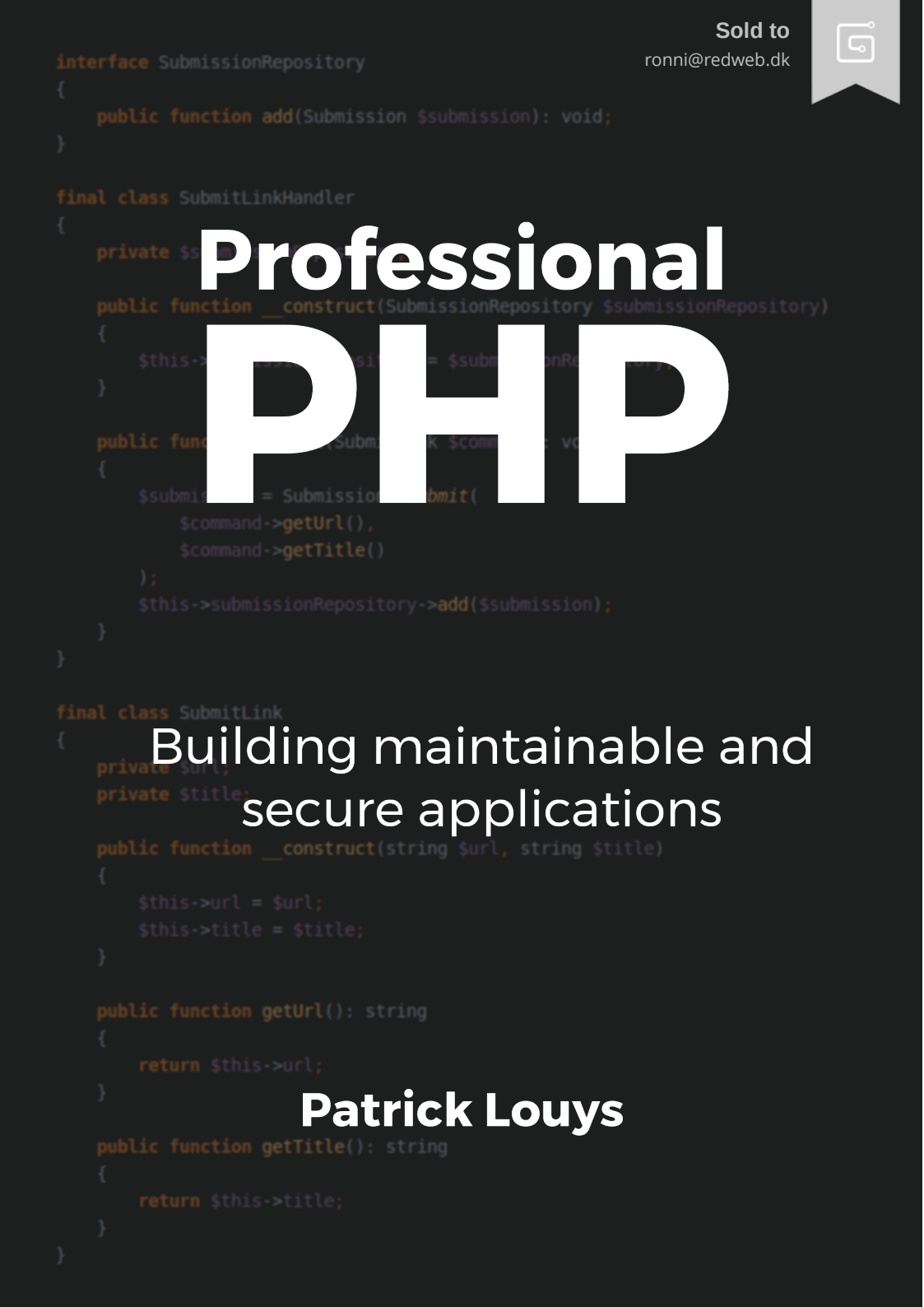


Professional PHP

Building maintainable and secure applications

Patrick Louys

Table of Contents

Part I: Theory

Part II: Tutorial

Foreword

My story

Hi! I'm Patrick, a software developer from Zurich, Switzerland.

I started programming sometime between 2005 and 2007. I can't remember the exact date. My first project was a simple PHP backend for the website of my counter-strike team (I was really into gaming back then).

My next project was a social network, but I abandoned that project before it even got close to completion. I could barely make the code work and didn't know anything about security, so I guess that's a good thing. My only programming education was an introduction class during my systems engineering apprenticeship (which wasn't exactly great).

I never really planned to become a programmer. During my apprenticeship I was only drawn towards programming because I wanted to build things, so I kept teaching myself.

After the apprenticeship, I joined the database team of the company for 3 months until I had to go join the military. My first project for them was a monitoring web application that was built in PHP. That's when I made up my mind and decided that I was going to become a software developer instead.

After over a year in the military I had to clear my head. With the money that I had saved up, I went to travel around the world for almost two years.

Traveling was fun, but I started to miss programming after a while. That's when I founded my own company and started to freelance for clients. I learned a lot during that time and I moved from the dreaded old-style PHP to writing code on top of the CodeIgniter framework. I also dabbled in other languages, but always ended up coming back to PHP.

When I got back home, I found a job at a company that was running a custom online shop on top of the CodeIgniter framework. Soon after, a new developer joined the team and showed me how to do object oriented programming in the right way. He even brought his copy of Uncle Bob's Clean Code to work.

We improved the codebase a lot while I was there. A few years later, I moved to a company that sells a learning management system. They just started to move from old-style PHP to the Symfony framework and I was brought in to help with that process.

I never really had a lot guidance in my career. I had to teach myself everything from books, videos and online tutorials.

Often the resources that I used to learn were written for other programming languages. I had to piece knowledge together from many different sources. This is why I ended up writing the no-framework-tutorial (https://github.com/PatrickLouys/no-framework-tutorial), which in turn was the inspiration for this book.

Why is clean code important?

When I first learned PHP, there weren't many good resources available. Object oriented programming was this big scary thing that nobody really seemed to understand. I listened to what the community recommended and ended up as a CodeIgniter developer. That was the hip framework back then, full of good practices. At least that's what everyone told me.

I completed several small projects during that time, either by myself or together with a frontend developer. I noticed a pattern over and over again: progress in the beginning was always very fast, but then the development speed kept slowing down and it became really hard to finish the project. I talked to other developers in the same position and they noticed the same. We thought it was a normal part of development software.

I was disgusted by my own code whenever I had to do maintenance work on a finished project, even if the project was only a few months old. I thought that the reason for this was because I had become a better developer in the meantime.

But when I look back to that time period now, I can see that I did not grow much as a developer. The new code that I was writing was not any better than what I had written before. I kept repeating the same mistakes over and over again.

Things finally started to improve when the company that I was working at hired a new developer. His background was not limited to PHP and he introduced me to many new concepts.

After I realized how little I knew, I started a learning binge that went on for over a year. I buried myself in the OOP classics and spend many days watching talks on Youtube about programming.

I was lucky, because I had the opportunity to apply what I was learning at my job. We didn't get everything right from the start, but the code quality steadily improved a lot over the next two years.

We also noticed that our development speed slowly started to increase again. Changing the refactored code was much easier and less stressful compared to the CodeIgniter part. It took much less time and we introduced fewer bugs.

About the book

When I first started to write the book, I had only planned to include theory. The code examples were random and with no big picture in sight. After getting some feedback, I realized that I was on the wrong path.

Most of the interest in the book was generated from my no-framework-tutorial on Github. I realized that I had to go back to the roots and turn the theory book into a tutorial. It's now much easier to follow along and we will hopefully share some common ground when I mention a concept. The theory is still in the book, but now it's mixed into the tutorial and the topics are only introduced at a point when they make sense.

I wrote this book for my past self. This is the book that I wanted to read when I first learned about clean code. Hopefully it will resonate with you and I hope that you'll get a lot out of it.

Enjoy the book :)

Part I: Theory

1. Introduction

This book is split into two parts. In the first part, I'll be covering a few fundamental programming concepts. The second part is a tutorial where you are going to build an application from scratch. The more advanced concepts will be introduced during the tutorial.

The tutorial chapters are named after the main topic that they cover, not after what you are building in the chapter. This should make it easier to use the book as a reference if you need to look up something.

I highly recommend that you follow the tutorial while you read the book. You will learn much more if you write the code and follow along. It will be harder to understand the content, if you only read the book passively.

And who doesn't like the feeling of having a project completed? You can publish the code on Github and add more features to make the project really your own. Or you can repurpose parts of it to realize one of your own project ideas.

> Tell me and I forget. Teach me and I remember. Involve me and I learn.
>
> *Benjamin Franklin*

Don't stop learning after finishing the book and tutorial. Try to apply the new things that you learned on a new and unrelated project. This will be much more challenging than just following the tutorial and you might not get everything right on the first try, but you will gain a lot of experience.

Whenever you learn something new, try to teach it to others. Teaching something really solidifies your understanding of the topic. Teach your coworkers, your programming friends and random people online that you see struggling with something that you are familiar with. You will benefit a lot from paying it forward and learn the material even better in the process.

> To teach is to learn twice.
>
> *Joseph Joubert*

What is maintainable code?

Your code should be easy to read and understand. Not just for you, but for your whole team. Even for the new junior developer that just started this week.

That doesn't mean that all your code should be procedural, just because there are people who don't understand object oriented programming. Your target audience are other developers after all. But for someone who is familiar with PHP and object oriented programming, it should be effortless to read and understand your code.

Maintainable code also has to be flexible. Business requirements change all the time and your code will have to adapt to those changes. You can't predict how the business requirements will change, but your code should be written so that it's easy to modify it later.

You also get a lot of benefit from writing code that's easy to test, even if you don't use unit tests. Testable code is a side effect of writing maintainable code, the two go hand in hand.

When you write maintainable code, bugs stand out and faulty code is easy to locate. Because everything is very simple and easy to understand, it is also easy to see when something is wrong.

We spend much more time reading code compared to writing code. This is even more emphasized during debugging where a few hours of reading code can lead to a bugfix that only takes a minute to implement. It takes some effort to keep the code maintainable, but it will save you a lot of time in the long run.

Writing completely bug free code for a large codebase is unrealistic and should not be the goal. Instead of writing perfect bug-free code, you should aim to write good code that is easy to read, understand and maintain.

There are a few industries and applications where bugs are really expensive, but those need to take a different approach than the one outlined in this book. In those cases, you want to be able to mathematically proof that your code is correct.

Technical Debt

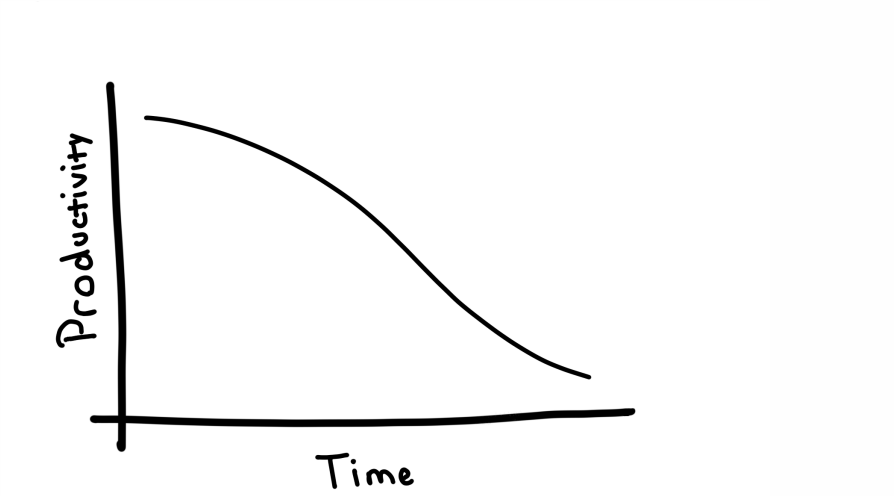
Whenever you take a shortcut and don't write maintainable code, you increase your technical debt. To pay off your technical debt, you can refactor your code to make it more maintainable.

Just like monetary debt, technical debt accumulates interest. The longer you wait before you refactor the code, the longer it will take.

When you take a shortcut during development, you are taking out a loan. Your future self has to repay that loan at some point - with interest.

It would be nice to avoid all technical debt, but that is not always possible. Sometimes you have to take on technical debt to meet an important deadline and you will have to refactor the code after the deadline.

Always try to get rid of technical debt as soon as possible. If you ignore your technical debt and it keeps piling on, your productivity slows down. It will take longer and longer to implement new features as time goes on.



When productivity slows down over time, managers often assume that the developers just got lazy. You should do your best to help the people in charge of the project understand technical debt, otherwise the slowdown of productivity will be blamed on the developers and not the faulty process.

> Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. The danger occurs when the debt is not repaid.
>
> *Ward Cunningham*

If you decide to take on technical debt, you should still write the code as clean as possible. Accepting technical debt is a trade-off and not a free pass to create a mess. Always code with the future refactoring in mind, otherwise you end up with code that is impossible to fix in a reasonable timeframe. If the refactoring takes too long, you will never find the time to take care of it.

Hide your bad code behind a clean interface and keep it separated from the good code. That keeps the mess contained and makes it easy to clean it up later.

How to deal with a deadline

Let's have a look at the following scenario: Your project manager keeps bothering you to finish a new feature, as soon as possible.

You have to display a list of upcoming events on the company website and the events will be listed in more than one location. There is also an admin area planned, where the events can be managed. But for now, you just have to display a few events.

You could just hardcode the list of events in all the places where it has to be displayed. But this approach will lead to duplication and it will be a lot of work to refactor the code later.

During the refactoring, you will have to go through the whole codebase to find all the hardcoded lists. If you are lucky, a simple search and replace will be able to do the job. If not, then it will be a tedious and time-consuming job to refactor the code.

If we hardcode the lists in multiple places, then we are not containing the mess. Instead, it is spilling into our otherwise clean codebase. A better approach hides the hardcoded list behind an abstraction.

We can create a class that encapsulates the hardcoded list of events and use this class when we need to display a list. The development for this approach takes slighly longer than hardcoding everything, but it still takes much less time compared to a complete solution where the events are properly stored (in a database for example).

To refactor the contained parts into a complete solution, you only have to change that one class. Compared to the other approach, you are much less likely to miss a piece of code that has to be refactored.

> An ounce of prevention is worth a pound of cure.
>
> *Benjamin Franklin*

To Rewrite or To Refactor

It is easy to apply the principles for maintainable code to a new codebase, but in the professional world we have to deal with a lot of legacy code. If you take over an old project with a lot of technical debt, often the first thing that comes to mind is that you have to rewrite everything from scratch.

You can rewrite everything from scratch if the codebase is very small or if it is just a small side project. For larger projects, a complete rewrite is rarely a good idea.

When you do a complete rewrite, you still have to maintain the old codebase while you write the new code. If you develop a new feature, you have to duplicate it for both codebases. You are doing twice the maintenance work and a complete rewrite, all at the same time.

To refactor a large legacy codebase efficiently is an art and there are entire books dedicated to the topic. It is possible to rewrite your application part by part, without breaking everything. If you have to tackle a large refactoring at some point in your career, then I highly recommend that you read a book or two on refactoring legacy code before you get started.

A lot of teams don't handle technical debt well and the amount of bad code in their codebase is always increasing. If there is a deadline lingering or an urgent bug has to be fixed, shortcuts are taken and the team can never find the time to refactor the code.

The boy scouts have a rule: "Always leave the campground cleaner than you found it". Whenever you find a piece of trash, you clean it up. Even if it was left there by someone else.

We should follow the same rule as responsible developers. If you limit yourself to only check in code that is cleaner than it was before, then the overall code quality will slowly increase over time.

Even if you don't have the time for a larger refactoring, you can still improve your code line by line. It might not seem like much, but if you follow the boy scout rule strictly, then your code quality will improve as time goes on.

2. Concepts

Before we dive into the more practical parts of the book, I want to cover some basic programming concepts in this chapter. If you are already familiar with some of the topics, feel free to skip them.

This chapter is just here to make sure that we speak the same language and to make sure that you won't get confused when we talk about concepts like abstraction and coupling later in the book.

Abstraction

Software becomes more complex when you add more features. The art of programming is managing that complexity and to make sure that the code is still easy to understand when the codebase grows larger and larger.

When we abstract something, we create a construct to hide the implementation details. Abstraction is not limited to programming, we do this every day through language.

If I mention the word laptop, you immediately know what I am talking about. That is an abstraction.

When your boss tells you to bring your laptop to a meeting, he doesn't care about the brand of your laptop, the screen size, the CPU or other small details. What he cares about is the abstract idea of a laptop.

We are not limited to real world objects when we create an abstraction. Part of a programmers job is to create new useful abstractions, to make it easier to think and reason about the code. It doesn't matter whether a real-world counterpart for the abstraction exists.

Let's assume that we are tasked with writing a program that reacts to a click on the track-pad. At first the track-pad might look like the correct abstraction, but after we think about it for a little longer, it becomes clear that there are other devices that fulfill the same function - like an external mouse for example.

Our code shouldn't care whether the click happened through the track-pad or the mouse. In this case, we can create a new abstraction with no direct counterpart in real life. We invent the pointing-device, an abstraction that groups together pointing devices like track-pads and mice.

Abstraction simplifies your code, even if you end up with more classes and lines of code. The code that uses your new pointing-device abstraction doesn't have to care about the details of track-pads and mice - the complexity is hidden behind the tracking-device abstraction.

The rabbit hole goes much deeper, we are not limited to one layer of abstraction. The track-pad is nothing more than annother abstraction of smaller components and those consist of even smaller components. Taken to the extreme, we can go down to the specific arrangement of molecules and atoms.

PHP and other programming languages are also just abstractions. They make our lives easier because we don't have to write assembler or processor instructions ourselves. The lower level programming languages are hidden behind the abstractions of the higher level languages.

Coupling

Coupling describes how dependent two components are on each other. Maintainable code has very low coupling between different components.

> Low and high coupling are also sometimes referred to as loose and tight coupling.

We can use the laptop again for our example. This time, an engineer screwed up and your monitor is highly coupled to your track-pad. The monitor is not connected to the power source directly, but through the track-pad instead.

This works well for a while and nobody notices it. The engineer is proud of himself, he made the laptop a couple of cents cheaper and saved himself some time.

After some time passed, the original track-pad is not produced anymore, but it has a better successor. The successor still looks the same, but it was improved internally and a few unnecessary pieces were removed - one of them the connector where the monitor was attached to.

A few years later a customer sends in a laptop for repair - with a broken track-pad. The original track-pads are out of stock and one of the new ones is used instead.

After the repair, the track-pad works fine but now the monitor stopped working. The engineer that designed the original monitor has left the company a while ago and the new engineers are left scratching their heads. They have to take the whole laptop apart until they find the problem.

Coupling unrelated things is bad - you make a change in one place and something completely unrelated breaks. It makes developers afraid of touching the code and refactoring becomes really hard.

Cohesion

Cohesion describes how close the elements of a component are to each other. We just talked about keeping unrelated things apart from each other, but if two pieces actually belong together, they should be in one place to achieve high cohesion. Aim for high cohesion and low coupling when you write code.

Let's come back to the laptop example one last time. If the laptop was designed properly, all the track-pad parts are closely grouped together - most likely located on the same board. Everything that belongs to the track-pad is soldered together and then connected as a unit to the rest of the laptop.

We have a clear separation between the components and a strictly defined interface to the outside world. If the track-pad depends on something from the outside, power for example, then it has a connector that fulfills a clearly defined specification.

The track-pad is responsible for one thing and nothing else. All the track-pad parts can be found in one place and there is nothing on the track-pad board that is not there to help the track-pad do its job.

Writing components with high cohesion leads to robust and maintainable code. It reduces complexity - each component has a small area of expertise where it excels. It also makes the component easy to reuse if the same functionality is needed in multiple places.

High cohesion and low coupling are closely related. Improving one of them will often improve the other one.

Don't repeat yourself

Duplication can turn into a maintenance nightmare - whether it's done on purpose or by accident.

I have been unlucky enough in the past to work with code where the same snippet was copy-pasted about 10 times inside the same class - and a few more times into other classes. Then the business requirements changed and the code had to be refactored.

The programmer that worked on it refactored most of the snippets, but also missed a few of them. This was repeated a couple of times and then nobody could remember which ones were correct and which ones were not.

It took a lot of effort to clean up that code and to make sure that the software still ran correctly. This could have been easily avoided if the "don't repeat yourself" principle (DRY) had been followed.

Duplication can arise from more than just a simple copy and paste, it is not just limited to code. The whole ecosystem of your codebase can be affected - including things like documentation and data storage.

A very common DRY violation is the duplication of information in comments and code. If you write a comment that describes what the code is doing, then you are repeating the same information that is already communicated through the code.

Comments still have their place of course, as long as they explain things that can't be communicated through code. Why something was done, for example.

If you come across a comment that explains what the code is doing, then you need to refactor until the code is self-explaining. I will show you how to do that later in the book.

Following the DRY principle doesn't mean that you can't have caching, etc. Duplication is fine, as long as it is generated from a single authoritative source.

Not all code that looks the same should be deduplicated. You can have the exact same code in different places and each copy represents a different piece of knowledge. Just as you can have code that looks different, but represents the same piece of knowledge. DRY is about knowledge, not how similar code looks.

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
>
> *The don't repeat yourself principle*

3. Methods

Naming your methods

A function that belongs to a class is called a method. PHP uses the function keyword for methods, so that can be a little confusing. In the official PHP documentation the name method is used and we'll stick to that.

A method represents an action and the name of a method should always contain a verb to describe what the method is doing. `$user->name()` does not tell us a lot. But `$user->getName()` clarifies what the method is doing.

Make names descriptive, even if they result in longer method names. Make them as long as necessary, but no longer.

Be consistent with your names. You could call a method that reads data from a database `findUser()`, `fetchUser()` or `retrieveUser()`, but you should use only one variation. Pick one word per concept and stick with it, otherwise you just make it harder to read and write the code.

Number of parameters

Try to limit the number of your method parameters. Every additional parameter adds a little bit of mental overhead.

Sometimes, when a method has too many parameters, you can reduce them by grouping them together in an object. You will learn more about that during the tutorial.

```
// before
$user->setFavoriteRgbColor($red, $green, $blue);

// after
$color = new RgbColor($red, $green, $blue);
$user->setFavoriteColor($color);
```

Only do this with parameters that are closely related and belong together. If you find it hard to come up with a good name for the object, the parameters might not be a good fit to be grouped together.

If a method doesn't use a parameter, get rid of it. It can be tempting to keep it to avoid refactoring the method calls, but it's easy to refactor with an integrated development environment (IDE) and keeps technical debt low.

If you don't remove unused parameters, you might end up writing a lot of unecessary code to provide the value for a parameter that is not even being used anymore.

Flag parameters

Flag parameters are parameters that change what a method is doing. In the example below, there are two different methods merged into one.

```
public function sendNotification(Message $message, bool $sendAsEmail): void {
    if ($sendAsEmail) {
        // send email notification...
        return;
    }

    // send internal notification...
}
```

The flag parameter adds unnecessary mental overhead - you have to know what changes when you flip the switch. It's easy to remove the flag parameter if you refactor the method into smaller methods.

```
public function sendEmailNotification(Message $message) : void {
    // send email notification...
}

public function sendInternalNotification(Message $message) : void {
    // send internal notification...
}
```

Now we have two distinct methods, each one is responsible for a different action. You can refactor all the shared code into a separate private method to avoid code duplication.

Type declarations

Type declarations enforce the preconstraints of your methods and act as documentation of the API at the same time. If you are not using type declarations, then you have no guarantee that the user is passing the correct types into your method. They help with debugging and if a wrong type is passed into a method, the error is caught early and not deep into a stack trace.

```
public function setFavoriteColor($color);

public function getFavoriteColor();
```

Methods without their types declared are harder to understand. You have to dig into the code to figure out what's going on. If you want to know all the possible return types, then you have to go through all the code that is being called - line by line.

```
public function setFavoriteRgbColor(RgbColor $color): void;

public function getFavoriteColor(): RgbColor;
```

With type declarations added, the API is now clear and there is no room for misinterpretation. The methods state clearly what is allowed to be passed into them and what you have to expect in return. You don't have to look into every method before you call it.

Type declarations also assist your IDE. You don't have to open a separate file to look at the types. Your IDE can show you the complete method signature when you start typing.

If you can't specify a type, use a docblock comment to add the missing information.

```
/**
  * @param Pixel[] $pixels
  */
public function drawPixels(array $pixels): void;
```

Method visibility

PHP supports three different visibilities (private, protected and public). You can use them to declare from where your method can be accessed.

Private methods can only be accessed by the class itself. Protected methods can also be accessed by inheriting and parent classes. Public methods can be accessed from everywhere.

PHP will default to public visibility if you don't declare one. I recommend that you always add the visibility keyword - consistency is good and it clearly shows your intent. If you don't always declare the visibility, then it will be hard to spot a mistake where you forgot to declare it.

Your classes should be black boxes to the outside world. For this reason, the default visibility should be private. From the outside, it should only be possible to see the API that you want to provide.

Your protected methods are part of your API (if your class is not final). You have to provide backwards compatibility, documentation, etc for protected methods.

Inheritance should only be used when the class is designed for it. When you use composition, you can (and should) limit yourself to public and private methods.

Keep your methods small

Your methods should only be responsible for a single action and they should be kept small. Small methods are easier to understand and you have to keep fewer things in your head when you read them.

Ideally, most of your methods should be around 10 lines or less. If one is growing over 20 lines, you should have a closer look and consider whether you should refactor it. The amount of lines are only rough guidelines. The goal is better readability and sometimes a slightly longer method is easier to understand.

You also want to avoid too many levels of indentation in your methods. If you have too much indentation, it often makes sense to extract the inner part into its own method.

> If you need more than 3 levels of indentation, you're screwed anyways, and should fix your program.
>
> *Linus Torvalds*

As an example, we are going to refactor the following code to increase its readability.

```
public function sendBill(Order $order): void
{
    if ($order->getTotal() > 0) {
        $emailAddress = $order->getEmailAddress();
        if ($emailAddress !== null) {
            $subject = "Bill for order #{$order->getId()}";
            $content = $this->renderer->render(
                'Documents/Email/Bill.html',
                ['order' => $order]
            );
            $email = new Email(
                $emailAddress,
                $subject,
                $content
            );
            $this->emailQueue->send($email);
        } else {
            $content = $this->renderer->render(
                'Documents/Print/Bill.html',
                ['order' => $order]
            );
            $pdf = $this->htmlToPdfConverter->convert($content);
            $this->printQueue->addPdf($pdf);
        }
    }
}
```

The multiple levels of indentation make it hard to read the code. There are multiple code paths that can be followed and the method is responsible for more than one action (the Email/PDF decision, sending Email and printing PDF).

To make the code easier to read, we can create separate methods for printing and emailing the bills.

```
private function sendEmailBill(Order $order): void
{
    $subject = "Bill for order #{$order->getId()}";
    $content = $this->renderer->render(
        'Documents/Email/Bill.html',
        ['order' => $order]
    );
    $email = new Email(
        $order->getEmailAddress(),
        $subject,
        $content
    );
    $this->emailQueue->send($email);
}

private function sendPrintedBill(Order $order): void
{
    $content = $this->renderer->render(
        'Documents/Print/Bill.html',
        ['order' => $order]
    );
    $pdf = $this->htmlToPdfConverter->convert($content);
    $this->printQueue->addPdf($pdf);
}
```

Now the original method can be refactored so that it only contains the decision-making code. The sending and printing is deferred to the new methods.

```
public function sendBill(Order $order): void
{
    if ($order->getTotal() <= 0) {
        return;
    }

    if ($order->getEmailAddress() !== null) {
        $this->sendEmailBill($order);
        return;
    }

    $this->sendPrintedBill($order);
}
```

I also added guard statements during the refactoring (early returns from a method). They reduce the need for `else` clauses and simplify the code paths.

You don't have to worry about orders with total of zero or less after seeing the guard statement. You can be sure that after this point, the total is always higher than zero.

Because the decision making and sending are separated, debugging becomes easier. If you come across a decision making bug, you have very little code to look at and bugs will be visible and stand out.

If you have to debug the previous code, you need to separate the decision making code from the rest in your head when you read the code. That makes debugging harder for no good reason.

Comment why, not what

Small methods also help to document your code. New developers often use comments to describe what the next few lines are doing. Those comments can usually be removed when those lines are moved into their own method with a good name.

> If you have to spend effort into looking at a fragment of code to figure out what it's doing, then you should extract it into a function and name the function after that "what".
>
> *Martin Fowler*

When I first learned to program, my teacher forced me to comment every single line of code. This habit didn't stick with me for long, but still I wrote way too many comments during my first few years as a developer.

It's a good rule of thumb to only comment the reasoning behind why you are doing something. Avoid comments that explain what your code is doing. If you need a comment to understand what the code does, you should rewrite the code until you don't need those comments anymore.

You can make some exceptions to this rule, but only if they are really necessary. We looked at one exception earlier, with the docblock that added additional type information to the `array` type declaration.

I'm not saying that you should just delete all your comments. If a comment was necessary to understand the code, only remove it after you refactored the code to make it unecessary.

A good comment explains why something was done in a certain way, to make sure that you and other developers are not confused when they come across the code a year later. If applicable, a comment can also include a link to documentation or a reference to a related issue in your project management system.

If you have too many comments in your code, the important ones don't stand out from the crowd and readers will skip over them. Thats another good reason why you should limit yourself to only use why comments - to make sure that they are noticed.

Deduplicate your code

Code deduplication is another big benefit of small methods. All these small methods can be reused by other methods. Whenever you notice multiple code blocks that look similar, ask yourself if it can be refactored into a method that can be called from multiple places.

You have to remember that you should only refactor the code blocks into a single method if they represent the same piece of knowledge. Otherwise you can run into problems later when the logic of one part changes, but not the other.

Most of the time, refactoring similar code into a shared method is the right choice. There are always exceptions to the rules, but make sure you understand the rules before you think about making exceptions.

Command-query separation

The command-query separation (CQS) principle was devised by Bertrand Meyer. It states that a function or method should either be a command or query, but not both at the same time.

- A command changes state, but does not return anything
- A query returns a result, but does not change state

> Asking a question should not change the answer.
>
> Bertrand Meyer

A query should always return the same result, even if it is called multiple times in a row. No side effects should be triggered by the query method calls.

A command on the other hand should not return anything. If you are using PHP 7.1 or higher, then you should always declare that those methods return `void`.

CQS is a very useful guideline and in most cases, you should strictly follow it. But don't treat it as a dogma, there are some cases where an exception to the rule makes sense.

One example that Martin Fowler (a well known developer) likes to use is the pop method. In a single method call, it removes and returns an item from a stack. That is a convenient method, but it could be avoided by splitting it into two separate methods.

If you think that violating the CQS leads to cleaner code in a particular example, make sure that the name reflects what the method is doing. Creating a separate query method is probably a good idea too, to make sure that it's possible to read the state without modifying it.

A method like `pop` might never create any problems, but you should stick to the command-query separation by default. Each exception has to be carefully considered.

4. Objects

When you create a new instance of a class, that instance is called an object. The non-static state of an object is independent from other objects, even if they are from the same class.

As a general principle, you should hide as much information as possible behind your classes, to reduce the number of things that you have to keep in your head when you work with the code. It is much easier to work with a class that hides everything except a few public methods, compared to a class that has all of its inner workings exposed.

We talked about abstraction and the different parts of a laptop in an earlier chapter - track-pad, mouse, pointing device, etc. Those things, whether they have a real-world equivalent or not, can be represented by a class in your code.

Properties

The member variables of a class are called properties. These are the attributes of your object. If the class represents a laptop, the properties will be things like the components, name of the model and version number.

It's a good idea to keep the type of a property consistent. Don't use the same member variable to hold an integer, string and `DateTime` object. It will be easier to understand your code when you stick to one type.

The exception to this is `null`. It's perfectly fine to make your type nullable.

Many languages have a messed up concept of `null` and you might come across blog posts that tell you to never use `null` for that reason. In Java for example, every object is nullable. If you have a type declaration for `UserId`, you can always pass in `null` instead without triggering an error during compiling. This makes the code very fragile and you have to do a lot of `null` checking.

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965.
>
> *Tony Hoare*

It turned out that `null` wasn't really the problem, but making everything nullable was.

PHP doesn't have this problem. Type declarations only accept a `null` value if they are explicitly allowed with `?` or `$var = null`. We can use `null` to represent a non-existing value, without having to worry about the problems that other languages have.

Property Visibility

If you are doing object oriented programming with PHP, then you should never use the public keyword for your class properties. Never is a strong word, but let me explain my reasoning behind this statement.

When you use a public property, you are exposing the inside of your class to the public. Everyone that is using your class can read and modify that variable - it becomes a part of the public API of your class.

If your code is being used by other people, then you are breaking backwards compatibility for them if you change your API. Everyone that uses your code has to change their own code whenever you break backwards compatibility.

Instead of making your class properties public, you can use methods to change the state of your object. Then you can always change the inner workings of your class without having to break backwards compatibility.

The same argument applies to protected variables. They also make it impossible change them without breaking backwards compatibility in the process. You can use protected methods to give child classes access to a parents property.

Because of the reasons above, all your properties should be private. No exceptions.

The same applies to constants. With PHP 7.1 visibility modifiers were added. You can now make your constants private and protected.

Constants are constant, as the name implies. Even if someone uses a public constant from your class, they can't change the state of your object. But they are still part of your API. Keep that in mind if you decide to use them.

Prohibit inheritance by default

A lot of people think that object oriented programming is about inheritance, but this assumption is wrong. There is even a principle that's called "composition over inheritance" (we'll have a closer look at it later in the book).

Inheritance is only one of many tools. It should be used sparingly and only when it's right for the job. Only use it when a class is designed for inheritance. A class should not be inheritable by default.

PHP provides us with a keyword that prohibits inheritance: `final`. It would be better if there was a keyword to mark a class as extensible, instead of the other way around. But we have to work with what PHP gives us.

By default you should make all your classes final. It might seem a bit tedious at first to add the keyword whenever you create a new class. But it will soon become second nature and the intentions of your code will be communicated better.

If your code is going to be used by other people, you need to be careful when you use the final keyword. You have to make it possible to modify the behaviour of your component, ideally through interfaces. When you use final classes and provide an interface, other people can write their own implementations and wrapper classes.

Don't take it as a disadvantage of final that it's impossible to modify your code through extension - that's the main benefit. If people are using inheritance to modify the behaviour of your classes, then you have to provide backwards compatibility for all the non-private internals of your classes - not just the public methods. Think about this when you design reusable code and provide interfaces where they make sense.

You can always make a final class non-final. But it is much harder to do the opposite if your code is already being used by other people.

When a class is marked final, you don't have to mark all the methods as final - inheritance is already prohibited. But when you create a class with inheritance in mind, you should default to private variables and declare all your methods final. Then think about what the child classes should be able to do and remove the final keyword for the methods that you want child classes to overwrite.

Principles

There are a lot of principles that help you write good object oriented code. For example there is the well-known SOLID acronym.

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

And there are the general responsibility assignment software patterns (GRASP).

- Controller
- Creator
- High cohesion
- Indirection
- Information expert
- Low coupling
- Polymorphism
- Protected variations
- Pure fabrication

And there are also other principles, like the Law of Demeter. We are not going to cover these principles in detail in the theory part. They will be introduced during the tutorial, when we come across a good example where the principle is being used.

These principles have helped me a lot on my journey to write better code. They are a great teaching tool and easy to reference when you are unsure about how to write a particular piece of code.

But remember that only a Sith deals in absolutes - it is easy to become overzealous after you learn something new. The goal is to write clean code and the principles should help you with that, but don't make your code worse by blindly following rules. There are always exceptions.

Good code is the result of more than just following a bunch of rules. You will have to make many decisions and often there will be more than one way to solve a problem. Follow your gut and experience in those cases and remember that refactoring is always an option if you find out later that you went with a suboptimal choice.

But don't let the paragraphs above devalue the principles, they have proven their worth and in the vast majority of cases, your code will be better if you follow them.

You ain't gonna need it

This is an important counterpart to the "don't repeat yourself" (DRY) principle, but it also applies to other areas.

When you first learn about DRY, it is easy to take it too far. Of course you should always deduplicate your logic when an opportunity presents itself, but don't get too far ahead of yourself. Sometimes developers become infatuated with the DRY principle and they always try to create a reusable solution, even when there is no need for one.

Do not try to anticipate the future and don't write code just because you might use it in the future.

Trying to future-proof yourself against duplicate code is a recipe for disaster. You only have one piece of reference and even if you are really good at predicting the future, you will get it wrong more often than not.

> The cheapest, fastest, and most reliable components of a computer system are those that aren't there.
>
> *Gordon Bell*

Write code to solve your current problem and nothing more. Refactoring is something that is done with the benefit of hindsight, not in advance.

The "you ain't gonna need it" principle (YAGNI) is meant to keep overengineering in check. If you follow the principles in this book, then it will be very easy to refactor your code when you have to change something. Don't be afraid of refactoring.

Writing code is like writing a book - most good writers don't get it right on the first try. They rewrite their chapters many times, until they are happy with the result. Develop software in the same way.

This doesn't mean that you should throw planning completely out of the window. If something was properly planned, the specification is written and you know it will be implemented in a week or two, go ahead and write your code with that in mind.

It also doesn't mean that you shouldn't build flexibility into your code. YAGNI only works if it's easy to refactor your code. Follow the object oriented programming principles and the code will be easy to modify later.

> Always implement things when you actually need them, never when you just foresee that you need them.
>
> *Ron Jeffries*

The YAGNI principle also applies to performance optimizations. I have seen countless developers who are trying to fix performance issues before there is a problem. I am not talking obvious things like nested loops, but small things like trying really hard to avoid an additional DB query.

If your program is slow, benchmark it first and then figure out where the bottleneck is. Fix it and then repeat this process until your software is fast enough. Don't spend a lot of time worrying about performance issues when it is very likely that no one will ever encounter them.

You can also look at it from a project management perspective. You were assigned a number of tasks and they were prioritized in order of what benefits the business the most, even if you don't agree with the priorities.

It would be irresponsible to start working on something that hasn't been properly planned and assigned to you. Focus on completing your current task and only do what is necessary for that.

Part II: Tutorial

1. The Front Controller

The project

During the tutorial we are going to build a small social news website, like hackernews or reddit. We will only implement a few basic features, the goal is not to build a large application, but to make you a better developer over the course of the book.

The focus of this book is on PHP, for this reason we will keep the rest of the website very simple. We will use a very minimal design and use as little JavaScript as we can get away with.

I highly recommend that you use version control, like Git, during the project. If you are not familiar with Git, now would be the perfect time to learn how to use it. I can recommend the Github tutorial (https://try.github.io), but there are also other good tutorials that you can find online.

Even though I highly recommend testing your application with unit, functional and integration tests, they won't be a part of this tutorial.

You can use any editor or IDE that you like, but I highly recommend PhpStorm from JetBrains. I am much more productive thanks to PhpStorm.

During the tutorial we are not going to use a framework. As you will see, you should decouple your application logic from your framework anyway. It doesn't really matter which framework you use, as long as you follow the principles presented in this book.

Not using a framework doesn't mean that you will have to write all the code from scratch. Instead, we will use Composer to install 3rd party packages. There is no need to reinvent the wheel.

Random thought: "Don't reinvent the wheel" is a weird saying, the wheel has been reinvented many times and was improved a lot in the process.

Whether you prefer to use a framework or not, the lessons from this book will be directly applicable to your work and the skills that you gain will be transferable even if you have to switch to a new framework for your next job.

The code of the completed tutorial project is available on Github (https://github.com/PatrickLouys/professional-php-sample-code).

The development environment

We will use the built-in PHP web server and SQLite for our database. I don't recommend to use them for production, but they are perfect for this tutorial. You don't have to spend much time to set everything up before you can start writing code.

This tutorial is written for PHP 7.2, but 7.1 should also work. If you haven't installed it already, you can find the instructions for your operating system on the PHP website (http://php.net/manual/en/install.php). Make sure that you also have SQLite installed and enabled on your system.

You need Composer (https://getcomposer.org) for dependency management. Follow the instructions on the website to install it either in the tutorial project folder, or system-wide. That's up to you, either way will work.

I will refer to composer commands with just `composer ...` in the rest of the tutorial. But depending on how you set it up, you might have to run those commands as `php composer.phar ...` instead.

That is all. You're now ready to start writing code.

A little history lesson

Back in the early days of PHP it was very common to have multiple PHP files, where each one served as an entry point for a different page.

If a user navigated to `/submit.php`, it would open up the corresponding `submit.php` file. The URL did not look very nice, so we removed the `.php` with a rule in the `.htaccess` file. If we wanted to share something between pages, we had to include the same file in all those scripts.

This approach can still work and with proper care, you might end up with reasonably maintainable code. But nowadays almost all newly written applications use a different pattern.

Introducing the front controller

The front controller pattern turns the old approach on its head. Instead of having all those different entry points, we route all the requests through a single file. It is then up to your application to make sure that each URL path maps to the correct method call.

> Front controller - a controller that handles all requests for a web site.
>
> *Patterns of Enterprise Application Architecture*

A front controller gives you a lot of freedom in regard to how you structure your URIs. You can use `/submissions/42/comment`, where the number is a variable value. If you wanted to do that without a front controller, this would have to be a custom rule in your web server config. That would couple your application to your web server. It is much more reasonable to keep all the logic in your PHP code, it's easier for developers and changing to a different web server becomes a trivial problem.

Decoupling is a common theme that we will talk about a lot during this book. We always try to keep coupling to a minimum between components - in this case the web server and your application code. That is one of the core principles of writing maintainable code.

We will use the PHP built-in web server for this tutorial. We won't have to set up any special routing rules, it works out of the box. The built-in web server will always try to route the requests to an `index.php` file if the URI does not specify a file.

> The PHP built-in web server is not meant for production use. It works great for development and trying out things, but it's not a fully featured web server and should not be used in a public network.

In an ideal world you would want your development environment to be a perfect copy of your production environment. But for this tutorial we will keep things simple. I don't want you to read a whole chapter about setting up your environment before you can start writing your first line of code. That's not why you opened this book.

Of course you're free to use whatever web server that you prefer for the tutorial, but then you are on your own. I had a lot of people contact me who had trouble with my `no-framework-tutorial` on Github, and in most cases it turned out that they used a different web server, and they did not configure it correctly. You can avoid those problems by sticking with the built-in web server until the end of the tutorial. If you choose to use a different server despite my warnings, please make sure you understand how to set it up correctly for use with a front controller before you continue with the tutorial.

Setting up the front controller

You must be getting anxious, so many pages and you haven't written a single line of code yet. There is one last thing that we need to cover, before you can write your first line. Where do we put the front controller?

If you have used a recent framework, you probably already know that you should not put it in the root directory of your project.

Why not? The `index.php` is the starting point, so it has to be inside the web servers public directory. That means that the web server has access to all the subdirectories of your application. If everything is set up properly, you can prevent users from accessing your subfolders where your application files are.

But sometimes things don't go as planned. If something goes wrong and your files are set up as decribed above, your whole application source code could be exposed to visitors.

Instead of doing that, create a `public` folder inside your projects root folder. This is the home of your `index.php` file and static assets like JavaScript files, CSS files and images.

Let's start with a very simple `index.php`, just to make sure that everything is working.

public/index.php

```
<?php declare(strict_types=1);

echo 'Hello, World!';
```

Open up a terminal window and navigate into your projects public folder. In there, type `php -S localhost:8000` and press enter. This will start the built-in webserver and you can access your page in a browser of your choice with `http://localhost:8000`. You should see the "hello world" message dislayed.

> `declare(strict_types = 1)` sets the current file to strict typing. In this tutorial we are going to use this for all PHP files. The result of this is that you can't just pass an integer as a parameter to a method that requires a string. If you don't use strict mode, it would be automatically casted to the required type. With strict mode, it will throw an Exception when you pass a wrong type.

Only displaying "Hello World" is pretty boring. Let's write some more exciting code, but remember that anything in the public folder could be exposed to visitors. For this reason we will limit the content of the `index.php` to only a single `require` and move all the application code into a different directory.

Navigate back to the root directory of your project and create a `src` directory. In there, create a `Bootstrap.php` file with the following content:

src/Bootstrap.php

```
<?php declare(strict_types=1);

echo 'Hello, from the bootstrap file :)';
```

Now modify your `index.php` to match the following code:

public/index.php

```
<?php declare(strict_types=1);

ini_set('display_errors', '1');
error_reporting(E_ALL);

require __DIR__ . '/../src/Bootstrap.php';
```

Now refresh your browser tab and make sure that the new text shows up.

> `__DIR__` is a magic constant that contains the path of the directory. By using it, you can make sure that the `require` always uses the same relative path to the file it is used in. Otherwise, if you call the `index.php` from a different folder it will not find the file.

We add the error reporting code here, otherwise you would be shown a blank page if there is an error in your `Bootstrap.php` file. If you ever come across an unexpected blank page, check the console window where the built-in server is running to see the actual error.

Your project folder structure should now look like the following:

```
public/index.php
src/Bootstrap.php
```

2. Bootstrapping

Introduction

The term “bootstrapping” comes from the old saying “pull yourself up by your own bootstraps”. In computing, it refers to the first piece of code that is being run when the system is started. It is often shorted to “booting”, which is a term that you probably have heard before.

In a web application, the bootstrap file is responsible for setting up your application. After that it will map the request to the correct function call, usually with the help of a separate routing component.

The bootstrap file is the only place in our code where you will have procedural code. After the bootstrapping is done, all your code will be object oriented.

Setting up Composer

We don’t want to write everything from scratch. Routing is a solved problem for example, it is something that every web application needs. So instead of writing our own routing code, we can use an existing library instead.

To install and to keep those libraries up to date, we are going to use Composer as our dependency manager. You should have installed it already if you followed all the steps of the “the development environment” subchapter. If not, go back and do it now.

Create a new file in your project root directory with the name `composer.json`. This is the Composer configuration file that will be used to configure your project and its dependencies. It must be valid JSON or the Composer commands are going to fail.

composer.json

```
{
    "require": {
        "php": "~7.2.0"
    },
    "autoload": {
        "psr-4": {
            "SocialNews\\": "src/"
        }
    }
}
```

With the `require` key you are telling Composer which packages your project depends on. For now, the PHP version is all that we are going to specify. We will add more later.

Composer can also take care of your class autoloading, which is what the `autoload` key is for.

Autoloading

With an autoloader you won't have to use `require` to include every single file into your application. Instead, the fully qualified class name will be used to figure out which file to include. PSR-4 is an autoloading specification that describes how to autoload classes from filepaths (http://www.php-fig.org/psr/psr-4).

`SocialNews\Foo\Bar` tries to locate the file at `src/Foo/Bar.php`.

Autoloading is very useful, but it can also be a source of confusion if you are not used to it. If you get an error that a class can't be found, always check that the physical path matches the namespace. If you have a typo somewhere or placed something in the wrong folder, then it won't find the file.

Composer will automatically generate a `vendor/autoload.php` file whenever you run `composer install`. It will also place all the code for the dependencies of your project in there. Never edit any file that is inside that folder, it will get overwritten by Composer. You should also exclude this folder from your version control. Follow the instructions on the Composer website (`https://getcomposer.org`) to set up your version control correctly.

To include the autoloading in your project, run `composer install` in your project root folder and then require `autoload.php` in your `Bootstrap.php` file.

src/Bootstrap.php

```
<?php declare(strict_types=1);

define('ROOT_DIR', dirname(__DIR__));

require ROOT_DIR . '/vendor/autoload.php';

echo 'Hello from the bootstrap file :)';
```

The `ROOT_DIR` constant is a fixed reference point to the root of our project folder. If we would always use relative paths instead, then you wouldn't be able to move code around in your project without refactoring all the paths. Using the constant will give you a fixed starting point for your file directory paths.

Pretty error handling

An error handler allows you to customize what happens when an error happens. A nice error page, with a lot of information for debugging, will help you a lot during development.

For this tutorial, we will use the Tracy package from the Nette framework. It adds useful debugging tools in addition to making the error page more developer-friendly.

To install Tracy, run `composer require tracy/tracy` and then add the following line just below the `require` statement in your `Bootstrap.php`:

src/Bootstrap.php

```
// ...

\Tracy\Debugger::enable();

// ...
```

If you navigate to your project site in the browser now, it should display a floating debugger bar in the bottom right corner.

> Never leave your debugger turned on in your production environment! Showing a stack trace to a malicious user can give him information about how to attack your system, or even worse, display secret information like your database password if you left that in your code. Always show a nice error page to your visitors and log the stacktrace somewhere else that is only visible to you.

If you want to see what Tracy does to your error pages, add `throw new \Exception;` to the end of your bootstrap file and refresh the page. Don't forget the remove the exception from your code before you continue with the tutorial.

HTTP

In PHP, information about the current request can be accessed through superglobal variables like `$_GET` and `$_POST`. The response is generated by functions like `echo` and `header`. But outside of our bootstrap file, we want to have all our code written in an object oriented way. We need something with a cleaner interface than the superglobals provided by PHP.

Using the superglobals directly is not a good idea. You will never know if some other part of the application modified them. They can be useful for some quick debugging, but besides that, you should only use a request object in your code.

We are going to use a package that gives us an object oriented interface for everything that is HTTP related. Install the very popular HttpFoundation component from the Symfony framework by running the command `composer require symfony/http-foundation` in your console.

To try out the HttpFoundation component, replace the `echo` line in your bootstrap file with the following code.

src/Bootstrap.php

```
// ...

$request = \Symfony\Component\HttpFoundation\Request::createFromGlobals();

$content = 'Hello ' . $request->get('name', 'visitor');

$response = new \Symfony\Component\HttpFoundation\Response($content);
$response->prepare($request);
$response->send();
```

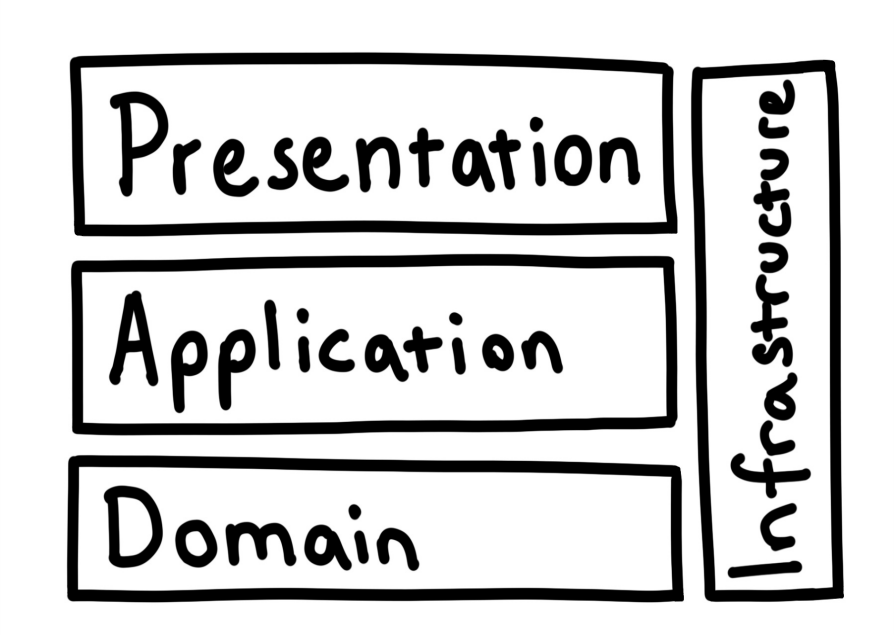
Your page should now display the message `Hello visitor`. Change the URL in your browser to `http://localhost:8000/?name=Anakin` to make sure that accessing the `GET` parameter also works.

Introducing controllers

The code in your bootstrap file should only be responsible for the setup of your application, it shouldn't do anything else. So we need a different place for the code that responds to a particular request.

There are a few different possible ways to set this up, but in the end all of them come down to calling a function or method. Using controller classes, that contain one or more methods, is the most common way. They work well and have few drawbacks, so that's what we are going to use in this tutorial.

Before we continue, we need a quick introduction to the layered architecture that we will use. Everything that we do at the moment is part of the presentation layer, the other layers will be introduced later.



Before we separate our code into layers, we want to separate it by context. As mentioned earlier, decoupling unrelated parts is a good idea. For this reason we are going to move every component into its own directory with its own layers.

Create a new directory 'FrontPage' in your 'src' directory. This component is responsible for rendering the front page of your social news site, where all the stories are aggregated. In there create a `Presentation` directory for the layer and place a `FrontPageController.php` file with the following content in there.

src/FrontPage/Presentation/FrontPageController.php

```
<?php declare(strict_types=1);

namespace SocialNews\FrontPage\Presentation;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

final class FrontPageController
{
    public function show(Request $request): Response
    {
        $content = 'Hello, ' . $request->get('name', 'visitor');
        return new Response($content);
    }
}
```

What are we doing here? First we have the namespace and the `use` statements of course. If you are not familiar with them, have a look at the corresponding PHP documentation pages before you continue.

If this is your first time using PHP 7, then `: Response` might throw you off a little. That is a return type declaration. If that is new to you, go check out all the useful new features that PHP 7.0, 7.1 and 7.2 brought to the language.

Instead of generating the response in the bootstrap file, it is now the responsibility of the controller methods to do that. The bootstrap file will pass the `Request` object to the method as a parameter and then expect a `Response` object in return.

src/Bootstrap.php

```
// ...

$controller = new \SocialNews\FrontPage\Presentation\FrontPageController();
$response = $controller->show($request);

if (!$response instanceof \Symfony\Component\HttpFoundation\Response) {
    throw new \Exception('Controller methods must return a Response object');
}

// ...
```

Add the code above to your bootstrap file. The whole file should now look like the code below.

src/Bootstrap.php

```
<?php declare(strict_types=1);

define('ROOT_DIR', dirname(__DIR__));

require ROOT_DIR . '/vendor/autoload.php';

\Tracy\Debugger::enable();

$request = \Symfony\Component\HttpFoundation\Request::createFromGlobals();

$controller = new \SocialNews\FrontPage\Presentation\FrontPageController();
$response = $controller->show($request);

if (!$response instanceof \Symfony\Component\HttpFoundation\Response) {
    throw new \Exception('Controller methods must return a Response object');
}

$response->prepare($request);
$response->send();
```

We added a few lines of code to make sure that the controller returns a `Response` object. Otherwise the error message could be a little cryptic if you return a wrong type.

Routing

Our current setup works well with the single controller method that we have. But if a different URL is being used, different controller methods should respond. Let's add a second controller.

First we need a new context for everything that has to do with submitting content to the website. Create the appropriate directories (`Submission` and `Presentation`) and place a new `SubmissionController.php` file in there.

src/Submission/Presentation/SubmissionController.php

```php
<?php declare(strict_types=1);

namespace SocialNews\Submission\Presentation;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

final class SubmissionController
{
    public function show(Request $request): Response
    {
        $content = 'Submission Controller';
        return new Response($content);
    }
}
```

How can we make this accessible with its own URL? You could try the following approach.

Don't do this

```php
if ($request->getPathInfo() === '/') {
    $controller = new \SocialNews\FrontPage\Presentation\FrontPageController();
    $response = $controller->show($request);
} elseif ($request->getPathInfo() === '/submit') {
    $controller = new \SocialNews\Submission\Presentation\SubmissionController();
    $response = $controller->show($request);
} //...
```

This code is very crude and it could be written in a nicer way, but there is no point in doing that. The whole approach is flawed. As more and more routes are added to the application, the code will grow out of control quickly and turn into an unreadable mess.

Routing is a solved problem, there are plenty of libraries available for this. So writing our own router would be a waste of time.

We will use FastRoute in this tutorial, which is a fast request router created by PHP contributor Nikita Popov. To install the package, run the command `composer require nikic/fast-route`.

src/Bootstrap.php

```
// ...

$controller = new \SocialNews\FrontPage\Presentation\FrontPageController();
$response = $controller->show($request);

// ...
```

Replace the two lines above with the code below.

src/Bootstrap.php

```
// ...

$dispatcher = \FastRoute\simpleDispatcher(
    function (\FastRoute\RouteCollector $r) {
        $r->addRoute(
            'GET',
            '/',
            'SocialNews\FrontPage\Presentation\FrontPageController#show'
        );
        $r->addRoute(
            'GET',
            '/submit',
            'SocialNews\Submission\Presentation\SubmissionController#show'
        );
    }
);

// ...
```

This part defines the routes. The third parameter for `addRoute` is the handler, which can be any variable with FastRoute. It could be a function, an array or, as in this case, a string. I went with `Controller#method` because I think it's easier to read compared to an array. For the tutorial I recommend that you stick with the convention that I set, just because it will be easier to follow along.

Add the following code just below the previous code block.

src/Bootstrap.php

```
// ...

$routeInfo = $dispatcher->dispatch(
    $request->getMethod(),
    $request->getPathInfo()
);

switch ($routeInfo[0]) {
    case \FastRoute\Dispatcher::NOT_FOUND:
        $response = new \Symfony\Component\HttpFoundation\Response(
            'Not found',
            404
        );
        break;
    case \FastRoute\Dispatcher::METHOD_NOT_ALLOWED:
        $response = new \Symfony\Component\HttpFoundation\Response(
            'Method not allowed',
            405
        );
        break;
    case \FastRoute\Dispatcher::FOUND:
        [$controllerName, $method] = explode('#', $routeInfo[1]);
        $vars = $routeInfo[2];

        $controller = new $controllerName;
        $response = $controller->$method($request, $vars);
        break;
}

// ...
```

In this code, we ask the router for the result and then react accordingly. If a route matched, we use the handler information to first instantiate the controller class and then call the appropriate method.

We pass the request object and the route variables to the controller method, so that we can access them from there. If `[$controllerName, $method]` is confusing you, that's just one of the new PHP features. It's doing exactly the same thing as `list($controllerName, $method)`.

To check that everything works, visit `/` and `/submit`. Also try a non-existing page to see if the 404 error is displayed.

The code works, but the routes don't really belong into the bootstrap file. We should move them into their own separate config file instead. Create a `Routes.php` file in your `src` folder.

src/Routes.php

```
<?php declare(strict_types=1);

return [
    [
        'GET',
        '/',
        'SocialNews\FrontPage\Presentation\FrontPageController#show'
    ],
    [
        'GET',
        '/submit',
        'SocialNews\Submission\Presentation\SubmissionController#show'
    ],
];
```

In there, we keep everything as a plain PHP array instead of dealing with the dispatcher object like we did in the bootstrap file. It's just a config file. You could also use JSON, XML, YAML or any other config file format, but to keep it simple we will just use a plain PHP array.

On larger applications a single route file can quickly grow too large to maintain. If that happens, you can create a separate route file for each component and then aggregate those together. But if it's just a small application then a single file is easier to maintain. Only start refactoring it when it becomes a problem.

Now we need to change a few lines in the bootstrap file to keep things working. Replace the dispatcher part with the following code.

src/Bootstrap.php

```
// ...

$dispatcher = \FastRoute\simpleDispatcher(
    function (\FastRoute\RouteCollector $r) {
        $routes = include(ROOT_DIR . '/src/Routes.php');
        foreach ($routes as $route) {
            $r->addRoute(...$route);
        }
    }
);

// ...
```

The splat operator (three dots) unpacks the array into arguments. So `...$route` results in `$route[0], $route[1], $route[2]` in this case. Check that your code still works before moving forward.

You did a lot of coding in this chapter. Most of the work for wiring things together is now done and you created your own little custom framework in the process.

That was pretty easy, wasn't it?

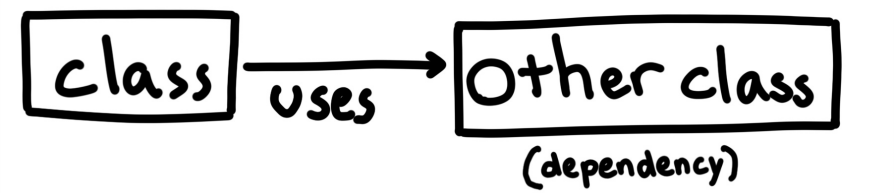
We will do some more changes in the bootstrap file in the upcoming chapter, but after that you shouldn't have to touch the bootstrap file anymore, unless you need to change the initialization of your application.

3. Dependency Injection

Introduction

Now we have our first two controllers, but they are not really doing anything. It would be nice if we could display a nice HTML template instead of just plain text.

A controller has the responsibility to take a request and then return an appropriate response. Rendering HTML is not something that the controller should do by itself, that would be the job of a different class. When one of your classes uses a different class, to help your class to do its job, that other class is called a dependency.



A few years ago, when I was still a CodeIgniter developer, the word dependency injection (DI) sounded scary to me. It made me imagine this large and confusing thing that I would have to learn about at some point, but I kept procrastinating.

With the power of hindsight, I now can see how wrong my preconceived notions of DI were. It is a really simple concept, hiding behind a name that makes it sound complicated.

The first dependency of our controller will be a template renderer. We need to find a way to give the controller access to an instance of a `TemplateRenderer`. One option would be to create a new instance whenever we need it, as shown in the following code.

Don't do this

```
final class FrontPageController
{
    private $templateRenderer;

    public function __construct()
    {
        $this->templateRenderer = new TemplateRenderer();
    }
}
```

This approach has a few drawbacks. The biggest one is that we create a strong coupling between the `FrontPageController` and `TemplateRenderer`. If the renderer has one or more constructor arguments, then the controller would have to supply those too. That makes it hard to change the constructor dependencies of the template renderer without also breaking all the other classes that use it.

By using dependency inversion, we invert the whole approach.

This code is not part of the tutorial

```
final class FrontPageController
{
    private $templateRenderer;

    public function __construct(TemplateRenderer $templateRenderer)
    {
        $this->templateRenderer = $templateRenderer;
    }
}
```

Your class now asks for an instance of its dependency, instead of creating one itself. It is then up to whoever creates a new instance of your class, to provide it with a correct set of dependencies. Never use the `new` keyword to add a dependency to a class.

That is all there is to dependency injection. Instead of creating the dependency, we ask the creator of the class to inject it instead.

If you are not used to programming like this, it might take a while to make the mental shift and to get used to it. But by the end of this tutorial, I can guarantee that using dependency injection will have become second nature to you.

I have seen some developers take this idea of not using `new` a little too far, they try to avoid the `new` keyword at all costs in their code. But you need to instantiate objects somewhere and there is nothing inherently bad about using `new'. Just don't use it to create the dependencies of your class from inside that class.

Using the `new` keyword is completely fine when you are creating an object that your method is going to return or if you need to create an object that you need to pass into a function or method call. In both those cases, your class is not using the object as a dependency, it is passing it along. This is fine.

> Always ask for your dependencies. Only instantiate objects that your method is going to return or pass to another method.

Depending on an interface

Using dependency injection adds other benefits as well. Instead of depending on a concrete class, the Twig renderer for example, your class can depend on an interface instead. You can think of an interface as a contract, that the implementing classes must fulfill.

But what's the point of all that? You would have to write much less code if you just skip that and use the Twig renderer directly. Why bother with an interface?

Depending on an interface, instead of a concrete class, gives you a lot of flexibility. You can switch to a completely different implementation without changing the class that uses it, as long as your new dependency implements the same interface.

You are probably not going to change the template renderer in the middle of a project, but it is still a good idea to start with the interface. When you think about using a component from a third party, using an interface forces you to think about what you actually need for that particular use case, before you look at the details of how you are going to implement it.

Your interfaces should be tailor-made for the code that consumes them. They should not just act as a collection of methods, of which only a few are used by the client code.

> No client should be forced to depend on methods it does not use.
>
> *Robert C. Martin*

I think the reason why many programmers end up with generalized interfaces is because they overemphasize the don't repeat yourself principle. If you focus on DRY too much, you end up with code that tries to solve too many problems at once.

A nice thing about interfaces is that you can implement more than one. That means that you can keep things DRY in your implementation class and provide multiple client-specific interfaces at the same time.

Where do we put the interface for the renderer? If you were to use a framework, this would be part of the framework and you would not have to create your own interface and class. We will stick to that theme.

Create a 'Framework' directory in your 'src' directory. In your `Framework` directory, create a `Rendering` directory and place a `TemplateRenderer.php` file in there, with the following code.

src/Framework/Rendering/TemplateRenderer.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rendering;

interface TemplateRenderer
{
    public function render(string $template, array $data = []): string;
}
```

For now, this is all that our renderer can do. It takes the name of a template, with some optional data that can be passed along, and then returns a string with the rendered HTML.

Update your front page controller and add the `TemplateRenderer` as a dependency. Look at the contructor dependency injection example that I showed earlier in this chapter if you get stuck. Don't forget add an `use` statement at the top to import the namespace of the renderer.

src/FrontPage/Presentation/FrontPageController.php

```php
<?php declare(strict_types=1);

namespace SocialNews\FrontPage\Presentation;

use SocialNews\Framework\Rendering\TemplateRenderer;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

final class FrontPageController
{
    private $templateRenderer;

    public function __construct(TemplateRenderer $templateRenderer)
    {
        $this->templateRenderer = $templateRenderer;
    }

    public function show(Request $request): Response
    {
        $content = 'Hello, ' . $request->get('name', 'visitor');
        return new Response($content);
    }
}
```

The interface suffix debate

The use of an `Interface` suffix has become quite common in the PHP community. The consensus is slowly shifting towards dropping the suffix, but when this book was being written, the use was still pretty widespread.

In C#, where the community uses an `I` prefix for interfaces, both `extends` and `implements` are mixed together. The prefix is a convention that helps to make the distinction between the two. In languages like Java and PHP, there is already a distinction with the explicit difference between `extends` and `implements`, so this is not a valid reason to use a suffix when using a language like PHP.

Interfaces are an important part of object oriented programming. They represent a type, just like a class does, and they should be named after the type that they represent. Naming them after what they are doesn't make any sense. We don't add a `Class` suffix to all our classes and a `Variable` suffix to all our variables, why would we make an exception for interfaces?

Let's compare the two versions and see how they look in our constructor.

```
public function __construct(TemplateRendererInterface $templateRenderer) {}

public function __construct(TemplateRenderer $templateRenderer) {}
```

The `Interface` word is just noise here and it doesn't add any information to the code. The client code does not care about the distinction between interfaces and classes, all it wants is an object of the `TemplateRenderer` type.

The class that is using a template renderer doesn't need a `TemplateRendererInterface`, it needs a `TemplateRenderer`. The class has no reason to make a distinction between interface and class, all it cares about is that it gets an instance which adheres to the contract.

By not using a suffix, we can easily refactor an interface into a class or the other way around. A proper IDE can take care of this even if you use a suffix, but if you have other code depend on your API, then this refactoring will break backwards compatibility.

One reasonable argument in favor of using a suffix is that it allows you to see which files are interfaces without having to open them. But if you use an IDE (highly recommended), then you can see this information even without the suffix.

The creation of an abstraction and then hiding the implementation details behind it is one of the core ideas of object oriented programming. By using an `Interface` suffix, you are leaking the implementation details. Client code doesn't care about whether your type is an interface or a class. By asking the client code to depend on a `TemplateRendererInterface`, you force it to tightly couple itself to the fact that the type is implemented with an interface instead of a class (and vice-versa).

A likely reason that some developers prefer the `Interface` suffix is probably laziness. It makes it easier to settle on a class name, even if that name is not ideal. Instead of spending additional time to think about what to call the implementation, they just end up with a `TemplateRenderInterface` and a `TemplateRenderer` implementation.

But the use of an interface implies that there can be multiple implementations. What makes the first implementation so special that it deserves the name `TemplateRenderer`? That name is also not a very precise name to describe an implementation.

If our renderer is using Twig, then using a name like `TwigTemplateRenderer` is much more descriptive compared to just naming it `TemplateRenderer`.

Whenever you implement an interface, you should make the name of the implementation a more specific version of the interface name. If you still have trouble coming up with a good name, you might want to reconsider whether an interface is the right choice in that particular case.

The suffix makes it easier to find a name, but you get less accurate names. Don't take the easy way out, make sure that you give everything a descriptive name. Sometimes you might have to take a few minutes, and maybe even talk to some other developers, before you settle on a name. Naming things is hard, but it's also very important if you want to write maintainable code.

The implementation

With the help of our new interface, we can see that Twig, like many other templating packages, is able to do the job. But because Twig doesn't implement our interface, we will have to write an adapter class to use it in our application. Install Twig by running `composer require twig/twig`.

Of course you don't need to do this if you are using a framework. Your framework probably already does this for you. The problem with Twig is that the package never adopted namespaces, it is stuck with the pre-namespace class naming that looks like `Twig_Loader_Filesystem`.

Because this is framework code, you can put the `TwigTemplateRenderer.php` file into the same folder as the interface.

src/Framework/Rendering/TwigTemplateRenderer.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rendering;

use Twig_Environment;

final class TwigTemplateRenderer implements TemplateRenderer
{
    private $twigEnvironment;

    public function __construct(Twig_Environment $twigEnvironment)
    {
        $this->twigEnvironment = $twigEnvironment;
    }

    public function render(string $template, array $data = []): string
    {
        return $this->twigEnvironment->render($template, $data);
    }
}
```

As you can see, using the adapter pattern is very straightforward. The controllers are completely decoupled from the implementation and it would be easy to switch out the renderer without having to change controller code.

Even if we stick with Twig forever, we can use that flexibility to add additional functionality, like caching or logging, to the renderer without changing any other code by using the decorator pattern.

Injecting an instance

The controller and renderer are now ready, but your homepage will show an error if you try to visit it.

```
Too few arguments to function
SocialNews\FrontPage\Presentation\FrontPageController,
0 passed in .../src/Presentation/Bootstrap.php on line 41 and exactly 1 expected
```

This was to be expected, we haven't actually created an instance of the `TwigTemplateRenderer` anywhere. We will have to configure Twig before we can use it. It doesn't really make sense to put all that code into the bootstrap file, so let's move that code into a factory.

By using a separate factory class, we can completely separate the construction logic of an object from the class itself. A factory is an object that has the responsibility of creating another object. It can have its own dependencies, if it needs them.

Factories are very useful, especially when there is some logic involved in setting up an object. You could even replace the dependency injector with only factories if you really wanted, but you would have to write a factory for every single class that needs to be instantiated. This would lead to a lot of boilerplate code and I don't recommend it, but its an interesting thought nonetheless.

Because the factory is closely coupled to the implementation, you can put it into the same folder as the `TwigTemplateRenderer`.

src/Framework/Rendering/TwigTemplateRendererFactory.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rendering;

use Twig_Loader_Filesystem;
use Twig_Environment;

final class TwigTemplateRendererFactory
{
    public function create(): TwigTemplateRenderer
    {
        $loader = new Twig_Loader_Filesystem([]);
        $twigEnvironment = new Twig_Environment($loader);
        return new TwigTemplateRenderer($twigEnvironment);
    }
}
```

Change the code that creates the controller in the bootstrap file to the following.

src/Bootstrap.php

```
// ...

$factory = new SocialNews\Framework\Rendering\TwigTemplateRendererFactory();
$templateRenderer = $factory->create();
$controller = new $controllerName($templateRenderer);

// ...
```

Check whether your pages are working before you continue with the tutorial.

A lot of controllers will have similar dependencies. For example we are going to use the renderer in most controllers, to help us return HTML to the browser.

Your first thought might be that it would be sensible to create an abstract controller, which all the others can extend. A lot of frameworks recommend this approach in their documentation. Here is an example of such a controller (in a simplified version).

Don't do this

```
abstract class Controller
{
    protected $templateRenderer;

    public function __construct(TemplateRenderer $templateRenderer)
    {
        $this->renderer = $templateRenderer;
    }
}
```

I have used this approach myself and I quickly learned why it is not a very good idea. We kept changing things and the abstract controller always broke the code. After a few months of using it, we had to refactor our code to remove the abstract controller. The result was cleaner code and fewer problems.

Still, there is a lot of successful software out there that uses an abstract controller. In the following part, I will show you how they make it work.

Service locators

A service locator is a class that knows how to instantiate all your classes (or at least a subset) and it is used by your classes as a dependency. We can inject this class into the abstract controller and in return, all our controllers will have access to any dependency they might need.

Don't do this

```
abstract class Controller
{
    protected $serviceLocator;

    public function __construct(ServiceLocator $serviceLocator)
    {
        $this->serviceLocator = $serviceLocator;
    }
}
```

You could then retrieve a template renderer like this.

Don't do this

```
final class FrontPageController
{
    public function show(Request $request): Response
    {
        $renderer = $this->serviceLocator->get(TemplateRenderer::class);
        // ...
    }
}
```

Why do frameworks like to use this approach? It lowers the barrier of entry and new developers won't have to understand constructor injection, they can start to write code straight away.

The problem with the service locator is that it hides the dependencies of your class. Instead of depending on the objects that your class needs to do its job, it depends on a class which knows about your whole application.

As a result, you can't just look at your class from the outside and create a new instance. Instead, you have to look inside and read the code line by line, just to find all its dependencies.

When you use a service locator, you are coupling all your classes to this all-knowing object. This is the opposite of low coupling and high cohesion. Going through the service locator for your dependencies adds an unecessary indirection that makes it harder to read your code.

Service locators make it possible to write fewer lines of code, which is something that is attractive for frameworks. Even more so when the framework is marketing itself as a tool for fast prototyping.

For the reasons above, the service locator is regarded as an anti-pattern by most senior developers when the goal is to write maintainable code. Because service locators got such a bad rap, newer frameworks usually avoid association with them. But if you take Laravel for example, their so called facades (not to be confused with the facade pattern) are just acting as a static service locator.

> Don't confuse the service locator anti-pattern with dependency injection containers. Using a container to create your classes and their dependencies is fine. The container only becomes a service locator when you inject the container itself into non-factory classes.

If you only use your controllers as glue code between HTTP and the rest of your application, then using service locators in your controllers can be ok. I would still not recommend that approach, but you can build large applications without adding a lot of technical debt in the process, as long as you are limiting the use to controllers.

Your framework could also be forcing you to use a service locator in your controllers because it doesn't support any other way. Maybe your team decides that its better than using constructor injection and you need to play along. If that is the case, just make sure that you keep the use of the service locator limited to the controllers, you will be fine.

Using a dependency injector

Having to write a factory for every single class can become cumbersome fast. To avoid this, we can use a recursive dependency injector. With the help of some configuration, the injector will wire together all our objects. You can also use factories together with the injector where it makes sense.

For this tutorial, we will use the auryn dependency injector package. Install it with `composer require rdlowrey/auryn`. As we did with the routes, we will put the dependency configuration into its own file. Create a `Dependencies.php` file in your `src` folder.

src/Dependencies.php

```
<?php declare(strict_types=1);

use Auryn\Injector;
use SocialNews\Framework\Rendering\TemplateRenderer;
use SocialNews\Framework\Rendering\TwigTemplateRendererFactory;

$injector = new Injector();

$injector->delegate(
    TemplateRenderer::class,
    function () use ($injector): TemplateRenderer {
        $factory = $injector->make(TwigTemplateRendererFactory::class);
        return $factory->create();
    }
);

return $injector;
```

Because we have a factory. we can use the `delegate` method of Auryn. The factory creation is wrapped into a closure so that we don't create a factory object when we don't need it.

In your bootstrap file, change the route found part to the following.

src/Bootstrap.php

```
// ...

case \FastRoute\Dispatcher::FOUND:
    [$controllerName, $method] = explode('#', $routeInfo[1]);
    $vars = $routeInfo[2];
    $injector = include('Dependencies.php');
    $controller = $injector->make($controllerName);
    $response = $controller->$method($request, $vars);
    break;

// ...
```

Visit your homepage and make sure that your site still works before you continue.

Composition over inheritance

You might have noticed that I have marked all the classes in this tutorial as `final`. I did that because I am a big proponent of composition over inheritance.

Sadly a lot of developers think that object oriented programming means that you should use a lot of inheritance. That's how you end up with code like the following

Don't do this

```
class Core extends Database {}
class Controller extends Core {}
class AuthController extends Controller {}
class AdminController extends AuthController {}
class DashboardController extends AdminController {}
```

You should only use inheritance when you, or someone else, designed a class explicitely with inheritance as a goal. That means by default, a class should not be used as a parent class. If you want to reuse code, use composition and not inheritance. I am not saying that you should never use inheritance, but use it very sparely. When you use inheritance, design those classes very carefully.

In my opinion, it would be better if we had to mark classes that are open for extension, instead of the other way around. But we have to work with what the languages gives us. Always using `final` might feel a bit forced in the beginning, but after a while it becomes second nature and you will immediately notice a missing `final`.

Some developer like to use inheritance to add functionality to an existing class and by using `final` we make this impossible. This encourages us, and other developers, to use composition and the decorator pattern instead.

One example of inheritance that I come across often is when a developer tries to share code between classes, but that is not a valid use case for inheritance. If you need to share functionality across several other classes then refactor that functionality into its own class and make that a dependency of the other classes.

Adding functionality to final classes is still possible, you just have to use the decorator pattern. A decorator adds functionality to a class by wrapping it, instead of extending it. The original class is being used as a constructor dependency and its public methods are called from the decorators methods, which might add some functionality before or after the method call.

And if you really need to, you can always remove the `final` from a class without breaking backwards compatibility. Even if you use `final` for everything, it doesn't mean that you can't extend one of your classes in the future.

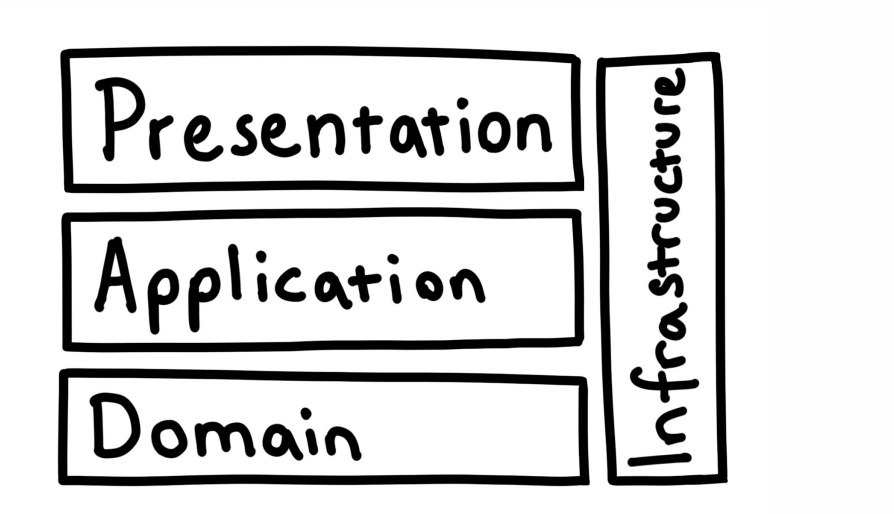
Having to remove the `final` keyword is a good reminder to have a closer look at the class before opening it up. Even if you want to make inheritance possible, it's also likely that you don't want to open up everything. You probably still want some things private and maybe even mark some methods `final`.

> Favor object composition over inheritance
>
> *Gang of Four (Design Patterns)*

4. Templating and Cross-site Scripting

The presentation layer

As mentioned earlier, during this tutorial we are going to use a layered architecture. This kind of architecture has many different names like hexagonal architecture, ports and adapters, onion architecture or clean architecture. But all those names try to convey the same idea of layers and how they are related to each other.



The first layer called the presentation layer. This layer encompasses everything from receiving a request from a user to returning an appropriate response. This layer is the glue between the medium that is used to access the application (like HTTP) and the application itself.

A few years ago, MVC (model-view-controller) was a very common buzzword in the PHP world. This architectural pattern was originally intended for implementing user interfaces in desktop applications. With the ruby on rails craze, it got popularized in the web application world. But the pattern was originally not created with a request-response environment in mind, so it had to be adapted.

The ruby on rails way was to simply have controllers, views (which were just templates) and models (which were just active record entities). But as the web development communities became more experienced, they started to learn about how to write better object oriented code.

When some developers started to replace active record entities with a proper object relational mapper (Doctrine ORM for example), they started to realized that the MVC pattern was not sufficient anymore. They started to realize that the model is an entire layer, not just an active record model.

The next step was the addition of an additional layer, because the "thin controller, fat model" approach became popular. So all the logic that didn't belong to the domain model, but also not into a controller, was moved into its own layer as application services.

As you can see, the MVC pattern slowly lost its meaning over time in the request-response environment. It will be hard to find two developers who can agree on what exactly MVC means. For this reason I am in favor of retiring the term, while keeping the good ideas from the pattern.

Having a clear separation between controllers, templates, application and the business logic is a very useful idea. Writing good software is all about separation of concerns, and MVC describes one way to achieve that separation. Don't get too hung up on whether your application is MVC or not, focus on the underlying principles that lead to maintainable code instead.

By the way, MVC is going through a little revival in the JavaScript world. While the original pattern does not play well with a request-response environment, it works perfectly well in the browser for the frontend part of the application. By retiring the term for the request-response backend, we will have less confusion when we talk about the frontend part.

The first template

In the last chapter, we made the `TemplateRenderer` a dependency of the front page controller. But we also need some templates before we can use the renderer.

To keep it simple, we will create a `templates` directory in your project root with a `FrontPage.html.twig` file inside. We won't bother with subdirectories for the templates because we are building a very small website with very few different pages. If it becomes a problem later down the road, it will be trivial refactor and add subdirectories later.

You could also put the templates into the folders of each component, but that would complicate the Twig setup. Both approaches have their own benefits. Putting the templates into the component folder makes everything very cohesive, but a separate folder will make it easier if you have a frontend developer working on the project.

templates/FrontPage.html.twig

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Social News</title>
</head>

<body>
<h1>Social News</h1>
<ol>
    <li>
        <a href="http://google.com">Google</a>
    </li>
    <li>
        <a href="http://bing.com">Bing</a>
    </li>
</ol>
</body>
</html>
```

In your front page controller, change the `show` method to render the template and then return the returned `string` as the response.

src/FrontPage/Presentation/FrontPageController.php

```
// ...

public function show(): Response
{
    $content = $this->templateRenderer->render('FrontPage.html.twig');
    return new Response($content);
}

// ...
```

Twig doesn't know where it has to look for the templates. We have to configure this in the `TwigTemplateRendererFactory`.

Value objects

We need to let the factory know where the templates are located. We already added a `ROOT_DIR` global constant that we can use to build the path, but instead of using the constant directly from multiple places in our code, let's create a class instead that we can inject.

We could create a generic `RootDirectory` class, but the template factory doesn't really need to now where the root directory is. The factory only needs to know the location of the template directory. Instead of building that path in the factory, let's move that logic into our new object.

src/Framework/Rendering/TemplateDirectory.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rendering;

final class TemplateDirectory
{
    private $templateDirectory;

    public function __construct(string $rootDirectory)
    {
        $this->templateDirectory = $rootDirectory . '/templates';
    }

    public function toString(): string
    {
        return $this->templateDirectory;
    }
}
```

You might ask yourself, why all this code when you could just use a simple string?

This is a value object, a class that represent a value. Value objects can be a simple scalar value, like the string in the example above, or they can be a combination of multiple values. They have some advantages in comparison to just using a scalar value.

The largest benefit is that you can declare the type for them. That makes your code much more explicit and easier to understand. Compare the two method calls below.

This code is not part of the tutorial

```
// scalars
$canvas->drawPixel(50, 25, 31, 132, 255);

// value objects
$canvas->drawPixel(new Coordinates(50, 25), new RGB(31, 142, 255));
```

The second one is easier to understand and it's much harder to mix up the parameter order. You could go even further in that example and add separate value objects for the remaining scalars.

There is no hard rule about how fine-grained you should go with your value objects. If you are not sure, always pick the option that leads to more readable code, even if it means writing a little more code.

Another advantage is that you can add some validation to your value objects, to make sure that they will never be in an invalid state.

This code is not part of the tutorial

```
public function __construct(int $red, int $green, int $blue)
{
    foreach ([$red, $green, $blue] as $color) {
        if ($color < 0 || $color > 255) {
            thrown new InvalidArgumentException(
                'Color values must be between 0 and 255'
            );
        }
    }

    // ...
}
```

We won't need any additional validation for our `TemplateDirectory` value object at the moment, any string will be valid for the root directory argument. To make it injectable and to provide a value for the root directory path, you have to configure it in the dependencies file.

src/Dependencies.php

```
use SocialNews\Framework\Rendering\TemplateDirectory;

// ...

$injector->define(TemplateDirectory::class, [':rootDirectory' => ROOT_DIR]);

// ...
```

In your `TwigTemplateRendererFactory`, add the `TemplateDirectory` value object as a constructor dependency. After that, you can add the templates path to the filesystem loader.

src/Framework/Rendering/TwigTemplateRendererFactory.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rendering;

use Twig_Loader_Filesystem;
use Twig_Environment;

final class TwigTemplateRendererFactory
{
    private $templateDirectory;

    public function __construct(TemplateDirectory $templateDirectory)
    {
        $this->templateDirectory = $templateDirectory;
    }

    public function create(): TwigTemplateRenderer
    {
        $templateDirectory = $this->templateDirectory->toString();
        $loader = new Twig_Loader_Filesystem([$templateDirectory]);
        $twigEnvironment = new Twig_Environment($loader);
        return new TwigTemplateRenderer($twigEnvironment);
    }
}
```

Refresh the front page and you should see the HTML page rendered by your browser.

Reuse template code

You don't want to write every HTML page from scratch. Usually there are a lot of shared elements on pages, a lot of code is written for the page layout and not the individual page itself. If you were to repeat that for every single page, it would be a maintenance nightmare.

As with any other code, we have to make sure that information is not duplicated. One approach that I've seen some people use is to require header and footer files from above and below the page content. This approach can work, but Twig offers a better alternative.

With Twig you can use template inheritance. That means that you can create a `Layout.html.twig` and in there you define blocks for the child template to fill in.

templates/Layout.html.twig

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Social News</title>
</head>

<body>
<h1>Social News</h1>
{% block content %}{% endblock %}
</body>
</html>
```

Now you just need to change the `FrontPage.html.twig` file to extend the layout.

templates/FrontPage.html.twig

```
{% extends 'Layout.html.twig' %}

{% block content %}
    <ol>
        <li>
            <a href="http://google.com">Google</a>
        </li>
        <li>
            <a href="http://bing.com">Bing</a>
        </li>
    </ol>
{% endblock %}
```

The nice thing about this approach is that you can define multiple blocks. You can define additional blocks if you want other information passed along to the parent template. A good use of this feature is adding a block for page-specific JS and CSS files to the layout.

Passing data to the template

Right now the list of links is hardcoded in the front page HTML code. Instead, we should be passing those links along as variables to the template.

Change your front page template to the following.

templates/FrontPage.html.twig

```
{% extends 'Layout.html.twig' %}

{% block content %}
    <ol>
        {% for submission in submissions %}
            <li>
                <a href="{{ submission.url }}">{{ submission.title }}</a>
            </li>
        {% endfor %}
    </ol>
{% endblock %}
```

Now you need to pass the submissions to the template in the controller.

src/FrontPage/Presentation/FrontPageController.php

```
// ...

public function show(): Response
{
    $submissions = [
        ['url' => 'http://google.com', 'title' => 'Google'],
        ['url' => 'http://bing.com', 'title' => 'Bing'],
    ];
    $content = $this->templateRenderer->render('FrontPage.html.twig', [
        'submissions' => $submissions,
    ]);
    return new Response($content);
}

// ...
```

Of course it is not really the controller's job to know what links belong to the front page, it should only pass those variables along to the template. In the following chapter, we will move the logic for fetching the links into its own class. But before we continue, let's talk a little bit about security.

How to prevent XSS attacks

XSS is short for cross-site scripting. XSS vulnerabilities make it possible for an attacker to inject a client-side script into your website code.

Whenever you display data that was entered by a user at some point, you need to make sure that the variables are escaped properly. Twig conveniently escapes our variables for HTML automatically, but that doesn't mean that you don't have to think about XSS vulnerabilities when you write code.

Add the `raw` filter in the twig template to disable the automatic escaping from Twig.

templates/FrontPage.html.twig

```
{# ... #}

<a href="{{ submission.url }}">{{ submission.title|raw }}</a>

{# ... #}
```

And now modify the title a little bit.

src/FrontPage/Presentation/FrontPageController.php

```
// ...

['url' => 'http://google.com', 'title' => 'Google<script>alert(1)</script>'],

// ...
```

Reload your homepage.

Whoops.

The attacker can run any script and do whatever he wants to your site. He could steal the passwords of your users, by sending them to an URL of his choice on your website. If it's hidden well, nobody would find out.

XSS is one of the most common security vulnerabilities on the web and I have seen countless examples in production code. Back in 2007, Symantec reported that 84% of all security vulnerabilities were because of XSS vulnerabilities. Hopefully things have changed for the better in the last decade, with improved developer education and packages like Twig, but it's still a very common oversight by developers.

In an attempt to make their code more secure, many developers and frameworks try to sanitize all `GET` and `POST` variables. They create elaborate rules, to try to prevent any possible attack scenario. As a result, they mangle the user input and what's worse, attackers still find their way around the sanitization rules.

`GET` and `POST` are also not the only way that untrusted input can get into your system. Any interface to another system can also have an opportunity to inject code. Please avoid this approach and use something a little more sensible for your security.

Properly preventing XSS is not hard, you just have to escape all your output for the appropriate format. But that also means that there is not a single way to do it, you need to be aware of the context of where the output will be displayed and escape accordingly.

The escape rules for HTML content are not the same as the ones for a HTML attribute. And those rules are different again for JavaScript and CSS. Twig comes with a few handy filters for this, with the HTML filter applied by default.

```
|e('html')
|e('js')
|e('css')
|e('url')
|e('html_attr')
```

Let's fix our template to make it secure again and let's use the `html_attr` filter for the `href` attribute. The `url` filter is only meant to escape URL parts, so it's not applicable for a whole URL like in our case.

templates/FrontPage.html.twig

```
{# ... #}

<a href="{{ submission.url|e('html_attr') }}">{{ submission.title }}</a>

{# ... #}
```

Refresh the homepage to see the escaping in action. The script won't work anymore and it's rendered as plain text by the browser instead.

> XSS is the most common vulnerability in websites and web applications. Always escape data that is introduced by users (or other systems) when you generate output. You only need to forget it once to create an opening for an attacker.

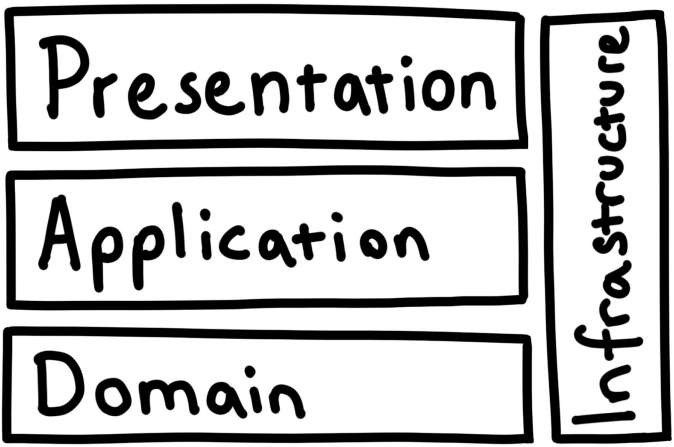
The XSS part was very short, but it's one of the most important parts of this book. If you just glanced over it, please reread it carefully.

If your code is not written in a clean way, then you might create some technical debt or introduce a bug or two. But if you have a security issue, the possible outcomes are much worse. If it gets exploited, a bad vulnerability can be the death warrant of a company or product.

5. Application Layer

Introduction

The name application layer comes from the domain driven design community and it describes the layer that separates your controllers from your business logic. Other common names for this layer include service layer and use case layer.



The application layer represents the possible interactions between the outside world and your application - the queries and commands that can be executed by the frontend. This is why it is sometimes called the use case layer.

The application layer methods and classes can be reused in multiple controllers. You can use the same application layer class from a HTTP controller, a CLI controller and for your JSON API. The layer also has the benefit of making behaviour testing very easy and convenient.

It is common for new developers to have all their logic in their controllers, they grow and grow until the controllers are several thousand lines long. The controller methods end up doing everything from displaying HTML to calling the database and sending emails. This is not object oriented programming, it's just procedural code in a class.

The presentation layer controllers are the glue that connects your application to the outside world. They only receive a request and then return a response. They can contain presentation logic, but business or application logic doesn't belong into the controllers.

To separate the responsibilities, our front page controller needs a new dependency that can return a list of submissions. Our controller will receive a request, fetch the submissions from this dependency and then return the content as a HTML page.

Single Responsibility Principle

The single responsibility principle (SRP), states that a class should only have one responsibility. You can think of the principle as separation of concerns for classes. But sometimes it can be unclear what exactly a single responsibility is, so Robert C. Martin came up with the following description for the principle.

> A class should have only one reason to change.
>
> *Robert C. Martin*

The single responsibility principle also relates to the basic programming principles of low coupling and high cohesion. You want to put all the things that change for the same reason into the same class (high cohesion) and all the things that change for a different reason into separate classes (low coupling).

Some developers are scared of breaking up their code into too many classes. They are worried that a lot of classes will make the system harder to understand. Enterprise Java code is often used as an example of OOP that has been taken too far.

Putting all your code into one single class is clearly not the way to go, that would not be object oriented at all. On the other hand, a million separate classes for a relatively small application sounds just as bad. We have to draw the line somewhere and find a good compromise between the two extremes.

The code will contain the same amount of moving parts, whether it's in one large class or in many small ones. You need to organize your code in a way that makes it easy to understand those moving parts. It will be much easier to navigate through a lot of small and well-named classes, compared to a few large ones with thousands of lines each.

If every class has only a single responsibility, it will be much easier to find a fitting name for that class. And a good class name makes it easier to find a specific piece of code during debugging or development.

Finding a good name

The classes in the application layer are often called services, which is why some developers call it the service layer. But don't take this too literal and just suffix all your application layer classes with `Service`.

A `Service` suffix makes it faster to write code, because you don't have to think about choosing a good class name. But when you come back to the code a few months later and you notice a `SubmissionService`, then you probably won't remember what the class does. A good name describes what an object is and a name like `SubmissionService` does not communicate much useful information.

Now you might be wondering if the same doesn't apply to controllers too, after all they share a `Controller` suffix. The difference is that the `Controller` suffix adds meaning. If one of your coworker spots a class with a controller suffix, then he immediately knows more about the class. On the other hand, the `Service` suffix doesn't add meaning. Service is an overused word in the developer world and it just ends up being a noise word.

We need a good name for our class and it has to be more meaningful than `SubmissionService`. A better pick for the name would be `SubmissionReader`, with a method that returns an array. That is more specific than a service suffix, but there is still room for improvement.

If we take a step back, we can see that we are usually doing one of two things in the application layer. Either we want to request some information or we want to change the state of the application. It's not that different from what HTTP does, it has GET on one side and POST, PUT, PATCH, DELETE on the other side.

The clear distinction between read and write operations was originally formalized by Bertrand Meyer, as the Command-query separation (CQS) principle. But CQS usually refers to methods and not classes.

> Asking a question should not change the answer.
>
> *Bertrand Meyer*

Some developers noticed that the same approach also had benefits when it was applied to other things. Originally it was also called CQS when the principle was used to separate classes and components, but that kept confusing developers. As a response to that confusion, Greg Young came up with the term Command-query responsibility segregation (CQRS) to distinguish the two.

If you read up on the internet on what CQRS is, you might get the impression that it's very complicated. But it's just about separating your read from your write side.

You probably already know where I am going with this. Instead of a generic reader class with many different read operations, we can model the queries as objects instead.

Interface segregation

The interface segregation principle (ISP) stand for the I in SOLID. It states that no client should be forced to depend on methods that it does not use.

This is the principle that I see violated most often from developers who otherwise try to follow the SOLID principles. These violations are either services or repositories that do a lot of different things. The ISP violations in repositories usually happen because there is a one to one relationship between entity and repository, but multiple classes need to modify and read the state of that entity.

We are going to look at entities and repositories in a later chapter when the domain layer is introduced. For now, we only need to read a list of submissions. The interface for a `SubmissionReader` could end up looking like the following example.

This code is not part of the tutorial

```
public function getSubmissions(): array;
public function getSubmissionsFromUser(UserId $userId): array;
public function getSubmission(SubmissionId $submissionId): Submission;
```

If you take the interface segregation principle into consideration, the problem should be obvious. Our front page controller only needs access to `getSubmissions()`.

On the other hand, we could have a controller to view a single submission and another one to view a user profile. Each one of those controllers only needs access to one of the methods, but if they depend on the `SubmissionReader`, then they are forced to depend on all of them. This complicates unit testing and makes it harder to refactor your code down the road.

We could try to split the reader into multiple classes, but that makes it hard to come up with good names. One class would end up as `SubmissionsFromUserReader` with a `getSubmissionsFromUser()` method for example. That approach goes into the right direction, but let's see how the same would look if we model it as a query instead.

This code is not part of the tutorial

```
interface SubmissionsFromUserQuery
{
    public function toIterable(UserId $userId): array;
}
```

The focus is now on the object and not the method. It is easy to follow the ISP when you use query objects - you have a separate interface for each query.

But as with everything, there is a tradeoff. It requires more effort to deduplicate code between query objects and sometimes you might have to depend on multiple objects, when before a single reader would have been sufficent. I believe that those are fair tradeoffs, because you get a cleaner interface for your application layer in return.

The query object

Before we create a query object, we need a simple value object to represent a submission. Create an `Application` directory in your `FrontPage` directory and then create a new `Submission.php` file. This value object will represent a single submission on the front page and it consists of an URL and a title text.

src/FrontPage/Application/Submission.php

```
<?php declare(strict_types=1);

namespace SocialNews\FrontPage\Application;

final class Submission
{
    private $url;
    private $title;

    public function __construct(string $url, string $title)
    {
        $this->url = $url;
        $this->title = $title;
    }

    public function getUrl(): string
    {
        return $this->url;
    }

    public function getTitle(): string
    {
        return $this->title;
    }
}
```

We need to return more than one submission, so we need an iterable collection of submissions. We could create a separate object for this, but in most cases a simple array works just as well. You just have to add a docblock to add additional type information.

In the same directory, create an interface for the query. I recommend a `Query` suffix for the queries, because otherwise you can get a naming conflict with one of your collection value objects.

src/FrontPage/Application/SubmissionsQuery.php

```
<?php declare(strict_types=1);

namespace SocialNews\FrontPage\Application;

interface SubmissionsQuery
{
    /** @return Submission[] */
    public function execute(): array;
}
```

Why do we use an interface instead of a concrete class?

I like to separate the layers as good as possible. The application layer only concerns itself with the use cases of the application and the infrastructure layer is the only one that knows about the database.

Sometimes you might want to reuse infrastructure code between multiple queries and those shared classes would only distract from the essential things in the application layer.

Now we need to make the query a dependency of the controller and use it instead of the hardcoded list of submissions from the last chapter.

src/FrontPage/Presentation/FrontPageController.php

```php
<?php declare(strict_types=1);

namespace SocialNews\FrontPage\Presentation;

use SocialNews\Framework\Rendering\TemplateRenderer;
use SocialNews\FrontPage\Application\SubmissionsQuery;
use Symfony\Component\HttpFoundation\Response;

final class FrontPageController
{
    private $templateRenderer;
    private $submissionsQuery;

    public function __construct(
        TemplateRenderer $templateRenderer,
        SubmissionsQuery $submissionsQuery
    ) {
        $this->templateRenderer = $templateRenderer;
        $this->submissionsQuery = $submissionsQuery;
    }

    public function show(): Response
    {
        $content = $this->templateRenderer->render('FrontPage.html.twig', [
            'submissions' => $this->submissionsQuery->execute(),
        ]);
        return new Response($content);
    }
}
```

We have the interface, but there is still no implementation. Later during the project we will fetch the submissions from a database, but for now we just need a hardcoded list to continue with the development.

An object that simulates the behaviour of a real object is called a mock object. Usually we use those for unit tests to test a class in isolation. We are not doing any unit testing in this tutorial, but we still need to manually test whether our application code works.

In your front page directory, create an `Infrastructure` directory and then create a `MockSubmissionsQuery.php` file in that directory. You could also add subdirectories for the layers in there, but because the component is very small, there is no need for that. Just something to keep in mind if your infrastructure directory grows out of control.

src/FrontPage/Infrastructure/MockSubmissionsQuery.php

```
<?php declare(strict_types=1);

namespace SocialNews\FrontPage\Infrastructure;

use SocialNews\FrontPage\Application\Submission;
use SocialNews\FrontPage\Application\SubmissionsQuery;

final class MockSubmissionsQuery implements SubmissionsQuery
{
    private $submissions;

    public function __construct()
    {
        $this->submissions = [
            new Submission('https://duckduckgo.com', 'DuckDuckGo'),
            new Submission('https://google.com', 'Google'),
            new Submission('https://bing.com', 'Bing'),
        ];
    }

    public function execute(): array
    {
        return $this->submissions;
    }
}
```

The Auryn dependency injector container is smart and it wires everything together by itself if it can figure out what concrete class you want to have injected. But because we are using an interface for `SubmissionsQuery`, we need to help the injector out.

To specify which implementation is going to be injected when we use the interface for the type declaration, add the following lines to your dependencies file.

src/Dependencies.php

```
// ...

use SocialNews\FrontPage\Application\SubmissionsQuery;
use SocialNews\FrontPage\Infrastructure\MockSubmissionsQuery;

// ...

$injector->alias(SubmissionsQuery::class, MockSubmissionsQuery::class);
$injector->share(SubmissionsQuery::class);

// ...
```

Now the `MockSubmissionsQuery` implementation has become the default. You can still override this on a per class basis if you come across a case where you need a different implementation.

What is happening here? The `alias()` method marks `MockSubmissionsQuery` as the default implementation for the `SubmissionsQuery` interface. Auryn can now automatically inject it whenever you depend on that interface. Of course you can always override the alias for a specific class if you want another implementation injected.

The `share()` method prevents Auryn from creating a new instance whenever an object is injected. The same instance of the object is reused for all classes that use this dependency.

Refresh your front page and make sure that the code works before you continue with the next chapter.

6. Infrastructure Layer

We only displayed hardcoded data for the homepage in the previous chapters. But we want to build a community site, where users will be able to submit content themselves and we need to store this content somewhere.

For a large part of my software development career, I always started a project by thinking about the database first. I designed all the tables and relations, before I wrote my first line of code.

The database first approach leads to not so great code from my experience. Everything turns into simple CRUD (create, read, update and delete) and the domain logic becomes an afterthought.

I recommend that you invert that approach and start with the code instead. Think about your use cases and objects first. What do you want your system to do? What state changes do you want to track?

> Invert, always invert.
>
> *Carl Gustav Jacob Jacobi (mathematician)*

In the end, it doesn't really matter where the state is persisted. It could be in a database, but it could also be stored in a text file or somewhere else.

For this tutorial, we will use an Sqlite database for simplicity's sake. For a real project, PostgreSQL is a good default choice. Sqlite can do a decent job for a small website (https://sqlite.org/whentouse.html), but I would still use PostgreSQL for most applications.

This is one of the benefits of a separate infrastructure layer, all the code that talks to the database is in one place. Refactoring is always an option, especially if your application is small.

Accessing the database

How do we talk to the database? You could just use PDO. It is provided by PHP and it can work well, but the code can become a little messy if you try to reuse parts of your SQL. With PDO you will be passing a lot of SQL strings around.

I had much better experiences with a simple database access layer (DBAL). I like the DBAL component from the Doctrine team. You can install it with the command `composer require doctrine/dbal`. With the DBAL, you can use objects to build your queries.

We have to configure the DBAL before we can use it, just as we did with Twig. We are going to use a factory, but before we do so let's start with a new value object to represent the database URL.

Create a new `Dbal` directory in your `src/Framework` directory and create a new `DatabaseUrl.php` file in there.

src/Framework/Dbal/DatabaseUrl.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Dbal;

final class DatabaseUrl
{
    private $url;

    public function __construct(string $url)
    {
        $this->url = $url;
    }

    public function toString(): string
    {
        return $this->url;
    }
}
```

The database URL contains all the information that the driver manager from the DBAL package needs to create a new connection. The URL format can be used to connect to many different databases, it's flexible and suits our case well.

Create a new `src/Framework/Dbal/ConnectionFactory.php` file. We move the construction of the `Connection` instance into the factory, because that simplifies the code in the dependencies file.

src/Framework/Dbal/ConnectionFactory.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Dbal;

use Doctrine\DBAL\Configuration;
use Doctrine\DBAL\Connection;
use Doctrine\DBAL\DriverManager;

final class ConnectionFactory
{
    private $databaseUrl;

    public function __construct(DatabaseUrl $databaseUrl)
    {
        $this->databaseUrl = $databaseUrl;
    }

    public function create(): Connection
    {
        return DriverManager::getConnection(
            ['url' => $this->databaseUrl->toString()],
            new Configuration()
        );
    }
}
```

Create a `storage` folder in your project root folder and place an empty `.gitignore` file in there (if you are using git). Git only keeps track of files and not folders, so this is necessary so that the empty folder is committed to the git repository. Add `db.sqlite3` to the `.gitignore` file, you don't want to commit every change in your database to your version control.

storage/.gitignore

```
db.sqlite3
```

After you have sorted out your version control, add the following code to your dependencies file.

src/Dependencies.php

```
// ...

use Doctrine\DBAL\Connection;
use SocialNews\Framework\Dbal\ConnectionFactory;
use SocialNews\Framework\Dbal\DatabaseUrl;

// ...

$injector->define(
    DatabaseUrl::class,
    [':url' => 'sqlite:///' . ROOT_DIR . '/storage/db.sqlite3']
);

$injector->delegate(Connection::class, function () use ($injector): Connection {
    $factory = $injector->make(ConnectionFactory::class);
    return $factory->create();
});

// ...
```

You don't have to worry about connection parameters with our very simple SQLite setup, but if you use another database, this becomes an issue. Whatever you do, never put your password into source control. You won't believe how many real passwords you can find on Github...

Even if you don't plan to make your repository public, never put any sensitive information into source control. Once it's in the history, it won't disappear. Even if you remove it in a second commit. If it happened anyways, change the passwords and other sensitive information immediately.

You also don't want your passwords in your code, they could be exposed in a stack trace if an error happens. Of course you should never show a stack trace to your users, but sometimes things go wrong. By keeping the sensitive data completely separated, you avoid that danger.

The recommended approach is to move all that information into environment variables. To learn more about this, visit `http://12factor.net`. A slightly less ideal approach, but still much better than other options, would be to have a separate config file that is not checked into source control.

Keeping the information separate not only helps with security, but also with setting up multiple environments. Each server can have its own configuration, so setting up a separate development and production environment is easy.

Singletons

You don't want to create a new connection to the database for every single query that you have to run, that would put a lot of stress on your server for no good reason. Set up a single connection instead and then reuse it for all your queries.

One way to reuse a database connection is by using the singleton pattern. A singleton is a class with a private constructor. The singleton can only be instantiated through a special method that will make sure that you always get the same instance, `Database::getInstance()` in the example below.

Don't do this

```
final class Database
{
    private static $instance;

    public static function getInstance()
    {
        if (self::$instance === null) {
            self::$instance = new Database();
        }
        return self::$instance;
    }

    private function __construct() {}
}
```

If you don't know about dependency injection, this might look like a reasonable way to handle database access. You don't want to set up a new database connection for every database call and with a singleton you can reuse the same one.

But the singleton pattern comes with a couple of problems, one of them is global state. You create a new instance of the singleton from inside your own class, but something else can manipulate that same instance without your knowledge.

Another problem with singletons is that at some point you might want more than one instance. Right now you might think that you will never need more than one database, but requirements change.

There have been attempts at solving those issues with different variations of the singleton pattern, but it's still the wrong tool for the job. If you want only one instance of a class, then just instantiate it once and pass that instance around your code. When you know how to use dependency injection, then it doesn't make sense to the singleton pattern.

To reuse your database connection, add the following line to your dependencies file.

src/Dependencies.php

```
// ...

$injector->share(Connection::class);

// ...
```

The injector will now reuse the connection, instead of creating a new one from scratch every time that a connection is injected.

Designing the table

Before we can read data from the database, we need to create a table. Because we already started with the object, let's use that as a starting point. We need at least two fields, one for the title and one for the URL of the submission.

It would also be useful if we had a way to display the submissions in order, starting from the newest one. To be able to do this, we need to store the creation date, either as a timestamp or datetime field. And we need an identifier (ID), so that we can view a submission on a separate page or use the ID as a reference point for comments and votes.

```
submissions
-----------
id
title
url
creation_date
```

We are going to use an UUID (universally unique identifier) for our ID, also know as GUID (globally unique identifier). The name GUID is usually only used by developers working with Microsoft technologies, while everyone else uses UUID. There are different versions for UUIDs and not all of them are random, the one we are talking about in this book is version 4.

An UUID consists of 128 random bits. When represented as a string, an UUID could look like `0794d9c4-d603-4a86-8f5a-0e7da8df91de` for example. Instead of starting with 1 and then incrementing that value, you create one at random when you create a new object or row. The probability for a collision when generating a new UUID is not zero, but it's close enough that you don't have to worry about it. If you wanted a one in a billion chance for a collision, you would have to generate 103 trillion ids.

If you are used to using incremental integers as IDs, using UUIDs might need some time to get used to. They are not as readable as an integer, but they come with many benefits.

One of the benefits is that you don't have to rely on the database anymore to get a new ID. Instead of returning the newly created ID after inserting a record, returning `void` is fine. Just pass in the already created ID as a parameter instead. It doesn't matter where the UUID was created, it could even be done in the frontend.

While the above might not sound like such a big benefit at first, it allows us to create immutable objects. With integer IDs you don't really have that option, you need to first create an object without an ID and then update the object after you get the ID back from the database.

A new object with an autoincrementing integer ID is not valid until it received it's ID. Any code that wants to read the ID will fail if the object hasn't been persisted and updated. With the UUID, you just generate a new one in the constructor and you don't have this problem, the object is always in a valid state.

UUIDs have some disadvantages as well. They need more space than an integer, you can't order by the ID and they don't look very nice if you use them as a part of your URL. As with so many things, it's a tradeoff.

Having the ID available before persisting to the database makes it easier to write simple and maintainable code, which is why I recommend using UUIDs for the default choice. If the disadvantages become a problem, you can always add a datetime or auto-incrementing field in addition to the UUID.

Migration

Now that we know what the table should look like, we have to find a way to create it in our Sqlite database. You could connect to the database directly and execute an SQL statement, but then you have a disconnect between your database and code.

When you keep your database migrations as part of your code, your database is versioned, just like your code. You can go back to any commit and reproduce the database, which is very useful for debugging purposes.

There are many database migration packages available and almost every framework comes with it's own. For this tutorial, we are going to keep the migrations very simple. We will just use DBAL and a small script. But for a real application I recommend that you use an existing and well-tested package instead, like `doctrine/migrations` for example.

Create a new `migrations` directory in your project root. In there you can create your first migration file. We need a way to version the migration files, to make sure that they can be executed in the correct order. We can do that by using date and time as part of the name, starting with the year so that it can be sorted. So for `2017-04-15 12:05` you append `201704151205`. Your first migration file will be `Migration201704151205.php` (replace the datetime with the current one).

migrations/Migration201704151205.php

```php
<?php declare(strict_types=1);

namespace Migrations;

use Doctrine\DBAL\Connection;

final class Migration201704151205
{
    private $connection;

    public function __construct(Connection $connection)
    {
        $this->connection = $connection;
    }

    public function migrate(): void
    {
        // migrate...
    }
}
```

This migration class is very simple. We only have to give it a DBAL `Connection` object and then we can call `migrate()`.

Let's add the code that actually creates the table. It will create the table schema SQL for your database platform and then execute the query to set up the table.

migrations/Migration201704151205.php

```
// ...

use Doctrine\DBAL\Schema\Schema;
use Doctrine\DBAL\Types\Type;

// ...

public function migrate(): void
{
    $schema = new Schema();
    $this->createSubmissionsTable($schema);

    $queries = $schema->toSql($this->connection->getDatabasePlatform());
    foreach ($queries as $query) {
        $this->connection->executeQuery($query);
    }
}

private function createSubmissionsTable(Schema $schema): void
{
    $table = $schema->createTable('submissions');
    $table->addColumn('id', Type::GUID);
    $table->addColumn('title', Type::STRING);
    $table->addColumn('url', Type::STRING);
    $table->addColumn('creation_date', Type::DATETIME);
}

// ...
```

But how do we start the migration script? It wouldn't make any sense to make the migration scripts available from the web user interface, after all a migration script is only used once.

What we need is a simple command line interface script to execute the migrations. By convention, CLI scripts are usually placed into a `bin` directory. Create one in your project root folder and in there, create a `Migrate.php` script.

bin/Migrate.php

```
<?php declare(strict_types=1);

use Migrations\Migration201704151205;

define('ROOT_DIR', dirname(__DIR__));

require ROOT_DIR . '/vendor/autoload.php';

$injector = include(ROOT_DIR . '/src/Dependencies.php');

$connection = $injector->make('Doctrine\DBAL\Connection');

$migration = new Migration201704151205($connection);
$migration->migrate();

echo 'Finished running migrations' . PHP_EOL;
```

The composer autoload doesn't know where to find the `Migration` namespace. We have to register that in your `composer.json`.

Update the autoload part of your composer file to the following and then run `composer dumpautoload`.

composer.json

```
...

"psr-4": {
    "SocialNews\\": "src/",
    "Migrations\\": "migrations/"
}

...
```

To run the migration script, navigate to your project root folder in your terminal and then execute the command `php bin/Migrate.php`. Check if `db.sqlite3` was created in your `storage` folder.

If you encounter an error when you run the command, make sure that you have the PHP SQLite extension installed and enabled.

Using DBAL

Now that everything is set up, it's time to replace the mock submissions query implementation with a real one that uses the database. The front page will be empty after we switch to the database, because we haven't added any data to the table yet.

Delete the `MockSubmissionsQuery.php` file and create a new implementation that uses the database.

src/FrontPage/Infrastructure/DbalSubmissionsQuery.php

```
<?php declare(strict_types=1);

namespace SocialNews\FrontPage\Infrastructure;

use Doctrine\DBAL\Connection;

use SocialNews\FrontPage\Application\Submission;
use SocialNews\FrontPage\Application\SubmissionsQuery;

final class DbalSubmissionsQuery implements SubmissionsQuery
{
    private $connection;

    public function __construct(Connection $connection)
    {
        $this->connection = $connection;
    }

// ...
```

Continued from the previous page

```
// ...

    public function execute(): array
    {
        $qb = $this->connection->createQueryBuilder();

        $qb->addSelect('title');
        $qb->addSelect('url');
        $qb->from('submissions');
        $qb->orderBy('creation_date', 'DESC');

        $stmt = $qb->execute();
        $rows = $stmt->fetchAll();

        $submissions = [];
        foreach ($rows as $row) {
            $submissions[] = new Submission($row['url'], $row['title']);
        }
        return $submissions;
    }
}
```

Change the alias in your dependencies file from `MockSubmissionsQuery` to `DbalSubmissionsQuery` and then visit the front page to check that there are no errors.

src/Dependencies.php

```
// ...

use SocialNews\FrontPage\Infrastructure\DbalSubmissionsQuery;

// ...

$injector->alias(SubmissionsQuery::class, DbalSubmissionsQuery::class);

// ...
```

Because there is nothing in the database, there list of submissions on the front page will be empty.

7. Cross-site Request Forgery

Introduction

In this chapter, we will create a form that makes it possible to add new submissions to the site. Create a new `Submission.html.twig` template in your templates directory.

templates/Submission.html.twig

```
{% extends 'Layout.html.twig' %}

{% block content %}
    <form method="post" action="">
        <label for="submission-title">Title</label>
        <input type="text" name="title" id="submission-title"><br>
        <label for="submission-url">URL</label>
        <input type="url" name="url" id="submission-url"><br>
        <input type="submit" value="Submit">
    </form>
{% endblock %}
```

Now we need to render the template from our `SubmissionController`. Refactor your controller until the code looks like the following.

src/Submission/Presentation/SubmissionController.php

```
<?php declare(strict_types=1);

namespace SocialNews\Submission\Presentation;

use Symfony\Component\HttpFoundation\Response;
use SocialNews\Framework\Rendering\TemplateRenderer;

// ...
```

Continued from the previous page

```
// ...

final class SubmissionController
{
    private $templateRenderer;

    public function __construct(TemplateRenderer $templateRenderer)
    {
        $this->templateRenderer = $templateRenderer;
    }

    public function show(): Response
    {
        $content = $this->templateRenderer->render('Submission.html.twig');
        return new Response($content);
    }
}
```

Visit `localhost:8000/submit` to check whether the form is displayed.

When you submit the form, you will get a "method not allowed" in response. This is because we did not yet create a `POST` route for `/submit`. Add one to your routes file.

src/Routes.php

```
// ...

[
    'POST',
    '/submit',
    'SocialNews\Submission\Presentation\SubmissionController#submit'
],

// ...
```

Then add a matching controller method (don't overwrite the existing `show()` method).

src/Submission/Presentation/SubmissionController.php

```
// ...

use Symfony\Component\HttpFoundation\Request;

// ...

public function submit(Request $request): Response
{
    $content = $request->get('title') . ' - ' . $request->get('url');
    return new Response($content);
}

// ...
```

We will refactor this method later. Displaying the title and URL will make it easier to demonstrate another security vulnerability.

CSRF explained

When you create a state-changing request, then you have to consider cross-site request forgery (CSRF) vulnerabilities. A CSRF attack tricks you into doing an action on a website like adding a new submission or changing your password. CSRF attacks are write-only and even a successful attack can't send information back to the attacker.

At the moment there are no users and the submission form is not login protected, but it is still important to protect the page from CSRF. A spammer could trick unsuspecting users into submitting his URLs to your website for example.

We will use the HTML form from earlier to demonstrate an attack. Create a simple `csrf.html` file outside of your project directory. This file will represent the website that is under control of the attacker.

csrf.html

```
<html><body>
  <form name="csrf" action="http://localhost:8000/submit" method="POST">
    <input type="hidden" name="url" value="http://patricklouys.com">
    <input type="hidden" name="title" value="Submitted by CSRF">
  </form>
  <script type="text/javascript">document.csrf.submit();</script>
</body></html>
```

If you open this file in your browser, it will automatically submit the forged request. You, as the user, don't have any control over what's submitted in your name. Use the developer console of your browser to inspect the request that was sent.

The attack from the example above is visible to the user because the page is being redirected. But it is easy to hide that from him, you just have to change the form target to a frame instead.

csrf.html

```
<html><body>
  <iframe style="display:none" name="frame"></iframe>
  <form name="csrf" action="http://localhost:8000/submit" method="POST"
  target="frame">
    <input type="hidden" name="url" value="http://patricklouys.com">
    <input type="hidden" name="title" value="Submitted by CSRF">
  </form>
  <script type="text/javascript">document.csrf.submit();</script>
</body></html>
```

Now the user can't see that a request happened. If you check the developer console, you will see that the request was still submitted.

Both `GET` and `POST` requests are vulnerable and can be forged. Some developers use `GET` requests to delete data, but they often forget protect those requests from CSRF attacks. You need to be very careful to avoid mistakes like those.

Change all your state-changing actions into `POST` requests and then enforce CSRF protection for all your`POST` requests. Only use `GET` requests for read operations.

The standard way to prevent CSRF attacks is by using a secret token. The token is submitted as part of the request and then validated in the controller. Because a CSRF attack is write-only, an attacker can't figure out the token value.

A fixed token per user gives you some protection from CSRF attacks, but if an attacker can get a hold of the secret token then he can forge requests for that user. This could happen through an additional XSS or man in the middle (MITM) vulnerability for example.

> If you have an XSS vulnerability on your website, an attacker can use it to figure out the secret CSRF token of a user.

An improvement from that approach is a token that expires after some time. Even if a token gets compromised, the window for an attack is much smaller.

You can also generate a separate token for each form. If an attacker is able to compromise one of your forms, he can't reuse the token to forge other requests.

A completely random token for every request and expiring the old ones immediately is the most secure approach. It is very hard for an attacker to circumvent this and if you are doing anything with sensitive data, this is the only approach that you should consider. But this approach has usability drawbacks, because if a user uses multiple tabs or the back button, the token is expired and the validation fails.

The following is my recommendation for CSRF protection measures. Use a per-session token for non-sensitive data and single-use tokens for sensitive data. By sensitive I mean admin functionality, an e-commerce checkout, e-banking, etc.

Generating a token

We need a random value for the token, but not all random number generators are made equally. PHP comes with multiple ways to generate them, like `rand()`, `mt_rand()`, `lcg_value()`, `openssl_random_pseudo_bytes()` and a few others.

All of the above random number generators have possible attack vectors that would make it possible for an attacker to make the output somewhat predictable. That's not ideal for a security relevant value like a CSRF token.

This is why `random_bytes()` and `random_int()` were added to PHP 7. They create cryptographically secure random values.

You aren't gonna need it

For our submission form, we need to generate a per-session token and a way to access the value from a template. We also need to validate the token from the controller.

We don't need a single use token for the submission form, so we are not going to implement it. This is the "You Aren't Gonna Need It" (YAGNI) principle in action.

> Always implement things when you actually need them, never when you just foresee that you need them.
>
> *Ron Jeffries*

It's easy to spend a few hours working on a feature that you could need at some point in the future, but there is no reason to do it before you actually need it. It will take roughly the same time to do it now compared to later and you don't have the proper requirements yet.

You are just guessing if you try to solve your problems in advance. And if your guess is wrong, then you will have to add additional time for the refactoring. And if you were wrong and you never need the feature, then you wasted time for nothing. Working on things that you might need in the future also delays your current work.

YAGNI only applies to writing code for an anticipated feature, it doesn't mean that you should not spend time on making your software easier to modify in the future. After all, YAGNI only works if your code is easy to refactor.

Generating a random token

We are going to start with the value object that will represent a token. Create the file in a new `src/Framework/Csrf` directory.

src/Framework/Csrf/Token.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Csrf;

final class Token
{
    private $token;

    public function __construct(string $token)
    {
        $this->token = $token;
    }

    public function toString(): string
    {
        return $this->token;
    }
}
```

To generate a new token, we are going to use a named constructor. We convert the bytes to hexadecimal to make the token value alphanumeric.

src/Framework/Csrf/Token.php

```
// ...

public static function generate(): Token
{
    $token = bin2hex(random_bytes(256));
    return new Token($token);
}

// ...
```

It's not enough to just create new tokens. We need to also store them, so that we can validate the tokens later.

src/Framework/Csrf/TokenStorage.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Csrf;

interface TokenStorage
{
    public function store(string $key, Token $token): void;

    public function retrieve(string $key): ?Token;
}
```

We use an interface for the `TokenStorage` because there can be multiple implementations. To keep it simple, we are going to store the tokens in the session. You can always switch to a cache later if performance becomes an issue. We are going to use the `SessionInterface` from Symfony to access the session.

src/Framework/Csrf/SymfonySessionTokenStorage.php

```php
<?php declare(strict_types=1);

namespace SocialNews\Framework\Csrf;

use Symfony\Component\HttpFoundation\Session\SessionInterface;

final class SymfonySessionTokenStorage implements TokenStorage
{
    private $session;

    public function __construct(SessionInterface $session)
    {
        $this->session = $session;
    }

    public function store(string $key, Token $token): void
    {
        $this->session->set($key, $token->toString());
    }

    public function retrieve(string $key): ?Token
    {
        $tokenValue = $this->session->get($key);

        if ($tokenValue === null) {
            return null;
        }

        return new Token($tokenValue);
    }
}
```

Now we are able to generate and store tokens, but there is still an important piece missing. To make it more convenient, we are going to create a class that allows us to retrieve a token if one exists and generate a fresh token if none exists.

src/Framework/Csrf/StoredTokenReader.php

```php
<?php declare(strict_types=1);

namespace SocialNews\Framework\Csrf;

final class StoredTokenReader
{
    private $tokenStorage;

    public function __construct(TokenStorage $tokenStorage) {
        $this->tokenStorage = $tokenStorage;
    }

    public function read(string $key): Token
    {
        $token = $this->tokenStorage->retrieve($key);

        if ($token !== null) {
            return $token;
        }

        $token = Token::generate();
        $this->tokenStorage->store($key, $token);

        return $token;
    }
}
```

Most of the time we are going to need the tokens for a Twig template. Instead of passing one into the template from a controller, we can extend Twig and add a function to create tokens.

We can register a new Twig function in our `TwigTemplateRendererFactory`. You might want to move the functions somewhere else if you end up with a lot of Twig functions, but we'll keep it simple for now.

Add `StoredTokenReader` as a dependency to the `TwigTemplateRendererFactory` and then register the new method in `Twig_Environment`. We don't want to manually convert the `Token` value object to a string whenever we need a token in Twig, we can do that conversion inside the function and return a string.

src/Framework/Rendering/TwigTemplateRendererFactory.php

```php
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rendering;

use SocialNews\Framework\Csrf\StoredTokenReader;
use Twig_Loader_Filesystem;
use Twig_Environment;
use Twig_Function;

final class TwigTemplateRendererFactory
{
    private $storedTokenReader;
    private $templateDirectory;

    public function __construct(
        TemplateDirectory $templateDirectory,
        StoredTokenReader $storedTokenReader
    ) {
        $this->templateDirectory = $templateDirectory;
        $this->storedTokenReader = $storedTokenReader;
    }

    public function create(): TwigTemplateRenderer
    {
        $loader = new Twig_Loader_Filesystem([
            $this->templateDirectory->toString(),
        ]);
        $twigEnvironment = new Twig_Environment($loader);

        $twigEnvironment->addFunction(
            new Twig_Function('get_token', function (string $key): string {
                $token = $this->storedTokenReader->read($key);
                return $token->toString();
            })
        );

        return new TwigTemplateRenderer($twigEnvironment);
    }
}
```

We use snake-case (`get_token`) for the twig function name because that's the Twig convention.

The `TwigTemplateRendererFactory` has a new dependency. Because it is created by the injector, the dependencies are resolved automatically. You don't have to change any code in the delegate function that creates the factory, bu there are a few aliases missing in the dependencies file.

src/Dependencies.php

```
// ...

use SocialNews\Framework\Csrf\TokenStorage;
use SocialNews\Framework\Csrf\SymfonySessionTokenStorage;
use Symfony\Component\HttpFoundation\Session\SessionInterface;
use Symfony\Component\HttpFoundation\Session\Session;

// ...

$injector->alias(TokenStorage::class, SymfonySessionTokenStorage::class);

$injector->alias(SessionInterface::class, Session::class);

// ...
```

We added a lot of new code during this chapter. Check your site to see if everything still works.

Add the following code inside of the form tag in to your `Submission.html.twig` template. The `submission` string keeps our tokens separate from other forms. If one form is compromised, it won't affect the others.

templates/Submission.html.twig

```
{# ... #}

<input type="text" name="token" value="{{ get_token('submission') }}"><br>

{# ... #}
```

Visit `http://127.0.0.1:8000/submit` in your browser and refresh the page check if the token stays the same. Visit the page in incognito mode and you should see a different token.

Token Validation

Only submitting the token does not prevent CSRF attacks, we also have to validate it. To do that, we have to compare the submitted token with the one in our `TokenStorage`. We could just read out both values and compare them ourselves, but that would not be very object oriented.

`if ($token->toString() === $storedToken->toString()) {}` violates the basic "tell, don't ask" principle of OOP. Instead of asking the the tokens for their value and then comparing them ourselves, we should move that logic into the `Token` class.

src/Framework/Csrf/Token.php

```
// ...

public function equals(Token $token): bool
{
    return ($this->token === $token->toString());
}

// ...
```

In this case the comparison is very simple. It might give off the impression that there is not much to gain by moving the logic into the value object. But comparison logic can be much more complicated than in this example.

Even if a comparison is very simple, it should be part of the object. The `equals` method makes sure that we are not repeating the same logic in multiple places.

Create a new file for the `StoredTokenValidator` class. The logic for this class is very simple - we get the appropriate token from the storage and then check to see if the provided token matches it.

src/Framework/Csrf/StoredTokenValidator.php

```php
<?php declare(strict_types=1);

namespace SocialNews\Framework\Csrf;

final class StoredTokenValidator
{
    private $tokenStorage;

    public function __construct(TokenStorage $tokenStorage)
    {
        $this->tokenStorage = $tokenStorage;
    }

    public function validate(string $key, Token $token): bool
    {
        $storedToken = $this->tokenStorage->retrieve($key);
        if ($storedToken === null) {
            return false;
        }
        return $token->equals($storedToken);
    }
}
```

Add it as a new constructor dependency to your `SubmissionController`.

src/Submission/Presentation/SubmissionController.php

```php
// ...

use SocialNews\Framework\Csrf\StoredTokenValidator;
use Symfony\Component\HttpFoundation\Response;
use SocialNews\Framework\Rendering\TemplateRenderer;

final class SubmissionController
{
    private $templateRenderer;
    private $storedTokenValidator;

    public function __construct(
        TemplateRenderer $templateRenderer,
        StoredTokenValidator $storedTokenValidator
    ) {
        $this->templateRenderer = $templateRenderer;
        $this->storedTokenValidator = $storedTokenValidator;
    }

// ...
```

Post/Redirect/Get

Post/Redirect/Get (PRG) is a web development pattern that prevents a user from submitting duplicate `POST` requests by accident. The pattern makes it possible to refresh the page without triggering another form submission. Use PRG whenever you handle a form submission.

PRG has a lot of benefits, but it makes it harder to pass messages back to the user. After the redirect, you can't access the form variables anymore. The best way to solve this problem is by using flash messages, they are stored in the session until they have been displayed to the user.

Symfonys HTTP foundation already provides us with flash storage functionality as part of their session object.

This code is not part of the tutorial

```
// set flash messages
$session->getFlashBag()->add('errors', 'Invalid token');

// retrieve messages
$flashMessages = $session->getFlashBag()->get('errors');
```

Add `Symfony\Component\HttpFoundation\Session\Session` as another dependency to your submission controller. We can't use `$flashbag->getSession()` or depend on the `SessionInterface` because the interface provided by Symfony doesn't include the `getFlashBag()` method.

We are validating the CSRF token from the controller because it's the responsibility of the controller to handle form input. Our application layer should not know anything about forms. To the application layer it does not matter where the data comes from, it can originate from a form or something else, like a command line interface for example.

If we would mix all the validation together in the application or the presentation layer, it would be harder to reuse the code. It's always a good idea to think about where a certain puzzle piece fits into the larger picture before you start to code.

src/Submission/Presentation/SubmissionController.php

```
<?php declare(strict_types=1);

namespace SocialNews\Submission\Presentation;

use SocialNews\Framework\Csrf\StoredTokenValidator;
use SocialNews\Framework\Csrf\Token;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use SocialNews\Framework\Rendering\TemplateRenderer;
use Symfony\Component\HttpFoundation\Session\Session;

final class SubmissionController
{
    private $templateRenderer;
    private $storedTokenValidator;
    private $session;

    public function __construct(
        TemplateRenderer $templateRenderer,
        StoredTokenValidator $storedTokenValidator,
        Session $session
    ) {
        $this->templateRenderer = $templateRenderer;
        $this->storedTokenValidator = $storedTokenValidator;
        $this->session = $session;
    }

    public function show(): Response
    {
        $content = $this->templateRenderer->render('Submission.html.twig');
        return new Response($content);
    }

// ...
```

Continued from the previous page

```
// ...

    public function submit(Request $request): Response
    {
        $response = new RedirectResponse('/submit');

        if (!$this->storedTokenValidator->validate(
            'submission',
            new Token((string)$request->get('token'))
        )) {
            $this->session->getFlashBag()->add('errors', 'Invalid token');
            return $response;
        }

        // save the submission...

        $this->session->getFlashBag()->add(
            'success',
            'Your URL was submitted successfully'
        );
        return $response;
    }
}
```

On larger projects, I prefer to create my own `FlashMessenger` class. I don't want to be forced to depend on the whole `Session` class when I just need the flash messages, but for this tutorial the Symfony session will do the job just fine.

We also have to display the messages in the templates. A simple, reusable template that can be included from other templates will help us with that.

But before we create the template for the flash messages, we need a way to access the flash messages from the templates. Add the session as a dependency to your `TwigTemplateRendererFactory` class.

src/Framework/Rendering/TwigTemplateRendererFactory.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rendering;

use SocialNews\Framework\Csrf\StoredTokenReader;
use Symfony\Component\HttpFoundation\Session\Session;
use Twig_Loader_Filesystem;
use Twig_Environment;
use Twig_Function;

final class TwigTemplateRendererFactory
{
    private $storedTokenReader;
    private $templateDirectory;
    private $session;

    public function __construct(
        TemplateDirectory $templateDirectory,
        StoredTokenReader $storedTokenReader,
        Session $session
    ) {
        $this->templateDirectory = $templateDirectory;
        $this->storedTokenReader = $storedTokenReader;
        $this->session = $session;
    }

// ...
```

After doing that, add the following function to the Twig environment in the `create()` method.

src/Framework/Rendering/TwigTemplateRendererFactory.php

```
// ...

use Symfony\Component\HttpFoundation\Session\Flash\FlashBagInterface;

// ...

$twigEnvironment->addFunction(
    new Twig_Function('get_flash_bag', function (): FlashBagInterface {
        return $this->session->getFlashBag();
    })
);

// ...
```

Then create a `FlashMessages.html.twig` file in your template directory.

templates/FlashMessages.html.twig

```
{% for message in get_flash_bag().get('errors') %}
    <span style="color:red">{{ message }}</span><br>
{% endfor %}

{% for message in get_flash_bag().get('success') %}
    <span style="color:green">{{ message }}</span><br>
{% endfor %}
```

Add `{% include 'FlashMessages.html.twig' %}` to your `Submission.html.twig` file, just after the `{% block content %}` tag.

templates/Submission.html.twig

```
{% extends 'Layout.html.twig' %}

{% block content %}
    {% include 'FlashMessages.html.twig' %}

{# ... #}
```

Now try it out and check if the CSRF protection works. Submit a correct form and one where you manipulated the CSRF token. You should see both the error and the success message.

If everything works, set the token field from `type="text"` to `type="hidden"`. Don't show a technical detail like a CSRF token to the user.

templates/Submission.html.twig

```
<input type="hidden" name="token" value="{{ get_token('submission') }}">
```

8. SQL injection

Introduction

There are many different approaches for saving data to the database. The two most common object-relational mappers (ORM) patterns are the data mapper pattern (Doctrine is an example) and active record (Eloquent is an example). We are not going to use an ORM package in this tutorial, we are going to do the mapping ourselves instead.

The active record pattern is simple and easy to learn, which is why it's used a lot in some parts of the PHP community. But the pattern brings a few problems with it. It couples your domain objects to your database tables and violates many of the OOP principles that are outlined in this book.

I have used active record in the past and I can't recommend it. It might save you a little time in the beginning, compared to alternative approaches. But you have to pay that time back with interest when your project matures. The technical debt quickly accumulates and becomes a problem. Because your objects are coupled to the database, it's very hard to refactor the code or change a database table.

A better alternative is the data mapper pattern. This pattern gives you much more flexibility compared to active record. Doctrine is an ORM that uses the data mapper pattern and it works well most of the time, but you can run into problems where you need a lot of Doctrine-specific knowledge.

After using both active record and data mapper based ORMs for a few years, I now prefer to not use an ORM package at all. I use the database abstraction layer (DBAL) from Doctrine directly and do the mapping manually. I have to write a few more lines of code, but the code is easier to follow and there is no magic happening behind the curtains.

If you decide to use an ORM in a project and you want to write maintainable code, use one that is based on the data mapper pattern.

The domain layer

The domain layer is the final piece of our layered architecture. It's the last layer to be introduced, but it's the most important one. The code for the business logic of your application resides in this layer.

The business logic for our social news website is very simple at the moment. We just want to be able to add a new submission to the site.

It is easy to imagine how the business logic could expand. A submission from a new user could trigger a manual review for example, which would be a perfect example of code that belongs into this layer.

There are a few common building blocks that are usually used in the domain layer. One of them is the entity, which is an object that is defined by a thread of continuity. Even if all its attributes change, it's still the same object. We have to assign a unique ID to each instance of an entity to guarantee this thread of continuity.

We are going to use an UUID to represent the identity of a submission, but we don't have a way to create one at the moment. It is not a good idea to write the code that creates an UUID yourself. Collisions become much more likely if there is a bug in the implementation that reduces the randomness.

Install the `ramsey/uuid` package and then create a new `Domain` directory in your submission context. In there, create a `Submission` class for the new entity.

src/Submission/Domain/Submission.php

```
<?php declare(strict_types=1);

namespace SocialNews\Submission\Domain;

use DateTimeImmutable;
use Ramsey\Uuid\UuidInterface;

final class Submission
{
    private $id;
    private $url;
    private $title;
    private $creationDate;

    public function __construct(
        UuidInterface $id,
        string $url,
        string $title,
        DateTimeImmutable $creationDate
    ) {
        $this->id = $id;
        $this->url = $url;
        $this->title = $title;
        $this->creationDate = $creationDate;
    }

    public function getId(): UuidInterface
    {
        return $this->id;
    }

    public function getUrl(): string
    {
        return $this->url;
    }

    public function getTitle(): string
    {
        return $this->title;
    }

    public function getCreationDate(): DateTimeImmutable
    {
        return $this->creationDate;
    }
}
```

Named constructors are very useful when you create a domain object as part of a business logic process. You can use the language of the business and be more descriptive than with a normal constructor.

src/Submission/Domain/Submission.php

```
// ...

use Ramsey\Uuid\Uuid;

// ...

public static function submit(string $url, string $title): Submission
{
    return new Submission(
        Uuid::uuid4(),
        $url,
        $title,
        new DateTimeImmutable()
    );
}

// ...
```

Don't forget to add the use statement for the `Uuid` class in addition to the one for the `UuidInterface`. I have no idea why the author of the package thought that it was a good idea to use an interface for a value object, but it's the de facto standard UUID package and it works well enough for our purpose.

After adding the named constructor, set the `__construct()` method to private. This communicates that only the named constructor should be used.

We are going to use the repository pattern to persist the submission entity. A repository is a collection-like interface that you can use to get, add and remove entities from your storage medium.

src/Submission/Domain/SubmissionRepository.php

```php
<?php declare(strict_types=1);

namespace SocialNews\Submission\Domain;

interface SubmissionRepository
{
    public function add(Submission $submission): void;
}
```

Before we continue with the implementation for the `SubmissionRepository`, let's call the domain code from the application layer.

The application layer

What we need is a class that can create a new instance of the `Submission` entity and then persist it with the help of our new repository.

The class could be called `SubmissionService`, but that's not a very descriptive name. We should try to find a name that describes the responsibility of the class better.

> There are only two hard things in Computer Science: cache invalidation and naming things.
>
> *Phil Karlton*

Another candidate for the name is `SubmissionWriter`. While that's a more descriptive name, it's still not a very specific one.

We want to describe a specific use case where the user submits a link. We could call the class `LinkSubmitter`, but `$linkSubmitter->submit($url, $title)` sounds very repetitive.

I struggled with naming application services for a long time. It's hard to come up with good names for them. Often parts of the class name are repeated in the method name, just as in the last example.

There is a more object oriented approach that can be used to design application services. In all the previous examples, our use case was only represented as a method. What if we turn it into an object instead? That leads us to commands and command handlers.

When I first learned about command query responsibility separation (CQRS), I assumed that it is a very complicated subject. The examples were handling asyncronous commands and had to deal with eventual consistency (changes only happen after a delay). But, unknown to me at the time, commands and command handlers can be very simple.

A command is represented by an immutable value object. It usually consists of only a constructor and a few accessor methods, but they can also be serializable and have their own ID in asyncronous systems.

The name of a command is always in the imperative. It describes the action that the user (or system) wants to execute.

Create an `Application` directory for your submission context and then create your first command in there.

src/Submission/Application/SubmitLink.php

```
<?php declare(strict_types=1);

namespace SocialNews\Submission\Application;

final class SubmitLink
{
    private $url;
    private $title;

// ...
```

Continued from the previous page

```
// ...

    public function __construct(string $url, string $title)
    {
        $this->url = $url;
        $this->title = $title;
    }

    public function getUrl(): string
    {
        return $this->url;
    }

    public function getTitle(): string
    {
        return $this->title;
    }
}
```

Every command needs a handler that executes it. There is always a one to one relationship between commands and handlers.

Command handlers don't contain any business logic. They only concern themselves with the glue code that calls the domain layer.

src/Submission/Application/SubmitLinkHandler.php

```
<?php declare(strict_types=1);

namespace SocialNews\Submission\Application;

use SocialNews\Submission\Domain\SubmissionRepository;
use SocialNews\Submission\Domain\Submission;

final class SubmitLinkHandler
{
    private $submissionRepository;

    public function __construct(SubmissionRepository $submissionRepository)
    {
        $this->submissionRepository = $submissionRepository;
    }

// ...
```

Continued from the previous page

```
// ...

    public function handle(SubmitLink $command): void
    {
        $submission = Submission::submit(
            $command->getUrl(),
            $command->getTitle()
        );
        $this->submissionRepository->add($submission);
    }
}
```

It's a good idea to declare a `void` return type for your command handler methods. The user input was already validated and if something else goes wrong, it's most likely not something where you have to inform the user with a nice error message.

You can use the command handler through a command bus or you can use it directly as a dependency of your controller class. We won't use a command bus in this tutorial, but for larger projects it's often a good idea to use one.

A command bus is a layer of indirection between your controllers and command handlers. It's very easy to add transactions, retrying of commands, logging, asynchronity and similar features to your application when you use a command bus. Instead of passing the command directly to its handler, you pass it to the bus instead. The bus then passes the command through the configured middlewares before the handler receives it.

Add the command handler as a constructor dependency to your submission controller. Then replace the `// save the submission` comment in your `submit()` method with the following code. As always, don't forget to add the use statements at the top.

src/Submission/Presentation/SubmissionController.php

```
// ...

use SocialNews\Submission\Application\SubmitLink;

// ...

$this->submitLinkHandler->handle(new SubmitLink(
    $request->get('url'),
    $request->get('title')
));

// ...
```

The submit form is still broken at this stage. We don't have an implementation of the submission repository yet. We are going to fix that in the next subchapter.

SQL injections

SQL injection is a code injection attack where an attacker can modify a SQL statement. After a successful attack, the attacker can tamper with your data or get access to data that he otherwise wouldn't have access to. A 2012 study showed that an average web application is the target of about 4 SQL injection attack campaigns per month (Imperva, 2012).

Injection attacks are not limited to SQL, they are just one common way to inject code. Cross-site scripting (XSS) works in a similar way and other examples are file upload code injection and command injection (console commands). You can prevent all code injection attacks if you escape untrusted input properly.

Now we are going to create an implementation for the submission repository. The following code includes the vulnerability that we are going to exploit in the next step.

Create the `Infrastructure` directory for your submission context and create a new `DbalSubmissionRepository` class.

src/Submission/Infrastructure/DbalSubmissionRepository.php

```
<?php declare(strict_types=1);

namespace SocialNews\Submission\Infrastructure;

use Doctrine\DBAL\Connection;
use SocialNews\Submission\Domain\Submission;
use SocialNews\Submission\Domain\SubmissionRepository;

final class DbalSubmissionRepository implements SubmissionRepository
{
    private $connection;

    public function __construct(Connection $connection)
    {
        $this->connection = $connection;
    }

    public function add(Submission $submission): void
    {
        $this->connection->exec("
            INSERT INTO
                submissions (id, title, url, creation_date)
            VALUES (
                '{$submission->getId()->toString()}',
                '{$submission->getTitle()}',
                '{$submission->getUrl()}',
                '{$submission->getCreationDate()->format('Y-m-d H:i:s')}'
            );
        ");
    }
}
```

Now we have to go back to the dependencies file and add a new alias.

src/Dependencies.php

```
// ...

use SocialNews\Submission\Domain\SubmissionRepository;
use SocialNews\Submission\Infrastructure\DbalSubmissionRepository;

// ...
```

Continued from the previous page

```
// ...

$injector->alias(SubmissionRepository::class, DbalSubmissionRepository::class);

// ...
```

Add a few submissions to your website to check that everything works fine. The new posts should appear on the homepage, with the newest one on top.

So far so good, but now let's try to inject some SQL. Add one more submission, but this time with the following title (make sure you get all the characters right).

```
MY BLOG', 'http://patricklouys.com', '2100-01-01 00:00:00'); /*
```

This new submission should appear on the homepage, together with all the others. The most recent submission always went to the top until now, pushing the others down. But you will notice that "MY BLOG" always stays on top, even if you add more regular submissions. With the SQL injection, we were able to manipulate its creation date and set it into the future.

This is only a very simple example, other SQL injection attacks can be much more devastating. An attacker could expose sensitive data, get admin access or delete all your data.

It is very easy to prevent SQL injections, we just have to use parameterized queries. Never concenate a query string with a variable that does not have its value hardcoded somewhere.

Let's rewrite the code to make it secure with named parameters. We can do this with Doctrine DBAL, as you can see in the following example. But you don't need DBAL to get named parameter support, you can use them with PDO too.

src/Submission/Infrastructure/DbalSubmissionRepository.php

```
// ...

public function add(Submission $submission): void
{
    $qb = $this->connection->createQueryBuilder();

    $qb->insert('submissions');
    $qb->values([
        'id' => $qb->createNamedParameter($submission->getId()->toString()),
        'title' => $qb->createNamedParameter($submission->getTitle()),
        'url' => $qb->createNamedParameter($submission->getUrl()),
        'creation_date' => $qb->createNamedParameter(
            $submission->getCreationDate(),
            'datetime'
        ),
    ]);

    $qb->execute();
}

// ...
```

Try the SQL injection attack again. This time, it won't work and the whole input from the title field will be correctly interpreted as the title.

Input validation

Your application is now protected against SQL injection attacks, but there is still an important puzzle piece missing. We have to validate the input from the user. Unlike other databases, Sqlite is very flexible with string length and by default a length of 1 billion characters is supported.

Disk space is cheap, but it's a still good idea to set a limit to what users can submit to our website. Let's limit the title and URL field to a maximum of 200 bytes each. While we are at it, we can also make sure that the fields are not empty.

> The `strlen()` function counts bytes and not characters. We declared that our charset is UTF-8 in our HTML and UTF-8 supports multi-byte characters. You need to install the `mbstring` PHP extension and use the `mb_strlen()` function if you want to count the characters correctly in all cases.

We are going to put the input validation into our `SubmissionController`. Commands should be fire and forget, which means that the input has to be validated before the command is fired.

Create a new `SubmissionForm` class in your `Presentation` directory. We could put the validation code directly into the controller, but it will get bloated quickly when you have large forms. A separate form object also makes it easy to reuse the validation logic.

src/Submission/Presentation/SubmissionForm.php

```
<?php declare(strict_types=1);

namespace SocialNews\Submission\Presentation;

use SocialNews\Framework\Csrf\StoredTokenValidator;

final class SubmissionForm
{
    private $storedTokenValidator;
    private $token;
    private $title;
    private $url;

    public function __construct(
        StoredTokenValidator $storedTokenValidator,
        string $token,
        string $title,
        string $url
    ) {
        $this->storedTokenValidator = $storedTokenValidator;
        $this->token = $token;
        $this->title = $title;
        $this->url = $url;
    }
}
```

The form needs access to the `StoredTokenValidator`, because we are going to move the CSRF token validation into this class. Having all the validation in one place keeps everything cohesive and it cleans up the controller method.

Add a method to get access to the validation errors.

src/Submission/Presentation/SubmissionForm.php

```
// ...

use SocialNews\Framework\Csrf\Token;

// ...

/**
 * @return string[]
 */
public function getValidationErrors(): array
{
    $errors = [];

    if (!$this->storedTokenValidator->validate(
        'submission',
        new Token($this->token)
    )) {
        $errors[] = 'Invalid token';
    }

    if (strlen($this->title) < 1 || strlen($this->title) > 200) {
        $errors[] = 'Title must be between 1 and 200 characters';
    }

    if (strlen($this->url) < 1 || strlen($this->url) > 200) {
        $errors[] = 'URL must be between 1 and 200 characters';
    }

    return $errors;
}

// ...
```

Add another method that checks whether a validation error happened. We are going to reuse the `getValidationErrors()` method to check for validation errors, to keep the code simple.

src/Submission/Presentation/SubmissionForm.php

```
// ...

public function hasValidationErrors(): bool
{
    return (count($this->getValidationErrors()) > 0);
}

// ...
```

After a successful validation, we want to turn the form data into a command. Right now, this code is in the controller method. But it makes more sense to move this code into our new form class.

src/Submission/Presentation/SubmissionForm.php

```
// ...

use SocialNews\Submission\Application\SubmitLink;

// ...

public function toCommand(): SubmitLink
{
    return new SubmitLink($this->url, $this->title);
}

// ...
```

Because `SubmissionForm` has a dependency on `StoredTokenValidator`, it doesn't make sense that we construct the form object directly from the controller. If we did that, the controller would have to depend on the validator - only to pass it along.

In your presentation directory, create a `SubmissionFormFactory`.

src/Submission/Presentation/SubmissionFormFactory.php

```
<?php declare(strict_types=1);

namespace SocialNews\Submission\Presentation;

use SocialNews\Framework\Csrf\StoredTokenValidator;
use Symfony\Component\HttpFoundation\Request;

final class SubmissionFormFactory
{
    private $storedTokenValidator;

    public function __construct(StoredTokenValidator $storedTokenValidator)
    {
        $this->storedTokenValidator = $storedTokenValidator;
    }

    public function createFromRequest(Request $request): SubmissionForm
    {
        return new SubmissionForm(
            $this->storedTokenValidator,
            (string)$request->get('token'),
            (string)$request->get('title'),
            (string)$request->get('url')
        );
    }
}
```

Replace the `StoredTokenValidator` dependency of your submission controller with a new dependency on the factory. Then refactor the `submit()` method until the code looks like the following.

src/Submission/Presentation/SubmissionController.php

```
<?php declare(strict_types=1);

namespace SocialNews\Submission\Presentation;

use SocialNews\Submission\Application\SubmitLinkHandler;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use SocialNews\Framework\Rendering\TemplateRenderer;
use Symfony\Component\HttpFoundation\Session\Session;

// ...
```

Continued from the previous page

```
// ...

final class SubmissionController
{
    private $templateRenderer;
    private $submissionFormFactory;
    private $session;
    private $submitLinkHandler;

    public function __construct(
        TemplateRenderer $templateRenderer,
        SubmissionFormFactory $submissionFormFactory,
        Session $session,
        SubmitLinkHandler $submitLinkHandler
    ) {
        $this->templateRenderer = $templateRenderer;
        $this->submissionFormFactory = $submissionFormFactory;
        $this->session = $session;
        $this->submitLinkHandler = $submitLinkHandler;
    }

    public function show(): Response
    {
        $content = $this->templateRenderer->render('Submission.html.twig');
        return new Response($content);
    }

    public function submit(Request $request): Response
    {
        $response = new RedirectResponse('/submit');
        $form = $this->submissionFormFactory->createFromRequest($request);
        if ($form->hasValidationErrors()) {
            foreach ($form->getValidationErrors() as $errorMessage) {
                $this->session->getFlashBag()->add('errors', $errorMessage);
            }
            return $response;
        }
        $this->submitLinkHandler->handle($form->toCommand());
        $this->session->getFlashBag()->add(
            'success',
            'Your URL was submitted successfully'
        );
        return $response;
    }
}
```

Make sure that your submission form still works. Try to submit titles and URLs that are both too long and too short.

9. Registration

Introduction

You completed the main feature of the application in the last chapter. It's now possible to add new submissions and you can view them on the homepage. But we don't want an anonymous social news website.

In this chapter, we are going to add users to the application. They will have to create an account before they are able to submit a new link. But before we start to code, let's have a look at how we're going to handle passwords.

Password Hashing

Insecure password management is a problem that is widespread in our industry. Even large companies don't always secure their user data properly. You can find many examples where plain-text passwords were dumped from a database, after a large company got hacked.

Why is it so important to handle passwords correctly? After all, a lot of the problems are only visible after you have been hacked already. And only other people get hacked, right?

When someone manages to leak a database dump to the internet, it's a PR disaster for the company. This can happen even if you avoid all the vulnerabilities that we talk about. Even if you write very secure software, you still have people who can access the data.

If best practices are followed, then only the hashes of the passwords can be leaked and the plain-text passwords have to be bruteforced. This buys the affected users some time to change their passwords.

Password should always be hashed, not encrypted. Encrypted content can be decrypted, but hashing only works one way. Even if an attacker manages to get a copy of your code and your database, he won't be able to recover a plain-text password.

Hashing the passwords also means that you don't know the passwords of your users. You can't send them their password if they lose it - you have to generate a new one for them. This is a good thing and it increases the security of your application.

With a fixed-length hash function, you don't have to worry about the length of the password or which characters are allowed. You only have to store the hash in your database, which is always in the same format.

You can't just use any random hash function, like `md5()` or `sha1()` for example. They are good for some things, but password hashing is not one of them. They are too fast and we need a slow hash function. A fast hash function makes brute force attacks very easy.

You also have to add a random string (salt) to the password of each user. This makes sure that an attacker can't just compare the hashes of common passwords with the hashes from your database (a hash table attack). It also makes it harder to reverse the hash function with a precomputed table (a rainbow table attack). A salt turns a common password into an uncommon one.

There many other possible attack vectors that an attacker can exploit. "Don't roll your own crypto" is what every expert will tell you. You can come up with an elaborate algorithm, but it will easily be broken by an experienced attacker.

> Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break.
>
> *Bruce Schneier (Schneier's Law)*

Use the `password_hash()` function from PHP to hash passwords. It uses bcrypt behind the scenes, which is a solid password hashing algorithm. A salt is generated by default and you don't have to create one yourself.

Registration

Create a new `User` directory in your `src` directory. Create the subdirectories for the different layers in there. Use the `Submission` context as a reference.

For our user accounts, we are only going to require a username and password. If you create an application that doesn't display usernames, then you can use an email address to identify a user instead.

Go to into your new domain directory and create a `User` entity.

src/User/Domain/User.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Domain;

use DateTimeImmutable;
use Ramsey\Uuid\UuidInterface;

// ...
```

Continued from the previous page

```
//...

final class User
{
    private $id;
    private $nickname;
    private $passwordHash;
    private $creationDate;

    private function __construct(
        UuidInterface $id,
        string $nickname,
        string $passwordHash,
        DateTimeImmutable $creationDate
    ) {
        $this->id = $id;
        $this->nickname = $nickname;
        $this->passwordHash = $passwordHash;
        $this->creationDate = $creationDate;
    }
}
```

As you can see in the previous example, we are only storing the password hash in the entity state. We don't have access to the plain text password when we create a new user instance with data from the database.

We are also going to create a named constructor that contains business logic. We convert the plain text password to the hash in there, with the help of the `password_hash` function.

src/User/Domain/User.php

```
// ...

use Ramsey\Uuid\Uuid;

// ...
```

Continued from the previous page

```
//...

    public static function register(string $nickname, string $password): User
    {
        return new User(
            Uuid::uuid4(),
            $nickname,
            password_hash($password, PASSWORD_DEFAULT),
            new DateTimeImmutable()
        );
    }

// ...
```

And of course we need a matching repository.

src/User/Domain/UserRepository.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Domain;

interface UserRepository
{
    public function add(User $user): void;
}
```

This is everything that we need for the user registration in the domain layer. Let's move to the application layer and add a command and handler.

The following steps are a little bit repetitive. The code is similar to the code that we wrote for the submission context. We'll move fast and I won't explain everything again.

src/User/Application/RegisterUser.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Application;

final class RegisterUser
{
    private $nickname;
    private $password;

    public function __construct(string $nickname, string $password)
    {
        $this->nickname = $nickname;
        $this->password = $password;
    }

    public function getNickname(): string
    {
        return $this->nickname;
    }

    public function getPassword(): string
    {
        return $this->password;
    }
}
```

Add a getter for both the nickname and password. Then continue with the command handler.

src/User/Application/RegisterUserHandler.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Application;

use SocialNews\User\Domain\UserRepository;
use SocialNews\User\Domain\User;

final class RegisterUserHandler
{
    private $userRepository;

// ...
```

Continued from the previous page

```
//...

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    public function handle(RegisterUser $command): void
    {
        $user = User::register(
            $command->getNickname(),
            $command->getPassword()
        );
        $this->userRepository->add($user);
    }
}
```

Now we can move to the presentation layer. Create a `Registration.html.twig` template.

templates/Registration.html.twig

```
{% extends 'Layout.html.twig' %}

{% block content %}
    {% include 'FlashMessages.html.twig' %}
    <form method="post" action="">
        <label for="nickname">Nickname</label>
        <input type="text" name="nickname" id="nickname"><br>
        <label for="password">Password</label>
        <input type="password" name="password" id="password"><br>
        <input type="hidden" name="token"
               value="{{ get_token('registration') }}">
        <input type="submit" value="Register">
    </form>
{% endblock %}
```

And of course we also need a controller to display the registration form.

src/User/Presentation/RegistrationController.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Presentation;

use SocialNews\Framework\Rendering\TemplateRenderer;
use Symfony\Component\HttpFoundation\Response;

final class RegistrationController
{
    private $templateRenderer;

    public function __construct(TemplateRenderer $templateRenderer)
    {
        $this->templateRenderer = $templateRenderer;
    }

    public function show(): Response
    {
        $content = $this->templateRenderer->render('Registration.html.twig');
        return new Response($content);
    }
}
```

And we need a new entry in the `Routes.php` file.

src/Routes.php

```
// ...

[
    'GET',
    '/register',
    'SocialNews\User\Presentation\RegistrationController#show'
],

// ...
```

Check `/register` in your browser to see if everything works so far. If you submit the form, you will get an error because we didn't add a `POST` route yet. Before we add the route and controller method, let's create another form object (similar to the `SubmissionForm` object).

src/User/Presentation/RegisterUserForm.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Presentation;

use SocialNews\Framework\Csrf\StoredTokenValidator;

final class RegisterUserForm
{
    private $storedTokenValidator;
    private $token;
    private $nickname;
    private $password;

    public function __construct(
        StoredTokenValidator $storedTokenValidator,
        string $token,
        string $nickname,
        string $password
    ) {
        $this->storedTokenValidator = $storedTokenValidator;
        $this->token = $token;
        $this->nickname = $nickname;
        $this->password = $password;
    }
}
```

Now add the same validation methods that we also used in our `SubmissionForm` object.

src/User/Presentation/RegisterUserForm.php

```
// ...

use SocialNews\User\Application\RegisterUser;
use SocialNews\Framework\Csrf\Token;

// ...

public function hasValidationErrors(): bool
{
    return (count($this->getValidationErrors()) > 0);
}

// ...
```

Continued from the previous page

```
// ...

/**
 * @return string[]
 */
public function getValidationErrors(): array
{
    $errors = [];

    if (!$this->storedTokenValidator->validate(
        'registration',
        new Token($this->token)
    )) {
        $errors[] = 'Invalid token';
    }

    return $errors;
}

public function toCommand(): RegisterUser
{
    return new RegisterUser($this->nickname, $this->password);
}
```

We want to limit the nickname to 3-20 alphanumeric characters. This is to make sure that the names work well with our user interface. Add the following two checks below the token validation in the `getValidationErrors()` method.

src/User/Presentation/RegisterUserForm.php

```
// ...

if (strlen($this->nickname) < 3 || strlen($this->nickname) > 20) {
    $errors[] = 'Nickname must be between 3 and 20 characters';
}

if (!ctype_alnum($this->nickname)) {
    $errors[] = 'Nickname can only consist of letters and numbers';
}

// ...
```

For the password, we only want to set the minimum length to 8. You could add other minimum password complexity rules there, but keep in mind that it has been shown that overly complex rules lead to passwords that are easy to guess.

If you make it hard for your users to remember their passwords, they will work against you and pick easy default combinations and/or write their passwords down. If you require upper/lower case, numbers and a symbol, a lot of people will just capitalize the first letter and suffix their password with 1! (or use another common variation).

src/User/Presentation/RegisterUserForm.php

```
// ...

if (strlen($this->password) < 8) {
    $errors[] = 'Password must be at least 8 characters';
}

// ...
```

It's a bad idea to limit the length of the password. The hash will be a fixed length anyways and we should encourage users to use long passwords, not discourage them.

We also have to add some validation logic to make sure that the usernames are unique, but for that we have to know where we store the user data. We are going to add this later.

Because we use the `StoredTokenValidator` class as a dependency, we also have to create a factory for our form object.

src/User/Presentation/RegisterUserFormFactory.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Presentation;

use SocialNews\Framework\Csrf\StoredTokenValidator;
use Symfony\Component\HttpFoundation\Request;

final class RegisterUserFormFactory
{
    private $storedTokenValidator;

    public function __construct(StoredTokenValidator $storedTokenValidator)
    {
        $this->storedTokenValidator = $storedTokenValidator;
    }

    public function createFromRequest(Request $request): RegisterUserForm
    {
        return new RegisterUserForm(
            $this->storedTokenValidator,
            (string)$request->get('token'),
            (string)$request->get('nickname'),
            (string)$request->get('password')
        );
    }
}
```

Now we can finally go back to the registration controller and add a `register()` method. But before you do so, add the `RegisterUserFormFactory` and `Session` as constructor dependencies to the controller.

src/User/Presentation/RegistrationController.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Presentation;

use SocialNews\Framework\Rendering\TemplateRenderer;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpFoundation\Session\Session;

// ...
```

Continued from the previous page

```
// ...

final class RegistrationController
{
    private $templateRenderer;
    private $registerUserFormFactory;
    private $session;

    public function __construct(
        TemplateRenderer $templateRenderer,
        RegisterUserFormFactory $registerUserFormFactory,
        Session $session
    ) {
        $this->templateRenderer = $templateRenderer;
        $this->registerUserFormFactory = $registerUserFormFactory;
        $this->session = $session;
    }

    public function show(): Response
    {
        $content = $this->templateRenderer->render('Registration.html.twig');
        return new Response($content);
    }

    public function register(Request $request): Response
    {
        $response = new RedirectResponse('/register');
        $form = $this->registerUserFormFactory->createFromRequest($request);

        if ($form->hasValidationErrors()) {
            foreach ($form->getValidationErrors() as $errorMessage) {
                $this->session->getFlashBag()->add('errors', $errorMessage);
            }
            return $response;
        }

        // register the user

        $this->session->getFlashBag()->add(
            'success',
            'Your account was created. You can now log in.'
        );
        return $response;
    }
}
```

Add a new entry to your routes file.

src/Routes.php

```
// ...

[
    'POST',
    '/register',
    'SocialNews\User\Presentation\RegistrationController#register'
],

// ...
```

You should now be able to test your validation. Go to `/register` in your browser and try a few different combinations.

If everything works, add `RegisterUserHandler` as a new dependency to your controller and replace the comment in the `register()` method with the following code.

src/User/Presentation/RegistrationController.php

```
// ...

$this->registerUserHandler->handle($form->toCommand());

// ...
```

This breaks the registration form, because we don't have an implementation for the `UserRepository` yet. Before we can create one, we have to add a new table to our database.

Database Migration

We need another migration to create a `users` table. Use the date and time for the class name, just as we did it for the first migration. Append `201706040802` for the date and time `2017-06-04 08:02` (use the current date and time).

migrations/Migration201706040802.php

```
<?php declare(strict_types=1);

namespace Migrations;

use Doctrine\DBAL\Connection;
use Doctrine\DBAL\Schema\Schema;
use Doctrine\DBAL\Types\Type;

final class Migration201706040802
{
    private $connection;

    public function __construct(Connection $connection)
    {
        $this->connection = $connection;
    }

    public function migrate(): void
    {
        $schema = new Schema();
        $this->createUsersTable($schema);

        $queries = $schema->toSql($this->connection->getDatabasePlatform());
        foreach ($queries as $query) {
            $this->connection->executeQuery($query);
        }
    }

    private function createUsersTable(Schema $schema): void
    {
        $table = $schema->createTable('users');
        $table->addColumn('id', Type::GUID);
        $table->addColumn('nickname', Type::STRING);
        $table->addColumn('password_hash', TYPE::STRING);
        $table->addColumn('creation_date', TYPE::DATETIME);
        $table->addColumn('failed_login_attempts', TYPE::INTEGER, [
            'default' => 0,
        ]);
        $table->addColumn('last_failed_login_attempt', TYPE::DATETIME, [
            'notnull' => false,
        ]);
    }
}
```

The first few fields should be self-explanatory. The last two are there for security reasons, they will make it possible to prevent brute force login attempts.

If you open `bin/Migrate.php` in your editor, you can see that we have to do a few changes to the file before we can support multiple migrations. Let's change the workflow of our script a little bit. When it is called, we are going to give the user a selection where he can select the migration that he wants to execute.

We have to do a major refactoring of the file to add this new functionality. To make it easier, delete all the content of your `Migrate.php` and start fresh with the following code.

bin/Migrate.php

```
<?php declare(strict_types=1);

define('ROOT_DIR', dirname(__DIR__));

require ROOT_DIR . '/vendor/autoload.php';

$injector = include(ROOT_DIR . '/src/Dependencies.php');
$connection = $injector->make('Doctrine\DBAL\Connection');
```

Now add the following function that returns the class names of all our migration files in chronological order.

bin/Migrate.php

```
// ...

function getAvailableMigrations(): array
{
    $migrations = [];
    foreach (new FilesystemIterator(ROOT_DIR . '/migrations') as $file) {
        $migrations[] = $file->getBasename('.php');
    }
    return array_reverse($migrations);
}
```

We also need a function that shows a selection to the user.

bin/Migrate.php

```
// ...

function selectMigration(array $migrations): int
{
    echo "[0] All" . PHP_EOL;
    foreach ($migrations as $key => $name) {
        $index = $key + 1;
        echo "[$index] $name" . PHP_EOL;
    }
    $selected = readline('Select the migration that you want to run: ');
    $selectedKey = $selected - 1;
    if ($selected !== '0' && !array_key_exists($selectedKey, $migrations)) {
        exit('Invalid selection' . PHP_EOL);
    }
    return (int)$selected;
}
```

Then we have to call those functions and execute the selected migration(s).

bin/Migrate.php

```
// ...

$migrations = getAvailableMigrations();
$selected = selectMigration($migrations);

foreach ($migrations as $key => $migration) {
    if ($selected !== 0 && $selected !== $key + 1) {
        continue;
    }
    $class = "Migrations\\$migration";
    (new $class($connection))->migrate();
    echo "Running $migration..." . PHP_EOL;
}
```

To try out your new migration script, delete your database file, use the command `php bin/Migrate.php` and execute the last migration. If you want, you can also delete your database file and run a complete migration.

> For a serious project, use the migration tools that come with your framework or have a look at the Phinx package.

The user repository implementation

Now we are almost ready to create the implementation for the user repository. We just have to add a few getters to our `User` entity. Otherwise, we won't be able to access the values that we want to write to the database.

src/User/Domain/User.php

```
// ...

public function getId(): UuidInterface
{
    return $this->id;
}

public function getNickname(): string
{
    return $this->nickname;
}

public function getPasswordHash(): string
{
    return $this->passwordHash;
}

public function getCreationDate(): DateTimeImmutable
{
    return $this->creationDate;
}

// ...
```

The new repository is going to look very similar to the repository that we created in a previous chapter.

src/User/Infrastructure/DbalUserRepository.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Infrastructure;

use Doctrine\DBAL\Types\Type;
use Doctrine\DBAL\Connection;
use SocialNews\User\Domain\User;
use SocialNews\User\Domain\UserRepository;

final class DbalUserRepository implements UserRepository
{
    private $connection;

    public function __construct(Connection $connection)
    {
        $this->connection = $connection;
    }

    public function add(User $user): void
    {
        $qb = $this->connection->createQueryBuilder();

        $qb->insert('users');
        $qb->values([
            'id' => $qb->createNamedParameter($user->getId()->toString()),
            'nickname' => $qb->createNamedParameter($user->getNickname()),
            'password_hash' => $qb->createNamedParameter(
                $user->getPasswordHash()
            ),
            'creation_date' => $qb->createNamedParameter(
                $user->getCreationDate(),
                Type::DATETIME
            ),
        ]);

        $qb->execute();
    }
}
```

We need only one more entry in your dependencies file before we can try the registration.

src/Dependencies.php

```
// ...

use SocialNews\User\Domain\UserRepository;
use SocialNews\User\Infrastructure\DbalUserRepository;

// ...

$injector->alias(UserRepository::class, DbalUserRepository::class);

// ...
```

Navigate to `/register` and register a new user. You can check if the user was created with an SQLite management tool, if you have one installed.

The registration is almost done, but there is one more thing that we have to sort out. At the moment, it's possible to register with a duplicate nickname. We have to add another validation rule to prevent that from happening.

Create an interface for the following query in the application directory of your user context.

src/User/Application/NicknameTakenQuery.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Application;

interface NicknameTakenQuery
{
    public function execute(string $nickname): bool;
}
```

After that, create the implementation in the infrastructure directory.

src/User/Infrastructure/DbalNicknameTakenQuery.php

```php
<?php declare(strict_types=1);

namespace SocialNews\User\Infrastructure;

use Doctrine\DBAL\Connection;
use SocialNews\User\Application\NicknameTakenQuery;

final class DbalNicknameTakenQuery implements NicknameTakenQuery
{
    private $connection;

    public function __construct(Connection $connection)
    {
        $this->connection = $connection;
    }

    public function execute(string $nickname): bool
    {
        $qb = $this->connection->createQueryBuilder();

        $qb->select('count(*)');
        $qb->from('users');
        $qb->where("nickname = {$qb->createNamedParameter($nickname)}");
        $qb->execute();

        $stmt = $qb->execute();
        return (bool)$stmt->fetchColumn();
    }
}
```

Add another alias to your dependencies file.

src/Dependencies.php

```php
// ...

use SocialNews\User\Application\NicknameTakenQuery;
use SocialNews\User\Infrastructure\DbalNicknameTakenQuery;

// ...

$injector->alias(NicknameTakenQuery::class, DbalNicknameTakenQuery::class);

// ...
```

Now you can add `NicknameTakenQuery` as a new dependency to your form object, just after the `StoredTokenValidator` dependency. You also have to add it to the `RegisterUserFormFactory` as a dependency and pass it along to the `RegisterUserForm` in the `createFromRequest()` method.

With that done, you just have to add another validation rule to the `getValidationErrors()` method of your `RegisterUserForm` class.

src/User/Presentation/RegisterUserForm.php

```
// ...

if ($this->nicknameTakenQuery->execute($this->nickname)) {
    $errors[] = 'This nickname is already being used';
}

// ...
```

That's it. Make sure that you can't register an account with a duplicate nickname and then continue with the next chapter.

10. Authentication

Creating the login form

In the previous chapter, we created our first user. In this chapter, we are going to create a login form and then we implement the authentication process.

> Always use HTTPS for your registration, login and protected pages. Otherwise an attacker can steal the session ID or login credentials through a man in the middle attack.

Start with a `Login.html.twig` file in your templates directory.

templates/Login.html.twig

```
{% extends 'Layout.html.twig' %}

{% block content %}
    {% include 'FlashMessages.html.twig' %}
    <form method="post" action="">
        <label for="nickname">Nickname</label>
        <input type="text" name="nickname" id="nickname"><br>
        <label for="password">Password</label>
        <input type="password" name="password" id="password"><br>
        <input type="hidden" name="token"
               value="{{ get_token('login') }}"><br>
        <input type="submit" value="Log in">
    </form>
{% endblock %}
```

It might seem a little counter-intuitive at first, but we have to protect our login form from CSRF attacks. An attacker can't to log into an account with a CSRF attack (he would need the password for this). But he can make the victim log into an account that he owns and then access the user history and similar things. This is not really a big security problem for our social news application, but it's very easy to prevent.

Create a `LoginController` class in the presentation layer of your user context.

src/User/Presentation/LoginController.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Presentation;

use SocialNews\Framework\Rendering\TemplateRenderer;
use Symfony\Component\HttpFoundation\Response;

final class LoginController
{
    private $templateRenderer;

    public function __construct(TemplateRenderer $templateRenderer)
    {
        $this->templateRenderer = $templateRenderer;
    }

    public function show(): Response
    {
        $content = $this->templateRenderer->render('Login.html.twig');
        return new Response($content);
    }
}
```

Add the following entry to your route file.

src/Routes.php

```
// ...

[
    'GET',
    '/login',
    'SocialNews\User\Presentation\LoginController#show'
],

// ...
```

Navigate to `/login` to check if the form is displayed correctly.

The login command

Create a `LogIn` command in the application directory. If you are wondering why we keep switching between `Login` and `LogIn`, it's because it's one word as a noun and two words when it's used as a verb.

src/User/Application/LogIn.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Application;

final class LogIn
{
    private $nickname;
    private $password;

    public function __construct(string $nickname, string $password)
    {
        $this->nickname = $nickname;
        $this->password = $password;
    }

    public function getNickname(): string
    {
        return $this->nickname;
    }

// ...
```

Continued from the previous page

```
// ...

    public function getPassword(): string
    {
        return $this->password;
    }
}
```

We need to create a matching command handler in the same directory.

src/User/Application/LogInHandler.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Application;

use SocialNews\User\Domain\UserRepository;

final class LogInHandler
{
    private $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    public function handle(LogIn $command): void
    {
        // log in
    }
}
```

Before we continue with the command handler code, let's expand the user entity to add the code for the authentication. Start by adding the missing fields to the constructor and making it public. We don't need a named constructor when we instantiate the user from existing data, no business logic is executed and we have all the required data available.

src/User/Domain/User.php

```
// ...

private $id;
private $nickname;
private $passwordHash;
private $creationDate;
private $failedLoginAttempts;
private $lastFailedLoginAttempt;

public function __construct(
    UuidInterface $id,
    string $nickname,
    string $passwordHash,
    DateTimeImmutable $creationDate,
    int $failedLoginAttempts,
    ?DateTimeImmutable $lastFailedLoginAttempt
) {
    $this->id = $id;
    $this->nickname = $nickname;
    $this->passwordHash = $passwordHash;
    $this->creationDate = $creationDate;
    $this->failedLoginAttempts = $failedLoginAttempts;
    $this->lastFailedLoginAttempt = $lastFailedLoginAttempt;
}

// ...
```

We also have to add them to the static constructor `register()`.

src/User/Domain/User.php

```
// ...

return new User(
    Uuid::uuid4(),
    $nickname,
    password_hash($password, PASSWORD_DEFAULT),
    new DateTimeImmutable(),
    0,
    null
);

// ...
```

Add a new `logIn()` method. If the password verification fails, we increase the failed attempts. If it is successful, we reset the failed attempts.

src/User/Domain/User.php

```
// ...

public function logIn(string $password): void
{
    if (!password_verify($password, $this->passwordHash)) {
        $this->lastFailedLoginAttempt = new DateTimeImmutable();
        $this->failedLoginAttempts++;
        return;
    }
    $this->failedLoginAttempts = 0;
    $this->lastFailedLoginAttempt = null;
}

// ...
```

We also need getters for the new fields.

src/User/Domain/User.php

```
// ...

public function getFailedLoginAttempts(): int
{
    return $this->failedLoginAttempts;
}

public function getLastFailedLoginAttempt(): ?DateTimeImmutable
{
    return $this->lastFailedLoginAttempt;
}

// ...
```

Now let's have a look at the `UserRepository`. We need two new methods in our user repository interface, one to retrieve an already existing user and another one to save an updated user.

src/User/Domain/UserRepository.php

```
// ...

public function save(User $user): void;

public function findByNickname(string $nickname): ?User;

// ...
```

Create the `save()` method in your `DbalUserRepository` class. You can find it in your infrastructure directory.

src/User/Infrastructure/DbalUserRepository.php

```
// ...

public function save(User $user): void
{
    $qb = $this->connection->createQueryBuilder();

    $qb->update('users');
    $qb->set('nickname', $qb->createNamedParameter($user->getNickname()));
    $qb->set('password_hash', $qb->createNamedParameter(
        $user->getPasswordHash()
    ));
    $qb->set('failed_login_attempts', $qb->createNamedParameter(
        $user->getFailedLoginAttempts()
    ));
    $qb->set('last_failed_login_attempt', $qb->createNamedParameter(
        $user->getLastFailedLoginAttempt(),
        Type::DATETIME
    ));

    $qb->execute();
}

// ...
```

We need a private method that creates a new user from a database row.

src/User/Infrastructure/DbalUserRepository.php

```
// ...

use DateTimeImmutable;
use Ramsey\Uuid\Uuid;

// ...

private function createUserFromRow(array $row): ?User
{
    if (!$row) {
        return null;
    }

    $lastFailedLoginAttempt = null;
    if ($row['last_failed_login_attempt']) {
        $lastFailedLoginAttempt = new DateTimeImmutable(
            $row['last_failed_login_attempt']
        );
    }

    return new User(
        Uuid::fromString($row['id']),
        $row['nickname'],
        $row['password_hash'],
        new DateTimeImmutable($row['creation_date']),
        (int)$row['failed_login_attempts'],
        $lastFailedLoginAttempt
    );
}

// ...
```

Now you can create the `findByNickname()` method.

src/User/Infrastructure/DbalUserRepository.php

```
public function findByNickname(string $nickname): ?User
{
    $qb = $this->connection->createQueryBuilder();

// ...
```

Continued from the previous page

```
// ...

    $qb->addSelect('id');
    $qb->addSelect('nickname');
    $qb->addSelect('password_hash');
    $qb->addSelect('creation_date');
    $qb->addSelect('failed_login_attempts');
    $qb->addSelect('last_failed_login_attempt');
    $qb->from('users');
    $qb->where("nickname = {$qb->createNamedParameter($nickname)}");

    $stmt = $qb->execute();
    $row = $stmt->fetch();

    if (!$row) {
        return null;
    }

    return $this->createUserFromRow($row);
}

// ...
```

After writing this code, we can go back to the `LogInHandler`. We have to retrieve the user, call authenticate and then save the user to the repository.

src/User/Application/LogInHandler.php

```
// ...

public function handle(LogIn $command): void
{
    $user = $this->userRepository->findByNickname($command->getNickname());

    if ($user === null) {
        return;
    }

    $user->logIn($command->getPassword());

    $this->userRepository->save($user);
}

// ...
```

Open your `LoginController` class. We have to add a few new dependencies to it. Add `StoredTokenValidator`, `Session` and `LogInHandler` as new dependencies and then create a new `logIn()` method that handles the `POST` request.

src/User/Presentation/LoginController.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Presentation;

use SocialNews\Framework\Csrf\StoredTokenValidator;
use SocialNews\Framework\Rendering\TemplateRenderer;
use SocialNews\User\Application\LogInHandler;
use Symfony\Component\HttpFoundation\Response;
use SocialNews\Framework\Csrf\Token;
use SocialNews\User\Application\LogIn;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Session\Session;

final class LoginController
{
    private $templateRenderer;
    private $storedTokenValidator;
    private $session;
    private $logInHandler;

    public function __construct(
        TemplateRenderer $templateRenderer,
        StoredTokenValidator $storedTokenValidator,
        Session $session,
        LogInHandler $logInHandler
    ) {
        $this->templateRenderer = $templateRenderer;
        $this->storedTokenValidator = $storedTokenValidator;
        $this->session = $session;
        $this->logInHandler = $logInHandler;
    }

    public function show(): Response
    {
        $content = $this->templateRenderer->render('Login.html.twig');
        return new Response($content);
    }

// ...
```

Continued from the previous page

```
// ...

    public function logIn(Request $request): Response
    {
        if (!$this->storedTokenValidator->validate(
            'login',
            new Token((string)$request->get('token'))
        )) {
            $this->session->getFlashBag()->add('errors', 'Invalid token');
            return new RedirectResponse('/login');
        }

        $this->logInHandler->handle(new LogIn(
            (string)$request->get('nickname'),
            (string)$request->get('password')
        ));

        // validate that the user was logged in

        $this->session->getFlashBag()->add('success', 'You were logged in.');
        return new RedirectResponse('/');
    }
}
```

And a new entry to your routes configuration file.

src/Routes.php

```
// ...

[
    'POST',
    '/login',
    'SocialNews\User\Presentation\LoginController#logIn'
],

// ...
```

Because we redirect to the front page after a successful login, we have to include the flash messages there to show the success message. Add the flash messages just after the content block opens.

templates/FrontPage.html.twig

```
{% extends 'Layout.html.twig' %}

{% block content %}
    {% include 'FlashMessages.html.twig' %}

{# ... #}
```

You can now navigate to `/login`. Make sure that there are no errors when you go through the login process. If you paid attention earlier, then you will have noticed that nothing happens after a successful login.

Authenticating the client

HTTP is stateless, it remembers nothing between requests. That means we have to return a value to the client, which he then he can return with every request to identify himself. This is usually done with a cookie.

Cookies can be modified by the client, which is why we can't just write the user ID into a cookie. It would be trivial for an attacker to get access to another account.

Session data is stored on the server and only a reference to it is stored on the client. We can store the ID of the authenticated user in there to make sure that it can't be manipulated by the user. The only thing that he can see and change is his session ID, not the session data.

In our repository, we need to know whether the user was was logged in successfully or not. We could try to figure that out from the failed login attempt variables, but that approach is a little fragile. It could lead to unexpected sideeffects when we save the user entity and the `logIn()` method was never called.

To keep track of the login action, we are going to introduce a domain event. A domain event is just a class that represents a change that happened. When you use a domain event for an audit log or event sourcing, they will contain data like the event ID, when it occurred and a payload with the changes. We don't need that, which is why the class is empty. The name of the class communicates all the information that we need.

src/User/Domain/UserWasLoggedIn.php

```
<?php declare(strict_types=1);

namespace SocialNews\User\Domain;

final class UserWasLoggedIn {}
```

Add a private variable to your user entity to store the recorded events. Then add the following two methods, so that we can read and clear the recorded events.

src/User/Domain/User.php

```
// ...

private $recordedEvents = [];

// ...

public function getRecordedEvents(): array
{
    return $this->recordedEvents;
}

public function clearRecordedEvents(): void
{
    $this->recordedEvents = [];
}

// ...
```

Put the following line at the end of the `logIn()` method, to record the event when the password was verified successfully.

src/User/Domain/User.php

```
// ...

$this->recordedEvents[] = new UserWasLoggedIn();

// ...
```

Add Symfony's `Session` as a dependency to your `DbalUserRepository` and then add the following code at the top of your `save()` method.

src/User/Infrastructure/DbalUserRepository.php

```
// ...

use SocialNews\User\Domain\UserWasLoggedIn;
use LogicException;

// ...

foreach ($user->getRecordedEvents() as $event) {
    if ($event instanceof UserWasLoggedIn) {
        $this->session->set('userId', $user->getId()->toString());
        continue;
    }
    throw new LogicException(get_class($event) . ' was not handled');
}
$user->clearRecordedEvents();

// ...
```

The user ID should not be stored in the session after a successful login. But our controller doesn't get any feedback from the command. This is by design, because we want to keep commands and queries separated. To check whether the user was logged in, we are going to check if the session value was set.

Showing an error message after a failed login

For many applications, it's a good idea to hide the reason why a login failed. An error message like "invalid nickname" makes it possible for an attacker to bruteforce the nicknames of your users. In our social news application, this doesn't really matter because the nicknames are public anyways. Hiding this information becomes more important when your application uses emails or another non-public login identifier.

You also have to think about other attack vectors that an attacker can use to bruteforce your login IDs. The registration form could be a weakness, if you check for duplicate login IDs there. You can reduce that risk by requiring a captcha on the registration form.

Hiding your login IDs makes a brute-force attack on your website a harder to execute. But it doesn't help a lot, an attacker might already know the login ID of his target or he might be able to guess it. Keep that in mind if you decide to trade usability for the additional security.

Replace the `// validate that the user was logged in` comment in your login controller with the following code.

src/User/Presentation/LoginController.php

```
// ...

if ($this->session->get('userId') === null) {
    $this->session->getFlashBag()->add('errors', 'Invalid username or password');
    return new RedirectResponse('/login');
}

// ...
```

Add the following line at the top of your `logIn()` method. Because we check this value later, we want to be sure that it wasn't already set in an earlier request.

src/User/Presentation/LoginController.php

```
// ...

$this->session->remove('userId');

// ...
```

The login process should now work. Register a new user and try it out with both correct and wrong credentials.

Session hijacking

Cookies can be stolen, whether that's through a man in the middle attack or another attack vector. If your application is dealing with sensitive data, then your cookie should expire at the end of the client session. This is fine for something like an e-banking application, but it can be annoying if it's an application where the user logs in daily.

If you want to provide a better user experience, you can try the approach that is popular with e-commerce sites. You authenticate the user through a separate persistent cookie when he comes back to your site, but you set a flag and mark him as auto-logged in. Then when the user tries to do something with sensitive data, you make him enter the password.

Preventing brute force attacks

It takes virtually no time to crack a weak password if you don't slow down an attacker, even if he uses an old and slow computer. The same applies to alphanumeric passwords up to 9 characters. Even passwords with numbers, symbols and a mix of upper and lowercase passwords are trivial to crack, if they are 7 characters or less. But even a short and weak password takes a very long time to crack when the attacker is limited to only one attempt per second.

A delay with `sleep()` doesn't work well, because the attacker can just do a multi-threaded attack. You can limit the number of open requests per IP address, but that won't protect you from an attacker with many IP addresses at his disposal (a botnet for example). You could discard all login attempts until the end of the delay is reached, but that locks out your real users too.

It is better to use a captcha after a number of failed login attempts is reached. Captchas are annoying for your users, but they will only see them during an ongoing brute force attack or after they failed multiple login attempts.

Captchas are not foolproof and weak ones can be solved automatically by software. The stronger ones can still be solved, by outsourcing it to a factory full of poor people who solve captchas all day (yes, this actually happens...). They are still useful as brute force prevention, because they make the attack much more expensive.

If you keep track of the known IP addresses of your users (only store hashes), then you can create a whitelist and skip the captcha when a known IP tries to log into the appropriate account. Captchas are annoying and with a whitelist you can make your application a little more user-friendly.

You can't prevent a brute force attack against someone with unlimited resources. But you can make an attack on you so expensive that it doesn't make economical sense.

We won't implement brute force prevention as part of the tutorial. A good solution will be complex enough that you could write a separate book just on that topic. For a simple solution, you can use the number of failed login attempts to ask for a captcha if someone tries to log into an account after a few failed attempts.

If a single user is targeted, we should be fairly safe with this prevention method. But an attacker can also choose a broader approach and attack multiple users at the same time. He can cycle through a large number of accounts and try to get access with commonly used passwords.

It is almost guaranteed that some of your users will have a common password. That's why we have to make it hard for an attacker to attempt to log into many accounts simultaneously.

You can use the last failed login attempt field to check whether there is an unusual number of failed login attempts happening. This will uncover a distributed brute force attack. If you detect one, activate the captcha for all users.

Password recovery

We also won't implement a password recovery process during this tutorial. If you decide to implement one on your own, then make sure that you don't send the current password to a user. You shouldn't be able to do that anyways, if you followed the previous guidelines. Make the user go through the steps below instead.

1. Ask for identity data
2. Send a token over a side-channel like email
3. Allow the user to change password in the existing session, if he provides the token

Make sure that the user has to change the password before he can continue to use your website. Log all password reset attempts, so that you can detect malicious behaviour.

11. Authorization

Introduction

Authentication identifies a user and authorization decides what he can do.

Just checking whether a user is logged in is not enough. We want a more fine-grained authorization approach, one that makes it possible to give different permissions to different users. It is very likely that you will have to add additional user roles in the future, like an administrator role for example.

Role based access control

Role based access control (RBAC) is one way to control access to your resources. With a RBAC system, every user can have multiple roles and every role can have multiple permissions. You can then decide whether a user can do an action or not by checking all indirectly assigned permissions.

It's important that you don't just check whether a user has a specific role. This approach might work fine in the beginning, but it's too inflexible when you start to add new roles. Check the permission instead, not the role.

> Sometimes you need a more complex authorization model. Consider an attribute based access control (ABAC) in those cases. ABAC is more flexible, but harder to implement.

The role based access control is not really a part of the user context. It is something that will be used from the whole application and we are going to make it a part of our framework.

There are some authentication libraries that use strings for permissions, but I don't like that approach. Authorization is not really the place where you should take a shortcut. A small typo could result in a security nightmare. It's slightly more work to create an object for every permission, but the result is more robust and less error-prone.

Earlier in the book, you learned that you should usually pick composition over inheritance. We are going to make an exception to that rule in the following code, because in this case inheritance is a good choice.

We need an abstract class for the permission. Our permissions are not much more than strings, we just need to be able to compare them. If two permission objects are of the same class, they are equal.

Create a new `Rbac` directory in your `src/Framework` directory and then create a new `Permission` class in there.

src/Framework/Rbac/Permission.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rbac;

abstract class Permission
{
    public function equals(Permission $permission): bool
    {
        return (get_class() === get_class($permission));
    }
}
```

Every role returns an array of its permissions. We can then use this array to check whether the role has a permission or not.

src/Framework/Rbac/Role.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rbac;

abstract class Role
{
    public function hasPermission(Permission $permission): bool
    {
        return in_array($permission, $this->getPermissions());
    }

    /**
     * @return Permission[]
     */
    abstract protected function getPermissions(): array;
}
```

Create both a `Permission` and `Role` subdirectory in your `Rbac` directory. We are going to place the concrete permission and role classes in there.

The first permission that we need is `SubmitLink`. Thanks to our abstract class, we just have to extend the `Permission` parent and we can call it a day.

src/Framework/Rbac/Permission/SubmitLink.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rbac\Permission;

use SocialNews\Framework\Rbac\Permission;

final class SubmitLink extends Permission
{
}
```

Then we can add `Author` as our first role. It's also very straight-forward, we just have to implement the `getPermissions()` method and return an array with all the permissions of the role.

src/Framework/Rbac/Role/Author.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rbac\Role;

use SocialNews\Framework\Rbac\Permission\SubmitLink;
use SocialNews\Framework\Rbac\Role;

final class Author extends Role
{
    protected function getPermissions(): array
    {
        return [new SubmitLink()];
    }
}
```

Now we are ready to add users. We are going to create two different implementations, one for authenticated users and one for guests. Guests are users that are not authenticated. This approach makes it easy to check for a permission, because there is always a current user. It also makes it possible to give guests certain permissions.

Let's start with a `User` interface.

src/Framework/Rbac/User.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rbac;

interface User
{
    public function hasPermission(Permission $permission): bool;
}
```

The guest implementation is very simple, we always return false on the `hasPermission()` check. If you want to give a guest account a permission, add the code for that to this class.

src/Framework/Rbac/Guest.php

```php
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rbac;

final class Guest implements User
{
    public function hasPermission(Permission $permission): bool
    {
        return false;
    }
}
```

The `AuthenticatedUser` is a little more complex. It also has a `hasPermission()` method, but this one loops through all the roles of the user and checks whether at least one of the roles has the required permission. In addition to that, we also added a getter for the ID. We can use that getter later to attribute things to the current user (like a submitted link for example).

src/Framework/Rbac/AuthenticatedUser.php

```php
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rbac;

use Ramsey\Uuid\UuidInterface;

final class AuthenticatedUser implements User
{
    private $id;
    private $roles = [];

    /**
     * @param Role[] $roles
     */
    public function __construct(UuidInterface $id, array $roles)
    {
        $this->id = $id;
        $this->roles = $roles;
    }

// ...
```

Continued from the previous page

```
// ...

    public function getId(): UuidInterface
    {
        return $this->id;
    }

    public function hasPermission(Permission $permission): bool
    {
        foreach ($this->roles as $role) {
            if ($role->hasPermission($permission)) {
                return true;
            }
        }
        return false;
    }
}
```

Checking for a permission

We need one more puzzle piece to make everything work: a factory that returns the current user. Because we have to talk to the Symfony session for this, we are going to create a `CurrentUserFactory` interface and a concrete `SymfonySessionCurrentUserFactory` class.

src/Framework/Rbac/CurrentUserFactory.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rbac;

interface CurrentUserFactory
{
    public function create(): User;
}
```

src/Framework/Rbac/SymfonySessionCurrentUserFactory.php

```
<?php declare(strict_types=1);

namespace SocialNews\Framework\Rbac;

use Ramsey\Uuid\Uuid;
use SocialNews\Framework\Rbac\Role\Author;
use Symfony\Component\HttpFoundation\Session\Session;

final class SymfonySessionCurrentUserFactory
{
    private $session;

    public function __construct(Session $session)
    {
        $this->session = $session;
    }

    public function create(): User
    {
        if (!$this->session->has('userId')) {
            return new Guest();
        }

        return new AuthenticatedUser(
            Uuid::fromString($this->session->get('userId')),
            [new Author()]
        );
    }
}
```

Because we have only one role at the moment, the right approach is to hardcode it. If you decide to add another role, you can store them in the database and retrieve the roles of a user before you return an authenticated user.

There is no single correct place where you can place the code that acts as a firewall and checks the permission. You could do the check before a controller method is called, the required permission could be stored in the routes file or in a separate configuration file. You could also do the check manually in the controller methods. If you are using a command bus, you can create a bus that authorizes the user before a command is fired.

We are going to put it into the controller methods, for the sole reason that I don't want to bother you with writing any more boilerplate code. After all, you are reading this book because you want to learn more about programming and not because you want to create a social news application.

Add the following to your `Dependencies.php` file.

src/Dependencies.php

```
// ...

use SocialNews\Framework\Rbac\User;
use SocialNews\Framework\Rbac\SymfonySessionCurrentUserFactory;

// ...

$injector->delegate(User::class, function () use ($injector): User {
    $factory = $injector->make(SymfonySessionCurrentUserFactory::class);
    return $factory->create();
});

// ...
```

Add the `User` as a dependency to your `SubmissionController`. Make sure that you import the RBAC user class, don't use the one from the user context. Then insert the following if-statement at the top of both public methods.

src/Submission/Presentation/SubmissionController.php

```
// ...

use SocialNews\Framework\Rbac\Permission;

// ...

if (!$this->user->hasPermission(new Permission\SubmitLink())) {
    $this->session->getFlashBag()->add(
        'errors',
        'You have to log in before you can submit a link.'
    );
    return new RedirectResponse('/login');
}

// ...
```

Have a closer look at the `use` statement in the previous code. Instead of importing `SubmitLink` directly, we are importing the permission namespace. This makes it less likely to be confused with the `SubmitLink` command, which is also called from this controller.

Navigate to `/submit` when you are not logged in. You should be redirected to the login page with an error message. Then try it again after logging in.

That's it. You have successfully implemented role based access control for your application.

Adding the user to the submitted link

Only authenticated users can submit links, but we can't track the author of a submission at the moment. There is a missing link between our users and their submissions.

Changing the database table

We have to add a new `author_user_id` column to our submissions table to establish the missing link. You need a proper migration plan if a change like that happens on a live website. You would have to attribute the existing posts to a user and make sure that you don't lose any data.

Because we are still working on the initial version of the website, we can take a shortcut and edit the original migration file. We don't have real data that we can lose.

Add the following line to the `createSubmissionsTable()` method in your first migration file.

migrations/Migration201704151205.php

```
// ...

$table->addColumn('author_user_id', Type::GUID);

// ...
```

Delete the `db.sqlite3` file in your storage directory and then run all your migrations again (with the `php bin/Migrate.php` console command).

Make sure that you are not logged into a non-existing account. Delete the session cookie or open an incognito window to be sure. Navigate to `/register` to create a new account and then go to `/login` to log in again.

Changing the business logic

The user ID is another UUID, just like the submission ID. If you have a look at the `Submission` entity in the domain directory of your submission context, you can see that we are already use the `UuidInterface` there.

Types like `string`, `DateTimeImmutable` and `UuidInterface` don't convey a lot of domain specific meaning to the reader of the code. It tells us what type a variable is, but we don't know what it represents.

In domain driven design (DDD), types like that are usually avoided in the domain and they are replaced by value objects. That approach requires more classes, but it adds a lot of meaning to the code. You don't have to wrap every single value into a value object, but the public API of your domain objects should use them generously.

This is not really a DDD tutorial. We won't refactor all the code that we have written so far, but we are going to add a value object to represent the user ID. But we want to use language that is a little more specific to our context, which is why we are going to call it the author ID instead.

Calling it `AuthorId` instead of a generic `UserId` does not only add more meaning to the value object, it also is very useful if you have to refer to multiple users in the future. You could also assign a reviewer to a submission for example. Calling both `UserId` will be confusing when someone is trying to understand the code.

In the domain directory of the submission context, create your new `AuthorId` value object.

src/Submission/Domain/AuthorId.php

```
<?php declare(strict_types=1);

namespace SocialNews\Submission\Domain;

use Ramsey\Uuid\UuidInterface;

final class AuthorId
{
    private $id;

    private function __construct(string $id)
    {
        $this->id = $id;
    }

    public static function fromUuid(UuidInterface $uuid): AuthorId
    {
        return new AuthorId($uuid->toString());
    }

    public function toString(): string
    {
        return $this->id;
    }
}
```

Add the `AuthorId` to the constructor method of the submission entity, just after the ID.

src/Submission/Domain/Submission.php

```
// ...

private $authorId;

// ...

private function __construct(
    UuidInterface $id,
    AuthorId $authorId,
    string $url,

// ...

    $this->authorId = $authorId;

//...
```

You also have to add it to the named constructor.

src/Submission/Domain/Submission.php

```
// ...

public static function submit(
    UuidInterface $authorId,
    string $url,
    string $title
): Submission {
    return new Submission(
        Uuid::uuid4(),
        AuthorId::fromUuid($authorId),
        $url,
        $title,
        new DateTimeImmutable()
    );
}

// ...
```

Add a `getAuthorId()` method to the entity that returns the author ID.

src/Submission/Domain/Submission.php

```
// ...

public function getAuthorId(): AuthorId
{
    return $this->authorId;
}

// ...
```

After doing that, edit your `DbalSubmissionRepository` from the infrastructure layer and add the author ID to the values that are inserted into the table.

src/Submission/Infrastructure/DbalSubmissionRepository.php

```
// ...

'author_user_id' => $qb->createNamedParameter(
    $submission->getAuthorId()->toString()
),

// ...
```

Wiring everything together

Change the constructor of your `SubmitLink` command to include the author ID. Then add a `getAuthorId()` method to the class.

src/Submission/Application/SubmitLink.php

```
// ...

use Ramsey\Uuid\UuidInterface;

// ...

private $authorId;
private $url;
private $title;

// ...
```

Continued from the previous page

```
//...

public function __construct(UuidInterface $authorId, string $url, string $title)
{
    $this->authorId = $authorId;
    $this->url = $url;
    $this->title = $title;
}

public function getAuthorId(): UuidInterface
{
    return $this->authorId;
}

// ...
```

In the `SubmitLinkHandler` you only have to add one new line to add the new argument.

src/Submission/Application/SubmitLinkHandler.php

```
$submission = Submission::submit(
    $command->getAuthorId(),
    $command->getUrl(),
    $command->getTitle()
);
```

Switch to the presentation layer of the submission context and edit the `SubmissionForm`. We are creating the command in there, but it doesn't make sense to add the user as a dependency to the form object. It is not used for anything form-related. Add it as an argument to the `toCommand()` method instead.

src/Submission/Presentation/SubmissionForm.php

```
// ...

use SocialNews\Framework\Rbac\AuthenticatedUser;

// ...

public function toCommand(AuthenticatedUser $author): SubmitLink
{
    return new SubmitLink($author->getId(), $this->url, $this->title);
}

// ...
```

Now we have to pass the user to the method call in the `submit()` method of your `SubmissionController`.

src/Submission/Presentation/SubmissionController.php

```
// ...

$this->submitLinkHandler->handle($form->toCommand($this->user));

// ...
```

It should work now. But if you use an IDE like PHPStorm, it will tell you that you are passing in the wrong type. We only have an instance of `User` in the controller, which could also be a `Guest` instead of an `AuthenticatedUser`. As long as we don't add the `SubmitLink` permission to the guest, nothing bad will happen.

But software is not static and things change. We can add a few lines of defensive code to make sure that we catch a bug like that early. Little things like that don't take much time and they have saved me a lot of debugging time over the years.

Add the following block just above the `handle()` method call.

src/Submission/Presentation/SubmissionController.php

```
// ...

use SocialNews\Framework\Rbac\AuthenticatedUser;

// ...

if (!$this->user instanceof AuthenticatedUser) {
    throw new \LogicException('Only authenticated users can submit links');
}

// ...
```

Navigate to `/submit` and add a new submission. It should be added to the front page, but you will notice that you still can't see who the author is.

Displaying the author

Open the `Submission` class from the front page context. Add the author to the constructor arguments and create a `getAuthor()` method.

src/FrontPage/Application/Submission.php

```
// ...

private $url;
private $title;
private $author;

public function __construct(string $url, string $title, string $author)
{
    $this->url = $url;
    $this->title = $title;
    $this->author = $author;
}

// ...

public function getAuthor(): string
{
    return $this->author;
}

// ...
```

`DbalSubmissionsQuery` also needs a few adjustments. We have to join the users table and add the nickname of the author to the select statement.

src/FrontPage/Infrastructure/DbalSubmissionsQuery.php

```
// ...

$qb->addSelect('submissions.title');
$qb->addSelect('submissions.url');
$qb->addSelect('authors.nickname');
$qb->from('submissions');
$qb->join(
    'submissions',
    'users',
    'authors',
    'submissions.author_user_id = authors.id'
);
$qb->orderBy('submissions.creation_date', 'DESC');

// ...
```

After doing that, pass the nickname to the construction of the `Submission` object.

src/FrontPage/Infrastructure/DbalSubmissionsQuery.php

```
// ...

$submissions[] = new Submission($row['url'], $row['title'], $row['nickname']);

// ...
```

To finish, add the code to display the author to your `FrontPage.html.twig`.

templates/FrontPage.html.twig

```
<li>
    <a href="{{ submission.url|e('html_attr') }}">{{ submission.title }}</a><br>
    <small>submitted by {{ submission.author }}</small>
</li>
```

If you visit your front page now, the author of each submission should be shown below each title.

This concludes the tutorial. Of course there are a lot of things that have to be done before the social news website is production-ready, but we will stop here. Feel free to continue working on it while you learn new things. Or delete all the code and apply the lessons that you learned to a new project.

Further learning

This book was intended as an introduction to intermediate programming concepts. The rabbit hole goes much deeper and there is much more to learn.

We are coming to the end of this book, but your learning should not stop here. We are going to wrap this up with a few introductions to more advanced programming topics. I am highlighting the following topics because they are related to the CQRS/DDD approach that was introduced in this book.

Keep in mind that the CQRS/DDD approach is mostly aimed at business software. It's all about solving the business problems in your code. There are other ways to program and every approach has its own set of drawbacks and benefits.

Command bus

You were already introduced to commands and command handlers during the tutorial. A command bus makes it easy to wrap your commands with additional functionality. Below I have listed a few things that can be added to all your commands with just a few lines of code.

- Logging
- Database transactions
- Error handling
- Queueing (to make commands asynchronous)
- Authorization

If you want to learn more about the command bus, there is a good talk by Ross Tuck on the Youtube ("Models & Service Layers; Hemoglobin & Hobgoblins"). Additionally, Mathias Noback (https://matthiasnoback.nl) has a series of good blog posts on the topic.

There are a few packages that you can use for your command bus, like `league/tactician`. It's very easy to implement a command bus yourself, so you don't have to use an existing package. But if you write your own, make sure that the code is properly tested and look to the existing packages for inspiration.

Unit testing

Every software developer needs to know how to write tests. Unit tests are just one piece of the puzzle, but they are a good starting point. PHPUnit is by far the most widespread testing framework in the PHP world. Check out their documentation and learn how to use it.

For an introduction to unit tests, search for "The Clean Code Talks - Unit Testing" on Youtube. After watching that, there are many other good resources available, many of them about test driven development (TDD).

I prefer to write my tests after writing most of the code, but TDD is a good way to learn about tests. The same applies to aiming for 100% code coverage. It's a good way to learn, but not really ideal in the real world.

Don't confuse TDD with unit tests. You can write unit tests without always writing the tests first.

After learning how to use unit tests, learn when to use them. Mathias Verraes has a good blog post on the topic (http://verraes.net/2014/12/how-much-testing-is-too-much). But in the end, experience will be your best teacher.

Behaviour driven development

Unit testing is about testing the application from the view of the developer. Behaviour driven development turns this around and looks at the application from the view of the product owner. Instead of testing technical details, you are testing the business requirements.

While unit and functional tests help you to avoid technical bugs, BDD makes sure that you develop the right thing.

You often hear that your tests should act as documentation. But from my experience, it's rare that a developer actually dives into the unit tests to figure out how something works. Unit tests are code and usually not very easy to read.

With a BDD testing framework like Behat, you split the specification from the implementation. That makes it really easy to follow the business requirements and they can be read even by non-technical people, thanks to the Gherkin language.

BDD fits in really well together with a CQRS/DDD approach. It's very easy to test the commands and business logic with it. But queries are not a great match for BDD tests.

```
Scenario: Jeff submits a new link
    Given that Jeff is logged in
    When the user submits a link with the title "foo" and URL "http://foo.com"
    Then a new link with the title "foo" was created
```

The Behat documentation is a good starting point if you want to learn more about how to do BDD

As I mentioned earlier, I don't think that TDD with unit tests works really well for a real application. The problem with TDD is that developers end up putting too much focus on the "how".

If you look past the semantics, BDD is just a new name for TDD. But it puts the focus on the features that are relevant for the business. BDD is TDD done right.

Domain driven design

It took me a few years from first learning about domain driven design until I finally understood it. I had always used some pieces of DDD, but it only really clicked when I started to use all the pieces together.

We have used some of those pieces in this book, but there is much more to learn. DDD is not only about the code, it is also about finding out what code to write and how to structure it.

The following three books are the three big DDD books. The blue book is the classic, but I wouldn't recommend it as a first introduction. It is not easy to read. The books by Vaughn Vernon are better structured and more up to date.

- Domain Driven Design Distilled, by Vaughn Vernon (the green book)
- Implementing Domain Driven Design, by Vaughn Vernon (the red book)
- Domain Driven Design, by Eric Evans (the blue book)

I recommend that you first read the green book to learn about the high level concepts. It's a very short book and an easy read. Then go through the red book and use it as a reference when you write your first DDD code - it includes a lot of code examples.

I can also recommend the DDD blog posts and talks from Mathias Verraes (http://verraes.net). If you want to learn more about event storming, check out the book by Alberto Brandolini on the topic (http://eventstorming.com).

Event sourcing

With event sourcing (ES), you don't just store the current state in the database. You keep the whole change history there. It is similar to accounting, where you never change or erase a record. Instead, you always add a new record with the change. If you want to know the current total, you add up the history.

A lot of possibilties are created when you have the whole history of the system available. You have a built-in audit log and you can travel back through time, which is very powerful if you have to undo or debug something.

Compared to just storing state, event sourcing is quite a bit more complicated. It is also not something that should be used for everything. For most applications it makes sense to only use event sourcing in the bounded contexts where the business value is created.

Some information about event sourcing is already covered in the DDD resources that I mentioned previously. If you want to dive deeper, Greg Young has many good resources available. For a good introduction to the topic, check out his "CQRS and Event Sourcing" talk on Youtube from Code on the Beach 2014.

Wrapping it up

That's it. There are many more things that I could talk about, but then this book would never get finished. I have to draw the line somewhere.

I will continue to blog (http://patricklouys.ch) and if this book is received well, it won't be my last. If you're not already on my mailing list, you can go to my blog and sign up. I rarely send out emails, but I'll make sure that you won't miss any of my content.

In our profession, it's important to invest in yourself and you have to always keep learning. This will not only result in jobs that pay better, but also jobs where you can do more meaningful and enjoyable work. It's easy to get an agency job where you create one simple website after another, but not so easy to get a job where you get to work in a good team on an interesting long-term project.

Learning doesn't mean that you should always jump from one new hot thing to the next. That gets tiring really quick and it's not something that you can keep up for your whole career. Focus on the fundamentals of programming instead. Learn concepts, principles and patterns that can be applied across many languages and frameworks. These skills will stay with you over the years, while learning framework X is only useful until framework Y takes over its marketshare.

I would love to hear from you (patrick@louys.ch). Please let me know what you think about the book, what you got out of it and how it helped you. Let me know what I could have done better and which parts you didn't like.

Until next time. Cheers.