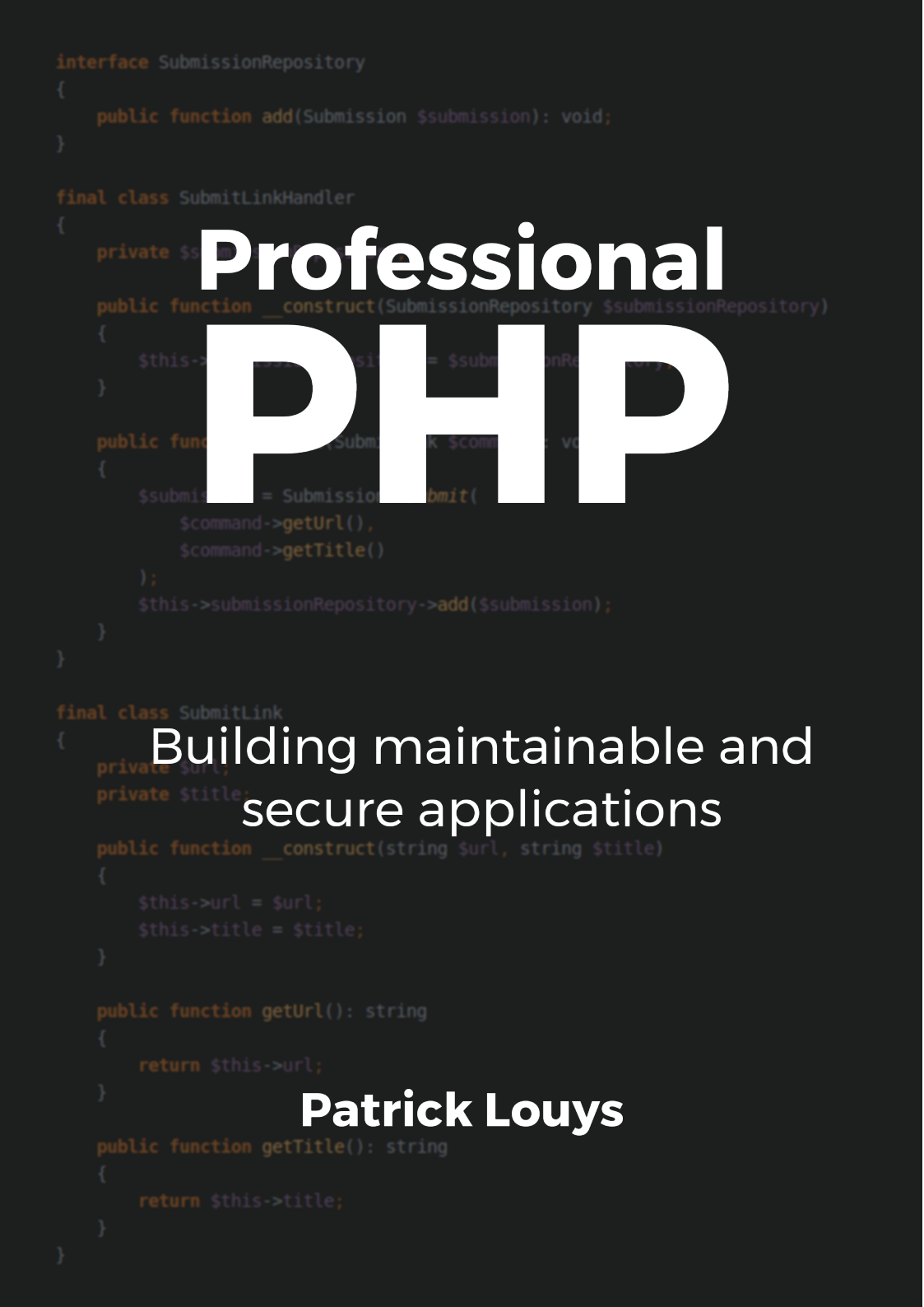


Professional PHP

Building maintainable and secure applications

Patrick Louys

Book Preview

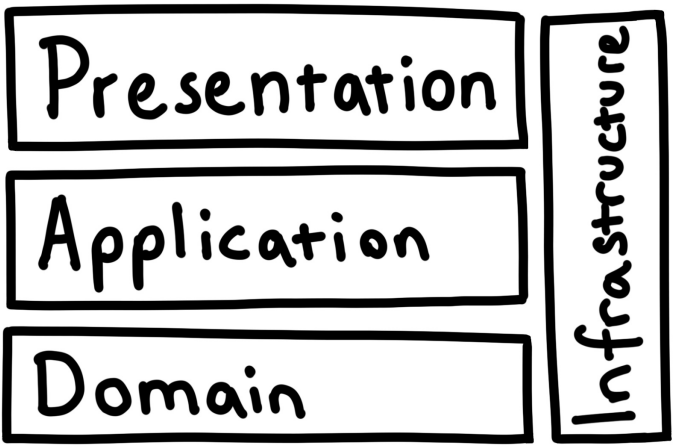
This is a sample chapter of “Professional PHP - Building maintainable and secure applications”. The book starts with a few theory chapters and after that it is structured as a tutorial. The more advanced programming concepts are covered as part of the tutorial.

I picked a random chapter from the tutorial to give you a sneak peak into the book. Without the preceding chapters, it lacks a bit of the surrounding context, but I hope that the sample chapter will give you a rough idea about what to expect from the book.

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.

If you want to read more

I hope that you enjoyed this sample chapter. The full book is available on my website at http://patricklouys.com/professional-php.

Cheers,
Patrick