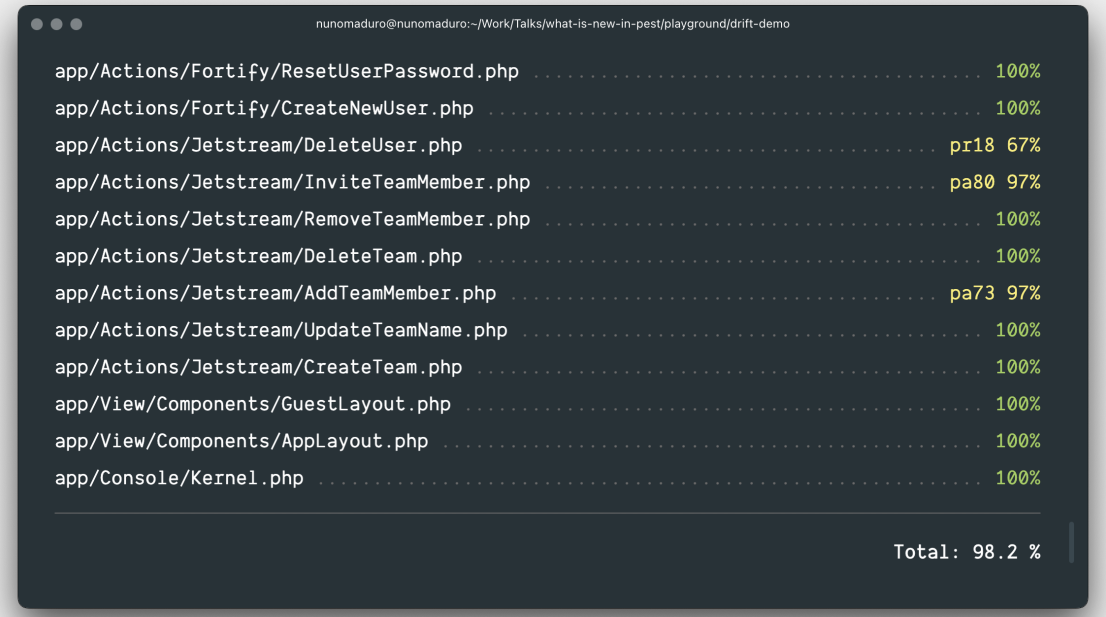
Type Coverage

To keep track of type usage in your applications, Pest offers a Type Coverage plugin that's easy to install and use:

```
composer require pestphp/pest-plugin-type-coverage --dev
./vendor/bin/pest --type-coverage
```

This plugin doesn't require any changes to your application or additional tests. It scans your codebase and generates a report on type coverage, showing a list of files with their coverage percentage and highlighting line numbers where type declarations are missing.

The screenshot below is from the `Type Coverage` plugin documentation page on the `Pest` website.



Data Handling with DTOs: A Path to Cleaner Code

In **Enterprise Applications**, managing data efficiently is critical. **Data Transfer Objects** (DTOs) are a design pattern that can help ensure data is handled consistently across your application.

DTOs act as containers for data, transferring it between different parts of the application without exposing sensitive information and giving context to the data. This makes your code cleaner and more organized, reducing the risk of errors and improving maintainability.

What are DTOs

Data Transfer Object (DTO) is a **Design Pattern** that's used to transfer data between different parts or layers of an application. Usually it's a simple object that contains a set of data represented by properties, that represents a specific entity or concept in the application. The main purpose of a DTO is to decouple the different layers or components of an application, allowing them to communicate and exchange data without needing to know the details of each other's implementation.

Why to use DTOs

Enterprise Applications usually have dozens or even hundreds of different features and actions. Also, we have multiple people implementing and maintaining an application. Having standards to create a good data structure for the application is critical for maintaining the quality of the code, improving the DX and the maintainability of it.

Let's check a really simple example on how DTOs can improve our applications.

```
class UserController extends Controller
{
     public function store(Request $request, CreateUser $action):
JsonResponse
    {
        return response()->json([
            $action->handle($request->all()),
            Response::HTTP_CREATED
        ]);
    }
}
```

The example above is just a simple action to crete a new user in the application, but even that simple action can get tricky if I created it one year ago, and now I have to change the logic. I probably don't remember the structure of the data from the **$request->all()**

anymore.

One way of solving this in Laravel is by creating a FormRequest class and then using it instead.

```
class UserController extends Controller
{
     public function store(CreateUserRequest $request, CreateUser
$action): JsonResponse
    {
        return response()->json([
            $action->handle($request->validated()),
            Response::HTTP_CREATED
        ]);
    }
}
```

With the code above, if I need to update the CreateUser I need to check the CreateUserRequest class to check what's the expected input. It already helps a little, but it still has some issues like the $request->validated() returns an array, we can't type the properties and while working with the code if we forgot any property, we would still need to check in the FormRequest class for it.

Applying the DTO pattern here we would solve these issues because we can map all the data as properties in it and we would get all the goodies like auto-completion in our IDE/Code Editor.

How to Use DTOs

A DTO can be a dead simple object, just to be a container to map the properties. The most basic implementation of a DTO would be something like this.

```
class CreateUserDTO
{
    public function __construct(
        public string $name,
        public string $email,
        public string $username,
        public string $password
    ) {}
}
```

But we can still improve this by making our **DTOs** as **readonly**, because immutability also brings another layer of security for your code. But why? With a readonly **DTO** you can make sure that after it's instantiated, the data won't be changing, if you do need to change the data, you'll need to create a new **DTO** instance instead.

```
readonly class CreateUserDTO
{
    public function __construct(
        public string $name,
        public string $email,
        public string $username,
        public string $password
    ) {}
}
```

Now that we have our **DTO**, we can update our example to use it, improving the quality and the maintainability of our application.

```
class UserController extends Controller
{
     public function store(Request $request, CreateUser $action):
JsonResponse
    {
        $request->validate([...]);

        return response()->json([
                                            $action->handle(new
CreateUserDTO(...$request->validated())),
            Response::HTTP_CREATED
        ]);
    }
}
```

Advanced DTO Usage

In the last example, we saw how we could create basic **DTOs** to improve the quality and maintainability of our applications. But we can still take this one step further by using "advanced" **DTOs**. For this we're going to use a package that I've created: Validated DTO, that allows us to create DTOs with features like:

- Data Validation
- Use **DTOs** in your Controllers like a **FormRequest**

- Type Casting

The package has a great documentation that you can get started pretty easily, but I'll show how we could use it to create a `DTO` for our last example.

First, let's install the package.

```
composer require wendelladriel/laravel-validated-dto
```

The package provides an `Artisan` command for creating DTOs:

```
php artisan make:dto CreateUserDTO
```

It's going to generate a `DTO` like this:

```
class CreateUserDTO extends ValidatedDTO
{
    protected function rules(): array
    {
        return [];
    }

    protected function defaults(): array
    {
        return [];
    }

    protected function casts(): array
    {
        return [];
    }
}
```

The `rules` method allows you to define validation rules in the same way you would do with a `FormRequest` class. The `defaults` method allows you to define default values for the properties if they're not set. The `casts` method allows you to define if any properties need to be cast before being set in the `DTO`.

For our example we're going to define the `rules` and the `defaults`.

```
readonly class CreateUserDTO extends ValidatedDTO
{
    public string $name,

    public string $email,

    public string $username,

    public string $password

    protected function rules(): array
    {
        return [
            'name' => ['required', 'string', 'min:3'],
            'email' => ['required', 'email'],
            'username' => ['sometimes', 'string', 'min:3'],
            'password' => [
                'required',
                Password::min(8)
                    ->mixedCase()
                    ->letters()
                    ->numbers()
                    ->symbols()
                    ->uncompromised(),
            ],
        ];
    }

    protected function defaults(): array
    {
        return [
            'username' => Str::snake($this->name),
        ];
    }

    protected function casts(): array
    {
        return [];
    }
}
```

With the `DTO` above we map the fields using public properties in the class, then we define the validation rules we want, and we even add that if the `username` is not passed we're

going to use a `snake_case` version of the `$name` property as the default value.

Now that we have our `DTO` created, we can update our example to use it.

```
class UserController extends Controller
{
     public function store(CreateUserDTO $dto, CreateUser $action):
JsonResponse
    {
        return response()->json([
            $action->handle(new CreateUserDTO($dto)),
            Response::HTTP_CREATED
        ]);
    }
}
```

As you can see, we're replacing the `Request` to use our `CreateUserDTO` instead because the `DTOs` created with this package can be used in this way, and we can have the data validation and the data wrapped in the same place, meaning that if we want to call the `CreateUser` from anywhere in the code, since it expects a `CreateUserDTO`, we will already make sure that the data is `ALWAYS` validated when the action is called.

This is just a simple example of what the `Validated DTO` package can do, you can check the documentation to learn about more features that can help you in your applications.

REAL-WORLD CASE: Array-Oriented Programming Problems

I once worked as a consultant in a project that was not even an `Enterprise Application`, but it was a medium-size project with some complexity and all the Controllers were passing to the service layer a `$request->all()` and all the validation and logic was done at the service level. Almost all the methods had something like this.

```
class BillingService
{
    public function processPayment(User $user, array $params): void
    {
        // CODE HERE
    }
}
```

The main issue was that the application had a lot of issues and making changes or fixing the bugs were taking a long time for the team because everytime they had to debug what was

inside the `$params`. At the end we had to refactor everything to use `DTOs`. It took two months to refactor the whole thing, but after one year that I was not consulting to the project anymore, the product owner reached me out and said that it was the best thing that happened, the amount of issues dropped a lot and the velocity of the team improved.

LESSON LEARNED: Having good standards for your data structure and investing some work on improving it can be a successful investment on the long run.

Boosting Code Organization and Clarity with Actions

In `Enterprise Applications`, keeping your codebase organized and maintainable is key. The `Action Pattern` is a great way to achieve this. By breaking down your application logic into small and reusable pieces, you improve code quality, making it easier to understand and maintain.

What is an Action

Think of an `Action` as a single piece of a puzzle. Each `Action` represents a distinct task or operation, like sending an email or processing a payment. By isolating these tasks into separate classes, you make your code modular and easier to manage and to maintain.

Why to use the Action Pattern

When we apply the `Actions Pattern`, we treat each individual `Action` as a building block. Just like combining puzzle pieces to form a complete picture, we can piece together `Actions` to build complex features. This approach not only enhances readability but also promotes reusability, allowing developers to focus on specific functionalities without the risk of breaking other parts of the application.

When implementing a feature, the developer can create one or multiple different `Action` classes to reach the final goal and then create tests for each of these separate pieces.

How to Implement the Action Pattern

There are multiple ways how you can handle your `Actions`, but the one that I like to use and it has been working well for all the projects so far that are using it is to have a single public method in the class called `handle` where this method will usually receive the logged user and a `DTO`, the return can vary depending on the type of the `Action`.

Let's use a feature of processing a payment as our example and let's see three different approaches depending on the requirements for this feature.

Simple Payment

The first example of requirement is a simple payment that we should only process the payment and send an email confirmation to the user. For this we can use a simple `Action` where the logic will be handled in the `handle` method.

```
class ProcessPayment
{
    public function handle(User $user, ProcessPaymentDTO $dto): void
    {
        // PROCESS PAYMENT AND SEND EMAIL
    }
}
```

Payment and Invoice - Simple

For this second example, we need to process the payment, but we also need to generate an invoice to the user and send it by email. The main point here is that this is the only time we would need to generate an invoice in our application. For this we can still use a single `Action` class, since every time we process a payment we will generate an invoice.

```
class ProcessPayment
{
    public function handle(User $user, ProcessPaymentDTO $dto): void
    {
        $paymentInfo = $this->processPayment($user, $dto);
        $this->generateInvoice($paymentInfo);
    }
     private function processPayment(User $user, ProcessPaymentDTO
$dto): PaymentInfoDTO
    {
        // PROCESS PAYMENT
    }
    private function generateInvoice(PaymentInfoDTO $dto): void
    {
        // GENERATE INVOICE
    }
}
```

As you can see, in this example we're splitting the logic into two different private methods just to keep the code organized and easy to maintain. If someone needs to change, improve or extend this feature, it would be an easier task because of the organization and the usage of the `DTOs`.

Payment and Invoice - Complex

For this third and last example, like the last one, we need to process the payment and generate an invoce for it. However, in this case we may need or not to generate the invoice

and also we may be able to trigger the invoice generation from the dashboard after a payment was already done. For this we're going to create not one, but two different **Actions** since we could use them individually in our application.

```
// ProcessPayment.php
class ProcessPayment
{
          public  function  __construct(GenerateInvoice
$generateInvoiceAction) {}

    public function handle(User $user, ProcessPaymentDTO $dto): void
    {
        // PROCESS PAYMENT

        if ($dto->needsInvoice()) {
            $this->generateInvoiceAction->handle($paymentInfo);
        }
    }
}

// GenerateInvoice.php
class GenerateInvoice
{
    public function handle(PaymentInfoDTO $dto): void
    {
        // GENERATE INVOICE
    }
}
```

As you can see, in this example we're splitting the logic into two different **Actions**, allowing us to use them individually, but also, when calling the **ProcessPayment**, when we need to generate an invoice from it, it will call the **GenerateInvoice** from within. This is like combining two puzzle pieces together to build a more complex solution.

REAL-WORLD CASE: Service Monsters

A typical approach on applications to avoid **Fat Controllers** is creating a **Service Layer**, where the Controllers use a **Service** class to wrap the business logic. I'm sure you already saw this before in some application you worked on.

This is a good way of wrapping the business logic decoupled from the Controllers, but for **Enterprise Applications** these files can get out of hand with time. I already worked in applications that had **Service** classes with more than 5.000 lines of code.

When this starts to happen it can affect a lot the DX for the developers, start creating issues in the application and also making the developers spend more time on maintenance than they should.

In a specific application, the adoption for the `Action Pattern` was something done in parts. The team started to use this pattern for all the new features and whenever they needed to touch the service classes to fix, improve or extend something, they refactored that into an `Action` class. After some months, we saw that the amount of bugs were decreasing and the time spent on maintenance was also going down.

LESSON LEARNED: Starting with small improvements is better than doing no improvements at all. Sometimes we just need to start to see the impact of a small improvement over time.

Empowering Your Code with Enums

In Enterprise Applications, handling fixed sets of related constants is common. Enumerations, commonly known as Enums, provide a clean and efficient way to manage these sets, making your code more readable and less prone to errors.

Enums allow you to define a group of named values, improving clarity and consistency across your application. They help prevent mistakes by ensuring that only valid values are used.

For quite some time, we didn't have native support for Enums in PHP, but in PHP 8.1 we got them added, bringing a lot of possibilities with them.

Starting to use Enums

Basically, an Enum is a data type that works as a collection of predefined constants. Imagine that we're building a Kanban board manager application. For that we can use an Enum to represent the different statuses of our tasks.

```
enum TaskStatus
{
    case TODO;
    case IN_PROGRESS;
    case DONE;
}
```

Before that, a lot of applications had these values set as separate constants, but wrapping them in an Enum adds context to the data, improving the readability, maintainability.

Not only that, we can improve further by using a Backed Enum, that can have a value assigned to each case. Internally they are still represented by objects, but since they are represented by a value, we can easily use this value to serialize the Enum for saving it to the database for example.

```
enum TaskStatus: string
{
    case TODO = 'todo';
    case IN_PROGRESS = 'in_progress';
    case DONE = 'done';
}
```

With Eloquent, we can define a property of our model to be cast as an Enum, so we can do

something like this.

```
class Task extends Model
{
    // Model definition here
    protected function casts(): array
    {
        return [
            ...,
            'status' => TaskStatus::class,
        ];
    }
}

$task = Task::query()->first();
$task->status; // This now will be a TaskStatus instance
$task->status->value; // This will return the string value for the
case
```

Another great thing using `Enums` is that we can use them to validate the data coming from the UI with the `Rule::enum` validation rule.

```
$request->validate([
    'status' => [Rule::enum(TaskStatus::class)],
]);
```

The code above will already validate if the value sent by the UI is among the allowed list. This improves a lot the maintainability and quality for your application. If a new status needs to be added, adding it to the `TaskStatus` enum would be the only change needed, because the validation will always be in sync when using an `Enum`.

Enum Methods

Like a "normal" class, we can also define methods in `Enums`, and that opens up a lot of possibilities, specially when we combine this with the power of the `match` operator.

Imagine that for our Kanban board manager application, we want to show each task in the UI in a different color based in their status and also a human readable label for it. We can do this easily by adding two methods to our `TaskStatus` enum.

```
enum TaskStatus: string
{
    case TODO = 'todo';
    case IN_PROGRESS = 'in_progress';
    case DONE = 'done';
    public function label(): string
    {
        return Str::headline($this->value);
    }
    public function color(): string
    {
        return match($this) {
            self::TODO => 'slate-500',
            self::IN_PROGRESS => 'sky-500',
            self::DONE => 'emerald-500',
        }
    }
}

$task = new Task([
    'name' => 'Implement Enums',
    'status' => TaskStatus::IN_PROGRESS,
]);

$task->status->label; // In Progress
$task->status->color; // sky-500
```

Advanced Enums

The examples we saw above already can improve a lot our applications, but having the ability to define methods in `Enums` can go to a next level.

Let's imagine a different scenario where we have an `Enum` for different types of `2FA` methods for our application.

```
enum TwoFactorMethod: string
{
    case EMAIL = 'email';
    case PHONE = 'phone';
    case APP = 'app';
}
```

And we have this set in our `User` model.

```
class User extends Authenticatable
{
    // Model definition here
    protected function casts(): array
    {
        return [
            ...,
            'two_factor_method' => TwoFactorMethod::class,
        ];
    }
}
```

During the login flow I want to send the `2FA` code based in the selected method. There are different ways on doing this, but we can take advantage of our `Enum` for helping us with that.

First thing we need is to define a contract on how this should work, so we can define a simple `Interface` for that.

```
interface TwoFactorCodeSender
{
    public function send(User $user): void;
}
```

With our contract defined, we need to implement each type of sender that we need, for the simplicity of this example, I'm not going to provide the implementation for these, just how we can chain this with our `Enum` and how we can use it.

```
enum TwoFactorMethod: string
{
    case EMAIL = 'email';
    case PHONE = 'phone';
    case APP = 'app';
    public function codeSender(): TwoFactorCodeSender
    {
        return match($this) {
            self::EMAIL => resolve(TwoFactorEmailSender::class),
            self::PHONE => resolve(TwoFactorPhoneSender::class),
            self::APP => resolve(TwoFactorAppSender::class),
        }
    }
}
```

Now, during the login flow, when we need to send the code to a user, we can simply do something like this:

```
// Login flow here
$user->two_factor_method->codeSender()->send($user);
```

As you can see, **Enums** opens a lot of possibilites for us to improve the overall structure and architecture of our applications.

Mastering Error Handling: The Power of Custom Exceptions

In `Enterprise Applications`, proper error handling is crucial. It helps ensure that your application can gracefully recover from unexpected situations. Without it, errors can lead to application crashes, poor user experiences, and potential data loss. By handling errors properly, you can maintain the reliability and stability of your application, even when things go wrong.

Using `Custom Exceptions` in `Enterprise Applications` provides more control over error handling. Custom exceptions allow you to create specific error types that relate directly to your application's logic. This makes it easier to identify, catch, and resolve issues quickly. With custom exceptions, you can provide clearer error messages and maintain cleaner code, making your application easier to debug and maintain.

How to Use Custom Exceptions

Let's imagine that we're building a feature to process a payment from the user. The initial code could be something like this, using generic Exceptions.

```
class ProcessPayment
{
    public function handle(ProcessPaymentDTO $dto): void
    {
        try {
            if ($this->isCardExpired($dto)) {
                throw new Exception('The given card has expired');
            }

            if (! $this->hasSufficientFunds($dto)) {
                  throw new Exception('Insufficient funds for this
transaction.');
            }
        } catch (Exception $exception) {
            // Error handling here
        }
    }
}
```

The issue with the code above when we need to handle it in a `try/catch` block is that we would need to validate the exception message before presenting it to the user, because if anything else goes wrong inside the `try/catch` block, we can risk showing sensitive information about our application if we don't handle it properly.

Since we already know that in these two scenarios, something can go wrong, we can create `Custom Exceptions` instead. This way we can easily identify which exception was raised and properly handling how to present it to the user. Besides that, we can improve our error messages by adding parameters to our `Custom Exceptions`.

```
class CardExpiredException extends Exception
{
    public function __construct(CardInformation $cardInfo)
    {
         parent::__construct("The card: {$cardInfo->identifier} has
expired.");
    }
}

class InsufficientFundsException extends Exception
{
     public function __construct(CardInformation $cardInfo, Money
$chargeAmount)
    {
         parent::__construct("The card: {$cardInfo->identifier} has
insufficient funds for the transaction of {$chargeAmount->value}.");
    }
}

class ProcessPayment
{
    public function handle(ProcessPaymentDTO $dto): void
    {
        try {
            if ($this->isCardExpired($dto)) {
                throw new CardExpiredException($dto->cardInformation);
            }

            if (! $this->hasSufficientFunds($dto)) {
                                                    throw    new
InsufficientFundsException($dto->cardInformation, $dto->chargeAmount);
            }
          } catch (CardExpiredException|InsufficientFundsException
$exception) {
            // Error handling here
        } catch (Exception $exception) {
            // Handle generic exceptions to not show sensitive data
        }
    }
}
```

As you can see in the example above, we improved the quality of our error handling, the security and also the overall organization of the application by using **Custom Exceptions**.

In this case, we can still improve a little further. When we have multiple `Custom Exceptions` related to the same domain, like this one that's related to a payment process, and we can handle them in the same way, we can create a single `Custom Exception` that can build different messages depending on the situation. This is how our example would look like after applying this approach.

```
class PaymentProcessException extends Exception
{
    public static function cardExpired(CardInformation $cardInfo)
    {
          return new self("The card: {$cardInfo->identifier} has
expired.");
    }

      public static function insufficientFunds(CardInformation
$cardInfo, Money $chargeAmount)
    {
          return new self("The card: {$cardInfo->identifier} has
insufficient funds for the transaction of {$chargeAmount->value}.");
    }
}

class ProcessPayment
{
    public function handle(ProcessPaymentDTO $dto): void
    {
        try {
            if ($this->isCardExpired($dto)) {
                                                                throw
PaymentProcessException::cardExpired($dto->cardInformation);
            }

            if (! $this->hasSufficientFunds($dto)) {
                                                                throw
PaymentProcessException::insufficientFunds($dto->cardInformation,
$dto->chargeAmount);
            }
        } catch (PaymentProcessException $exception) {
            // Error handling here
        } catch (Exception $exception) {
            // Handle generic exceptions to not show sensitive data
        }
    }
}
```

Using **Custom Exceptions** is a powerful strategy for robust and efficient error handling. They provide additional context and information about errors, making debugging easier and more efficient. They also increase the readability of our code, as they can clearly signal what kind of error occurred. From small applications to large-scale ones, using the power of

`Custom Exceptions` can significantly improve the overall code quality and maintainability, and for an `Enterprise Application`, this is a MUST.

REAL-WORLD CASE: Sensitive Exceptions

As we've seen in this section, if we don't properly handle errors, we can be compromising the security of our applications by exposing sensitive data that shouldn't be accessed by the users.

I worked as a consultant in a company that said they had issues with `SQL Injection` attacks, but they were not understanding what was causing that. After researching the codebase, there were only two issues, but the combination of these two issues led them to this scenario.

The first issue was that for a specific part of the application, the error was being handled with a generic Exception and the `$exception->getMessage()` was being sent directly to the user and in some cases that was sending information of two table structures from the database.

The second issue was that in another part of the application, instead of using `Eloquent` for the query, it was using a raw SQL query using data coming from the UI without any sanitization on it.

`LESSON LEARNED:` Keeping attention to security-related subjects when implementing or reviewing code can keep you and your applications safe from attacks.

Optimizing Performance and Integrations with Queues

In `Enterprise Applications`, managing tasks efficiently is vital for performance. `Queues` help handle background tasks, ensuring your application remains fast and responsive. They are essential for processing jobs that don't need to be done immediately, allowing your application to scale smoothly.

In this section, I'll focus on the advantages on why you should be using `Queues` in your applications and some small tips to better use them in an `Enterprise Application`. I won't focus on how to use `Queues` in `Laravel`, because they have an awesome documentation on this.

Advantages of Using Queues

`Queues` are important because they allow your application to handle time-consuming tasks without slowing down user interactions. Operations like sending emails, processing jobs, or generating reports can be offloaded to queues. Here are some advantages of using them:

- `Improved Performance:` they keep your application fast by processing tasks in the background, freeing up resources for immediate user interactions.
- `Scalability:` they allow your application to handle more requests by distributing workload efficiently across multiple workers.
- `Reliability:` they help ensure that tasks are completed even if the system faces issues, as they can retry failed tasks automatically.
- `Resource Management:` by offloading tasks to queues, you reduce load on your server during peak times, improving overall stability.
- `User Experience:` they maintain a smooth experience for users by handling complex processes without impacting response times.

Tips and Tricks for Queues in Laravel

Implementing `Queues` in `Laravel` is really easy because of their awesome documentation and elegant implementation. But there are some things you may miss from the documentation or things that for an `Enterprise Application` you can apply to save some headaches later on.

Jobs and Actions

In an `Enterprise Application`, it can be common that certain functionalities could be

triggered from multiple places or even multiple clients like Mobile Applications, CLI, etc and to sometimes the same action could be needed in a sync way while in others it can be async. If the application you're working on supports this, one thing that can save some headaches is to wrap the needed logic in an Action class, and then if you need to run it in a sync way from a Controller for example, you can use it, but also call it from a Queued Job if needed.

Avoiding Large Jobs

When serializing a Job, all the Eloquent model relationships are also serialized, this can make your serialized job string become quite big. Not only that, but when deserializing it, all the relationships will be entirely retrieved from the database. For this you can use the withoutRelations method on the model in the job.

```
class ProcessVideo implements ShouldQueue
{
    use Queueable;

    private Video $video;

    public function __construct(Video $video)
    {
        $this->video = $video->withoutRelations();
    }
}
```

If you're using PHP 8 constructor property promotion, Laravel provides the Illuminate\Queue\Attributes\WithoutRelations attribute that you can use.

```
class ProcessVideo implements ShouldQueue
{
    use Queueable;

    public function __construct(
        #[WithoutRelations]
        Video $video
    ) {}
}
```

Keep an Eye in Your Queues

Another thing that's in the documentation, but sometime may be overlooked, is that Laravel provides out-of-the-box a way for monitoring your Queues and for you to get

notified if any of them exceeds a threshold of job count. This is a simple two-step process.

First, you need to schedule the `queue:monitor` command to run EVERY MINUTE. You can then pass a list of `Queues` you want to monitor and the threshold.

```
php artisan queue:monitor redis:default,redis:important --max=100
```

After that, you need to configure the notification to be sent in the `AppServiceProvider` by listening to the `Illuminate\Queue\Events\QueueBusy` event.

```
class AppServiceProvider extends ServiceProvider
{
    public function boot(): void
    {
        Event::listen(function (QueueBusy $event) {
            Notification::route('mail', 'dev.team@company.com')
                ->notify(new QueueHasLongWaitTime(
                    $event->connection,
                    $event->queue,
                    $event->size
                ));
        });
    }
}
```

REAL-WORLD CASE: The (Really) Long Polling

`Queues` can be used not only for improving the performance of an application, but also as a way to integrate with another application, where one application writes to the `Queue` and the other ones consumes from it, this can be used to have something like real-time updates from one application to the other.

But there are other techniques for that, being one of them the `Long Polling`. This is a technique where an application repeatedly request data from another application using HTTP requests.

I worked as a consultant in a company that they had two applications that were communicating using this technique. When I started working with them, they said that they were having a lot of issues with performance. During some times of the day they had some processes running that caused some peaks for this communication and this was really slow.

I proposed the solution to switch to `Queues`, where instead of the "client" application doing a `Long Polling` to see if there's new data to process from the "server" application, the

"server" application would add a message to a `Queue` and we would create a `Worker` to consume that in the "client" application, even running multiple workers in parallel. After the implementation was finished and released, the resource usage dropped by 70% and the average processing time dropped 85%.

LESSON LEARNED: Researching and trying other approaches to handle a solution can consume more time at the beginning, but save you A LOT of time and A LOT of headaches in the future.

Leveraging Base Classes for Cohesion

`Enterprise Applications` are usually big and complex applications, even if they don't start like that, since they are projects that live for a long time, they can turn into a MASSIVE codebase.

Using `Base Classes` is a great way to help the team to manage the codebase. They provide common functionality that can be shared across similar components, avoid duplication and keeping the code organized.

Advantages of Base Classes

- `Code Reusability:` they allow you to reuse code across different parts of your application, reducing duplication and effort.
- `Consistency:` they provide a consistent structure, ensuring that similar components follow the same rules and patterns.
- `Easier Maintenance:` with shared logic in one place, updates or fixes can be made easily without searching through multiple files.
- `Simplified Development:` they speed up development by providing ready-to-use methods and properties, allowing developers to focus on specific features.
- `Enhanced Collaboration:` with a clear structure in place, teams can work together more effectively, reducing misunderstandings and bugs.

Getting Started with Base Classes

If you're working on a new project that's designed to be a long-term application, or even working on an existing `Enterprise Application`, the best way to start working with `Base Classes` is starting with empty classes. Another important thing to note is that all these classes should be `abstract`, you should never need to instantiate one of these classes, because their purpose is only to group base and common functionalities like methods and properties that can be used by their multiple children.

Some of the `Base Classes` I always add to my projects are the ones below.

```
abstract class BaseController {}

abstract class BaseAction {}

abstract class BaseModel extends Model {}
```

How to use Base Classes

Base Classes should be used to define standards for common functionalities in the codebase. So the best way to work with them is starting them as an "empty container", then you'll start looking for repeated code or logic across similar components in your application. This can be either methods or properties. When you see that multiple places need to perform the same action, it's time to extend your Base Class.

One example from a project that I consulted for is how to handle downloads in a standard way across the application. The code below is a simplified version of the actual code, but you can get the idea.

```
abstract class BaseController
{
    protected function downloadFileWithCustomHeaders(
        string $fileContent,
        MimeType $mimeType,
        ?string $filename = null
    ): Response {
        $header = ['Content-Type' => $mimeType->value];

        if (! is_null($filename)) {
             $header['Content-Disposition'] = "attachment: filename
='{$filename}'";
        }

          return response()->make($fileContent, Response::HTTP_OK,
$header);
    }
}
```

Having a standard way of doing things in an Enterprise Application is something that will save you and your team a lot of headaches and lost time with debugging and maintenance, so mastering using Basse Classes is a great technique to handle this. In the next chapter we will be seeing an example showing how this can be used to handle a

tricky subject that's common in **`Enterprise Applications`**.

Enterprise-Level Tips

When we work with `Enterprise Applications`, a whole different set of challenges arise. Things that for "normal" applications are "nice to have", for `Enterprise Applications` can be A MUST, for example, applications may need to follow specific rules on how they log things, may need an easy way of enabling and disabling specific features for specific users or teams.

In this chapter I'll share some tips that I learned over the years working with different `Enterprise Applications` that seemed common topics for all of these different applications. The idea is not to provide full solutions, neither provide information on all topics regarding `Enterprise Applications` because that would turn this into a MASSIVE book. But I want to provide some foundations for you, so when you face a similar challenge, you'll already have a starting point.

All these tips are based in real-world case needs and solutions, so get ready to learn how to handle some topics for your `Laravel Enterprise Applications`.

Structure for Success: Implementing Layered Architecture

As we already talked about, **Enterprise Applications** can get massive, and having standards and rules can maintain the codebase healthy and easier to maintain and extend with time. Diving the application in layers is a great way of achieving this, by separating responsibilities we create a codebase that's easier to extend and maintain, but also reducing the complexity.

There are lots of books and articles out there that talks about this topic, here in this book, I'll show one simple, yet powerful architecture that I already applied in some **Enterprise Applications** that I worked with, and that went very well. It's an architecture composed of four layers: **Communication**, **Validation**, **Business** and **Data**.

Communication Layer

This is the simplest of the layers, it should be responsible only for the **IO** (Input/Output) of our application, receiving the data sent by the user and return a result back to it. This layer in most of the cases is represented by our **Controllers**. Here, we should use the **Thin Controller** approach, meaning that it should be as minimal as possible, like the example below.

```
class UserController extends BaseController
{
     public function store(CreateUserDTO $dto, CreateUser $action):
JsonResponse
    {
        return $this->successResponse(
            data: $action->handle($dto),
            code: Response::HTTP_CREATED
        );
    }
}
```

Validation Layer

This layer is responsible for validating and mapping the **Input** received by the **Communication Layer** before proceeding to the next layers. You may wonder why this is a separate layer instead of doing it in the **Communication Layer**, but there's a simple reason. By decoupling the validation from the **Communication Layer** I can reuse my logic across different parts of the application. This layer is represented by the **DTOs**. In the example above, that we have the **CreateUser** action, if I need to create a user from a CLI

or from another action in my application, since it expects a `CreateUserDTO` as a parameter, if I add my validation there, I can reuse it on all other places in my application, avoiding code duplication and reducing the risks of introducing bugs.

The code below shows an example of how we could define our `CreateUserDTO`. The example is using my Validated DTO package.

```
class CreateUserDTO extends ValidatedDTO
{
    use EmptyCasts,
        EmptyDefaults;

    public string $name;

    public string $email;

    public string $password;

    #[Cast(type: EnumCast::class, param: Role::class)]
    #[DefaultValue(Role::REGULAR)]
    public Role $role;

    protected function rules(): array
    {
        return [
            'name' => ['required', 'string', 'min:4'],
            'email' => ['required', 'email'],
            'password' => [
                'required',
                Password::min(8)
                    ->letters()
                    ->mixedCase()
                    ->numbers()
                    ->symbols()
                    ->uncompromised(),
                'confirmed',
            ],
            'role' => ['sometimes', 'string', new Enum(Role::class)],
        ];
    }
}
```

Business Layer

This is the most critical layer for the application, this is where we have all the **Business Logic**, the core of our application. This layer should receive the mapped and validated data by the **Validation Layer**, get the needed data from the **Data Layer**, apply the needed logic and send the result back to the **Communication Layer**. This layer is represented by our **Action** classes. To maintain a healthy **Business Layer**, you should apply what was already discussed in our **Use Actions** section from the previous chapter.

Data Layer

This is the layer that can read and write from our application's database. This can be a polemic topic in the **Laravel** community, because usually the **Repository Pattern** is seen as not needed and even as overengineering. I already worked in many applications that applied and others that didn't apply this pattern, and when we're talking about **Enterprise Applications**, keeping a single layer of our application responsible for the communication with the database can save A LOT of headaches and issues down the road.

With that in mind, I'm using a new approach that I think it's a middle ground. I don't apply the **Repository Pattern**, but I do something that I call **Model-Repository**, where I wrap all the communication with the DB inside methods in the **Eloquent Model** classes. If you don't like the **Repository Pattern**, this can be a solution for maintaining your code clean and easy to maintain without the overhead of this pattern.

```
class ProductListing extends BaseModel
{
    // ...

    public static function merchantProductsByDate(
        int $merchantId,
        CarbonImmutable $from,
        CarbonImmutable $to
    ): Collection {
        return self::query()
            ->where('merchant_id', $merchantId)
            ->whereBetween('date', [$from, $to])
            ->get();
    }
}
```

With this approach we can avoid code duplication, but also improve the maintainability of the application, because when someone needs to fix, extend or improve a query, they know that they should look only at the model classes, and not looking everywhere in the

application.

Track and Trust: Effective Data Auditing Practices

In `Enterprise Applications`, having control over data changes is essential. Whether changes come from users or internal staff, knowing who did what and when is crucial for maintaining accountability and security. `Data Auditing` helps track these changes, providing a clear record of modifications. This is important when applications need to keep in compliance with regulations, but also increases the trust and transparency within your application, both for clients and within the company.

There are many ways for doing this, and also a lot of packages out there that can help you. However, depending on your needs, you don't need a package full of features that you won't need or use and neither to pay for a third-party service. I'll show here a simple, yet powerful approach on how to have a great `Data Auditing` process with a few lines of code, leveraging the power of `Laravel` for that.

Implementing Data Auditing

The first step for a good `Data Auditing` process, is having records of who made changes to the data and when. This can be easily achievable with `Laravel`. Make sure that all your tables have the `created_at` and `updated_at` fields. This is easily achievable by adding the `$table->timestamps()` call in your `Migrations`.

Some applications may require that critical resources should be kept in record even after deletion, for that, `Laravel` also provides the `Illuminate\Database\Eloquent\SoftDeletes` trait. Just make sure to add the `$table->softDeletes()` call in your `Migrations`.

With these, we already cover the `when` part, now we need to handle the `who` part. For that, we can add some fields in our tables, following a standard close to the `timestamps`, we're going to add the: `created_by`, `updated_by` and `deleted_by` fields. For not having to manually create these in our `Migrations`, we can create a macro for them. In the `boot` method of your `AppServiceProvider` you can add this.

```
class AppServiceProvider extends ServiceProvider
{
    public function boot(): void
    {
        Blueprint::macro('userActions', function ($addDelete = false)
{
$this->foreignId('created_by')->nullable()->constrained('users');
$this->foreignId('updated_by')->nullable()->constrained('users');
            if ($addDelete) {
$this->foreignId('deleted_by')->nullable()->constrained('users');
            }
        });
    }
}
```

With this, now you can use the `$table->userActions()` to add the `created_by` and `updated_by` fields, and if you call `$table->userActions(true)` it will also add the `deleted_by` to your tables in your `Migrations`.

The timestamps are handled automatically by `Eloquent`, but we don't want to add manually the value for the `created_by`, `updated_by` and `deleted_by` fields as well, right? We can achieve this by creating a trait and using the technique of using `Base Classes`, we can add this trait to our `BaseModel`, so all our models will have this out-of-the-box.

First we create our trait.

```
trait UserActions
{
    public bool $disableUserActions = false;

    public static function bootUserActions()
    {
        static::creating(function (Model $model) {
            if ($model->disableUserActions) {
                return;
            }

            $model->created_by = Auth::id();
        });

        static::updating(function (Model $model) {
            if ($model->disableUserActions) {
                return;
            }

            $model->updated_by = Auth::id();
        });

        // Starting on Laravel 12.20 we have the method below
        // If you're using an older version, the same can be
        // achieved with this check
         // if (in_array('Illuminate\Database\Eloquent\SoftDeletes',
class_uses(static::class)))
        if (static::isSoftDeletable()) {
            static::softDeleted(function (Model $model) {
                if ($model->disableUserActions) {
                    return;
                }

                $model->deleted_by = Auth::id();
                $model->save();
            });
        }
    }
}
```

Then we can add this trait to our `BaseModel`, and all our model classes extending from it will have this feature enabled out-of-the-box.

```
abstract class BaseModel extends Model
{
    use UserActions;
}
```

If you paid attention, we also added a property called **$disableUserActions** in our trait. This is needed if you want to disable this behavior for a particular model by just setting this property to **true**.

```
class Product extends BaseModel
{
    public bool $disableUserActions = true;
}
```

Now we have already a way to check **when** and **who** did the changes in the data from our application. But sometimes we still need more, we need to have **what** was changed. We're going to follow a similar approach by using a trait for automatizing this task, but before that, we need to create a table to store this information, let's start with the **Migration**.

```
return new class() extends Migration
{
    public function up(): void
    {
        if (! Schema::hasTable('change_logs')) {
            Schema::create('change_logs', function (Blueprint $table)
{
                $table->id();
$table->foreignId('user_id')->nullable()->constrained();
                $table->bigInteger('record_id')->nullable();
                $table->string('table');
                $table->string('action', 50);
                $table->json('payload');
                $table->json('old_data')->nullable();
                $table->json('new_data')->nullable();
                $table->json('changed_data')->nullable();
                $table->timestamps();
                $table->index(['table', 'record_id', 'created_at']);
                   $table->index(['table', 'record_id', 'action',
'created_at']);
            });
        }
    }
};
```

The table has a simple structure, but it gives all we need for having the context of **what** was changed in our data, with the `Migration` created, let's create our `Model` and an Enum that we will use in it.

```php
// ChangeAction.php
enum ChangeAction: string
{
    case CREATE = 'CREATED';
    case UPDATE = 'UPDATED';
    case DELETE = 'DELETED';
}

// ChangeLog.php
class ChangeLog extends Model
{
    protected $fillable = [
        'user_id',
        'record_id',
        'table',
        'action',
        'payload',
        'old_data',
        'new_data',
        'changed_data',
    ];

    protected $casts = [
        'action' => ChangeAction::class,
        'payload' => 'array',
        'old_data' => 'array',
        'new_data' => 'array',
        'changed_data' => 'array',
    ];

    public function user(): HasOne
    {
        return $this->hasOne(User::class);
    }
}
```

Now, let's build the **magic** part with our trait.

```php
trait LogChanges
{
    public bool $disableChangeLogs = false;

    public bool $logCreateEvent = true;
```

```
    public bool $logUpdateEvent = true;

    public bool $logDeleteEvent = true;

    public static function bootLogChanges()
    {
        static::saved(function (Model $model) {
            if ($model->disableChangeLogs) {
                return;
            }

            if ($model->wasRecentlyCreated && $model->logCreateEvent)
{
                static::logChange($model, ChangeAction::CREATE);
            } else {
                if (! $model->getChanges()) {
                    return;
                }
                if ($model->logUpdateEvent) {
                    static::logChange($model, ChangeAction::UPDATE);
                }
            }
        });

        static::deleted(function (Model $model) {
            if ($model->disableChangeLogs) {
                return;
            }

            if ($model->logDeleteEvent) {
                static::logChange($model, ChangeAction::DELETE);
            }
        });
    }

     public static function logChange(Model $model, ChangeAction
$action): void
    {
        ChangeLog::query()->create([
            'user_id' => Auth::check() ? Auth::id() : null,
            'record_id' => $model->id ?? null,
            'table' => $model->getTable(),
            'action' => $action,
            'payload' => $model->toJson(),
              'old_data' => $action !== ChangeAction::CREATE ?
```

```
            $model->getOriginal() : null,
                        'new_data' => $action !== ChangeAction::DELETE ?
            $model->getAttributes() : null,
                       'changed_data' => $action === ChangeAction::UPDATE ?
            $model->getChanges() : null,
                    ]);
            }
        }
```

Now, we can just add this trait to our `BaseModel` to make this available to all of our models out-of-the-box.

```
abstract class BaseModel extends Model
{
    use ChangeLogs,
        UserActions;
}
```

Like the other `UserActions` trait, this one also has properties that we can disable all the logs, or only specific ones. Every time a model is going to be created, updated or deleted, you'll have the logs saved in this table. A little caveat here is that bulk actions won't trigger the `Eloquent` events, so in this case you can create some helper methods in your `BaseModel` for doing these bulk actions and insert the logs in the `change_logs` table.

With the solution from this section, you'll have already a good `Data Auditing` process in your application, where you can know `when`, `who` and `what` changes were done in the data in a simple way, without using any third-party services or packages.

Managing Releases: Unlocking the Power of Feature Flags

When we work with `Enterprise Applications`, one critical subject is how we properly release and manage features for the users. `Enterprise Applications` usually have dozens, sometimes even hundreds of different features, and having a standard and structured way to manage how to release and which users have access to different features is one of the challenges of the enterprise ecosystem.

What are Feature Flags

Think of `Feature Flags` as switches that you can use to enable and disable features from your application. These switches can be applied in different levels depending on the need for the application. The most two types I've seen in these years working with `Enterprise Applications` are `Feature Flags` in `Single Layer` or `Multiple Layers`.

Single Layer Feature Flags

This is the simplest system and that's used by many applications. In this system, the `Feature Flags` are handled in a single level, usually at the `User Level` or at the `Team/Organization Level`, meaning that when a `Feature Flag` is enabled or disable it will affect that specific user or all users in that specific team/organization.

Multiple Layers Feature Flags

In this system, we have different layers where a `Feature Flag` can be enabled and disabled, and where there's a hierarchy for these levels. For example a two-layer system where we have `Feature Flags` at both the `Team/Organization Level` and the `User Level`. In this case, we can have `Feature X` enabled for `Team 1`, but only `User 1` and `User 2` have access to it, while `User 3` has no access to it.

Why to Use Feature Flags

Having a robust `Feature Flag` system brings a lot of advantages, especially when we're talking about `Enterprise Applications`.

- `Safe Deployment:` they let you deploy new features safely by enabling them for a small group first. This minimizes risks and allows for testing in a real environment.
- `Quick Rollback:` if a feature causes issues, you can easily disable it with a flag, ensuring your application's stability.
- `Continuous Delivery:` you can release updates incrementally. This supports a

continuous delivery process, making deployments smoother and less disruptive.

- **Targeted Release:** you can control which users or groups see a new feature. This targeted approach helps gather specific feedback and make improvements before releasing it to everyone.

- **Product Validation:** they allow the team to have an internal validation process before releasing a new feature to customers as well as enabling the sales team to showcase features to potential customers.

How to Implement Feature Flags

There are tons of different ways we can implement a `Feature Flag` system, and there are also lots of different packages for this, but I highly recommend that you use `Laravel` first-party package for `Feature Flags` - Laravel Pennant.

I don't want to just copy things from the documentation into this book, and `Laravel Pennant` docs are already amazing, so I'm going just to add one tip that I think that could be relevant when we're talking about `Enterprise Applications`.

With `Laravel Pennant`, you can define `Feature Flags` in two ways. by using the `Feature::define()` method in the `boot` method of your `AppServiceProvider` or creating them as classes. As I said before, in `Enterprise Applications` we can have even hundreds of different `Feature Flags` in our applications, so use the class-based approach.

This approach also brings another advantage over defining them in the `AppServiceProvider`, you can define a `before` method in your `Feature Flag` class that will run before the check of the feature. This method always run in-memory, before retrieving the value from storage, and if any `non-null` value is returned from it, it's going to be used in the place of the value set in the storage.

Seamless Connections: Crafting Robust Integrations

A lot of applications need to communicate with other third-party services or other internal applications, and when we're working with `Enterprise Applications` this is guaranteed. When we add integrations with other services, they become like a dependency for our application, and we need to handle and manage this integration carefully if we want to avoid headaches down the road.

There are many ways on handling this in an application, here I'll show you an approach that's very simple, yet easy to maintain, extend and adapt. This approach is based in `Action` classes, that as we already saw before, are building blocks to build our application.

Configuring the Integration

All the integrations that we will implement will need some configuration values like URL, app key, credentials, etc. For this we will be using the `config/services.php`. For each integration we want to add to our application, we will create a new section there. For this example, let's imagine we want to integrate with the `HubSpot` API. For this, we will need to add some basic configuration values.

```
return [
    // Other services here

    'hubspot' => [
        'url' => env('HUBSPOT_URL'),
        'access_token' => env('HUBSPOT_ACCESS_TOKEN'),
    ],
];
```

One thing that I like to do is to add some additional configurations like a config for the `HTTP Header` to use for the authorization, I know that most of the services this will be something that won't change often, but I already had some bad experiences before and this can maybe help you. Another one I like to add is the `timeout` configuration we will use for the service.

```
return [
    // Other services here

    'hubspot' => [
        'url' => env('HUBSPOT_URL'),
        'access_token' => env('HUBSPOT_ACCESS_TOKEN'),
        'http_header' => env('HUBSPOT_HTTP_HEADER', 'Authorization'),
        'timeout' => env('HUBSPOT_TIMEOUT', 60),
    ],
];
```

Implementing the Integration

Now that we have the configuration for out integration sorted out, the next step is to implement it. We're going to use `Action` classes for this, but we're also going to use another technique already showed in this book, the `Base Classes`. The first thing we're going to build is a base class for our integration that will configure the `HTTP Client` in a standard way to be used by other classes.

```php
abstract class HubSpotAction
{
    protected string $baseURL;

    private string $token;

    private string $httpHeader;

    private int $timeout;

    public function __construct()
    {
        $this->baseURL = config('services.hubspot.url');
        $this->token = config('services.hubspot.access_token');
        $this->httpHeader = config('services.hubspot.http_header');
        $this->timeout = config('services.hubspot.timeout');
    }

    abstract protected function actionURL(): string;

    protected function request(): PendingRequest
    {
        return Http::timeout($this->timeout)
            ->withHeaders([
                 $this->httpHeader => "Bearer {$this->token}",
                'content-type' => 'application/json',
            ]);
    }
    protected function url(): string
    {
        return "{$this->baseURL}/{$this->actionURL()}";
    }
}
```

Now that we have our base class that handles the common logic for all the calls we will need to do for the **HubSpot** API, we can start creating the implementation for all the needed endpoints we need. The idea is to follow the same standards we discussed in the **Use Actions** section from the last chapter. Each **Action** will handle the integration with a single endpoint, using **DTOs** to improve the quality and maintainability. For this example, I'll show the action for creating a Visitor ID.

```
class CreateVisitorID extends HubSpotAction
{
    public function handle(CreateVisitorIDDTO $dto): void
    {
        $response = $this->request()
            ->post($this->url(), $dto->toArray());
        // Handle response here
    }

    protected function actionURL(): string
    {
                        return   'conversations/v3/visitor-
identification/tokens/create';
    }
}
```

As you can see, following this approach, the **Action** class that will implement the endpoint integration can reuse all the things we configured in our **Base Class**, making it very easy to integrate as many endpoints we need. Also, if any bugs are found in your application, it would be very easy to debug since you can go directly to the **Action** class responsible for the endpoint that's causing the issues.

The **Base Class** can be used to wrap all the common things you'll need in your endpoints' integrations, for example if instead of having an access key, you needed to authenticate in the service with credentials you could create the authentication part in the **Base Class**, adding a cache layer if needed to avoid multiple calls to authenticate when not need and reuse this logic in all the **Action** classes for the endpoints' integrations.

If you're already using the **Action Pattern** for your application, using this approach to your integrations will feel completely familiar to you. Is an approach that uses **Laravel** features, without the need of third-party packages, unless you want to use one for handling **DTOs** for example, but this won't be specific for the integrations as they'll be used across the whole application.

Alternative Approach

If you didn't like this approach, you can use a third-party package for handling integrations. The best one IMO for the **PHP** and **Laravel** ecosystem is Saloon, which is VERY POWERFUL and provides a lot of features. I won't extend in this topic because this package is packed with so many amazing features, that there's a whole book about this topic.

About the Author

Hey there, I'm Wendell Adriel!

I'm a **Software Engineer** from Brazil, living in Portugal since 2016.

I work with Software Development since 2009, and have more than a decade of experience working with **Enterprise Applications** from different industries.

I love topics regarding **Software Architecture** and **Software Design**, having experience with different languages like **Go**, **Node.js**, **Python** and **Java**, but being an expert in **PHP** and **Laravel**.

I already worked in roles like **Solutions Architect**, designing and building the architecture for **Enterprise Solutions** from scratch. Also as **Lead Software Engineer** and **Engineering Manager**, leading teams of engineers and projects of different sizes and complexity.

I love to share my knowledge with others, I already mentored dozens of engineers, written articles not only for my blog, but also for other websites and even the PHP Architect magazine.

I'm an Open Source enthusiast, I contributed and still contribute to some projects, and I even had already contributed with some features and fixes for the Laravel Framework. I also maintain some other projects that I created.

Follow me on X, where I share tips and insights on **PHP** and **Laravel**: wendell_adriel

Check my projects and contributions to Open Source projects in my GitHub Profile.

If you want to read my blog, where you can find articles about **PHP**, **Laravel** and **Software Development** in general, or check other contact information, you can access it here.