Recipes for Decoupling

Matthias Noback

Recipes for Decoupling

Matthias Noback

This book is for sale at http://leanpub.com/recipes-for-decoupling

This version was published on 2022-06-29

Also By Matthias Noback

Contents

Introduction

When you're writing a software application you'll need many, many things. Sometimes it's quite disappointing. Just look at how many packages need to be installed only to run a very basic web application with a framework like Symfony or Laravel, and a test framework like PHPUnit. Most projects will have fewer lines of code in their own `src` directory than in the `vendor/` directory.

If we don't install all those dependencies, we can still create an application. But we would have to "reinvent a lot of wheels". It would make development so inefficient, we would never be allowed to make another application. So we rightfully trade development speed against an increased dependency on other people's code.

Once we have decided on a certain framework, we experience the highest speed of development if we do everything "their way". We follow their documentation, community blog posts, video trainings, e-books, and conferences. We apply the best practices that are being spread by visible community members, and we always prefer so-called idiomatic use of the framework. There are many advantages to this approach: it will be easy to find answers on Stack Overflow ("How to do X with Laravel?"), you can add the framework to job descriptions ("5+ years of experience with Symfony"), you can even raise your team's standard by requesting everyone to join a certification program.

Coupling, Why is it Bad?

Going all-in on a framework also has downsides. It's commonly known as vendor lock-in. This term already has enough negative connotations to trigger some alarms. For almost 20 years now, I've worked on PHP projects that first benefited incredibly from strictly following the framework approach. In the end, they had become an unmaintainable mess and were sometimes considered total-loss.

Looking at the code of such projects, and I guess you also have experience with them, a lot of work would be needed to save them. You'd have to migrate to a different framework, a different ORM, a different test framework. All of this is very hard to do because the code is tangled up with all those frameworks. Business logic is not in plain sight, but hidden in controllers, hooks, SQL queries, foreign key constraints, migrations, even in templates. Tests are completely tied to the framework, and can't be run without it.

All code has the potential to become legacy code over time. Still, based on my experience with these projects, it's clear that tightly coupled code is more likely to become legacy code, and to be abandoned completely. Which is a waste of the expensive development effort that went into it. A part of the application (if not the whole application) still works and is relied upon by its users, and now it has to be rewritten. A rewrite is not guaranteed to be a successful process, and the user is likely to miss some features that weren't noticed by the developers because they were obscured by framework integration code.

How can we prevent this from happening in the future? I think we, as software developers *today*, should make an effort to decouple from our frameworks. All our domain logic should be in plain sight, and our tests should be executed with only the most basic test framework. When the time comes to migrate to a different framework, maybe even upgrade to the next major version, it should be an easy job, finished within the course of a few weeks. It should never endanger the life of the project itself.

Decoupling, How to Do it Efficiently?

The trick is to keep a safe distance from your framework, while using some of its powerful features, so you don't have to write them yourself. This can be achieved by *decoupling*, which has two components: you need to decouple your *code*, and you need to decouple your *approach* to software development from the approach advocated by the framework authors. You need to think about what your application has to offer to the user, then implement it with code that *leverages* the framework's powers, but only in a few places.

Complete decoupling is undesirable as it would lead to too much work. *Loose coupling* is what we're after. When you want to replace the thing you're coupled to, that shouldn't be too much work, and it should only involve a few local changes. In other words, the framework shouldn't be all over the place. If you manage to do so, an automated refactoring tool like Rector will help you transform all the old-style integration code to code that matches the expectations of the new framework.

Objection

Whenever I propose to decouple from a framework, there's always a common objection: "YAGNI" (You Ain't Gonna Need It). Although this is not really an argument by itself - it could benefit from a few extra words - I think it hints at a valid concern. Should we decouple, even if we don't know how long the project will live, or if we'll ever face a framework migration? It shouldn't be a surprise, but I think "yes". If we don't do it now, we're introducing *technical debt* in the project. Of course, we can take the shortcut now, and write the code a bit faster today, but we know that we'll have to do a lot of work later. As with any technical debt, we can accept it. However,

1. I'm not entirely sure that we will be so much faster. Yes, in the very beginning of the project we can quickly slap a prototype together, but soon we'll start running into framework limitations, quirks, and magic that we didn't understand correctly. Personally I've spent *a lot of time* debugging forms, validation, entity mapping, and templating issues, and I'm a certified Symfony developer (well, I was part of the first class, 11 years ago at SymfonyLive Paris, so maybe that doesn't count anymore).
2. I'm not entirely sure that everyone is aware of how much debt they introduce when relying on frameworks and ORMs, or that going all-in on a framework should even be considered technical debt. This is something you only realize when you look at one of those almost-dying projects.

I think I'm not the only one who thinks that decoupling deserves to be part of our daily work as a programmer. I think programmers have always done this, because I keep recognizing design patterns that seem to have been invented for the sake of decoupling. Consider patterns from the old "Gang of Four" book, like *Observer*, later evolved into *Event dispatcher*, but also *Adapter*, etc. Or from Domain-Driven Design, e.g. *Aggregate*, *Repository*, *Application service*, and so on. Domain-Driven Design itself seems to be an exercise in moving focus away from frameworks and databases, the things we developers like so much, and divert our attention to the business domain itself. The same goes for development approaches like BDD, which explicitly aim to decouple specifications written as scenarios from the underlying technology, to keep a clear focus on what we're doing.

With my writing I'm always looking to place myself as a developer in this long-standing tradition of software development that doesn't revolve around a specific language, nor a specific framework. How can we write software in a way that is timeless, focuses on the business domain that it tries to serve, and produce applications that are long-term maintainable?

What's Special About This Book?

In another book, Advanced Web Application Architecture[1], I've already described in detail how to create an application that primarily consists of a decoupled application core. It also

[1] https://matthiasnoback.nl/book/recipes-for-decoupling/link-registry/advanced-web-application-architecture-book

shows how you can wrap this core inside any framework or connect it to any ORM you like. That book presents one over-arching vision of what I consider a good approach for application design. It helps you do domain-oriented, test-first development.

In this book, “Recipes for Decoupling”, I’m taking a much more light-weight approach. The focus isn’t on design patterns or architecture. I will sometimes reference the relevant concepts or patterns, but only as a way to point out how a decoupling recipe is related to this larger culture of software design practices. Instead of spending much time exploring the theory, we’ll mostly focus on the practice of decoupling. We’ll consider various popular frameworks and libraries that are used in web applications, and how you can decouple from them. This is done using small refactoring steps, showing how you can improve the structure of your code, while preserving the existing behavior. Something that’s very important with legacy code, of course.

As a reader, you can pick specific topics that are of interest to you, or that pose a particular risk in your current project, like “ORMs”, or “mocking libraries”. The decoupling approaches shown in each chapter can be applied in isolation. You don’t have to decouple your entire application at once. You also don’t need to go all the way on each refactoring. Most chapters offer an incremental series of refactorings that allow you to turn the dial on the level of decoupling you need today. If you like, you can do some more work if it provides additional benefits in the area of maintainability or maybe testability. You decide when to stop.

How to Stay Decoupled?

A refactor a day keeps the doctor away, but once your code is decoupled, you’ll need to be very disciplined to keep it decoupled. For instance, if you don’t want the service container to be used outside of controller classes, you have to continuously be aware of this rule. Not just you, all the other team members need to remember that they’re not allowed to use the container anywhere else. This is a risk for your project. It’ll be difficult to maintain that discipline over a longer period of time. We may oversee a particular “violation” of this rule in a PR, and accidentally merge it into the `main` branch. This single mistake becomes a precedent and will be copied many times; apparently “this is allowed now”. When all the developers who cared about decoupling have left the team, we can be sure that in the end all the rules will be ignored.

I believe that this tendency towards deterioration of a code base can be overcome by automation. Many of us have already worked with a project that performs some kind of automated build for each commit that we push to a remote branch. The build often consists of installing all dependencies (and checking that none of them have security vulnerabilities), linting the PHP code to catch potential syntax errors, running the tests, and verifying that the coding standard has been applied. These tools catch a lot of potential issues before we release them to `main` or even deploy them to production. If there is an issue, the build turns "red" and merging or deploying isn't even possible.

With an automated build phase for a project, we have a way to *prevent* bad code from being merged. So if we can define our own decoupling rules as checks that run during the build, we have a way to prevent coupled code from being merged. We only need to find a relatively easy way to write those decoupling rules. Once we have a platform for this, we can specify rules and run them as part of the build, now and forever. This will prevent the developers from falling back into old habits, and will save our code from ever becoming coupled again.

While working on the Rector book[2]with Tomas Votruba, he told me how he deals with user-contributed PRs for the Rector project itself. He noticed that he had to point out the same issues again and again. This took a lot of time, and led to delayed merges, and some frustration on both sides. He started using PHPStan[3]in the project build to automatically report mistakes directly to the user, without human intervention.

PHPStan is a tool that statically analyses PHP code and points out potential problems related to incorrect types, argument counts, undefined methods, etc. It has a lot of built-in rules that can catch a lot of common programming mistakes before you ever run your code. On top of that, it also allows you to define your own rules. That's what Tomas did for the Rector project. It didn't only help its contributors; he was also able to worry less. If he'd forget about some rule, PHPStan would remind him of it.

Following Tomas' lead, I'm doing the same for this book. Whenever we discuss a refactoring that leads to decoupled code, we'll also look for ways to solidify the decoupling in the long run by writing a custom PHPStan rule. This is going to be much safer than relying on the personal discipline of every developer on the team.

Before we can do this, we should take a look at the process of writing a PHPStan rule. This involves a short introduction to PHPStan and how to write custom rules for it. You'll find this introduction in the first chapter.

In this book I adopt PHPStan because I'm familiar with it, and also because Rector is built on top of it. You can also use the popular alternative - Psalm[4]. Psalm can be extended with custom rules as well, and they won't be very different from the ones in this book.

[2]https://matthiasnoback.nl/book/recipes-for-decoupling/link-registry/rector-book
[3]https://matthiasnoback.nl/book/recipes-for-decoupling/link-registry/phpstan
[4]https://matthiasnoback.nl/book/recipes-for-decoupling/link-registry/psalm

Who Should Read This Book?

I hope this book will be relevant for any developer who has used a framework, has sometimes been bitten by its magic, or has been unable to upgrade to the next framework version. This book will be useful if you're looking for ways to make applications more maintainable in the long run. I expect only some experience with PHP and one of its currently popular frameworks.

Overview of the Contents

We start with a chapter that introduces PHPStan and how you can create custom rules for it. If you already use PHPStan and have some experience writing rules for it, feel free to skip this chapter.

Chapter 2 introduces common coupling issues related to web frameworks. We consider what would be needed to migrate a controller from a Symfony application to a Mezzio application, and use this process to gain some understanding about any potential framework migration. This gives us decoupling rules for dealing with request and response objects, service dependencies, and template rendering.

Chapter 3 covers similar issues, but for CLI frameworks. We take a look at a Symfony-based console command and find out how to decouple the business logic of this command from the terminal-based input and output objects.

Chapter 4 looks at form validation and how Laravel applications deal with this. We look for a way out of framework coupling by shifting our focus away from form validation. We introduce value objects and try to enforce constraints at the level of the model itself.

Chapter 5 zooms in on a type of library that often takes over a big part of our application and might therefore be considered a framework itself: the ORM. How to decouple from it in such a way that we don't lose all the benefits? Part of the solution is in separating the "write" from the "read" activities of the ORM. At some point we realize that maybe we don't need an ORM at all. This chapter has examples based on Laravel's Eloquent as well as Doctrine ORM, because they don't have much in common and both require a different approach to decoupling.

Just as tests are usually an after-thought (sadly so), the final chapter of this book dives into test frameworks and mocking libraries and describes some options for decoupling from them.

By means of a conclusion we'll look back at all the refactoring and decoupling activities we did in this book, and we try to extract some common aspects. Rephrased as decoupling strategies they should help you decouple from things that don't exist yet, or that we otherwise didn't cover in this book.

About the Author

Matthias Noback has been building web applications with PHP since 2003. He is the author of the *Object Design Style Guide* and *Advanced Web Application Architecture*. He's also a regular blogger, speaker and trainer[5]. In his spare time he plays the violin and builds wooden 17th-century ship models.

Changelog

If you've bought an early version of this book, you can download all future updates for free. After each update I'll describe what has been added or changed.

10 May 2022

Initial release.

- Added the *Introduction*
- Added Chapter 1, *Creating custom rules for PHPStan*

17 May 2022

- Added Chapter 2, *Web frameworks*

- Fixed the output of a PHPUnit run in Chapter 1 that was expected to show `echo`-ed node types (thanks for reporting, Christopher L Bray!).
- Changed the returned rule in `NoErrorSilencingRuleTest` to `NoErrorSilencingRule` (thanks for reporting, Quentin Delcourt!)

[5] https://matthiasnoback.nl/book/recipes-for-decoupling/link-registry/matthias-noback

26 May 2022

- Added Chapter 3, *CLI frameworks*
- Fixed some broken sentences, layout issues, and updated the link registry (thanks for reporting, Amir Ziapoor!)

31 May 2022

- Added Chapter 4, *Form validation*

7 June 2022

- Added Chapter 5, *ORMs and the database*

15 June 2022

- Added the missing sections about PHPStan rules to Chapter 5:
  - PHPStan rule: disallow auto-incrementing model IDs
  - PHPStan rule: only allow calls to fromDatabaseRecord() from repository classes
- Added Chapter 6, *Test frameworks*

- Processed feedback about Chapter 1 (thanks, Paul Rijke!)
  - Added a hint about running `composer dump-autoload`
  - Improved the comment about using `customRulesetUsed` instead of `level`
  - Added a hint about running only the rule tests
  - Improved the section about configuring a suffix

29 June 2022

- Added the Conclusion

1. Creating Custom Rules for PHPStan

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Introduction

As mentioned in the introduction, we're going to use PHPStan[1]in this book to verify that after decoupling our code it will remain decoupled forever. In order to do so we first need to install PHPStan, and configure it to analyze our project files, then practice a bit with writing custom rules. I recommend you to follow along with this chapter, implementing the examples yourself inside one of your projects.

Analyzing Code with PHPStan

PHPStan is a tool that statically analyzes your PHP code. This means it doesn't run your code, but only "reads" it. It then performs a number of checks on your code, which are called "rules". For instance, when your code calls some method, it checks that you supply the right number of arguments, and that the arguments match the expected types. If it finds any issues, PHPStan will report this as output in your terminal.

You'll find all the details about installing and configuring PHPStan on its website[2], but the basic steps are:

1. Install PHPStan as a Composer package in your project: `composer require --dev phpstan/phpstan`
2. Create a `phpstan.neon` configuration file in the root of your project
3. Run PHPStan: `vendor/bin/phpstan`

A basic `phpstan.neon` configuration file looks like this:

[1]https://matthiasnoback.nl/book/recipes-for-decoupling/link-registry/phpstan
[2]https://matthiasnoback.nl/book/recipes-for-decoupling/link-registry/phpstan

```
parameters:
    level: max
    paths:
        - src
        - tests
```

I prefer using the `max` level, so PHPStan can operate at maximum power. For existing code, this may be quite problematic - check out the documentation section about the Baseline[3]to find out how to deal with this.

We need to provide the paths that PHPStan should analyze (in this example `src` and `tests`). Every PHP file in the project, except `vendor` code should be analyzed, or we'll miss out on valuable feedback.

When we run `vendor/bin/phpstan` we'll get output similar to this:

```
------ ------------------------------------------------------------
 Line   src/example.php
------ ------------------------------------------------------------
 5      Parameter #1 $json of function json_decode expects string,
        string|false given.
------ ------------------------------------------------------------


[ERROR] Found 1 error
```

PHPStan lists all errors, the files in which they occur, and their line number. Every error needs to be fixed, in one way or another. In this example we get an error about the first argument provided to the `json_decode()` function call. It expects a string, but we pass `string|false` to it:

[3]https://matthiasnoback.nl/book/recipes-for-decoupling/link-registry/phpstan-baseline

```php
<?php

declare(strict_types=1);

json_decode(file_get_contents('some.json'));
```

Looking at the documentation of `file_get_contents()`[4], it turns out that this function returns a `string`, which is the contents of the requested file, or `false` if the file doesn't exist. In our code we need to deal with both cases. One way to do so is to throw an exception in case `file_get_contents()` returns `false`:

```php
<?php

declare(strict_types=1);

$filename = 'some.json';

$contents = file_get_contents($filename);

if ($contents === false) {
    throw new RuntimeException('Could not read file ' . $filename);
}

json_decode($contents);
```

PHPStan is smart enough to understand that below the `throw` statement it can be certain that `$contents` is no longer possibly `false`. According to the union type `string|false`, what remains is that `$contents` is a `string`, and it will be safe to pass that to `json_decode()`. So when we run `vendor/bin/phpstan` again, it shows no errors:

```
[OK] No errors
```

To ensure that fixing PHPStan-reported errors isn't an optional activity for developers, we have to commit `phpstan.neon` and add the `vendor/bin/phpstan` command to the project's build script.

[4]https://matthiasnoback.nl/book/recipes-for-decoupling/link-registry/php-file-get-contents

I recommend letting PHPStan analyze your entire project, and setting `level` to an appropriate value (see Rule Levels[5]). However, you're not strictly required to use any of PHPStan's built-in rules or rule levels. You can also instruct PHPStan to only run your custom rules against the code base. If that's what you want for your project, you can remove the `level` parameter in `phpstan.neon` and instead add `customRulesetUsed: true`.

Catching Specific Node Types

Now that PHPStan is ready to be used in your project, let's start by creating a custom rule. As a warm-up exercise, let's report any code that uses the error silencing operator `@` to suppress errors raised by PHP functions like `file_get_contents()`, as is done in this example:

src/error-silencing.php

```
<?php

declare(strict_types=1);

@file_get_contents('not-found.txt');
```

Instead of silencing errors, we'd like full exposure about everything that goes wrong in our code. We want to fix issues instead of ignoring them. So let's make a custom PHPStan rule that forbids the use of `@` anywhere in the project.

A PHPStan rule is a class that implements the `PHPStan\Rules\Rule` interface. The class should be auto-loadable. I prefer to keep rule classes outside the normal `src` folder though, so I have a dedicated line for PHPStan rules in the `autoload-dev` section of my `composer.json` file:

[5]https://matthiasnoback.nl/book/recipes-for-decoupling/link-registry/phpstan-rule-levels

```
{
    "require-dev": {
        "phpstan/phpstan": "^1.5"
    },
    "autoload-dev": {
        "psr-4": {
            "Utils\\PHPStan\\": "utils/PHPStan/src"
        }
    }
}
```

When you make a change to the `autoload` or `autoload-dev` section of your `composer.json` file, don't forget to run `composer dump-autoload` afterwards.

Our first rule class is going to be called `NoErrorSilencingRule`:

utils/PHPStan/src/NoErrorSilencingRule.php

```
<?php

declare(strict_types=1);

namespace Utils\PHPStan;

use PhpParser\Node;
use PHPStan\Analyser\Scope;
use PHPStan\Rules\Rule;

final class NoErrorSilencingRule implements Rule
{
    public function getNodeType(): string
    {
        return Node::class;
    }
    public function processNode(Node $node, Scope $scope): array
    {
        return [];
    }
}
```

For now, I've only added a minimal implementation of the interface. It doesn't do anything useful yet, but it also doesn't break anything. To make PHPStan aware of the new rule class, we should add it to `phpstan.neon`:

```
parameters:
    level: max
    paths:
        - src
rules:
    - Utils\PHPStan\NoErrorSilencingRule
```

Run PHPStan to check that it can find the new rule. If it couldn't find the rule class, it will show an error.

While practicing with custom rules, it will be a good idea to not analyze the full project, but only a single file that has a representative example of something we want to trigger an error for. You can do this by running `vendor/bin/phpstan analyze [file-path-to-analyze]` (note the additional `analyze`).

Later on we're also going to write tests for our custom PHPStan rules, which accomplishes the same thing, and is a much better alternative anyway.

We've set up a new rule class, now it's time to implement it. Note that the rule has two methods: `getNodeType()` and `processNode()`. These work in a similar fashion as your average event dispatcher does: you register the type of event you're interested in, and you will be notified when such an event occurs. For a rule you register the "node type" you're interested in, and when PHPStan encounters a *node* of that type, it will call `processNode()`. But what's a node?

Nodes are elements of the code that can be recognized as meaningful units. For example, a class is represented as a single node, and all its properties and methods are also nodes. Within every class method, each statement is recognized as a node, and also each expression. Nodes can contain other nodes, or lists of nodes, and so on. Each PHP file can be interpreted as one big tree of nodes.

PHP itself builds such a tree of nodes when it loads one of your `.php` files and runs it. It first needs to understand what's inside the file and if it makes any sense. In order to do so it *parses* the `.php` code, and as a results makes a node tree of it, which it can then turn into so-called byte code, which is executable by PHP's virtual machine. Instead of *running* code based on the node tree, which is called Abstract Syntax Tree (AST), we could also *analyze* the tree and spot potential errors, as PHPStan does.

PHPStan uses the PHP-Parser library[6]to parse your project's PHP code and create an AST for each file. It then walks over the tree of nodes, asking each rule: are you interested in this node (`getNodeType()`)? If so, here you have it (`processNode()`); please let me know if you want to report any errors about it!

To decide if the rule should trigger an error, the rule usually has to take a closer look at the node. Nodes themselves are simple PHP objects. The classes of those objects can be found in the PHP-Parser library. When you install PHPStan as a Composer package, all of its code is inside a single `.phar` file (`vendor/phpstan/phpstan/phpstan.phar`). This file also contains PHPStan's `vendor` dependencies, including the PHP-Parser library. If you use an IDE like PhpStorm you can inspect the contents of the `.phar` file and find this library. Look for `vendor/nikic/php-parser/lib/PhpParser/Node` and you'll find every possible node class.

When you're creating a new rule, you'll have to decide what type of node you want to target. Because we don't know much at this point, we pick the most general node type - the `PhpParser\Node` interface that all node classes implement, and return its name in `getNodeType()`. Since every node that PHPStan encounters implements this interface, it will call `processNode()` for every node in every PHP file we analyze. We can use this to gain some understanding about what kind of nodes we should be looking for. Let's print the type (i.e. class) of each node on screen for now:

[6]https://matthiasnoback.nl/book/recipes-for-decoupling/link-registry/php-parser

```php
<?php

declare(strict_types=1);

namespace Utils\PHPStan;

use PhpParser\Node;
use PHPStan\Analyser\Scope;
use PHPStan\Rules\Rule;

final class NoErrorSilencingRule implements Rule
{
    public function getNodeType(): string
    {
        return Node::class;
    }

    public function processNode(Node $node, Scope $scope): array
    {
        echo $node::class . "\n";

        return [];
    }
}
```

With this rule enabled, we shouldn't run PHPStan on our entire project, since that would give us too much output. Instead, we should analyze just a single sample file that uses a silencing error, `src/error-silencing.php`, which contains just the following code:

src/error-silencing.php

```php
<?php

declare(strict_types=1);

@file_get_contents('not-found.txt');
```

To allow direct output via `echo` we have to add the `--debug` option when running PHPStan. The full command is:

```
vendor/bin/phpstan analyze --debug src/error-silencing.php
```

The output:

```
src/error-silencing.php
PHPStan\Node\FileNode
PhpParser\Node\Stmt\Declare_
PhpParser\Node\Stmt\DeclareDeclare
PhpParser\Node\Scalar\LNumber
PhpParser\Node\Stmt\Expression
PhpParser\Node\Expr\ErrorSuppress
PhpParser\Node\Expr\FuncCall
PhpParser\Node\Arg
PhpParser\Node\Scalar\String_
--- consumed 6 MB, total 62 MB, took 0.09 s

 [OK] No errors
```

The result is a list of the *types* of nodes that were found in our `error-silencing.php` script. Again, each type is actually a class, and you can open each of them in your IDE to find out more about their internal structure. Comparing the list of node types to the code in `error-silencing.php` we can relate every node to a piece of code. E.g. we see at the end of the list a `FuncCall` with a single `Arg` which contains a `String_` node. These are all the nodes for the expression `file_get_contents('not-found.txt')`.

Right before the `FuncCall` node we notice the `ErrorSuppress` node. That's the one we're looking for, it's the `@` in front of the function call. PHPStan should trigger an error for any `ErrorSupress` node that PHPStan finds in our code. So we should change the `getNodeType()` function of our rule class to return the `ErrorSupress` node class instead:

```php
use PhpParser\Node\Expr\ErrorSuppress;

final class NoErrorSilencingRule implements Rule
{
    public function getNodeType(): string
    {
        return ErrorSuppress::class;
    }

    // ...
}
```

Next, we need to report an error for the line where we encountered an `ErrorSuppress` node. PHPStan offers a convenient error builder for this. We only have to provide an error message, and PHPStan will add the remaining information:

```php
use PHPStan\Rules\RuleErrorBuilder;

final class NoErrorSilencingRule implements Rule
{
    // ...

    public function processNode(Node $node, Scope $scope): array
    {
        return [
            RuleErrorBuilder::message(
                'You should not use the silencing operator (@)'
            )->build(),
        ];
    }
}
```

Running PHPStan again, we should now see that it reports an error for our `error-silencing.php` file:

```
------ -----------------------------------------------
 Line   error-silencing.php
------ -----------------------------------------------
 5      You should not use the silencing operator (@)
------ -----------------------------------------------


[ERROR] Found 1 error
```

Great, we can now run PHPStan on the entire project and find all the places that still use the error silencing operator! PHPStan sees everything, and will keep watching forever. Whenever someone makes the "mistake" to use `@` again, PHPStan will warn them about it.

Adding Automated Tests for a PHPStan Rule

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Deriving Types from the Current Scope

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Putting a Node in Context

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Generalizing a Rule

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Conclusion

2. Web Frameworks

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Introduction

For a long time my favorite framework was Symfony[1], until the day I started liking Laminas Mezzio[2]more. There have been many frameworks, and they have been the favorites of many developers, but occasionally one of them gets abandoned by its maintainers. And so the quest for the next “best” framework begins. Something similar that many of us have experienced: the maintainers were tired of version X, so they rewrote the whole thing, and it was released as version Y. Why indeed! The result was a lot of work.

My experience in web development has proven to me that we need to be ready to get rid of a framework. We should also be ready to upgrade our current framework to the next major version, which can be as drastic as switching to a completely different framework. In particular if we’ve coupled our code to the framework, in all the possible documented (and undocumented) ways.

It may be a cliché by now, but we need to keep the framework at a safe distance. If we do so, I’m sure we can save ourselves a lot of expensive development work, trying to get a project up-to-date again. In practice, this “safe distance” means we should not use any framework facilities in some parts of our code, while we allow other parts to make optimal use of the framework’s special abilities. This will give us the benefits of working with a framework (mostly that there’s no need to reinvent certain wheels), while giving us the flexibility to upgrade the framework without too much trouble, or even to migrate to a different framework.

The million-dollar question is: what are those parts, and where should the cut be? To find the answer, let’s first consider what web frameworks have offered us in the past, but also: what will they keep doing for us in the future? What’s the *common denominator*?

What I think a web framework will always have to do, is:

[1]https://matthiasnoback.nl/book/recipes-for-decoupling/link-registry/symfony
[2]https://matthiasnoback.nl/book/recipes-for-decoupling/link-registry/mezzio

- Match the URI of the HTTP request to one of our controllers based on some pattern (*routing*).
- Instantiate and invoke the matched controller, providing the request itself as input.
- Turn the response returned from the controller into a proper HTTP response that a browser can show to the user.
- Catch exceptions that occur inside the controller and turn them into error responses.

Controllers play an important role here: they are the place where, as developers, we can finally do something. But what we do is often quite repetitive: we load data from the database, apply the changes to it that were submitted using a form, we save the resulting data, and finally we render a nice page for the user. Framework authors try to make our controllers simpler by removing some of this duplication, and providing convenient shortcuts. This is often achieved with magical features that are implemented by running some code before and after our controller code. For instance, the framework may magically do the following things for us:

- Validate request data
- Load an entity based on a matched URI parameter and pass it as a controller argument
- Provide the logged-in user object as a controller argument
- Render a template based on an annotation

These features count as *magical* because for the reader it's not clear what happens, when, and based on what input.

Controllers

Show How the Response is Created

Only Use Constructor Injection for Dependencies

Make Every Step Explicit

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Controllers Have No Parent Class

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Every Action Has its Own Controller Class

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Should We Use an HTTP Abstraction Library?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Rules for Decoupled Controllers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Forbidden Parent Classes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Allowing Only Parameters of a Certain Type

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Enforcing Return Types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

One Action Per Controller

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Views

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Pass All the Data That the Template Needs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Don’t Pass Objects That Don’t Belong in a Template

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Rules for Decoupled Views

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Don’t Use Certain Global Variables in a Template

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Don’t Use Certain Functions in a Template

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Don’t Pass Entities to a Template

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

3. CLI Frameworks

Introduction

Web frameworks like Symfony and Laravel ship with a sub-framework that helps the developer add command-line-based tasks to their projects. Usually these tasks are implemented as "commands", which are classes that extend from some kind of abstract `Command` class. Out of the box a project already has some commands available, e.g. for migrating the database schema, or for setting up a new controller. When developers start adding their own commands to the project, those commands will be in one way or another coupled to the command-line-interface (CLI) framework of their project.

Decoupling from a CLI framework isn't always necessary, but it will certainly give you an advantage when the time comes to upgrade or switch. In that case, there will be several areas where work is needed:

1. In the code that *defines* the available command-line options and arguments supported by the command
2. In the code that *reads* the user-provided options and arguments based on these definitions
3. In the code that deals with the output streams (`stdout` and `stderr`)

Each framework will offer its own API for that, so these are the areas where we'll have to rewrite existing code. To make this easier for us, extracting input and producing output is something that should only happen in a few places in our codebase. If this is the case, there will be a large chunk of the code that doesn't have to be adapted when switching to a different CLI framework.

Input

Collect Input First

Jump to a Service

Output

Using an Observer for Showing Output

Generalizing the Solution with Event Dispatching

Turn the Command Class Itself Into an Event Subscriber

Controllers Should Call Framework-agnostic Services

Rules for Decoupling

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Use InputInterface and OutputInterface Only in Command Classes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

4. Form Validation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Introduction

Forms and validation go hand-in-hand in most (web) frameworks. A form is defined in HTML, its data encoded and submitted by the browser, as the body of an HTTP request. The framework usually converts the data into an associative array for us, so we can take from it what we need inside the controller. There are a number of things we always have to consider:

- We expect certain keys to exist, but maybe they don't, because the form didn't show a particular field, the browser didn't submit the data, or the user has tampered with the request body.
- We expect certain values to be provided by the user, and expect these values to be within certain lengths, ranges, etc. The user may not have filled in the form correctly, or again, they may have tampered with the request body.

So before we can use the submitted data we always have to be careful about verifying its correctness. We have to make sure that we don't process invalid data, or even save it into our database. This means we should stop the execution of our business logic as soon as some value is missing or looks wrong. The only way to reliably stop execution is to throw an exception, which will let us jump out of the current method immediately. However, if we stop execution at the first sign of a problem, we'll have a hard time communicating with the user. We can't just display any exception message to the user since that poses a security risk.

By relying solely on exceptions for validation, we may also get the user into an endless cycle of trial and error. They fill out the form, submit it, fix the first reported error, submit it again, fix the next reported error, and so on. This isn't a great user experience. Instead, we want to show nice and friendly (often localized or translated) error messages below each form field for which the user didn't provide a correct value.

Based on these considerations we may conclude that the two major reasons for performing validation on user-submitted data are:

1. To offer protection against invalid data being processed by our application (i.e. guarantee data consistency)
2. To help the user provide valid data (i.e. improve the user experience)

Form Validation

Protecting Data Inside the Model

Delegating Protection to Value Objects

Removing Duplicate Validation Logic

Defining an Explicit Shape for the Input Data

Rules for Decoupling

Don’t Pass a Single Array of Data to create()

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Enforcing the Use of Closure-based Form Validation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

5. ORMs and the Database

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Introduction

In the early 2000s we wrote all our SQL queries by hand. This was eventually recognized as a security hazard. We learned about using prepared statements. PHP introduced PDO, as a way to abstract away some database vendor-specific things that we did in our code. It turned out that PDO definitely wasn't a complete database abstraction library. There were still many aspects of abstracting the database that Doctrine DBAL (Database Abstraction Layer) was able to add. Meanwhile, programmers wanted to move to a more object-oriented style of programming. We needed a way to write their domain logic in classes, but still save the state of domain objects in database tables. In other words, we wanted to *map* objects to records in a relational database. Hence, the name ORM - Object-Relational Mapper. The ORM abstracts the work that needs to be done to "save an object to the database". But that's not all, the ORM also needs to retrieve records and reconstitute the original objects. Nowadays, ORMs also handle schema migrations for you.

Since ORMs do so many things, we often don't realize how much our domain model is coupled to it anymore. We only notice this when the need arises to migrate to a different ORM, because the old one uses out-dated programming techniques and design patterns, or is simply no longer maintained. We also don't realize how much the ORM itself is still coupled to the relational database. It may abstract from a specific database vendor, but it has not abstracted from persistence to tables. Using the same ORM, it's impossible to start saving object as, for instance, JSON files on disk, or to retrieve them from ElasticSearch. This shows how a domain model is often tightly coupled to the ORM and the database itself.

To find out how we can decouple from an ORM, we should indeed consider what happens if we remove it. Then we should consider how much work is needed if we want to switch from a relational to, say, a document database.

When scanning a code base for places where the model is coupled to the ORM, we'll find the following use cases for the ORM:

1. In some parts of the application we create an object and then save it to the database. Sometimes we load an existing object, modify it, and again, save it to the database.
2. In other parts we load one or more objects from the database and show parts of the object on some kind of view (e.g. an HTML page, a JSON response, an email body). We don't need to modify the object here, nor do we want to save anything to the database. We're only interested in retrieving and presenting information.

We can summarize these two basic use cases in a more abstract way, as the "needs" of our application. We need to:

1. *Persist* a domain model object, and to retrieve that object by its ID, so we can make further changes, persist it again, etc.
2. *Query* some data for the user to view.

I've made these descriptions somewhat abstract on purpose: they don't say anything about the source of the data itself. Does it come from a database? And if so, a relational, or a document database? It doesn't matter. Our application just needs these abilities, and we have to implement them somehow. Defining our needs as abstractions is a way to decouple from their underlying technical implementations. We first look at how to properly abstract our code so that the first need is addressed in a decoupled way. For this we'll need a repository, application-generated identifiers, and non-cascading updates.

Repository: an Abstraction for Persistence

Trying an Alternative Implementation

Application-generated IDs

PHPStan Rule: Disallow Auto-incrementing Model IDs

Defining Our Own Object API

Custom Mapping Code

PHPStan Rule: Only Allow Calls to fromDatabaseRecord() from Repository Classes

No Magic Persistence of Related Objects

Using Aggregate Design Rules

Limiting Changes to One Entity

Introducing View Models

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Provide Read Models When the Framework Needs Your Data

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Use Domain Events Instead of Persistence Events

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

PHPStan Rule: Forbidden Parameter Types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

6. Test Frameworks

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Introduction

So far I've used many frameworks for testing. Some that come to mind: SimpleTest, Lime, PHPUnit, Codeception, PHPSpec, Behat… Each has its own unique selling points and is probably easy to use, but do we really need so many test frameworks? Sure, every new framework introduces new ways of writing and structuring test code, but if it's new and better ways that we're looking for, one day we'll certainly want to migrate do a different framework that offers even more revolutionary ways to write test code.

Many projects suffer from this urge to use multiple test frameworks over the years. In the beginning a project has only one framework for writing tests (e.g. PHPUnit with a setup for browser-like testing), then it adds another one (e.g. Behat for high-level scenario tests), and maybe even another one (e.g. PHPSpec for class tests). Eventually, one of these frameworks is going to become obsolete. The maintainer of CoolTest 2000 is going to give up on the project. They liked inventing a new framework, and indeed, it's probably fascinating to do so, but at some point it becomes boring. Perhaps it wasn't the maintainer that gave up, but the development team who didn't like the framework anymore. Then the team has to adopt the next framework, which is going to do things in a completely different way.

The same is true for mocking libraries. If you use any number of mocks in your tests then you end up being invested in one particular mocking library, like Mockery, or PHPUnit mocks, or Prophecy. Unfortunately, when a new tool comes around, someone in the team wants to try it, or the team decides to switch but never takes the time to complete the migration. Now the project has two or three different mocking tools in use. All their related packages need to be updated regularly, and for (new) developers it's unclear what mocking tool they should use.

My hypothesis is that a project ends up with different test frameworks and mocking libraries because:

- Writing tests can feel cumbersome, we want it to go faster, and be more fun, so we install new and cool tools that promise these things.
- Tests aren't production code, so it feels like we can freely experiment with different approaches.
- A new tool is quickly installed and the enthusiasm is contagious, but no project manager is ever going to "give us time" to upgrade the old approach to the new one.

Use Only the Most Basic Features of a Test Framework

Declare Test Dependencies Explicitly

PHPStan Rule: Don't Allow PHPUnit Extensions

PHPStan Rule: Don't Allow Class-level Set-up and Tear-down Functions

Assertions

Write Tests in Plain Code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Handwritten Test Doubles

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

PHPStan Rule: Don’t Generate Mocks

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/recipes-for-decoupling.

7. Conclusion

8. The End of the Book