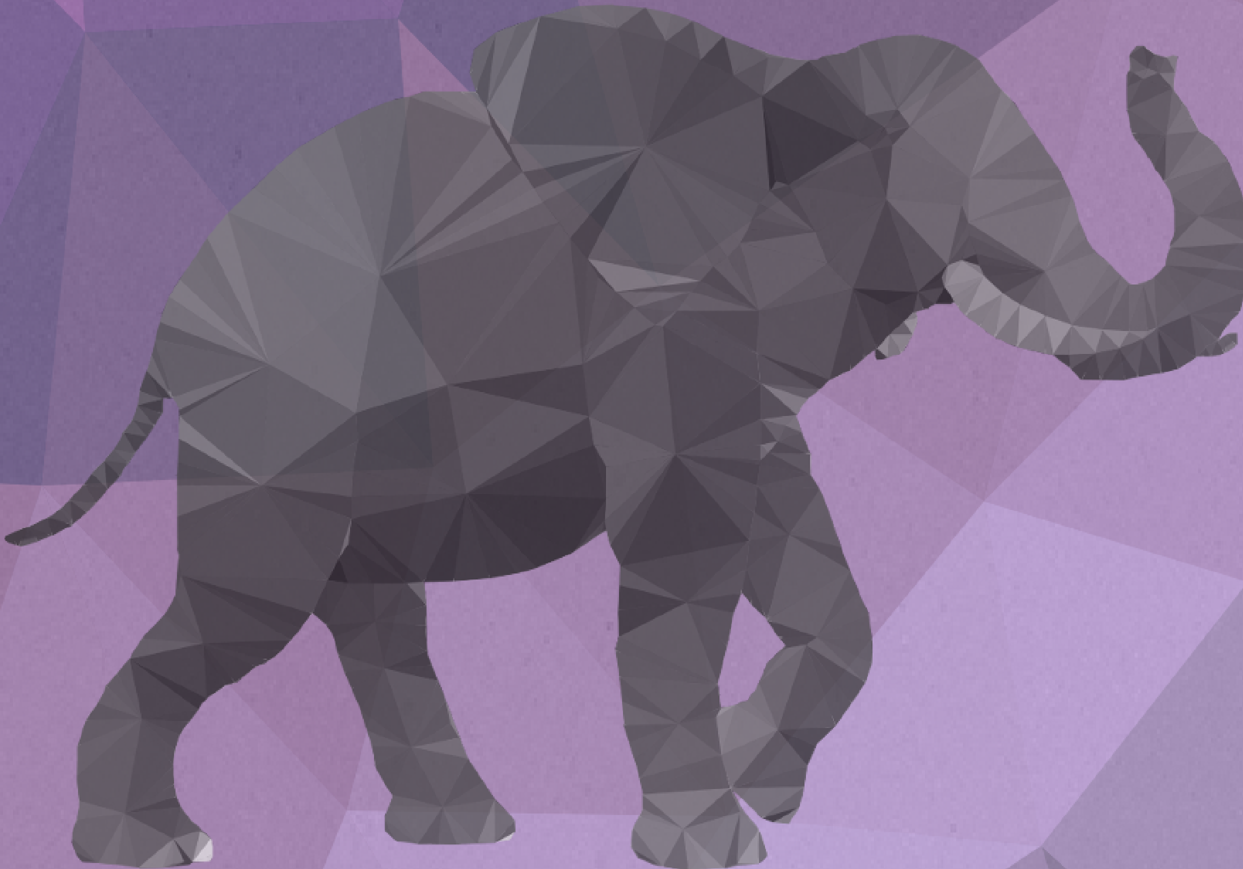
THE CLEAN ARCHITECTURE IN PHP



KRISTOPHER WILSON

The Clean Architecture in PHP

Kristopher Wilson

This book is for sale at http://leanpub.com/cleanphp

This version was published on 2015-04-24

Dedication

*First and foremost, I dedicate this book to my wife, **Ashley**. Thank you for allowing me to spend so much time staring at millions of dots on a screen.*

*Secondly, to **my parents**, who worked so hard to make sure their children had everything they needed and wanted, and for encouraging me to follow my dreams, however odd they may have been.*

Contents

Introduction

Figuring out how to architect a brand new application is a big deal. Doing it the wrong way can lead to a huge headache later. Testing can become hard – or maybe even impossible – and refactoring is an absolute nightmare.

While the methods outlined in this book aren't the only way to go about developing an application, they do provide a framework for developing applications that are:

1. Testable
2. Refactorable
3. Easy to work with
4. Easy to maintain

This book is for anyone wanting to build a medium to large sized application that must be around for a long time and/or be easily enhanced in the future. The methods outlined in this book aren't meant for all applications, and they might be downright over kill for some.

If your application is small, or an unproven, new product, it might be best to just get it out the door as fast as possible. If it grows, or becomes successful, later applying these principles may be a good idea to create a solid, long lasting product.

The principles outlined in this book involve a learning curve. Writing code this way will slow a developer down until the methods become familiar to them.

Organization

This book begins by discussing common problems with PHP code and why having good, solid, clean code is important to the success and longevity of an application. From there, we move on to discussing some principles and design patterns that allow us to solve problems with poor code. Using these concepts, we'll then discuss the Clean Architecture and how it further helps solve problems with bad code.

Finally, in the second half of the book, we dive into some real code and build an application following this architecture. When we're done with our case study application, we'll start swapping out components, libraries, and frameworks with new ones to prove out the principles of the architecture.

The Author

My name is Kristopher Wilson. I've been developing in PHP since around 2000. That sounds impressive on the surface, but most of those years involved writing truly terrible code. I would have benefited greatly from a book like this that outlines the principles of how to cleanly organize code.

I've done it all, from simple websites to e-commerce systems and bulletin boards. Mostly, I've concentrated on working on ERP (Enterprise Resource Planning) systems and OSS (Operational Support Systems) – software that runs the entire back office of large organizations, from manufacturing to telecommunications. I even wrote my own framework once. It was terrible, but that's another story.

I live in Grand Rapids, Michigan with my wife and our four cats (our application to become a registered zoo is still pending). I'm one of the founders of the Grand Rapids PHP Developers (GrPhpDev) group and am highly involved with organizing, teaching, and learning from the local community.

A Word about Coding Style

I strongly prefer and suggest the use of PSR-2 coding standards[1]. As a community, it makes it much easier to evaluate and contribute to one another's code bases if the dialect is the same. I also strongly suggest the use of DocBlocks and helpful comments on classes and methods.

However, for brevity, the code examples in this book make a few deviations from PSR-2 standards, namely involving bracket placement, and don't include many DocBlocks. If this is too jarring for the PSR-2 and DocBlock fan, like myself, I humbly apologize.

[1]https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-2-coding-style-guide.md

The Problem With Code

Writing code is easy. So easy that there are literally hundreds of books[2] claiming they can teach you to do it in two weeks, ten days, or even twenty-four hours. That makes it sound really easy! The internet is littered with articles on how to do it. It seems like everyone is doing it and blogging about it.

Here's a fun question: would you trust a surgeon or even dentist who learned their profession from a couple of books that taught them how to operate in two weeks? Admittedly, writing code is nothing like opening up and fixing the human body, but as developers, we do deal with a lot of abstract concepts. Things that only exist as a collection of 1s and 0s. So much so that I'd definitely want an experienced, knowledgeable developer working on my project.

The problem with code is that good code, code that serves it's purpose, has little or no defects, can survive and perform it's purpose for a long time, and is easy to change, is quite difficult to accomplish.

[2]http://norvig.com/21-days.html

Writing Good Code is Hard

> If it were easy, everyone would be doing it
>
> *-Somebody, somewhere in history*

Writing code is hard. Well, scratch that: writing code is easy. It's so easy everyone is doing it. Let me start this chapter over.

> If it were easy to be good at it, everyone would be good at it
>
> *-Me, I suppose*

Writing code, especially in PHP, but in many other languages as well, is incredibly easy. Think about the barrier to entry: all you have to do is go download PHP on your Windows machine, type:

```
php -S localhost:1337
```

Now all of your PHP code in that directory is suddenly available for you via a browser. Hitting the ground running developing PHP is easy. On Linux, it's even easier. Install with your distributions package manager and type the same command as above. You don't even have to download a zip file, extract it, worry about getting it in your path, etc.

Not only is getting a server up easy, but actually learning how to get things accomplished in PHP is incredibly easy. Just search the web for "PHP." Go ahead, I'll wait. From Google, I got 2,800,000,000 results. The internet is literally littered with articles, tutorials, and source code relating to PHP.

I chose my words very carefully. The internet is literally *littered* with PHP.

Writing Bad Code is Easy

Since PHP is incredibly easy to get started in, it makes sense that eventually it would gather a large following of developers. The good, the bad, and the ugly. PHP has been around since 1994 or so, as has been gathering developers ever since. At the time of this writing, that's twenty years worth of code being written in PHP.

Since then, an absolute horde of poorly written PHP has shown up on the web, in the form of articles, tutorials, StackOverflow solutions, and open source code. It's also fair to point out that some really stellar PHP has shown up as well. The problem is, writing code the good way (we'll talk about what that means soon) typically tends to be harder. Doing it the down and dirty, quick, and easy to understand way, is, well, easier.

The web has been proliferated with poorly written PHP, and the process of turning out poorly written PHP only naturally increases with the popularity and adoption of the language.

Simply put: it's just way too easy to write bad code in PHP, it's way too easy to find bad code in PHP, it's way too easy to suggest (via putting source code out there or writing tutorials) others write bad code, and it's way too easy for developers to never "level up" their skills.

So why is bad code bad? Let's discuss the results of bad code.

We Can't Test Anything

> We don't have time to write tests, we need to get working software out the door.
>
> -The Project Manager at a previous job

Who has time to write tests? Test are hard, time consuming, and they don't make anybody any money. At least according to project managers. All of this is absolutely correct. Writing good tests can be challenging. Writing good test can be time consuming. Very rarely will you come across an instance in your life where someone cuts you a check specifically to write software tests.

The Project Manager at my last job who, painfully, was also my boss, absolutely put his foot down to writing tests. Working software out the door was our one and only goal; making that paycheck. What's so incredibly ironic about this is that a robust test suite is the number one way to make it possible to write working software.

Writing tests is supremely important to having a stable, long lasting software application. The mountains of books, articles, and conference talks dedicated to the subject are a testament to that fact. It's also a testament to how hard it is to test, or, more correctly, how important it is to test effectively.

Testing is the single most important means of preventing bugs from happening in your code. While it's not bullet proof and can never catch everything, when executed effectively, it can become a quick, repetitive, and solid way to verify that a lot of the important things in your code – such as calculating taxes or commissions or authentication – is working properly.

There is a direct correlation between how poorly you write your code, and how hard it is to test that code. Bad code is hard to test. So hard to test, in fact, that it leads some to declare testing pointless. The benefits of having tests in place though, cannot be argued.

Why does bad code make tests so hard? Think about a taxing function in our software. How hard would it be to test that taxing functionality if it were spattered about controllers? Or worse yet, spattered about a random collection of `.php` files? You'd essentially have to CURL the application with a set of POST variables and then search through the generated HTML to find the tax rates. That's utterly terrible.

What happens when someone goes in and changes around the tax rates in the database? Now your known set of data is gone. Simple: use a testing database and pump it full of data on each test run. What about when designers change up the layout of the product page, and now your code to “find” the tax rate needs to change? Front-end design should dictate neither business nor testing logic.

It is nearly impossible to test poorly written code.

Change Breaks Everything

The biggest consequence of not being able to test the software is that change breaks everything. You’ve probably been there before: one little minute change seems to have dire consequences. Even worse, one small change in a specific portion of the application causes errors within a seemingly unrelated portion of the application. These are **regression bugs**, which is a bug caused after introducing new features, fixing other bugs, upgrading a library, changing configuration settings, etc.

When discovered, these regression bugs often lead to exclamations of “We haven’t touched that code in forever!” When they’re discovered, it’s often unknown what caused them in the first place, due to the nature of changes usually happening in “unrelated” portions of the code. The time elapsed before discovering them is often large, especially for obscure portions of the application, or very specific circumstances needed to replicate the bug.

Change breaks everything, because we don’t have the proper architecture in place to gracefully make those changes. How often have you hacked something in, and ignored those alarm bells going off in your brain? Further, how often has that hack come around to bite you later in the form of a regression bug?

Without good clean architecture conducive to change, and/or a comprehensive set of test suites, change is a very risky venture for production applications. I’ve dealt with numerous systems that the knowledgeable developer declared “stable and working” that they were utterly terrified of changing, because, when they do, “it breaks.” Stable, huh?

We Live or Die by the Framework

Frameworks are fantastic. If written well, they speed up application development tremendously. Usually, however, when writing software within a framework, your code is so embedded into that framework that you’re essentially entering a long term contract with that framework, especially if you expect your project to be long lived.

Frameworks are born every year, and die every once-in-awhile, too (read: CodeIgniter, Zend Framework 1, Symfony 1). If you’re writing your application in a framework, especially doing so the framework documented way, you’re essentially tying the success and longevity of your application to that of the framework.

We’ll discuss this much more in later chapters, and go into a specific instance where my team and I failed to properly prepare our application for the death of our chosen framework. For now, know this: there is a way to write code, using a framework, in such a way that switching out the framework shouldn’t lead to a complete rewrite of the application.

We Want to Use All the Libraries

Composer[3] and Packagist[4] brought with them a huge proliferation of PHP libraries, frameworks, components, and packages. It's now easier than it ever has been to solve problems in PHP. The wide range of available libraries, installed quickly and simply through Composer, make it easy to use other developer's code to solve your problems.

Just like the framework, though, using these libraries comes at a cost: if the developer decides to abandon them, you're faced with no choice but eventually replacing it with something else. If you've littered your code base with usages of this library, you now have a time consuming process to run through to upgrade your application to use some other library.

And of course, later, we'll describe how to gracefully handle this problem in a way that involves minimal rewriting, and hopefully minimal bugs if you've written a good test suite to verify your success.

Writing Good Code

Writing good code is hard.

The goal of this book is to solve these problems with bad code. We'll discuss how architecture plays a key role in solving both causing and solving these problems, and then discuss ways in which to correct or at least mitigate these issues, such that we can build strong, stable, and long-lasting software applications.

[3]https://getcomposer.org/
[4]https://packagist.org/

What is Architecture?

Whether you know it or not, any piece of software you have ever written has followed some sort of architecture, even if it were just your own. Software architecture is the structure that defines the flow of information through a software system. It is a set of decisions made about how software is organized and operates in order to meet the goals of that software.

Architecture can apply to the application as a whole, or might only apply to individual pieces of the application. Maybe you follow one architectural pattern on the client side of the application and a completely different one on the server side of the application. The means by which your client side application and server side application communicate can follow an architectural pattern as well.

What does Architecture Look Like?

Some examples of architecture include how you organize your files, whether you intermix your PHP code and your HTML code, or whether your code is procedural in nature or object-oriented. The architecture might be whether you interface with the database directly, or abstract it away in your code so that you're moving across several layers to get at the data. Or maybe you're interacting with an API on the front-end, using something like Angular JS, and your back-end PHP is simply an API layer that gives data to the front-end application.

All of these features of your application determine the architecture. Architecture is simply a set of attributes about how your code is laid out, organized, and how it interacts with other pieces or layers of the code.

Since describing your architecture can be pretty verbose, architectural patterns can also be named, and often are when they are shared and described within the industry. Following commonly defined architecture, rather than coming up with something on your own, makes your code easily readable and understandable by other developers, especially when that architecture is well documented, and makes it quite easy to describe your architecture.

For example, you might be able to just say you use the MVC architecture on the front-end and an API web service on the back-end. Developers familiar with these ideas should understand you pretty quickly.

Layers of Software

Often when people talk of software architecture, they mention layers of software. **Layers**, in object oriented programming, are groups of specific classes that perform the similar functions. Usually, layers are broken up by concerns, which is a particular set of function or information. Our concerns can be many, depending on the application, but can include: database interaction, business rules, API interactions, or the view, or UI.

In good software architecture, these concerns are each broken out out into layers. Each layer is a separate set of code that should, in a perfect world, loosely interact with the other layers. For instance, when a web request comes in, we might pass it off to the control layer that processes the request, which pulls any necessary data from the database layer, and finally presents it to the view layer to render a UI to the user.

In a perfect world, these layers and their interaction is kept fairly separate, and specifically controlled. As you're about to see, without these layers, software can become pretty messy and hard to maintain.

Examples of Poor Architecture

Before we start discussing how to cleanly build the architecture of your application, let's first take a look at some code with poor architecture. Analyzing the problems faced by poor architecture can help us in understanding why our good architectural decisions are important and what benefits we might gain from taking better architectural routes.

Dirty, In-line PHP

PHP has an easy entry point: it's not hard to get up and running, and the internet is littered with code samples. This is good for the language and community as it creates a low barrier for starting. Unfortunately, this has its drawbacks in that most of those code samples found on the web aren't of the highest caliber, and often lead new developers into writing code that looks similar to this:

```php
<body>
  <?php $results = mysql_query(
    'SELECT * FROM customers ORDER BY name'
  ); ?>

  <h2>Customers</h2>
  <ul>
    <?php while ($customer = mysql_fetch_assoc($results)): ?>
    <li><?= $customer['name'] ?></li>
    <?php endwhile; ?>
  </ul>
</body>
```

You'll find code just like this plastered all over blogs and tutorials, even on websites touting themselves as a professional resource (no names named). Sadly, there is quite a bit wrong with writing PHP this way, especially for anything but tiny applications.

The `mysql_` Functions are Deprecated

Right off the bat, the most glaring issue with this code is the use of the deprecated `mysql_`-functions. Even worse: the functions are being removed entirely in PHP 7, although you can install them as an extension via other channels.

They have been deprecated for a reason: they are considered unsafe and insecure to use, and suitable alternatives (PDO and the `mysqli_` functions) have been created.

Regardless of third party support, this will make upgrading PHP hard or impossible some day. Choosing to use these functions today is choosing heartache tomorrow.

One Layer to Rule Them All

This sample PHP is written in a monolithic style. A **monolithic application** is one with a single layer comprising everything that application does, with each different concern of the application squished together. These applications are not modular and thus do not provide reusable code, and are often hard to maintain. Remember we discussed that software is often layered, keeping the code responsible for interfacing with the data separate from the code responsible for displaying it to the user. This sample is the complete opposite of a layered approach.

Our sample has some code which retrieves data from a database back-end and a view layer that is responsible for displaying this information to the user, all together as one single layer. We're literally querying right in the middle of our HTML. This monolithic approach is the biggest problem with this sample, and it is the direct cause of the next two problems.

A Refactoring Nightmare

Refactoring is pretty much out of the window. Consider these scenarios and what we would have to do to accomplish them:

- What if a table name or column name changes? How many different files will we have to update to make that change? What about switching to a library like PDO? What if we want to get data from a different data source, like a RESTful API?
- What if we decided we wanted to start using a templating language, like Twig or Blade? Our database logic is so tightly wound into our HTML that we would have to rewrite the application to get it out so that we can have nice templates to represent the data.
- What if we wanted to provide some kind of standardized way of displaying user names (like *Kristopher W.* instead of *Kristopher Wilson*)? How many places might we have displayed a users name that now has to be updated?

Simply put, the ability to refactor code is hampered by our lazy, dirty architecture that throws everything together without any concern for the future.

An Untestable Pile of Mush

This approach results in code that is virtually untestable. Sure, some end-to-end tests, like Behat, will probably allow us to test this, but unit tests are out the window. We can't test anything in isolation. How do we ensure that we got the expected number of users back? Using a live database, what *is* the expected number of users? And since we can't test the `$results` variable directly, do we have to parse the HTML DOM? And what happens when the DOM changes?

These are going to be some poor, slow, error prone tests.

Testing only works when it can be executed repetitively and quickly. If it takes a long time to run the tests, the developer simply cannot rely on them through the process of development as it would incur too much time waiting to discover if some seemingly minute change broke anything unexpected elsewhere in the application.

Poor Man's MVC

Usually, the next progression in architecture in the world of PHP development is the adoption of an MVC style architecture, which we'll talk about in MVC. While this is a big step up from in-line, procedural PHP, it can still have several issues, especially if not implemented correctly. Take this example:

```
class CustomersController {
  public function indexAction() {
    $db = Db::getInstance();

    $customers = $db->fetchAll(
      'SELECT * FROM customers ORDER BY name'
    );

    return [
      'customers' => $customers
    ];
  }
}
```

```
<h2>Customers</h2>
<ul>
  <?php foreach ($this->customers as $customer): ?>
  <li><?= $customer['name'] ?></li>
  <?php endforeach; ?>
</ul>
```

This code looks better. We're using controllers and views, so we've separated the presentation logic from the control logic. The controller's `indexAction()` grabs all the customers and then returns them, which gets passed to the view so it can render the data. This should make the code much easier to test and refactor. Except it doesn't.

Still Hard-coding Queries

This is obvious. We still haven't solved the problems of having hard-coded queries in layers that don't concern the database, such as controllers. See the comments above in *A Refactoring Nightmare* for more details.

Strong Coupling to the `Db` Class

We've moved away from using the deprecated `mysql_` functions and instead have abstracted away the database into some `Db` class, which is good. Except we still suffer the pitfalls of not being able to refactor this code. Our same questions as above in *A Refactoring* nightmare still apply. We can hardly change anything about our database layer without having to touch a large amount of files that use that database layer to do so.

Still Hard to Test

We've made testing much easier now simply by extracting our code out of the HTML. We now have a controller doing all the processing, and then passing that data off to the view. This is much easier to test, as we can simply instantiate `CustomersController`, call the `indexAction()` method, and analyze the return value. But how many customers should we expect, and what are their names? Again, we can't know this unless we go the complicated route of setting up a test database (a known state) before running our tests.

Since we are declaring our `Db` class right in the `indexAction()` method, there's no way to mock that. If there were, we could simply set it up to return a known set of customers, and then validate that the `indexAction()` properly retrieved them.

Two Very Large Layers

1. This code is hard to test in isolation as it declares it's database dependency in-line, and thus can't test without it. We can't override it. We could override the configuration so we could use a properly staged test database, which is good for integration testing, but unit testing is impossible. We simply can't test this controller without the database class.
2. We're still hard coding queries, which ties us to a database and specific database flavor at that.
3. We're retrieving an instance of the `Db` class, which tightly couples this implementation to that class. We talk about this in more detail in Coupling, The Enemy, but for now, understand that it makes it very hard to test this controller without bootstrapping our database class as well.
4. If we decide to rewrite our application layer, we lose everything. This is because our data domain is wrapped so tightly into our application services. Let's say for an instance that we're using Zend Framework and this is a Zend Framework controller. What happens when we want to switch to Laravel? This would require us to rewrite our entire controllers, but since our data access logic is stored right in the controller, we have to rewrite that, too, especially if we switch to using Eloquent ORM, which ships with Laravel.

Poor Usage of Database Abstraction

Finally, we get smart and abstract away the data source using the Repository design pattern:

```
class CustomersController {
  public function usersAction() {
    $repository = new CustomersRepository();
    $customers = $repository->getAll();

    return [
      'customers' => $customers
    ];
  }
}
```

```
<h2>Customers</h2>
<ul>
  <?php foreach ($this->customers as $customer): ?>
  <li><?= $customer['name'] ?></li>
  <?php endforeach; ?>
</ul>
```

This code is much better than our original example, and even better than our second. We're slowly coalescing to some good application architecture.

Instead of interfacing directly with the database, we've abstracted it away into a Repository class. The repository is responsible for understanding our datasource and retrieving and saving data for us. Our controller doesn't have to know anything about where the data comes from, so we've removed the bad, hard-coded queries from the controller. We could easily refactor `CustomersRepository` to get its data from a different source, but wouldn't have to touch any code that uses the repository so long as the `getAll()` method's signature and return result are still the same.

While this is much better architecture, it still suffers some issues:

Strong Coupling to `CustomersRepository`

We're using a concrete instance of the `CustomersRepository`, which means the controller is still tied to that implementation. For instance, this `CustomersRepository` probably connects to a database of some sort to retrieve the information. Now our controller is permanently tied to this implementation, unless we refactor it away. If we're going to change out where or how our data is stored, we're probably going to write a new class instead of completely changing the existing one. We discuss how to solve this in Dependency Injection.

Continuing Dependency Issues

We're still declaring our dependency (`CustomersRepository`) right in our method, which makes it impossible to mock and test the `usersAction()` method in isolation (remember, we'd have to setup an entire known state in the database for this to work). This might be great for end-to-end testing of our application, but it isn't so great for unit testing our application.

We'll also talk about how to solve this in Dependency Injection.

So how Should this Code Look?

It's pretty easy to pick apart some sample code and explain why it needs improvement, but it's much harder to simply provide good code samples without going into quite a bit of discussions first. We're going to solve this exact problem (listing customers) once we get to our Case Study at the end of the book, which will build on concepts we discuss in the next few chapters.

However, just like the days leading up to a holiday, everybody loves a sneak peek. We were actually really close in the last sample, and only had to make a few tweaks to make this some rock solid architecture. This is how we will solve this problem later:

```php
class CustomersController extends AbstractActionController {
  protected $customerRepository;

  public function __construct(CustomerRepositoryInterface $repository) {
    $this->customerRepository = $repository;
  }

  public function indexAction() {
    return [
      'users' => $this->customerRepository->getAll()
    ];
  }
}
```

We've solved several problems:

1. We're no longer tightly coupled to any repository implementation by using an interface. Whatever we get will be required to implement `CustomerRepositoryInterface`, and that should give us our data. We don't care what, how, or where.
2. We can easily test now as we can mock the class being used by the controller and make it return a known set of data. Then we can test that the controller properly passes it off to the view.
3. We have nothing in here that should prevent us from ever upgrading to newer versions of PHP or libraries, unless PHP or some library drastically change how they work, which would require a big rewrite regardless of how we wrote our code.
4. Queries? We're not even using queries. Again: we don't even know where our data comes from at this layer. If we suddenly need to get data from a different place, no big deal: simply pass us a different object that implements `CustomerRepositoryInterface`.

If some of this doesn't make much sense, don't worry. We're about to cover it all in-depth in the next chapters.

Costs of Poor Architecture

As we've just seen, taking a bad approach when developing your application can lead to several problems. Classically, using a bad architecture can lead to the following common problems, although it entirely depends on how the application was written:

1. **Untestable**. Poor architecture often results in code that is difficult to test. This especially happens when things are tightly coupled together and cannot be tested in isolation. We'll talk about this in Coupling, the Enemy. Inability to test can lead to an unstable application.
2. **Hard to Refactor**. Developers tend to make iterative changes to the software they build as their understanding of and solution to a problem is strengthened. Users often request additional features and changes to existing applications. Both of these instances are known as refactoring, and software written with a poor architecture is hard to refactor, especially without a strong test suite to guarantee nothing breaks. See #1.
3. **Impossible to Upgrade**. Code architected and written poorly is often very hard to upgrade, either to new versions of PHP, new versions of underlying frameworks and libraries, or switching to new frameworks and libraries entirely. This can cause projects to end up in an impossible upgradeable limbo.

Coupling, The Enemy

The main issue we were dealing with when we looked at various examples of poor architecture is coupling. **Coupling** is the amount of dependency one component has on another. If one component simply cannot function without another, it is **highly coupled**. If one component, loosely depends on another and can function without it, it is **loosely coupled**.

The looser the coupling within your code base, the more flexibility you have in that codebase. With a high amount of coupling, refactoring, such as extending new functionality to existing code, becomes a very dangerous task. With loosely coupled code, it becomes much easier to change this around and swap out solutions as the code using that solution is not fully dependent upon it.

To get a better understanding of coupling, let's look at two very different examples of highly coupled code.

Spaghetti Coupling

```
<body>
  <?php $users = mysqli_query('SELECT * FROM users'); ?>

  <ul>
    <?php foreach ($users as $user): ?>
    <li><?= $user['name'] ?></li>
    <?php endforeach; ?>
  </ul>
</body>
```

This example, which is very similar to our first example of code with poor architecture, has a lot of coupling. Can the application function in any respect without the database? No, we have queries all over the code. It is highly coupled to the database.

Can the application function without a web browser? Well, technically yes, but who wants to read straight HTML? Can we get a list of users without it being formatted in HTML? No, we cannot. Our application is highly coupled to the browser.

OOP Coupling

```
class UsersController {
  public function indexAction() {
    $repo = new UserRepository();
    $users = $repo->getAll();

    return $users;
  }
}
```

In this example, we have a class, called `UsersController`, that uses another class, called `UserRepository` to get all the users. This code looks much better than the first example, but it still has a high level of coupling.

Can the `UsersController` function without the `UserRepository`? Definitely not, it's highly coupled to it.

Why is Coupling the Enemy?

So what's the big deal about all this coupling anyway? Who cares?

People who care about having loosely coupled code are:

1. **Developers who refactor their code**. Do you always get it right the first time? Do requirements never change on you? We often need to move things around or rework them, but that's often hard to do when the code you're reworking is so tightly bound to code in several other places. One little change here, a couple dozen regression bugs there.
2. **Developers who like to test their code**. Testing code can be an absolute pain if the code is tightly coupled. Often, we want to test just one component of an application at a time, in isolation – unit testing. But that's impossible when one class requires a dozen other classes to run, and instantiates them itself.
3. **Developers who like to reuse their code**. Reusing code is great! Writing the same code twice sucks. Reusing one piece of code is absolutely impossible when it is tightly coupled to the rest of your application. You can't just copy the class out and drop it in another project without either hacking away it's coupling, or bringing everything else with it. For shame.

Simply put, coupling is the enemy of developers everywhere as it makes their future lives incredibly difficult. Don't screw over your future self.

How do we Reduce Coupling?

There are quite a few ways we can reduce the amount of coupling within our codebase, but we'll cover four basic, easy solutions:

1. **Have less dependencies**. This sounds like a no brainer. Having less dependencies reduces the amount of coupling in your code by reducing the amount of things to couple to. This does not mean, however, that we need to stop using dependencies. By making sure our classes and method are short, and only have one purpose, and by breaking out complex routines into several classes and methods, we can reduce the amount of dependencies each class itself needs, which makes it much easier to refactor classes in isolation.
2. **Use Dependency Injection**. We'll cover this in the next chapter. Dependency injection provides us with a means with move the control of dependencies outside of our class and giving it to a third party.
3. **Use Interfaces, not Concrete Classes**. As much as possible, we want to couple ourselves to interfaces, which provide a sort of contract of what to expect. Used together with with dependency injection, we can write classes that know nothing about our dependencies, only that they request a specific format for the dependency, and let something else provide it. We'll also cover this in the next chapter.
4. **Use Adapters**. Instead of coupling to something, instead couple to an adapter, which takes some third party code and transforms it into what we'd expect to to look and behave like. Combine this with #2 and #3 above, and we can safely use third party code without tightly coupling to it. We'll cover this in Abstracting with Adapters.

Your Decoupling Toolbox

We've uncovered various ways in which poor design decisions can lead to code that is hard to maintain, hard to refactor, and hard to test. Now we're going to look at some guiding principles and design patterns that will help us alleviate these problems and help us write better code. Later, when we talk about architecture, we'll apply these principles further to discover how to create truly uncoupled, refactorable, and easily testable code.

Design Patterns, A Primer

A **design pattern** is a specific solution to a commonly occurring problem in software development. These design patterns form a common language among software developers that can be used to discuss problems and describe solutions. They are transferable across languages and are not specific to PHP.

Design patterns as we know them today are heavily influenced by "the Gang of Four" who were instrumental in their popularization starting in 1994 with their book *Design Patterns: Elements of Reusable Object-Oriented Software*. The GoF are: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

In their book, the GoF described twenty three design patterns organized into three categories: Creational, Structural, and Behavioral. It would be impossible to cover all of these patterns in any detail in this book, however, as they are instrumental to nearly every chapter that follows, we're going to briefly cover a few of them here.

1. **Factory**: An object responsible for instantiating other objects
2. **Repository**: Not actually a GoF design pattern, this is an object responsible for transitioning data to and from its storage
3. **Adapter**: An object that encapsulates another object to make it conform to a desired API
4. **Strategy**: Encapsulates a behavior or a set of behaviors, allowing them to be used interchangeably

For a lighter, and maybe funner approach to design patterns, you can also checkout Head First Design Patterns[5] by Freeman, Bates, Sierra, and Robson.

The Factory Patterns

Factories are objects that are responsible for creating other objects when needed, just like factories in real life are responsible for creating tangible things.

In real life, a Car Factory is responsible for creating cars. If we wanted this concept in code, it'd be the same. We'd have a `CarFactory` object that would create `Car` objects for us.

Factories in code can be applied to things that generally wouldn't make sense in real life, such as a `CustomerFactory` or `ConnectionStrategyFactory`.

Typically, if we wanted to create a `Customer` object, we'd just instantiate one:

[5]http://www.headfirstlabs.com/books/hfdp/

```
$customer = new Customer();
```

And this is fine for very simple classes with little configuration. But what if we had to bootstrap our `Customer` object, so to speak? Let's say, for instance, that by default, a `Customer` has a `$0` credit limit, is of status "pending" and is randomly assigned an account manager. Do we really want to litter our code with this logic in all the places we might create a new `Customer` object? What happens when these business rules change?

Of course, we're going to use a factory to handle creating `Customers` for us:

```
class CustomerFactory {
  protected $accountManagerRepo;

  public function __construct(AccountManagerRepository $repo) {
    $this->accountManagerRepo = $repo;
  }

  public function createCustomer($name) {
    $customer = new Customer();
    $customer->setName($name);
    $customer->setCreditLimit(0);
    $customer->setStatus('pending');
    $customer->setAccountManager(
      $this->accountManagerRepo->getRandom()
    );

    return $customer;
  }
}
```

The business logic for creating new `Customer` objects is encapsulated within the `CustomerFactory` object.

There are several benefits to using factories:

1. **Reusable code**. Any place you have to create an object, the same logic is applied. No duplicate logic.
2. **Testable code.** Factories make creational logic easy to test. If this code were directly within some other class somewhere, it wouldn't be possible to test in isolation.
3. **Easy to change**. If the logic ever changes, the change only needs to occur in one place.

Static Factories

Often, you'll see factories in the wild setup as "static classes" (i.e.: classes with only static methods):

```php
class CustomerFactory {
  public static function createCustomer($name) {
    $customer = new Customer();
    $customer->setName($name);
    $customer->setCreditLimit(0);
    $customer->setStatus('pending');

    return $customer;
  }
}
```

Which would be called simply:

```php
$customer = CustomerFactory::createCustomer('ACME Corp');
```

At first, this seems much cleaner and easier to use than instantiating the factory every time we need it. However, our first `CustomerFactory` had additional dependencies it needed to create `Customer` objects, namely the `AccountRepository`. We could pass this dependency to the method each time we call it, but that would be a mess to cleanup if we ever changed the name of `AccountRepository` or switched to another paradigm for data management.

I'll leave it up to you whether you want to use static factories or not. For simple factories, there's probably no negatives to doing so. For involved factories, especially those with dependencies, it can lead to some pretty smelly code.

Types of Factories

The `CustomerFactory` is typically how repositories are thought of by many, but it's not exactly how the Factory Pattern was described in the Gang of Four design patterns book. In fact, they outlined two different types of factories.

Typically, the type of factory we created above is called a Simple Factory. A **simple factory** is a class responsible for creating another object.

Let's look at the two factories patterns defined by the GoF:

1. The **Factory Method** pattern
2. The **Abstract Factory** pattern

Factory Method Pattern

According to the Gang of Four, the intent of the **Factory Method Pattern** is to:

> Define an interface for creating an object, but let subclasses decide which class to instantiate.

This type of factory isn't standalone like the Simple Factory we describe above. Instead, it's embedded in either an abstract class or interface, or even within a concrete class. Whatever class needs to define a subordinate class can have a factory method.

```
class Document {
  public function createPage() {
    return new Page();
  }
}
```

In this example, we have a simple `createPage()` method on `Document` which returns a new page.

However, if we want to be able to create multiple different types of documents, we could make this an abstract class:

```
abstract class AbstractDocument {
  public function render() {
    $this->addPage(1, $this->createPage());
  }
  public function addPage(1, AbstractPage) {
    // ...
  }

  abstract public function createPage();
}
```

Now if we wanted to create a `ResumeDocument` and a `PortfolioDocument`:

```
class ResumeDocument extends AbstractDocument {
  public function createPage() {
    return new ResumePage();
  }
}

class PortfolioDocument extends AbstractDocument {
  public function createPage() {
    return new PortfolioPage();
  }
}
```

Of course, we need those subordinate objects being created by the factory methods:

```
interface PageInterface {}
class ResumePage implements PageInterface {}
class PortfolioPage implements PageInterface {}
```

The benefit of using a factory method like this is that it allows us to keep the bulk of our common functionality within the `AbstractDocument` class, as it's inherent to both `ResumeDocument` and

`PortfolioDocument`, but not have `AbstractDocument` reliant on the actual concrete document it's working with.

Whenever `AbstractDocument` needs to generate a new page for whatever concrete it's part of, it simply calls `createDocument()` and gets one.

The factory pattern works best when a class doesn't know which type of objects it is working with and needs to create.

Abstract Factory Pattern

According to the Gang of Four, the intent of the **Abstract Factory Pattern** is to:

> Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

With the Abstract Factory pattern, we're interested in devising a solution to handle the creation of multiple different related objects, when the type of objects that need to be created isn't known.

The Abstract Factory Pattern typically has several different players:

1. **Client Code**, which is an object or code that needs to use the factory to create other objects
2. **Object Interface**, which defines the structure of the actual objects to be created
3. **Concrete Objects**, which implement the Object Interface with specific implementation details
4. **An Abstract Factory**, which declares an interface to define how objects (Object Interfaces) should be created
5. **Concrete Factories**, which implement the Abstract Factory interface and create specific types of objects (Concrete Objects)

Let's say we are writing some code to generate various different makes and models of cars, but we want the code to be adaptable to anything we throw at it.

Let's say we have a `Building` class that is responsible for generating a specific car manufacturer's line. This is our *Client Code*:

```
class Building {
  public function createCars() {}
  public function createTrucks() {}
}
```

We want this `Building` to be able to create `Cars` and `Trucks` as necessary. The first thing we need to do for the Abstract Factory Pattern to work is create an interface that defines these objects. These are our *Object Interfaces*:

```
interface CarInterface {}
interface TruckInterface {}
```

Let's say we want to create both Ford and Chevrolet vehicles. We'll need to define a concrete class for each manufacturer and each vehicle type. These are our *Concrete Objects*:

```
class ChevyMalibu implements CarInterface {}
class ChevySilverado implements TruckInterface {}

class FordFiesta implements CarInterface {}
class FordF250 implements TruckInterface {}
```

We're going to need an assembly line to create each of these, so let's define an interface for that too. This assembly line will be the basis for our actual factories. This interface is our *Abstract Factory* player.

```
interface AssemblyLineInterface {
  public function createCar();
  public function createTruck();
}
```

And of course, we'll need some concrete classes of the `AssemblyLineInterface` to create each manufacturer's line. These are our *Concrete Factories*:

```
class ChevyAssemblyLine implements AssemblyLineInterface {
  public function createCar() {
    return new ChevyMalibu();
  }

  public function createTruck() {
    return new ChevySilverado();
  }
}

class FordAssemblyLine implements AssemblyLineInterface {
  public function createCar() {
    return new FordFiesta();
  }

  public function createTruck() {
    return new FordF250();
  }
}
```

Now, our `Building` class can be supplied a specific `AssemblyLineInterface`, and start making vehicles:

```
class Building {
  protected $assemblyLine;
  protected $inventory = [];

  public function __construct(
    AssemblyLineInterface $assemblyLine
  ) {
    $this->assemblyLine = $assemblyLine;
  }

  public function createCars() {
    for ($i = 0; $i < 20; $i++) {
      $this->inventory[] =
        $this->assemlyLine->createCar();
    }
  }

  public function createTrucks() {
    for ($i = 0; $i < 15; $i++) {
      $this->inventory[] =
        $this->assemlyLine->createTruck();
    }
  }
}
```

We could call the code as such:

```
$building = new Building(new FordAssemblyLine());
$building->createCars();
```

The Abstract Factory pattern is useful when several different objects need to be created independent from the system that creates them. If we're only concerned with created one object, then the Abstract Factory isn't a very suitable solution.

Abstract Factory uses Factory Methods

You might have noticed that the `create()` methods in the assembly lines above look a lot like Factory Methods. That's because they are! The Abstract Factory Pattern actually uses the Factory Method Pattern.

Repository Pattern

A **Repository** is an object that allows for the retrieval and persisting of data to a data store. The Repository Pattern is described in great detail in Eric Evan's *Domain-Driven Design: Tackling Complexity in the Heart of Software*:

> A REPOSITORY represents all objects of a certain type as a conceptual set (usually emulated). It acts like a collection, except with more elaborate querying capability.
>
> *Domain-Driven Design, Eric Evans, p. 151*

When working with repositories, one stops thinking of getting data as "querying the database," but instead thinks of the process as retrieving data from the repository.

Repositories usually have methods to retrieve data and methods to persist data. They're usually called different things depending on the implementation.

Common retrieval methods include:

```php
class MyRepository {
  public function getById($id);
  public function findById($id);
  public function find($id);
  public function retrieve($id);
}
```

I prefer either the `get()` or `find()` variants. You'll often encounter other methods in repositories aimed at retrieving data in different ways:

```php
class MyRepository {
  public function getById($id);
  public function getAll();
  public function getBy(array $conditions);
}
```

Common persistence methods include:

```php
class MyRepository {
  public function persist($object);
  public function save($object);
}
```

I prefer to use `persist()`, but these decisions are entirely up to you.

Repositories should only reference one object class. Thus, for each object you need to retrieve or persist, you should have a separate repository for them. If we're working with a `Customer` object, we should have a `CustomerRepository`.

```
public function saveAction() {
  $customer = $this->customerRepository->getById(1001);
  $customer->setName('New Customer Name');

  $this->customerRepository->persist($customer);
}
```

This client code is simple to use; it does not need to know the mechanics or language of the data store. It only needs to know how to use the repository, which it does through a simple API.

Repositories and Factories are often used together. The factory is responsible for created the object, and the repository is responsible for persisting it. When an object already exists in the data store, the repository is also responsible for retrieving a reference to it, and later likely responsible for saving changes made to it.

How does a Repository Work?

This is all well and good, but how exactly does a repository work?

> Objects of the appropriate type are added and removed, and the machinery behind the REPOSITORY inserts them or deletes them from the database.
>
> *Domain-Driven Design*, Eric Evans, p. 151

What is this machinery behind the repository? Making repositories work can be simple. Making repositories that work very well can be quite complicated. It's best to rely on an already established framework, especially when you're just starting out. Doctrine ORM[6] provides a great Data Mapper implementation, while Eloquent ORM[7] and Propel[8] provide implementations of the Active Record pattern, and Zend Framework 2[9] provides a Table Data Gateway implementation.

However, if you really want to dive in with creating your own repository backend, I'd highly suggest reading Martin Fowler's Patterns of Enterprise Application Architecture[10]. This book covers how to properly implement various patterns that would sit behind a repository, including Data Mapper, Table Data Gateway, and Active Record.

Adapter Pattern

The **Adapter Pattern** allows for encapsulating the functionality of one object, and making it conform to the functionality of another object. This pattern is sometimes also referred to as the **Wrapper Pattern**, as it involves wrapping one object with another.

Let's say we have one class whose interface looks something like this:

[6]http://www.doctrine-project.org/projects/orm.html
[7]http://laravel.com/docs/eloquent
[8]http://propelorm.org/
[9]http://framework.zend.com/
[10]http://martinfowler.com/books/eaa.html

```
class GoogleMapsApi {
  public function getWalkingDirections($from, $to) {}
}
```

We have an interface in our project that defines the structure of an object that gets us distances:

```
interface DistanceInterface {
  public function getDistance($from, $to);
}
```

If we want to provide a `DistanceInterface` concrete class that gives the walking distance between two points, we can write an adapter to use the `GoogleMapsApi` class to do so:

```
class WalkingDistance implements DistanceInterface {
  public function getDistance($from, $to) {
    $api = new GoogleMapsApi();
    $directions = $api->getWalkingDirections($from, $to);

    return $directions->getTotalDistance();
  }
}
```

This class inherits from our `DistanceInterface`, and properly returns the distance using the `GoogleMapsApi` class to first get the walking directions, and then return the distance contained in that result.

The Adapter Pattern allows us to take one object, and adapt it to fit the interface of another object, making them compatible in whatever context we're trying to use them.

Adapters are discussed in more detail in the chapter Abstracting With Adapters, including some very valid use cases for this pattern.

Strategy Pattern

The **Strategy Pattern** allows the behavior of an algorithm to be determined during runtime of an application. Strategies are usually a family of classes that share a common interface, that each encapsulate separate behavior that can be interchangeable at runtime.

Let's say we need to develop an invoicing process for our customers, who have the option of choosing between two methods of receiving their invoices: email or paper.

```php
public function invoiceCustomers(array $customers) {
  foreach ($customers as $customer) {
    $invoice = $this->invoiceFactory->create(
      $customer,
      $this->orderRepository->getByCustomer($customer)
    );

    // send invoice...
  }
}
```

Our `InvoiceFactory` takes care of generating an invoice from all the orders returned by our `OrderRepository`, so how do we go about sending those invoices?

Using the Strategy Pattern, we first define an interface that describes how all invoices should be sent, regardless of delivery method:

```php
interface InvoiceDeliveryInterface {
  public function send(Invoice $invoice);
}
```

We have two possible methods of delivering an invoice: email or print. Let's define a strategy for each.

```php
class EmailDeliveryStrategy implements InvoiceDeliveryInterface {
  public function send(Invoice $invoice) {
    // Use an email library to send it
  }
}


class PrintDeliveryStrategy implements InvoiceDeliveryInterface {
  public function send(Invoice $invoice) {
    // Send it to the printer
  }
}
```

Optimization Opportunity

How these two classes manage to delivery the invoice aren't important for this exercise, but consider how you might use third party libraries to send emails and use some kind of printing service (maybe through an API).

Could the Adapter Pattern be a good way to bring those libraries into your code base and use them within these strategies?

Our calling code (the client) now needs to make a determination of which strategy to use:

```php
public function invoiceCustomers(array $customers) {
  foreach ($customers as $customer) {
    $invoice = $this->invoiceFactory->create(
      $customer,
      $this->orderRepository->getByCustomer($customer)
    );

    switch ($customer->getDeliveryMethod()) {
      case 'email':
        $strategy = new EmailDeliveryStrategy();
        break;
      case 'print':
      default:
        $strategy = new PrintDeliveryStrategy();
        break;
    }

    $strategy->send($invoice);
  }
}
```

This code now delivers invoices using the two new strategies. The Strategy Pattern has allowed us to encapsulate the behavior and make a determination at runtime of which strategy to use.

The code isn't perfect, though. Is this the correct place to make the determination of how to send an invoice to the customer? Could we maybe utilize one of these design patterns to make this code better? Of course!

Instantiating the correct strategy to use is the perfect place to use a factory.

```php
class InvoiceDeliveryStrategyFactory {
  public function create(Customer $customer) {
    switch ($customer->getDeliveryMethod()) {
      case 'email':
        return new EmailDeliveryStrategy();
        break;
      case 'print':
      default:
        return new PrintDeliveryStrategy();
        break;
    }
  }
}
```

The actual logic within the `InvoiceDeliveryStrategyFactory` class isn't any different than what we had in the `invoiceCustomers` method, but now it's reusable (if that were even necessary in this case), and it's independently testable. It's a great use of a factory.

This simple code example now shows the usage of repositories, factories, strategies, and, if you followed our tip, maybe even adapters!

```php
public function invoiceCustomers(array $customers) {
  foreach ($customers as $customer) {
    $invoice = $this->invoiceFactory->create(
      $customer,
      $this->orderRepository->getByCustomer($customer)
    );

    $strategy = $this->deliveryMethodFactory->create(
      $customer
    );
    $strategy->send($invoice);
  }
}
```

Learning More Design Patterns

Design patterns help us write clean, understandable, concise code that makes refactoring, testing, and maintainability possible. It also gives us a common language to use when discussing ideas with other developers.

This chapter has only scratched the surface of the design patterns presented, and only presented a handful of the design patterns in the wild. Even beyond those introduced by the Gang of Four book, others have defined their own design patterns, and some of those have gained traction.

I highly recommend you pick up at least one of the two books mentioned in the beginning of this chapter:

1. Design Patterns: Elements of Reusable Object-Oriented Software[11]
2. Head First Design Patterns[12]

Design patterns are a great, tried-and-true way to solve common coding problems. We'll use these patterns throughout the rest of this book.

[11]http://amzn.to/XO8DB4
[12]http://www.headfirstlabs.com/books/hfdp/

SOLID Design Principles

Much like design patterns are a common language of constructs shared between developers, SOLID Design Principles are a common foundation employed across all types of applications and programming languages.

The **SOLID** Design Principles are a set of five basic design principles for object oriented programming described by Robert C. Martin[13]. These principles define ways in which all classes should behave and interact with one another, as well as principles of how we organize those classes.

The SOLID principles are:

1. **S**ingle Responsibility Principle
2. **O**pen/Closed Principle
3. **L**iskov Substitution Principle
4. **I**nterface Segregation Principle
5. **D**ependency Inversion Principle

Single Responsibility Principle

The **Single Responsibility Principle** states that objects should have one, and only one, purpose. This is a principle that is very often violated, especially by new programmers. Often you'll see code where a class is a jack of all trades, performing several tasks, within sometimes several thousand lines of code, all depending on what method was called.

To the new OOP developer, classes are often viewed at first as a collection of related methods and functionality. However, the SRP advocates against writing classes with more than one responsibility. Instead, it recommends condensed, smaller classes with a single responsibility.

What is a responsibility?

In his description of the Single Responsibility Principle, Robert Martin describes a responsibility as "a reason for change." Any time we look at a given class and see more than one way in which we might change it, then that class has more than one responsibility.

Another way to look at a responsibility is to look at the behavior of a class. If it has more than one behavior, it is violating SRP.

Let's look at a class that represents a `User` record stored in a database:

[13]http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

```
class User {
  public function getName() {}
  public function getEmail() {}

  public function find($id) {}
  public function save() {}
}
```

This `User` class has two responsibilities: it manages the state of the user, and it manages the retrieval from and persistence to the database. This violates SRP. Instead, we could refactor this into two classes:

```
class User {
  public function getName() {}
  public function getEmail() {}
}

class UserRepository {
  public function find($id) {}
  public function save(User $user) {}
}
```

The `User` class continues to manage the state of the user data, but now the `UserRepository` class is responsible for managing the retrieval and persistence to the database. These two concepts are now decoupled, and the two classes conform to SRP.

When we look at the `UserRepository` class, we can make a determination that retrieving and persisting data to the database are the same responsibility, as a change to one (such as changing where or how the data is stored) requires a change to the other.

Breaking up Classes

In order to apply the SRP principle to existing classes, or even when creating new classes, it's important to analyze the responsibility of the class. Take for instance a customer invoicing class, like the one we looked at in the previous chapter:

```
class InvoicingService {
  public function generateAndSendInvoices() {}
  protected function generateInvoice($customer) {}
  protected function createInvoiceFile($invoice) {}
  protected function sendInvoice($invoice) {}
}
```

Already it's plainly obvious that this class has more than one responsibility. Just looking at the method name of `generateAndSendInvoices()` reveals two. It's not always readily apparent from class and method names how many responsibilities there are, though. Sometimes it requires looking at the actual code within those methods. After all, the method could have simply been named `generateInvoices()`, hiding the fact that it was also responsible for delivering those invoices.

There are at least four separate responsibilities of this class:

1. Figuring out which invoices to create
2. Generating invoice records in the database
3. Generating the physical representation of the invoice (i.e.: PDF, Excel, CSV, etc)
4. Sending the invoice to the customer via some means

In order to fix this class to conform to SRP, we'll want to break it up into smaller, fine tuned classes, each representing one of the four responsibilities we identified above, plus the `InvoicingService` class that ties this all together.

```
class OrderRepository {
  public function getOrdersByMonth($month);
}

class InvoicingService {
  public function generateAndSendInvoices() {}
}

class InvoiceFactory {
  public function createInvoice(Order $order) {}
}

class InvoiceGenerator {
  public function createInvoiceFormat(
    Invoice $invoice,
    $format
  ) {}
}

class InvoiceDeliveryService {
  public function sendInvoice(
    Invoice $invoice,
    $method
  ) {}
}
```

Our four classes here represent the responsibilities of the previous `InvoicingService` class. Ideally, we'd probably even have more than this: we'll probably want strategy classes for each format needed for the `InvoiceGenerator` and strategy classes for each delivery method of the

`InvoiceDeliveryService`. Otherwise, these classes end up having more than one responsibility as they're either generating multiple file formats, or utilizing multiple delivery methods.

This is a lot of classes, almost seemingly a silly number of classes. What we've given up, however, is one very large, monolithic class with multiple responsibilities. Each time we need to make a change to one of those responsibilities, we potentially risk introducing an unintended defect in the rest of the seemingly unrelated code.

Why does SRP matter?

Why are we concerned with making sure a class only has one responsibility? Having more than one responsibility makes those responsibilities coupled, even if they are not related. This can make it harder to refactor the class without unintentionally breaking something else, whereas having a separate class for each responsibility shields the rest of the code from most of the risk.

It's also much easier to test a class with only one responsibility: there's only one thing to test, although with a potential for many different outcomes, and there's much less code involved in that test.

Generally, the smaller the class, the easier it is to test, the easier it is to refactor, and the less likely it is to be prone to defects.

Open/Closed Principle

The **Open/Closed Principle** states that classes should be open to extension, but closed to modification. This means that future developers working on the system should not be allowed or encouraged to modify the source of existing classes, but instead find ways to extend the existing classes to provide new functionality.

The Strategy Pattern introduced in the previous chapter of Design Patterns provides a great example of how the Open/Closed Principle works. In it, we defined strategies for mechanisms to deliver invoices to customers. If we wanted to add a new delivery method, perhaps one via an EDI (Electronic Data Interchange), we could simply write a new adapter:

```
class EdiStrategy implements DeliveryInterface {
    public function send(Invoice $invoice) {
        // Use an EDI library to send this invoice
    }
}
```

Now the invoice process has the ability to deliver invoices via EDI without us having to make modifications to the actual invoicing code.

The OCP in PHPUnit

The PHPUnit testing framework provides a great example of how a class can be open to extension, but closed for modification. The `PHPUnit_Extensions_Database_TestCase` abstract class requires that each individual test case provide a `getDataSet()` method, which should return an instance of `PHPUnit_Extensions_Database_DataSet_IDataSet`, an interface. PHPUnit provides several implementations of this interface, including a `CsvDataSet`, `XmlDataSet`, `YamlDataSet`, etc.

If you decided you wanted to provide your data sets as plain PHP arrays, you could write your own data set provider class to do so, simply by implementing the `IDataSet` interface. Using this new class, we could provide the `TestCase` class with a data set, parsed from PHP arrays, that works and acts like any of the built-in PHPUnit data sets.

```
class MyTest extends DatabaseTestCase {
  public function getDataSet() {
    return new ArrayDataSet([]);
  }
}
```

The internal code of PHPUnit has not been modified, but now it is able to process pure PHP arrays as data sets. [14]

https://github.com/mrkrstphr/dbunit-fixture-arrays.

Why does OCP matter?

The benefit of the Open/Closed Principle is that it limits the direct modification of existing source code. The more often code is changed, the more likely it is to introduce unintended side effects and cause defects. When the code is extended as in the examples above, the scope for potential defects is limited to the specific code using the extension.

Liskov Substitution Principle

The **Liskov Substitution Principle** says that objects of the same interface should be interchangeable without affecting the behavior of the client program [15].

This principle sounds confusing at first, but is one of the easiest to understand. In PHP, interfaces give us the ability to define the structure of a class, and then follow that with as many different concrete implementations as we want. The LSP states that all of these concrete implementations should be interchangeable without affecting the behavior of the program.

So if we had an interface for greetings, with various implementations:

[14]If you want to see a complete example of this concept in action, checkout

[15]http://www.objectmentor.com/resources/articles/lsp.pdf

```php
interface HelloInterface {
  public function getHello();
}

class EnglishHello implements HelloInterface {
  public function getHello() {
    return "Hello";
  }
}

class SpanishHello implements HelloInterface {
  public function getHello() {
    return "Hola";
  }
}

class FrenchHello implements HelloInterface {
  public function getHello() {
    return "Bonjour";
  }
}
```

These concrete `Hello` classes should be interchangeable. If we had a client class using them, the behavior shouldn't be affected by swapping them for one another:

```php
class Greeter {
  public function sayHello(HelloInterface $hello) {
    echo $hello->getHello() . "!\n";
  }
}

$greeter = new Greeter();
$greeter->sayHello(new EnglishHello());
$greeter->sayHello(new SpanishHello());
$greeter->sayHello(new FrenchHello());
```

While the output may be different, which is desired in this example, the behavior is not. The code still says "hello" no matter which concrete instance of the interface we give it.

LSP in PHPUnit

We already discussed an example of the Liskov Substitution Principle when we discussed the Open/Closed Principle. The `ArrayDataSet` class we defined as an instance of PHPUnit's `IDataSet` is returned from the `getDataSet()` method of DbUnit's `DatabaseTestCase` abstract class.

```
class MyTest extends DatabaseTestCase {
  public function getDataSet() {
    return new ArrayDataSet([]);
  }
}
```

The PHPUnit `DatabaseTestCase` class expects that the `getDataSet()` method will return an instance of `IDataSet`, but doesn't necessarily care what implementation you give it, so long as it conforms to the interface. This is also referred to as design by contract, which we'll talk about in much more detail in Dependency Injection.

The key point of the Liskov Substitution Principle is that the behavior of the client code shall remain unchanged. Regardless of what implementation of `IDataSet` we return from `getDataSet()`, it will result in the data set being loaded into the database for unit tests to be run against. It doesn't matter if that data came from CSV, JSON, XML, or from our new PHP array class: the behavior of the unit tests remain the same.

Why does LSP matter?

In order for code to be easily refactorable, the Liskov Substitution Principle is key. It allows us to modify the behavior of the program, by providing a different instance of an interface, without actually modifying the code of the program. Any client code dependent upon an interface will continue to function regardless of what implementation is given.

In fact, as we've already seen, the Liskov Substitution Principle goes hand-in-hand with the Open/Closed Principle.

Interface Segregation Principle

The **Interface Segregation Principle** dictates that client code should not be forced to depend on methods it does not use[16]. The principle intends to fix the problem of "fat" interfaces which define many method signatures. It relates slightly to the Single Responsibility Principle in that interfaces should only have a single responsibility. If not, they're going to have excess method baggage that client code must also couple with.

Consider for a moment the following interface:

```
interface LoggerInterface {
  public function write($message);
  public function read($messageCount);
}
```

This interface defines a logging mechanism, but leaves the details up to the concrete implementations. We have a mechanism to write to a log in `write()`, and a mechanism to read from the log file in `read()`.

Our first implementation of this interface might be a simple file logger:

[16] http://www.objectmentor.com/resources/articles/isp.pdf

```php
class FileLogger implements LoggerInterface {
  protected $file;

  public function __construct($file) {
    $this->file = new \SplFileObject($file);
  }

  public function write($message) {
    $this->file->fwrite($message);
  }

  public function read($messageCount)
  {
    $lines = 0;
    $contents = [];

    while (!$this->file->eof()
      && $lines < $messageCount) {

      $contents[] = $this->file->fgets();

      $lines++;
    }

    return $contents;
  }
}
```

As we continue along, though, we decide we want to log some critical things by sending via email. So naturally, we add an `EmailLogger` to fit our interface:

```php
class EmailLogger implements LoggerInterface {
  protected $address;

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

  public function write($message) {
    // hopefully something better than this:
    mail($this->address, 'Alert!', $message);
  }

  public function read($messageCount)
  {
     // hmm...
  }
}
```

Do we really want our application connecting to a mailbox to try to read logs? And how are we even going to sift through the email to find which are logs and which are, well, emails?

It makes sense when we're doing a file logger that we can easily also write some kind of UI for viewing the logs within our application, but that doesn't make a whole lot of sense for email.

But since `LoggerInterface` requires a `read()` method, we're stuck.

This is where the Interface Segregation Principle comes into play. It advocates for "skinny" interfaces and logical groupings of methods within interfaces. For our example, we might define a `LogWriterInterface` and a `LogReaderInterface`:

```
interface LogWriterInterface {
  public function write($message);
}

interface LogReaderInterface {
  public function read($messageCount);
}
```

Now `FileLogger` can implement both `LogWriterInterface` and `LogReaderInterface`, while `EmailLogger` can implement only `LogWriterInterface` and doesn't need to bother implementing the `write()` method.

Further, if we needed to sometimes rely on a logger that can read and write, we could define a `LogManagerInterface`:

```
interface LogManagerInterface
  extends LogReaderInterface, LogWriterInterface {
}
```

Our `FileLogger` can then implement the `LogManagerInterface` and fulfill the needs of anything that has to both read and write log files.

Why does ISP matter?

The goal of the Interface Segregation Principle is to provide decoupled code. All client code that uses the implementation of an interface is coupled to all methods of that interface, whether it uses them or not, and can be subject to defects when refactoring within that interface occur, unrelated to what implementations it actually uses.

Dependency Inversion Principle

The **Dependency Inversion Principle** states that [17]:

> A. High level modules should not depend upon low level modules. Both should depend upon abstractions.

and that:

> B. Abstractions should not depend upon details. Details should depend upon abstractions.

This principle is very core to The Clean Architecture, and we'll discuss how it fits in great detail in that leading chapter.

Imagine a class that controls a simple game. The game is responsible for accepting user input, and displaying results on a screen. This `GameManager` class is a high level class responsible for managing several low level components:

```
class GameManager {
  protected $input;
  protected $video;

  public function __construct() {
    $this->input = new KeyboardInput();
    $this->video = new ScreenOutput();
  }

  public function run() {
    // accept user input from $this->input
    // draw the game state on $this->video
  }
}
```

This `GameManager` class is depending strongly on two low level classes: `KeyboardInput` and `ScreenOutput`. This presents a problem in that, if we ever want to change how input or output are handled in this class, such as switching to a joystick or terminal output, or switch platforms entirely, we can't. We have a hard dependency on these two classes.

If we follow some guidelines of the Liskov Substitution Principle, we can easily devise a system in which we have a `GameManager` that allows for the input and outputs to be switched, without affecting the output of the `GameManager` class:

[17] http://www.objectmentor.com/resources/articles/dip.pdf

```
class GameManager {
  protected $input;
  protected $video;

  public function __construct(
    InputInterface $input,
    OutputInterface $output
  ) {
    $this->input = $input;
    $this->video = $output;
  }

  public function run() {
    // accept user input from $this->input
    // draw the game state on $this->video
  }
}
```

Now we've inverted this dependency to rely on `InputInterface` and `OutputInterface`, which are abstractions instead of concretions, and now our high level `GameManager` class is no longer tied to the low level `KeyboardInput` and `ScreenOutput` classes.

We can have the `KeyboardInput` and `ScreenOutput` classes extend from these interfaces, and add additional ones, such as `JoystickInput` and `TerminalOutput` that can be swapped at run time:

```
class KeyboardInput implements InputInterface {
  public function getInputEvent() { }
}

class JoystickInput implements InputInterface {
  public function getInputEvent() { }
}

class ScreenOutput implements OutputInterface {
  public function render() { }
}

class TerminalOutput implements OutputInterface {
  public function render() { }
}
```

We're also utilizing what's known as Dependency Injection here, which we'll talk about in the next chapter, conveniently called Dependency Injection.

If we can't modify the input and output classes to conform to our interfaces, if they're maybe provided by the system, it would then be smart to utilize the Adapter Pattern we previously discussed to wrap these existing objects and make them conform to our interface.

Why does DIP matter?

In general, to reach a decoupled code base, one should get to a point where dependency only flows inward. Things that change frequently, which are the high level layers, should only depend on things that change rarely, the lower levels. And the lower levels should never depend on anything that changes frequently, which is the higher level layers.

We follow this philosophy to make it easier for change to happen in the future, and for that change to have as little impact upon the existing code as possible. When refactoring code, we only want the refactored code to be vulnerable to defects; nothing else.

Applying SOLID Principles

The SOLID principles work tightly together to enforce code that is easy to extend, refactor, and test, which ultimately leads to less defects and quicker turn around time on new features.

Just as we continued building on our Design Patterns in this chapter, we'll continue building on the principles of SOLID as we discuss Inversion of Control and the Clean Architecture later. The SOLID principles are the founding principles that make the Clean Architecture work.

Dependency Injection

One of the worst forms of coupling we encounter in object oriented programming deals with instantiating classes directly within other classes. The quickest way to find instances of this is to simply look for the `new` operator:

```
class CustomerController {
  public function viewAction() {
    $repository = new CustomerRepository();
    $customer = $repository->getById(1001);
    return $customer;
  }
}
```

In this `CustomerController`, there is a dependency on `CustomerRepository`. This is a hard, concrete dependency; without the existence of `CustomerRepository`, the `CustomerController` simply will not work without it.

Instantiating dependencies within classes introduces several problems:

1. **It makes it hard to make changes later**. Refactoring is difficult when classes manage their own dependencies. If we wanted to change the method by which we retrieved data from the database, such as switching out the underlying library, or switching to a different database storage, we'd have to find all the instances within our code where we declared a `CustomerRepository` and make those changes. This would be a tedious and error-prone process.
2. **It makes it hard to test**. In order for us to write any kind of unit tests for the `CustomerController` class, we have to make sure that not only `CustomerRepository` is available to the test, but all dependencies that `CustomerRepository` relies on – such as a database full with testable data – have to be available as well. Now we're doing full stack testing instead of simple unit testing. Coupling makes it very hard to test components in isolation.
3. **We have no control over dependencies**. In some instances, we might want a class dependency to be configured differently, depending on various circumstances, when it is used within another class. This becomes very awkward to develop when a class is declaring and configuring the declaration of its own dependencies.

These can turn out to be some pretty big problems, especially in larger applications. It can severely inhibit a developer from making even small changes to an application in the future.

Inversion of Control

Inversion of control is the process by which the instantiation and configuration of dependencies is moved from the dependent class and given instead to some external means. There are several of these "external means," and we'll talk about two of the most popular: the *service locator pattern* and *dependency injection*.

Using a Service Locator

One method of achieving inversion of control is to use the Service Locator pattern. A **Service Locator** is a registry of resources that the application can use to request dependencies without instantiating them directly. When using this pattern, our code simply pulls its dependencies out of the service locator registry.

```
public function viewAction() {
  $repository = $this->serviceLocator->get('CustomerRepository');
  $customer = $repository->getById(1001);
  return $customer;
}
```

Now, instead of instantiating its own dependencies outright, this controller pulls the dependent object out of the service locator. Depending on the implementation or framework that you use, you'll probably have some code that registers `CustomerRepository` with the service locator, so that the service locator knows what to return to the calling code:

```
$serviceLocator->setFactory('CustomerRepository', function($sl) {
  return new Path\To\CustomerRepository(
    $sl->get('Connection')
  );
});
```

In this theoretical service locator, an anonymous function is registered with the key `CustomerRepository`, and that function is responsible for building and configuring the repository and returning it. Further, the `CustomerRepository` itself has a dependency on something called `Connection` that we'll just assume is defined elsewhere. The point here is to know that dependencies can have dependencies of their own. When they do, instantiating them directly when needed becomes even more overwhelming.

This code provides a couple several benefits.

1. There is now only have one place that is responsible for instantiating a `CustomerRepository` so that whatever code needs it will always have a properly configured repository. If we ever need to change anything about *how* that repository is created, there is only one place to go to do so. This makes refactoring the code base much easier.
2. Another benefit is that it makes the `CustomerController` code much easier to test. When testing the controller above, the test code can simply give it a different implementation of the service locator, one that has a mocked repository that can control what data it returns.

```
$serviceLocator->setFactory('CustomerRepository', function() {
  return new Path\To\MockCustomerRepository([
    1001 => (new Customer())->setName('ACME Corp')
  ]);
});
```

This `MockCustomerRepository` simply returns the test data "stored" based on customer ID, so that when testing the controller, it will return a `Customer` object with the name of *ACME Corp*. The controller is now tested separately from the actual repository, and the test is now only concerned with what is returned, and not how it is retrieved.

This code is still only marginally better than the factory example and the direct instantiation of the original code:

1. It still requests its own dependencies as needed through the service locator. This is better than instantiating the dependencies directly as at least we have control over what is actually instantiated within the service locator, which should make refactoring and testing easier.
2. In order to test, we have to go through the cumbersome process of setting up a fake service locator and registering required services with it just to get the controller to work.

Using Dependency Injection

Dependency Injection is a process by which a dependency is injected into the object that needs it, rather than that object managing its own dependencies.

Another way to think of dependency injection is to call it "third-party binding," as some third party code is directly responsible for providing the dependency to the class. As Nat Pryce describes it[18], the use of the term injection can be seen as a misnomer, as it isn't really injected into the code, but instead declared as part of the code's API.

There are two methods of dependency injection that we'll discuss:

1. **Setter injection** - Using this method, dependencies can be provided to the object through a set method on the class, which would be stored for later use in a class member variable.
2. **Constructor injection** - Using this method, dependencies are provided to the object through its constructor, which would also be stored for later use in a class member variable.

Using Setter Injection

Using Setter Injection, we would update the mechanism responsible for instantiating the `CustomerController` object (either a routing or dispatching process in our framework) to call some new set methods and provide the object with its dependencies:

[18]http://www.natpryce.com/articles/000783.html

```
$controller = new CustomerController();
$controller->setCustomerRepository(new CustomerRepository());
$customer = $controller->viewAction();
```

And of course, the `CustomerController` would be updated to have this `setCustomerRepository` method:

```
class CustomerController {
  protected $repository;

  public function setCustomerRepository(CustomerRepository $repo) {
    $this->repository = $repo;
  }

  public function viewAction() {
    $customer = $this->repository->getById(1001);
    return $customer;
  }
}
```

By the time the `viewAction()` method is called, this class should have been injected an instance of `CustomerRepository` through the `setCustomerRepository` method. Now the processes of retrieving dependencies has been completely removed from the `CustomerController` class.

When testing, we can see how much easier it is to mock the repository to provide a stable testing state:

```
$repository = new MockCustomerRepository([
  1001 => (new Customer())->setName('ACME Corp')
]);
$controller = new CustomerController();
$controller->setCustomerRepository($repository);
$customer = $controller->viewAction();

assertEquals('ACME Corp', $customer->getName());
```

There is still one drawback here, though, and that is that using setter injection does not make the dependencies required. This puts us in a situation that can lead to hard-to-detect defects when we forget to inject some dependency:

```
$controller = new CustomerController();
$customer = $controller->viewAction();
```

This code will throw errors as, without calling the `setCustomerRepository()` method, the class `$repository` variable will be null when it is used. Given how small this application is, we'll likely find the problem easily, but in a larger system with more involved code, this could lead to a rough time debugging where something went wrong.

Using Constructor Injection

Using Constructor Injection, the problem presented with setter injection is solved in that, in order for the object to be instantiated, the dependencies must be injected via the constructor. If they aren't, a very helpful, specific fatal error is thrown by PHP.

Providing that dependency would look something like this:

```php
$controller = new CustomerController(new CustomerRepository());
$customer = $controller->viewAction();
```

The `CustomerController` is updated to require an instance of `CustomerRepository` be passed along to the constructor. We use type-hinting here so that an actual instance of `CustomerRepository` is given, not just any variable.

```php
class CustomerController {
  protected $repository;

  public function __construct(CustomerRepository $repo) {
    $this->repository = $repo;
  }

  public function viewAction() {
    $customer = $this->repository->getById(1001);
    return $customer;
  }
}
```

The `CustomerController` is now guaranteed to have an instance of `CustomerRepository` ready to be used. This controller can test this just like it was with setter injection using a mocked repository, except it would be provided to the constructor rather than the set method:

```php
$repository = new MockCustomerRepository([
  1001 => (new Customer())->setName('ACME Corp')
]);
$controller = new CustomerController($repository);
$customer = $controller->viewAction();

assertEquals('ACME Corp', $customer->getName());
```

Dependency injection helps to both loosely couple our code base, as well as provides the ability to test individual components in isolation, without having to construct, prepare, and pass around their individual dependencies. Since constructor injection forces the object to be injected with its dependencies, it will be preferred over setter injection throughout the rest of this book.

When to use Dependency Injection

Now that we've discussed what dependency injection is and the problems it can help solve in your code, it's important to take a moment think about where it is appropriate to use this technique, and where it is not. This is a bit of a blurry line, and whoever you talk to, you'll probably get a different opinion.

Here's where I think you should use dependency injection:

1. **When the dependency is used by more than one component**. If it's being used by many things, it probably makes sense to abstract out the instantiation and configuration and inject the dependency, rather than doing it when needed. Without this, we make it hard to refactor our code that uses this dependency. However, if the dependency is only used once, you then only have one place to go when refactoring the code.
2. **The dependency has different configurations in different contexts**. If a dependency is configured differently in the different places it is used, then you definitely want to inject it. Again, this configuration shouldn't take place in the dependent class. Not only does it make refactoring difficult, but can make testing difficult as well. You'll probably also want to invest in a real factory that can create these dependencies under different scenarios to abstract further the creation of those dependencies.
3. **When you need a different configuration to test a component**. If testing a dependent object requires that the dependency needs to be configured differently to be tested, you'll want to inject it. There isn't any other good way around this. Think back to our example of mocking the `CustomerRepository` to control what data it returned.
4. **When the dependency being injected is part of a third party library**. If you're using someone else's code, you shouldn't be instantiating it directly within the code that depends on it. These dependencies should absolutely be injected into your code. The usage of someone else's code is something we might refactor often. Special attention needs to be paid to this scenario, and we'll discuss that in detail in Abstracting with Adapters using the Adapter Pattern we've already discussed.

But there are definitely some scenarios where you probably don't want to use dependency injection.

If a dependency doesn't need to be configured in anyway (meaning, no constructor arguments or set methods that need to be called to set it up), and doesn't have dependencies itself, it might be safe to go ahead and just instantiate that dependency within the class that needs it. This is especially true when the dependency is the component within the same layer of the application (we'll get to talking about various layers of software in a little bit).

For example, if our framework requires that our controller actions return some kind of view or response object, it makes perfect sense to just instantiate and return that right in the action.

```
public function viewAction() {
  $customer = $this->repository->getById(1001);

  return new ViewModel([
    'customer' => $customer
  ]);
}
```

There is little gained value from injecting this `ViewModel` class into the controller. It has no configuration, it has no dependencies itself, and in the case of testing, we probably just want to verify that an instance of `ViewModel` was returned with an instance of `Customer` stored within it.

Using Factories to Create Dependencies

If we have a dependency that needs configuration, it doesn't always make sense to inject that dependency itself. Sometimes it might be a better idea to inject an object that knows how to build this dependency for us. This is useful when the way in which that dependency is configured is controlled by scenarios within the class itself.

Looking at the example above: what if we wanted to return a different type of response based on a context requested? Maybe by default we would return HTML, but based on context, might return JSON or XML as well?

In this case, we may want to inject a factory that knows how to build the response based on the requested context:

```
class CustomerController {
  protected $repository;
  protected $responseFactory;

  public function __construct(
    CustomerRepository $repo,
    ResponseFactory $factory
  ) {
    $this->repository = $repo;
    $this->responseFactory = $factory;
  }

  public function viewAction() {
    $customer = $this->repository->getById(1001);
    $response = $this->responseFactory->create($this->params('context'));
    $response->setData('customer', $customer);
    return $response;
  }
}
```

Previously, our dependency was the specific response we were returning. Now, however, it's this new `ResponseFactory` class that builds a response for us.

The important thing here is that we are abstracting out the logic that determines how to build the dependency, and instead depending on that abstraction. We're also injecting that in so that if the logic or methodology ever changes, we just have to change the implementation instead of updating an unknown number of locations that were previously building their own responses.

Handling Many Dependencies

It's very easy to let dependency injection get out of control:

```
public function __construct(
  CustomerRepository $customerRepository,
  ProductRepository $productRepository,
  UserRepository $userRepository,
  TaxService $taxService,
  InvoiceFactory $invoiceFactory,
  ResponseFactory $factory,
  // ...
) {
  // ...
}
```

This is usually indicative of having a class that violates the Single Responsibility Principle and does too much and contains too much logic. There's no hard rule about how many dependencies a class can have. Generally, the fewer the better, however, it's entirely possible to go too far in the other direction as well, and end up with a hierarchy of tiny classes that are hard to follow, refactor and test.

After understanding the concepts of this book, you can start to use the "Feels Bad" policy of knowing when to refactor a bad situation into good, healthy code.

Are we still coupling?

Our whole reason for going down this path of dependency injection was to reduce coupling in our code. Recall that our initial example, pre-inversion of control looked like this:

```
class CustomerController {
  public function viewAction() {
    $repository = new CustomerRepository();
    $customer = $repository->getById(1001);
    return $customer;
  }
}
```

In this code we are highly coupled to the `CustomerRepository` as we are declaring a concrete dependency right in the middle of our code. By switching to using dependency injection, we ended up with this:

```
class CustomerController {
  protected $repository;

  public function __construct(CustomerRepository $repo) {
    $this->repository = $repo;
  }

  public function viewAction() {
    $customer = $this->repository->getById(1001);
    return $customer;
  }
}
```

Now we're being provided *some kind of class* that is an instance of `CustomerRepository`. This is much looser coupling, but it's still coupling. We still need something that is or extends from `CustomerRepository`. There's no real way around that. And since this repository will likely sit within our persistence implementation, we're also coupling to a whole infrastructure layer that talks to a database.

There is one level farther we can go with decoupling our dependencies by using interfaces to define contracts.

Defining a Contract with Interfaces

In the previous chapter we discussed the principle of inversion of control, and describe how dependency injection can make refactoring and testing code easier. We started out by discussing object coupling, and presented dependency injection as a method of limiting coupling. However, when we reached the end, we realized that we hadn't entirely removed the coupling, we only made it weaker by moving it to the constructor from the methods of the class.

```
class CustomerController {
  protected $repository;

  public function __construct(CustomerRepository $repo) {
    $this->repository = $repo;
  }

  public function viewAction() {
    $customer = $this->repository->getById(1001);
    return $customer;
  }
}
```

This class is still well coupled to the `CustomerRepository` class. Since the `CustomerRepository` class is responsible for, at least through its dependencies, connecting to and retrieving data from a database, this means that the `CustomerController` is also coupled to that database.

This can be problematic if we ever decided to switch data sources, such as moving to a NoSQL variant, or switching to some API service to provide our data. If we do that, we'll have to then go modify all the code across our entire code base that uses this `CustomerRepository` to make it use something else.

Further, if we want to test this `CustomerController`, we still have to give it an instance of `CustomerRepository`. Any mock object we create will need to extend from `CustomerRepository` to pass the type-hint check. That means our simple mock repository will still have to have the entire database back-end tied to it, even if we are overriding everything it does. That's pretty messy.

Interfaces in PHP

Recall that in PHP, an **interface** is a definition of a class, without the implementation details. You can think of it as a skeleton of a class. An interface cannot have any method bodies, just method signatures.

```
interface Automobile {
  public function drive();
  public function idle();
  public function park();
}
```

Any class implementing an interface *must* implement all the methods of that interface.

```
class Car implements Automobile {
  public function drive() {
    echo "Driving!";
  }

  public function idle() {
    echo "Idling!";
  }

  public function park() {
    echo "Parking!";
  }
}
```

Any number of implementations may exist for the interface `Automobile`:

```
class DumpTruck implements Automobile {
  public function drive() {
    echo "Driving a Dump Truck!";
  }

  public function idle() {
    echo "Idling in my Dump Truck!";
  }

  public function park() {
    echo "Parking my Dump Truck!";
  }
}
```

The two classes `Car` and `DumpTruck` are considered compatible as they both define the `Automobile` interface, and either could be used in any instance where an `Automobile` is necessary.

This is known as **polymorphism**, where objects of different types can be used interchangeably, so long as they all inherit from a common subtype.

Using Interfaces as Type Hints

The usage of interfaces comes in handy when trying to reduce coupling within a class. We can define an interface of some dependency, and then reference only that interface. So far, we've been passing around concrete instances of `CustomerRepository`. Now, we'll create an interface that defines the functionality of this repository:

```
interface CustomerRepositoryInterface {
  public function getById($id);
}
```

We have a simple interface with one method, `getById()`, which returns a `Customer` object for the customer data identified by `$id`. As this is an interface, we cannot provide any implementation details, so this class provides no information about where the data comes from, or how it is retrieved.

Now in our controller, we use PHP's type-hints for methods and functions to declare that the argument passed to the `__construct()` method *must* be an instance of our new interface, `CustomerRepositoryInterface`.

```
class CustomerController {
  protected $repository;

  public function __construct(CustomerRepositoryInterface $repo) {
    $this->repository = $repo;
  }

  public function viewAction() {
    $customer = $this->repository->getById(1001);
    return $customer;
  }
}
```

Now the `CustomerController` class is only coupled to the `CustomerRepositoryInterface`, but that's okay: this interface isn't a concrete implementation, it's just a definition of an implementation. We can couple to this and, in fact, we should, as it defines how our application interacts, without referencing the specific implementations.

Whatever mechanism is responsible for instantiating the `CustomerController` can still provide it with the concrete `CustomerRepository`, so long as that class implements the 'CustomerRepositoryInterface'.

```
class CustomerRepository implements
  CustomerRepositoryInterface {

  public function getById($id) {
    // get and return the customer...
  }
}
```

The `CustomerRepository` provides the implementation of the `getById()` method and fulfills the requirements of the interface.

When we want to test this controller now, we can instead use a mock instance of `CustomerRepositoryInterface:`

```
class MockCustomerRepository implements
  CustomerRepositoryInterface {

  public function getById($id) {
    if ($id == 1001) {
      return (new Customer())
        ->setId(1)
        ->setName('Customer #1001');
    }
  }
}
```

While this may not be the greatest code (and using a mocking library like Mockery[19] or the mocking utilities provided by PHPUnit would be better[20]), we can nevertheless pass this in to `CustomersController` to fulfill the `CustomerRepositoryInterface` type hint. Now the controller is being tested in isolation, without the specific configuration and dependencies of the real `CustomerRepository`.

The `CustomersController` really doesn't care what object you end up injecting it with. So long as that object meets the required interface (and it will, otherwise a fatal error will be thrown), the controller should function just the same. This also assumes, of course, that the dependencies being injected actually work and return the data they should.

The Liskov Substitution Principle

If this sounds familiar, it should. We already discussed these principles when we talked about the **Liskov Substitution Principle**. Our various concrete repositories are interchangeable as they both extend the same interface.

The **Dependency Inversion Principle** also applies here, as we're modifying our high level code (the controller in our example) to not depend upon low level code (the repository), and instead depend upon an abstraction.

[19]https://github.com/padraic/mockery
[20]http://phpunit.de/manual/4.1/en/test-doubles.html

Using Interfaces as a Contract

Another way to think about using interfaces with dependency injection is that they are fulfilling a contract. Our interface is a contract between the supplier, our code instantiating a dependency and injecting it, and our client, the class with a dependency need. The contract is fulfilled when the correct object is injected into the object.

This concept has been described as **programming by contract**. It's an interesting way to think about interfaces and dependency injection.

Making Third Party Code Conform to Contracts

Using interfaces to define contracts is easy when it's our own code, but how do we make use of a third party library and make it conform to an interface for dependency injection? After all, we shouldn't simply open up the third party source code and modify it to extend from one of our interfaces.

The answer is to use the Adapter Pattern.

Abstracting with Adapters

Interfaces have provided the means to completely decouple our code from concrete implementations of their dependencies. So far, we've only show how to do this with our own low level code. What if we want to use some third party library instead?

Let's say we find a great third party library through Packagist called *Bill's Geocoder* that validates addresses with Google Maps or USPS or some other service.

```
class AddressController extends AbstractController {
  protected $geocoder;

  public function __construct(BillsGeocoder $geocoder) {
    $this->geocoder = $geocoder;
  }

  public function validateAddressAction() {
    $address = $this->vars()->fromPost('address');
    $isValid = $this->geocoder->geocode($address) !== false;
  }
}
```

We're using some dependency injection into the controller, which is great. This is a step in the right direction and solves some problems for us, but it's still strongly coupling our controller to whoever Bill is. What if he goes away? What if you figure out `DavesGeocoder` is so much better because it supports Zip+4, which `BillsGeocoder` didn't? And what if you just happen to use this geocoder all over the place and now you have to go update all those references? What if `DavesGeocoder` doesn't have a `geocode()` method but instead has `validateAddress()`. You've run into a refactoring nightmare.

Setting up the Adapter

Recall back to our discussion on design patterns, specifically the Adapter Pattern. Adapters are perfectly suited to solve this problem as they allow us to "wrap" the functionality of the third party code, and by doing so, make it conform to an interface we define, so that we can inject that adapter to fulfill the interface.

This is exactly what we did when we discussed the Adapter Pattern. We started by defining our interface:

```
interface GeocoderInterface {
  public function geocode($address);
}
```

Then, we'll go ahead and make our controller depend only upon this interface:

```
class AddressController extends AbstractController {
  protected $geocoder;

  public function __construct(GeocoderInterface $geocoder) {
    $this->geocoder = $geocoder;
  }

  public function validateAddressAction() {
    $address = $this->vars()->fromPost('address');
    $isValid = $this->geocoder->geocode($address) !== false;
  }
}
```

Finally, we'll create an adapter to wrap `BillsGeocoder` and make it conform to our `GeocoderInterface` that is required by our `AddressController` class:

```
class BillsGeocoderAdapter implements GeocoderInterface {
  protected $geocoder;

  public function __construct(BillsGeocoder $geocoder) {
    $this->geocoder = $geocoder;
  }

  public function geocode($address) {
    return $this->geocoder->geocode($address);
  }
}
```

In our `geocode()` method, we're simply passing off the processing to our instance of `BillsGeocoder`, which we take through the constructor.

We can use dependency injection to inject an instance of `BillsGeocoderAdapter` into our `AddressController`, which allows us to use a third party library but makes sure it conforms to the interface we need.

How does this help?

This method of using adapters with third party libraries allows us to remain decoupled and free from dependence on those third party libraries. It allows us to freely swap out those dependencies without having to rewrite any code that uses them, and it allows us to easily test our application and its use of those dependences without actually having to test those dependencies ourselves. We only have to test that we're properly utilizing them.

We'll later discuss the importance of External Agency Independence when we discuss The Clean Architecture.

The Clean Architecture

We've explored some messy architecture and some all around bad code. We've also explored some great design patterns and development principles that can lead to good, quality code.

Now we're going to take a look at some good architectural decisions that utilize these design patterns and principles to write good, clean, solid code.

This architecture is called the **Clean Architecture**.

MVC, and its Limitations

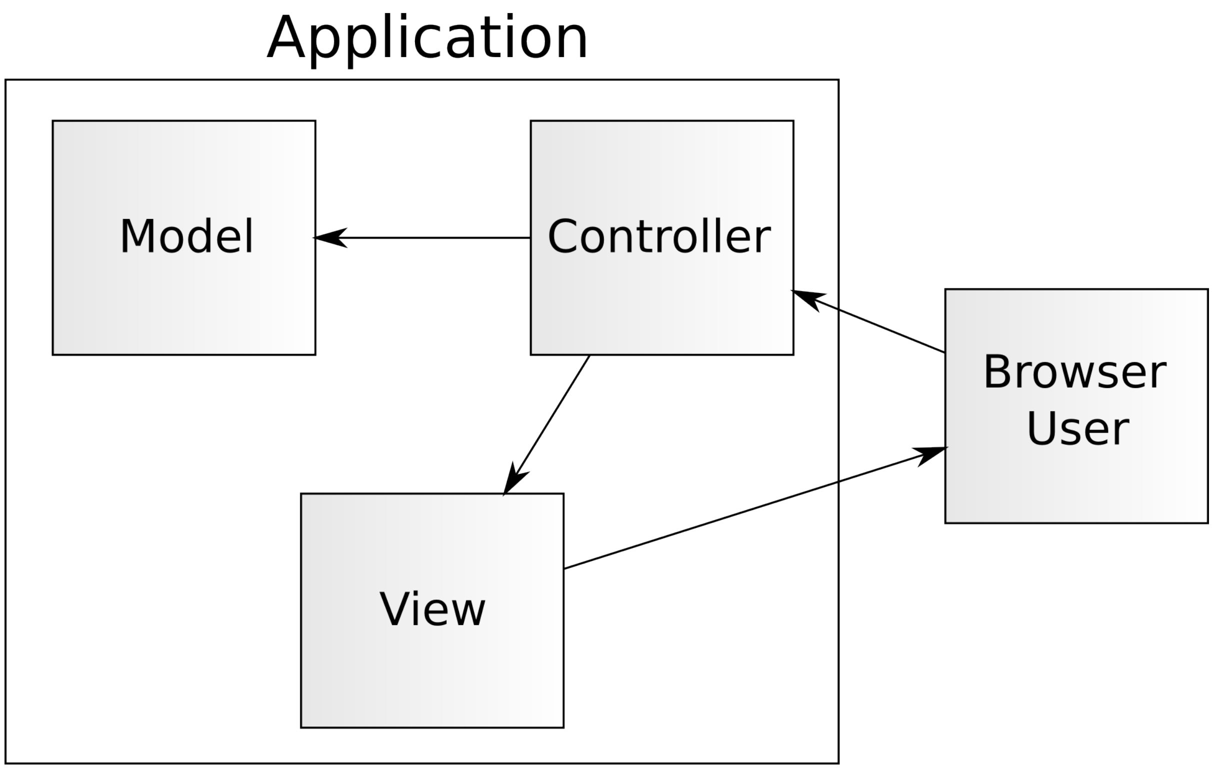
When building an application, there are often several different things going on. There's the HTML, CSS and JavaScript that presents your application to the user. There's usually an underlying data source, whether it's a database, an API, or flat files. Then there's the processing code that goes in between. The code that tries to figure out what the user requested, how to act upon their request, and what data to display to them next. Finally, there's also the business rules of the application. The rules that dictate what the application does, how things relate to one another, and what the confines of those relationships are.

When one first starts out attempting to better their code (or maybe they're lucky enough to learn of it straight on), they quickly come across the MVC architecture. The **Model-View-Controller** architecture dictates a strong separation of concerns by separating the database logic, control/processing logic, and the view/UI logic. The MVC architecture does have some faults, which we'll discuss in the next chapter, but it does provide a pretty good framework for cleanly separating code.

For those who think they already know about the MVC architecture and don't need a refresher, feel free to skip this chapter and head on to the next.

MVC in a Diagram

Let's have a look at a pretty picture of MVC. It commonly looks like this:



Briefly, the **controller** is the section of the codebase that analyzes a request and determines what to do. The user kicks off a controller by hitting a specific URL in their web browser, which then routes to the specific controller designed to handle that request (we'll talk about routing in a bit).

The controller then manipulates some **models**, which are representations of data. For instance, if we hit a controller that is meant to save a new user, the controller would populate a `User` model with the data supplied in the post, and then persist it to the database.

Finally, the controller returns a **view** to the user, which is the PHP, HTML, CSS, JavaScript, and images that represent the request. In our example, after creating the new user, we might redirect off to a View User action which would display our new user's information and give us further actions to take upon that user.

The MVC Components

The best way to discuss the MVC architecture is to discuss each component individually. We'll start with Model and View, because, hey, that's how the acronym goes, and we'll end with the Controller, which is quite fitting as it ties the whole thing together, as you'll see.

The Model

The Model in MVC is the portion of your application that represents your data. A specific model is a class that represents data. Consider this `User` class:

```
class User extends AbstractModel {
  public $id;
  public $alias;
  public $fullName;
  public $email;
}
```

This is a model representing a user in our system. Usually the model layer has some means for creating and hydrating these model records, as well as persisting them to the actual data storage. We'll talk about this in more detail in Database Independence.

As a developer, we can manipulate this model like any PHP class:

```
$user = new User();
$user->alias = 'billybob';
$user->fullName = 'William Bob';
$user->email = 'william.bob@bobcorp.com';
```

Compare this to traditional use of `PDO` or the straight `mysql_*` or `mysqli_*` methods: here, we're using fully backed objects that represent our data, rather than querying for it and dealing with arrays of data. We actually have representational data using this method.

The View

The View in MVC is simply what is presented to the user. In the PHP world, it is mostly composed of the HTML, CSS, and JavaScript that drive the UI. The view is also responsible for the user interaction with the application, through the use of links, buttons, JavaScript, etc. These actions may be handled entirely in the view layer, or they may make additional requests to the web server to load other data and views.

The View is also responsible for taking models and representing them to the user of the application. For instance, for our `User` model, we may have a page that iterates through a collection of users and displays them in a grid:

```
<table>
  <thead>
    <tr>
      <th>ID</th>
      <th>Full Name</th>
      <th>Email</th>
      <th>&nbsp;</th>
    </tr>
  </thead>
  <tbody>
    <?php foreach ($this->users as $user): ?>
    <tr>
```

```
      <td><?= $user->id ?></td>
      <td><?= $user->fullName ?></td>
      <td><?= $user->email ?></td>
      <td>
        <a href="/users/edit/<?= $user->id ?>">
          edit</a>
      </td>
    </tr>
    <?php endforeach; ?>
  </tbody>
</table>
```

Our view layer is very purposeful: it is meant to display data to the user. It does no processing outside of simple loops and conditionals. It doesn't query anything directly, just manipulates the data that is given to it.

But just how does the view get its data?

The Controller

The controller is responsible for interpreting the user request and responding to it. It can load specific models relevant to the request and pass it off to a view for representation, or it can accept data from a view (via something like an HTTP POST request) and translate it to a model and persist it to the data storage.

Controllers come in many forms, but one of the most common form is an action controller. These controllers are classes that contain one or more methods, each method representing a specific request.

If we continue our user example, we might have a `UsersController` that is responsible for dealing with requests relevant to users:

```
class UserController extends AbstractController {
  public function indexAction() {}
  public function viewAction() {}
  public function updateAction() {}
}
```

This example controller has three actions:

- `indexAction` is responsible for listing all Users
- `viewAction` is responsible for viewing a User
- `updateAction` is responsible for updating a User

On a specific request, let's say the view customer request, the corresponding action will be called, which would process the request and prepare the required view. This might look something like:

```php
public function viewAction() {
  $id = $this->params('id');
  $user = $this->repository->getByid($id);

  $view = new View();
  $view->setFile('users/view.phtml');
  $view->setData(['customer' => $user]);

  return $view;
}
```

This pseudo-controller action in this sample code retrieves the passed ID from some mechanism, then uses the stored `UserRepository` to retrieve that user. Finally, it instantiates a new view, sets the view file to render, and passes the data off using `setData()`.

Here, we can see that the controller only cares about responding to requests. It uses the model layer to retrieve data, and then passes it off to the view layer for processing and display.

Routing

This all starts to make much more sense when you consider how routing works in web-based MVC applications. We tend to lean towards using clean URLs these days, where our URI looks like this:

```
/users/view/1
```

Or, if you're defining RESTful URIs:

```
/users/1
```

This is a URI that would route to the `UserController`'s `viewAction` in our examples above. Traditionally when routing clean URLs to controllers, the first part of the URI, `/users` maps to the controller, `UsersController` in our example, while the part, `/view`, maps to the action, which, of course, is the `viewAction`.

This isn't always the case, however. Must frameworks allow routing to be whatever you make it, such that any URI can map to any controller or action.

Some frameworks make this explicit and do not require any additional setup. They simply map the URI to a corresponding controller and action. Most modern frameworks require you to setup some sort of routing table that tell it which URIs map to which portions of code.

In Zend Framework 2, that looks something like this:

```
return [
  'router' => [
    'routes' => [
      'user' => [
        'type' => 'Literal',
        'options' => [
          'route' => '/users',
          'defaults' => [
            'controller' => 'App\Controller\Users',
            'action' => 'index'
          ]
        ],
        'may_terminate' => true,
        'child_routes' => [
          'view' => [
            'type' => 'Segment',
            'options' => [
              'route' => '/view/[:id]',
              'defaults' => [
                'action' => 'view'
              ],
              'constraints' => [
                'id' => '[0-9]+',
              ]
            ]
          ]
        ]
      ]
    ]
  ]
];
```

This long-winded block of code defines two routes, `/users`, which routes to `UsersController::indexAction()`, and `/users/view/[:id]`, which routes to `UsersController:viewAction()`. Both are `GET` requests.

You can see how flexible this can be in defining routes as they don't have to match the controller structure whatsoever. But it is pretty verbose.

Laravel, on the other hand, takes a much simpler approach to routing:

```
Route::get('user/view/{id}', function($id) {
  return 'Viewing User #' . $id;
});
```

This routing is much simpler and expressive. Any time a URI matching `/users/view/{id}` is hit, the anonymous function runs and returns `Viewing User #{id}`.

MVC Isn’t Good Enough

The MVC architecture is a great start to building robust and adaptable software, but it isn’t always enough. With only three layers in which to organize all code, the developer usually ends up with too many details in one of the layers. It’s fairly presumptuous to think that everything should be able to fit into three buckets. Either it’s a model, or a view, or a controller. The view is usually saved from being slaughtered as its role is pretty well defined, so either the controllers become overwhelmed with business logic, or the model absorbs it all. The community has for quite some time adopted the mantra of “fat model, skinny controller.”

Obese Models

This “fat model, skinny controller” mantra is all well and good, until the model layer also becomes the database abstraction and persistence layer, and now the core business logic is tightly coupled to a data source. It ultimately becomes the **obese model** approach. This is bad news, as it makes it difficult for us to swap out that data layer, either the actual database itself or the library that powers the abstraction of it.

As it doesn’t make any sense to put database configuration and interaction in the view layer, and it becomes messy and not reusable to place it within controllers, everything related to the database, querying, and the representation of the database, the model or entity, gets shoved into the model. However, this tightly coupling of the representative data model to the actual data source is problematic.

Let’s say, for example, through the first iteration of our application, that we store and retrieve all of our data from a relational database system. Later down the road, however, our needs and number of applications might grow, and we might decide to build a web service API to manage the interaction of data with all the applications. Since we tightly coupled our business logic and data access together in the form of a model, it becomes difficult to switch over to our API without having to touch a lot of code. Hopefully we wrote a big test suite to help us.

Maybe instead of switching primary data sources, you simply find a much better database abstraction library that you want to use. Perhaps it is a well written library that better optimizes queries, saves resources, executes faster, and is easier to write. These are some great reasons to switch. However, if you initially went down a path of merging your database implementation details with your data representation, you might end up having to rewrite the whole entire model layer just to switch to a better library.

This becomes a clear issue of violating the Single Responsibility Principle.

A nice, clean model might look like this:

```
class User {
  public $id;
  public $alias;
  public $fullName;
  public $email;
}
```

How exactly does data get into these variables from the database, and how do we save changes we make back to that database? If we add in methods and mechanisms to this `User` class to persist our data to the database for us, we're then making it very hard to test and switch data sources or database abstract layers later. We'll cover various approaches and solutions to this problem in Database Independence.

Model Layer vs Model Class vs Entities

The solution to this problem is to realize there's a difference between the Model Layer of MVC, and the Model Class. What we have in our example above is a Model class. It's a representation of the data. The code responsible for actually persisting that data to the database storage is part of our model layer, but **should not** be part of our model class. We're mixing concerns there. One concern is a representative model of the data, and the other is the persistence of that data.

From now on, we're going to refer to the actual model class, like the one above, as an entity. **Entities** are simply representational states of things that have identities and attributes unique to that identity. For instance, an Order, User, Customer, Product, Employee, Process, Quotes, etc., can all be entities.

We're also going to stop referring to the model layer as the persistence layer. The **persistence layer** is simply the layer of code that is responsible for persisting data, entities, back to the data store, as well as retrieving entities from the data store based on their identity.

From here-forth, you can pretend like the word model doesn't exist, and with that, we can drop all baggage of the obese model.

More Layers for All of the Things!

Of course our solution here is to really add another layer to the MVC paradigm. We now have EPVC, which might stand for **Entity-Persistence-View-Controller** if that were a thing. This doesn't mean that every problem can be solved simply by throwing another layer at the problem. But it does make sense to split up our representation of the data with the persistence of the data as they really are two different things.

Doing this allows us to move the database away from being the core of our application to being an external resource. The entities now become the core, which leads to an entirely different way of thinking about software applications.

The Clean Architecture

So if MVC isn't enough, if it doesn't give us enough organization and segregation of our code base, what is the solution, and what is enough?

The Clean Architecture

The solution to this problem is what I'm going to refer to as The Clean Architecture. Not because I named it so, as near as I can tell that honor goes to "Uncle" Bob Martin who wrote in 2012[21] about a collection of similar architectures that all adhered to very strong forms of separation of concerns, far beyond what traditional MVC describes.

Uncle Bob describes these architectures as being:

> **Independent of Frameworks**. The architecture does not depend on the existence of some library of feature laden software. This allows you to use such frameworks as tools, rather than having to cram your system into their limited constraints.
>
> **Testable**. The business rules can be tested without the UI, Database, Web Server, or any other external element.
>
> **Independent of UI**. The UI can change easily, without changing the rest of the system. A Web UI could be replaced with a console UI, for example, without changing the business rules.
>
> **Independent of Database**. You can swap out Oracle or SQL Server, for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.
>
> **Independent of any external agency**. In fact your business rules simply don't know anything at all about the outside world.

Framework Independence

Framework independence is huge. When a developer initially starts using frameworks for their projects, they might think that the framework is the end game for their application. They make a choice and they're sticking with it. But that's a terrible decision.

Framework's live and die, and even when they don't die, they change, leaving the project that depends on them out in the cold. Take Zend Framework, for example. The release of Zend

[21]http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html

Framework 2 was such a massive change and shift, that it was near impossible to upgrade any ZF1 application to ZF2 without a complete rewrite. Especially if you based your domain around the `Zend_Model` family of classes.

Not only that, but new frameworks come out all the time. The PHP framework scene is more active than it has ever been, with previous unknowns like Laravel surging in popularity, and micro-frameworks, such as Silex, starting to get due respect. The best decision yesterday is not always the best decision today.

This is why it is the utmost importance to make sure that your applications are written as framework agnostic as possible. We'll talk about this in Framework Independence

Testable

Testability in applications of any size of are of extreme importance. Software tests are the single most important means to preventing regression errors in an application. To people who sell their software, regression bugs often mean warranty work and a loss of billable time. And for all software developers, bugs are simply a pain to deal with, often a pain to troubleshoot and fix, and, without tests, more often than not a guessing game.

When we write software, we refactor code quite often. In fact, the process of software development in general can be seen as an endless series of refactoring. As we continually write code, we're continually evolving the application, tearing down previous implementations and replacing them with newer, better ones, or enhancing them to account for some new feature set.

Since we're always changing what already exists, having a suite of tests that are readily available, fast to run, and comprehensive becomes very important. It can literally be the difference between quickly preventing many bugs in the first place and spending hundreds of hours trying to fix them later.

While this book doesn't have a dedicated chapter to testing, concepts discussed throughout the book will often discuss how they help the refactoring and testing process, and why they are good practices for developing a robust and highly available test suite.

Database Independence

Database independence may not seem like an important issue when building the architecture of an application. To some developers, it matters a great deal, such as those who are developing an open source project that users may want to deploy on a variety of database systems. But if you're developing in house software where the architecture is controlled, and often vetted and picked for very specific reasons, this doesn't seem like a big deal.

However, just like frameworks, databases can live and die. And just like frameworks, new platforms can spring up that suddenly look like a much better solution to your problem. Take for instance the rise of NoSQL alternatives to traditional relational database management systems (RDMS) in the past decade. For some situations, a NoSQL like Mongo or Couch might be a great alternative to MySQL or PostgreSQL. But when you've integrated MySQL into the heart of your application, it's a daunting task to try to rip it out and replace it with something else.

When writing a Customer Relationship Management (CRM) application, my colleagues and I went down the common path of using a relational database, PostgreSQL as our data backend. Through the course of developing the application over the next six months, we quickly realized that other applications and systems, including mobile device applications, were going to need to interface with this data.

After analyzing the situation, we decided that what we should really do is build a RESTful API and have the CRM application sit on top of it. The API would be a middle layer between the application and the database, and provide a common way of interacting with the data across all of the applications.

Since we had already developed a good chunk of the application, this could have proven challenging. However, since we had created a separate domain layer with interface contracts that our persistence layer adhered to, we were able to simply rewrite our small persistence layer to pull data from the API rather than from the database, and never had to touch any other code, like the controllers, services, etc.

We'll discuss how we can accomplish database independence in Database Independence.

External Agency Independence

The PHP ecosystem has recently exploded with a plethora of libraries and frameworks, most of which are now, thanks to PHP-FIG and Composer, easily plopped into every project you develop. These libraries are fantastic, and speed up the development of projects for you by providing proven solutions.

But should you use them? Absolutely! You just have to be careful how you do.

Just as these wonderful libraries have sprung up, they also die off and become forgotten just as fast. What could be more frustrating than having to refactor hundreds of files because you littered usage of someone else's library all over your code, and they decided to let it rot?

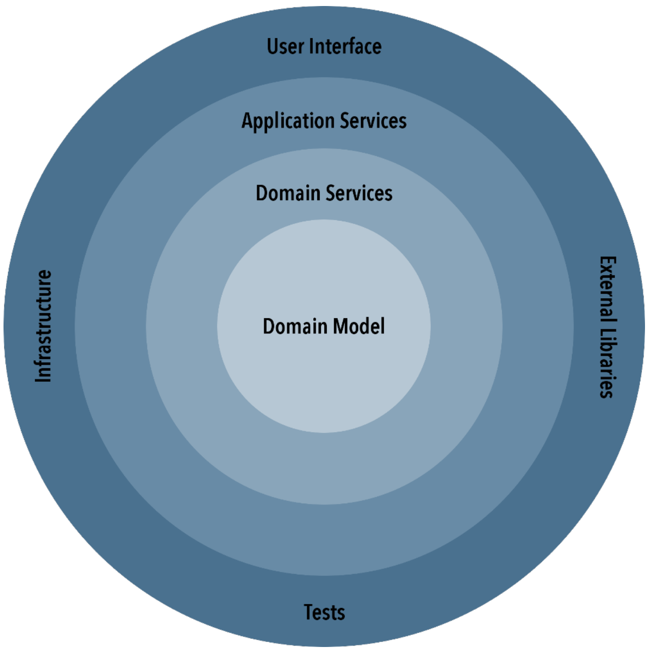
Using some tried and true design patterns, we can lessen this problem by wrapping our usage of these libraries. We'll discuss this in intricate detail in External Agency Independence.

The Onion Architecture

One off-shoot of the Clean Architecture, and the first one I came across before I found Uncle Bob's article, is The Onion Architecture[22], described by Jeffrey Palermo. While the naming of this architecture may make one think it is satire, it is, in fact, a pretty descriptive way to describe how software architecture should be built.

Palermo described the layers of software like the layers of the onion: moving out from the core of the onion, each layer depends on the layers deeper for integrity and purpose, but the inner layers do not depend on the outer layers. It's best illustrated by a diagram:

[22]http://jeffreypalermo.com/blog/the-onion-architecture-part-1/



In traditional application development, the database is central to the application. It's often designed first, and many application structure decisions are made around the database. Take it out, and the whole thing crumbles from within.

In the Onion Architecture, the domain is core to the application, and it is completely decoupled from the database. The database is no longer central, or even a required component of the application; it's secondary. It can be swapped out at will without touching the rest of the application. It is supreme Database Independence. We'll touch on how this works in Database Independence.

The Domain Model and Domain Services Layers

At the core of your onion is your domain model layer. This layer contains pure models or entities that are representative of the business objects in your application. They interact with one another to define relationships, and interact with nothing else. They are pure

Outside of that is your domain services layer. These are things like factories, repositories, and other services that use the core domain model

Coupled together, the domain model and domain services make up the core, most central layer of any application. They contain the business logic, and almost every other layer of the application depends on it. For instance, in a CRM application, you probably have a `Customer` model and various services for interacting with it that are used across many of the controllers, views, etc. of the other layers.

For all intents and purposes, the domain core *is* your application. Everything else is simply an extension or a client that interacts with your actual application.

The Application Services Layer

Outside of the Domain Services layer exists the Application Services. This layer is composed of application implementation. In a traditional MVC, this is your Controller layer. It is the router, which responds to an HTTP request and routes it to a specific controller and action, which may use other application services, such as authentication, data parsers or translators (maybe we're exporting a CSV document?) and do everything necessary to bootstrap and provide the view, which is our UI.

It is important to know that nothing in our application should depend on this layer. The Application Services layer should be consider a layer that merely bootstraps our real application, which is the Domain Model and Domain Services application. Like the database, we should be able to swap this out at will with minimal effort. Maybe we started with Symfony but really want to switch to Laravel. That shouldn't be a hard thing to do if the application is written correctly. This allows us to achieve Framework Independence, which, I'm sure you've guessed, we'll talk about in more detail in Framework Independence.

The User Interface

One of the final and outermost layers of the onion is the User Interface. The UI has nothing dependent upon it, but is dependent upon every deeper layer of the onion. It requires the Application Services to give it meaning, which route through the Domain Services to get to the Domain, our meat and potatoes of our application. The view, at its core, is a representation of our domain for the user.

It sits on the outside and has nothing dependent upon it. We can refactor and hack it up at will without consequence. We could try a new templating language, or a new JavaScript framework every week, and the stability of our core application will not change.

We won't spend any time in this book specifically talking about the User Interface, but it will be touched on in Framework Independence and during our Case Study at the end of the book.

Infrastructure

The infrastructure layer resides in an outer layer of our onion. Its responsibility is of grave importance: it provides for our domain. It defines where our data comes from and facilities the retrieving and saving of it to whatever data source it may come from, either Database, Web Services (APIs) or something else entirely. Those things sit on the outside of the application. They're used by the Infrastructure layer, but they aren't part of our onion. They're simply data providers.

The Infrastructure relies on the Domain Services and Domain Model layer as they provide a contract for how the Infrastructure must work. This is done through interfaces, and we'll cover that in the chapters on Interfaces and Dependency Injection and Database Independence.

The Infrastructure also depends upon the Application Services layer, as it is usually where the infrastructure is configured and "hooked up" to the domain. It is usually where our configuration is defined, and where our Dependency Injection Container or Service Locator is setup to provide services. We'll see this at work in our Case Study at the end of the book.

External Libraries

External Libraries are great, and very important to our application as we discussed above. These libraries sit in a layer on the outside of our onion, and are used, much in a similar way that

Infrastructure is, by the Application Services layer to provide meaning and implementation to some portion of the application. For instance, we might use a Google Maps API library to provide geocoding for addresses entered by a user.

We'll talk about these in much detail in External Agency Independence.

Tests

Finally, our tests sit outside of the application in an outer layer of the onion. They, in various incarnations, depend on various layers of the onion in order to test them. This is not a book about testing, but just know that if your application ever relies on the existence of tests to properly function, you've done something horribly wrong.

Framework Independence

Frameworks like Laravel, Symfony, Zend Framework, et al provide great value to the modern day PHP developer. Frameworks tackle common problems, like authentication, database interaction, API usage, MVC, routing, etc., by providing proven solutions so that you, the developer, can get down to brass tax and start actually building the software you set out to build.

Using a framework can also go a long way to help teach and enforce good principles to inexperienced developers. You can really learn a lot by looking at and using a good PHP framework's code base. Often, these frameworks even force you into these design patterns or principles, otherwise the framework simply won't work.

Finally, frameworks can often speed up development time and allow the developer to ship the product faster. It makes sense: with less overhead of having to implement those common problems that frameworks handle for you, you have more time to develop your actual product. This, of course, is ignoring the ramp up period where you actually have to learn the framework, but eventually, this speed up can be realized.

The Problem with Frameworks

With all the benefits provided by these frameworks, it is often very easy to ignore the one giant negative about them. The negative that exists no matter what framework you use and no matter how good that framework is: coupling.

Having your application tightly coupled to a particular framework makes it very hard to leave that framework. You will, eventually, want to leave your beloved framework. A new framework might come out that makes development much easier and much quicker. Or a framework like Phalcon might come out that makes your code run faster. Or your framework might even disappear, either being abandoned by its developers, or reach end of life after a new version is released.

My coworkers and I once had a large application written using Zend Framework 1 that was very successful. When ZF1 reached end of life, and Zend Framework 2 was released, we were very excited to begin investigating upgrading. That excitement faded very quickly when we realized that ZF2 was such a backwards compatibility break that we were going to need to rewrite our application from scratch.

That's costly.

Frameworks can be a blessing, and frameworks can be a curse if not used properly. If we had paid attention to writing our application better, in a way that did not rely so heavily and fully upon our framework, our transition to Zend Framework 2 would have been much quicker, cheaper, and a lot less stressful.

Framework Independence

The phrase "framework independence" can be quite jarring at first to the framework developer. We're not talking about the type of independence sought by early colonists in North America. Instead, we're talking about the ability to switch at will, easily, between one framework or another, or to using no framework at all. In a software application, the ability to leave a framework with minimal effort is a very powerful position to be in.

When writing medium to large applications, it's important to know that the pieces of the application that are implemented using your framework aren't your application, or at least they shouldn't be. Any collection of controllers, forms, helpers, database abstraction, etc., is not your business application. They exist simply as a means to hydrate and display data to your users.

The domain model and domain services layers are your application. This collection of services, repositories, factories and entities are your application. The rest, the stuff the framework provides, is just a gateway or GUI sitting on top of your real application.

If your project is an inventory management system, the code that represents the inventory, the locations, and the means by which they relate to one another, is your application. This code should continue to function correctly if you were to remove the framework.

In order to gain such independence from a framework, there are several things we must remember when developing our applications and using these frameworks.

Abstract the Usage of the Framework

It is very important, as much as possible, to abstract the usage of the framework itself. Every line of code you write that directly uses a component of any framework is code you will have to rewrite if you ever try to switch frameworks.

We use several tactics to abstract the usage of a framework:

- **Use Interfaces Liberally** We previously discussed how we can use interfaces to define base functionality we require, type-hint to those interfaces, and then pass in concrete implementations of those interfaces using dependency injection (or some other method of inversion of control).
- **Use the Adapter Pattern** We also discussed the usage of the Adapter design pattern to wrap the functionality of one class and make it conform to the specification of another, such as an interface.
- **Follow the principles of clean code and SOLID** Writing clean code, and following the principles of SOLID, allow us to have nicely organized and grouped code, and when implemented correctly, code that doesn't depend strongly on the framework to function.

Combining these first two tactics allows us to create a set of interfaces that define the functionality we need to use, and write classes that implement these interfaces, and simply wrap framework classes and map to their functionality to meet those interface requirements.

Additionally, making sure that our code is single-purposed, clean and short, and independent of other parts, will allow us to easily refactor away the framework usage later.

Let's see how this works in different parts of a framework we might use.

Routes and Controllers

Many of us rely heavily on a framework's router and controller mechanisms, as the vast majority of PHP frameworks are MVC-oriented.

How do you go about abstracting the usage of routes and controllers so that you don't tightly couple yourself to them? This is the hardest part of an application to implement in the Clean Architecture. Routes and controllers are the entry point of your application. It is the basis by which all other work in the application is triggered.

It is pretty hard to decouple yourself completely from the framework, unless you actually stop using it. Just think about it: your first step after defining some route is to define some controller logic. This usually involves extending from some base or abstract controller:

```
class CustomersController extends BaseController { }
```

How do we decouple this from the framework? We're immediately extending from the framework, meaning that every piece of code we write in this class from here on out is going to be coupled to our controller. It is not just the extension of `BaseController`, either; it is all the other mechanisms that this class provides us that our code begins to rely on.

Using Adapters to Wrap Your Controller

One approach you can take is to write controllers completely removed from your application. These essentially become very similar to services:

```
namespace MyApp\Controller;

class Customers {
  public function index() {
    return [
      'users' => $this->customerRepository->getAll()
    ];
  }
}
```

With an adapter that looks something like:

```
class CustomersController extends AbstractActionController {
  protected $controller;

  public function __construct(Customers $controller) {
    $this->controller = $controller;
  }

  public function indexAction() {
    return $this->controller->index();
  }
}
```

This seems entirely like overkill. All we're doing is simply wrapping one class in another just to call a method and pass it through. It's a bunch of busy work just to claim we're highly decoupled?

If you're using a lighter framework, like Silex[23], this might be a little easier as controllers are a developer concept, not a framework concept proper. Those controllers only become as coupled as you make them.

Keep Your Controllers Small

Our best bet when dealing with controllers is to make sure we actually minimize the controller code. We want to follow very closely the Single Responsibility Principle we discussed previously. Each controller, and each action in the controller, should have as little code in it as possible.

Controllers should be thought of as response factories. They are responsible for building a response based on the input (the HTTP request). All logic should be passed off to either Domain Services or Application Services for processing, and the data returned from them loaded into a response and returned.

Ruby on Rails was big on the mantra of "fat model, skinny controller," and that is at least partially sound. Having a skinny controller is very important. Having a fat model? Well, that depends on what "fat" and "model" mean to you. We'll discuss this in depth in Database Independence. Having a skinny controller means that a controller doesn't really do much. Let's take this controller action, for example:

```
class CustomersController extends AbstractActionController {
  public function indexAction() {
    return [
      'users' => $this->customerRepository->getAll()
    ];
  }
}
```

This is a simple GET request to an index action (maybe `/customers`) that simply lists all customers. The controller uses the dependency injected `CustomerRepository` (not shown) and its `getAll()` method to retrieve all the customers.

[23]http://silex.sensiolabs.org/

Obviously we're going to have more complex actions than `indexAction()`, but the point is that we want to pass all of the logic and processing to our Domain Services layer, and keep the controllers as small, tight, and single purposed as possible.

Views

Your view layer should be primarily composed of HTML, JavaScript and CSS. There should be no business logic contained within the views; only display logic. These views should be mostly transferable when switching frameworks. The only questionable part is the means by which data is presented to the view from the controller, which will probably vary by framework.

The big thing to watch out for is view services, view helpers, and/or view plugins, which many frameworks provide. These components help generate common or recurring HTML, such as links or pagination, or even forms (which we'll talk about next). The method by which these exist will likely vary wildly by framework, and could cause quite a headache if they were very heavily relied on.

If you're writing your own helpers, which many frameworks allow you to do, make sure that you're writing the bulk of the code without relying on the framework, so that you can easily move this helper to a new framework. If possible, also consider writing interfaces and/or an adapter that turns your helper into something the framework expects.

Do as much as you can to make leaving the framework easy.

Another option would be to forgo your framework's built-in view layer and use a third party library, such as Plates[24]. This will allow you to keep your view layer intact when switching frameworks.

Forms

In my experience, forms have been one of the hardest things to deal with in projects. Doing forms cleanly and independent from the framework is near impossible. Again, we can write a bunch of adapters to abstract the usage of the framework, but that will almost certainly negate the time savings given by the framework.

The biggest rule is to make sure that no business logic whatsoever exists within the form code. Remember: business logic belongs in the Domain layer. Aside from validation rules and filtering, no logic should be contained within the forms.

Outside of that: just use them. If the bulk of your work when switching frameworks is porting the forms, you're in pretty good shape.

Another solution to try would be to use some third party form library, so that you're not coupled to your framework's form classes. Something such as Aura.Input[25] from the Aura components would suffice, and allow you to keep that code when switching frameworks. Some form libraries are light enough that you might even be able to write an adapter around them, but only do so if you'll be able to accomplish that quickly.

[24]http://platesphp.com/
[25]https://github.com/auraphp/Aura.Input

Framework Services

Most frameworks provide helpful services to make writing day to day code much easier. These can include services that run HTTP requests or even do full API calls, components that implement an OAuth2 client to log in to those APIs, services that generate PDFs or barcodes, send emails, retrieve emails, etc. While these services provide a quick and fast way to get the job done by preventing you, the developer, from writing extraneous code, they are also a great place to run into coupling problems.

How do you run into coupling problems with these components? Simply by using them.

Consider Laravel's `Mail` facade:

```
Mail::send('emails.hello', $data, function($message) {
  $message->to('you@yoursite.com', 'You')->subject('Hello, You!');
});
```

Laravel makes it extremely simply to send emails, but the minute we drop this in a controller or service, we're tightly coupling ourselves to Laravel. Remember, our goal with controllers and services are to have them as small, lightweight, and uncoupled as possible so that we can mitigate the work needed to migrate that code over to another controller system, should we ever switch framework or underlying controller/routing mechanism.

How do we solve this? Using the previously discussed adapter design pattern, the correct way to handle these services are to define an interface outlining the functionality that we need, and writing an adapter that wraps the framework code to implement that interface. Finally, the adapter should be injected into whatever client object needs it.

```
interface MailerInterface {
  public function send($template, array $data, callable $callback);
}
```

Our adapter would then implement this interface:

```
class LaravelMailerAdapter implements MailerInterface {
  protected $mailer;

  public function __construct(Mailer $mailer) {
    $this->mailer = $mailer;
  }

  public function send($template, array $data, callable $callback) {
    $this->mailer->send($template, $data, $callback);
  }
}
```

The adapter should then be injected into our controller and used, instead of directly using the `Mail` facade:

```
class MailController extends BaseController {
  protected $mailer;

  public function __construct(MailerInterface $mailer) {
    $this->mailer = $mailer;
  }

  public function sendMail() {
    $this->mailer->send('emails.hello', $data, function($message) {
      $message->to('you@yoursite.com', 'You')->subject('Hello, You!');
    });
  }
}
```

To make this work, we'll register our interface with Laravel's IoC container so that when the controller is instantiated, it'll get a proper instance of `MailerInterface`:

```
App::bind('MailerInterface', function($app) {
  return new LaravelMailerAdapter($foo['mailer']);
});
```

Now the controller gets a concrete instance of `LaravelMailerAdapter`, which conforms to the controller's dependency requirement of `MailerInterface`. If we ever decide to switch mailing mechanisms, we simply write a new adapter and change the binding of `MailerInterface`, and all our client code that previously got injected with `LaravelMailerAdapter` now gets whatever this new implementation is.

The popular Symfony YAML[26] component is another great example of where a quick and easy adapter and interface allow you to be completely decoupled from the concrete implementation that Symfony provides.

The component is extremely simple to use:

```
$data = Symfony\Component\Yaml\Yaml::parse($file);
```

That's all it takes to turn a YAML file into a PHP array. But when using this in our code, we'll first want to create an interface to define this functionality that we need:

```
interface YamlParserInterface {
  public function parse($fileName);
}
```

We then implement this interface with an adapter:

[26] http://github.com/symfony/yaml

```
class SymfonyYamlAdapter implements YamlParserInterface {
  public function parse($fileName) {
    return Yaml::parse($file);
  }
}
```

Then we simply utilize dependency injection to provide an instance of `SymfonyYamlAdapter` into any code that needs it:

```
class YamlImporter {
  protected $parser;

  public function __constructor(YamlParserInterface $parser) {
    $this->parser = $parser;
  }

  public function parseUserFile($fileName) {
    $users = $this->parser->parse($fileName);

    foreach ($users['user'] as $user) {
      // ...
    }
  }
}
```

Now we're harnessing the power of Symfony's YAML parser, without coupling with it. The point of doing this, again, is so that if we ever need to switch to a different YAML parsing solution in the future – for whatever reason – we can do so without changing any of our client code. We would simply write another adapter, and dependency inject that adapter in `SymfonyYamlAdapter`'s place.

Database Facilities

Most PHP frameworks come bundled with some sort of Database Abstract Library (DBAL), and sometimes a Query Builder library that makes it easier to build SQL queries, or maybe an Object Relational Mapping (ORM) library, either conforming to the Active Record or Data Mapper patterns. Taking advantage of these libraries can lead to easy, and rapid development of database-powered applications.

As always, though, we want to watch out for coupling too tightly to these database layers.

This is such an important topic, that the entire next chapter, Database Independence[27] is devoted to it.

27

This is a Lot of Work

Are there any instances in which it's okay to simply couple yourself to the framework? Of course. The principles outlined above only apply, in varying degrees, to the size and complexity of your application. If your application is so small in scope and function that it wouldn't take you very long at all to entirely can it and rewrite it in a new framework, then by all means, go ahead and coupled to the framework.

It does take a lot of extra work to write code this way, so it is up to you to do a cost analysis for the project. If it is a small registration application that simply collects attendee information and saves it to the database, it's going to be quicker to just completely rewrite it later, rather than go through all of these steps.

When the application is large in scope, like a Customer Relationship Management (CRM) system or an Enterprise Resource Planning (ERP) system, it probably makes a lot of sense to think about the future of the code right away. It probably makes a lot of sense to write the code in a way that it survives the chosen framework.

Database Independence

Often times when developing an application that uses a database for storage, the database easily becomes the center and focal point of the application. At first, this makes sense: we're building an application on top of the database. Its sole purpose is to display and manipulate that data stored in the database. Why wouldn't it be central to the application?

An application where the database becomes the central focal point suffers from a few problems:

1. The code is often littered with database interaction code, often times raw SQL or at least direct references and instantiation of classes that query the database. If the database, or database abstraction code, is literally all over the code base, it becomes nearly impossible to refactor that code without a time consuming effort.
2. Testing this code without using the database because very hard, if not impossible. Testing using databases is painful: you have to setup a known state for each and every test case as the code is modifying the contents of the database with each test. This can become slow to run, and test suites, to be successful and helpful, need to be fast.

So if the database shouldn't be central to the application, what should take it's place at the core?

Domain Models

If you paid attention to our previous discussion about the Clean Architecture, you should already know: the Domain Model is the core of our application, and central to everything else around it. Everything builds from it. So what is it exactly?

The **domain model layer** is a collection of classes, each representing a data object relevant to the system. These classes, called Models or Entities, are simple, plain old PHP objects.

By definition, the domain model layer, being the core of the application, cannot be dependent upon any other layer or code base (except the underlying language, such as PHP). This layer is wholly independent of anything. This makes it completely uncoupled, transferable, and quite easily testable.

A Sample Domain Model

A sample domain model for a customer might look something like this:

```
class Customer {
  protected $id;
  protected $name;
  protected $creditLimit;
  protected $status;

  public function getId() {
    return $this->id;
  }

  public function setId($id) {
    $this->id = $id;
    return $this;
  }

  public function getName() {
    return $this->name;
  }

  public function setName($name) {
    $this->name = $name;
    return $this;
  }

  // ...
}
```

This Domain Model is a pure PHP object. It has no dependencies nor coupling, other than to PHP itself. We can use this unhindered in any of our code bases, easily transfer it to other code bases, extend it with other libraries, and very easily test it.

We've achieved some great things following this Domain Model implementation, but it's only going to get us so far. Right now, we just have a simple PHP object with some getters and setters. We can't do much with that. We're going to have to expand on it, otherwise it's going to be terribly painful to use.

Domain Services

Domain Services, being the next layer in the onion of our architecture, is meant to expand on this and provide meaning and value to the Domain Model. Following the rule of our architecture, layers can only have dependency upon layers deeper in the onion than it is. This is why our Domain Model layer could have no dependencies, and why our Domain Services layer can only depend upon, or couple to, the Domain Model layer.

The Domain Services layer can consist of several things, but is usually made up of:

- **Repositories**, classes that define how entities should be retrieved and persisted back to some data storage, whether it be a database, API, XML or some other data source. At the Domain Services layer, these are simply interfaces that define a contract that the actual storage mechanism must define.
- **Factories**, are simply classes that take care of the creation of entities. They may contain complex logic about how entities should be built in certain circumstances.
- **Services**, are classes that implement logic of things to do with entities. These can be services such as invoicing or cost buildup or classes that build and calculate relationships between entities.

All of these are implemented only using the Domain Model and other Domain Services classes. Again, they have no dependence or coupling to anything else.

Repositories

Repositories are responsible for defining how Domain Model entities should be retrieved from and persisted to data storage. At the Domain Services layer, these should simply be interfaces that some other layer will define. Essentially, we're providing a contract to follow so that other layers of our application can remain uncoupled to an implementation. This allows that implementation to be easily changed, either by switching out what is used in production, or maybe just switching out what storage is used during testing.

A sample repository might look like this:

```php
interface CustomerRepositoryInterface {
  public function getAll();
  public function getBy($conditions);
  public function getById($id);
  public function save(Customer $customer);
}
```

As you can see, again, we have a simple PHP interface, uncoupled from anything but the Domain Model (through the usage of `Customer`). This interface doesn't do anything. It simply defines a contract to be followed by an actual implementation. We'll talk about that implementation in *Database Infrastructure / Persistence* in a little bit.

The Domain Services layer should contain a definition of all the repositories an application will need to properly function, as well as each repository containing all the methods that the application will need to interact with the data.

Factories

Factories are responsible for creating objects. At first, that seems a little silly as creating an object is as simple as:

```
$customer = new Customer();
```

It's not always this easy, however. Sometimes, complex logic goes into creating a customer. If you notice above when we defined the `Customer` class, we gave it a credit limit and status attribute. It might be that all customers get set to a certain state when they're created such that these two attributes are always set to a predefined value. If we were to continue with simple instantiation:

```
$customer = new Customer();
$customer->setCreditLimit(0);
$customer->setStatus('pending');
```

Now let's assume we might have several different places in the code where we create customers. We now have to repeat this code all over the place. If these default rules ever change, we then have several places we need to go change the code. If we miss some, now we have bugs in the code.

Using a factory lets us consolidate that code and make it reusable:

```
class CustomerFactory {
  public function create() {
    $customer = new Customer();
    $customer->setCreditLimit(0);
    $customer->setStatus('pending');

    return $customer;
  }
}
```

Now, wherever we need to create a customer, we can simply use our factory:

```
$customer = (new CustomerFactory())->create();
```

If our business logic ever changes, all we need to do is simply update the factory, and each customer creation point will follow the new rules.

The skillful developer might see an even easier solution to this problem: just throw the defaults in the class or constructor:

```
class Customer {
  protected $id;
  protected $name;
  protected $creditLimit = 0;
  protected $status = 'pending';
}
```

This is true: that does look much simpler, and it's completely due to the simplicity of the example. However, let's say we have an account manager that needs to be assigned to every new customer, and to pick them, we need to find the next available account manager (whatever that might mean in the context of our application):

```
class CustomerFactory {
  protected $managerRepository;

  public function __construct(AccountManagerRepositoryInterface $repo) {
    $this->managerRepository = $repo;
  }

  public function create() {
    $customer = new Customer();
    $customer->setAccountManager(
      $this->managerRepository->getNextAvailable()
    );
  }
}
```

As our business rules and domain logic become more complex, using these factories start to make sense. The important thing to remember is that these factories are completely decoupled from the actual data storage. Their only dependence is upon the Domain Model.

So how exactly does an instance of `AccountManagerRepositoryInterface` get into the `CustomerFactory`? And what exactly is the implementation of that interface? We'll cover that soon in *Database Infrastructure / Persistence*.

Services

Services, simply put, are responsible for doing things. These are usually processes, such as invoice runs or some kind of cost build up analysis. Anything that involves business logic that is not either creational (which belongs in a factory), or retrieving or persisting data (which belongs in a repository). These services can depend on repository interfaces and factories to do their work.

Let's look at an example service for generating invoices based off orders:

```
class BillingService {
  protected $orderRepository;
  protected $invoiceRepository;
  protected $invoiceFactory;

  public function __construct(
    OrderRepositoryInterface $order,
    InvoiceRepositoryInterface $invoice,
    InvoiceFactory $factory
  ) {
    $this->orderRepository = $order;
    $this->invoiceRepository = $invoice;
    $this->invoiceFactory = $factory;
  }
  public function generateInvoices(\DateTime $invoiceDate) {
    $orders = $this->ordersRepository
      ->getActiveBillingOrders($invoiceDate);

    foreach ($orders as $order) {
      $invoice = $this->invoiceFactory->create($order);
      $this->invoiceRepository->save($invoice);
    }
  }
}
```

This service is pretty simple, because it leverages the power of the `OrderRepository` given to it to retrieve orders, and the `InvoiceFactory` to generate invoice objects. It then simply persists them to the database using the `InvoiceRepository`. The `BillingService` can now be used anywhere that invoices need to be generated for whatever means of ordering the system needs implement. This is abstracted away into the service, so that the code is not repeated all over the place.

Further, this code does not depend on a specific data store whatsoever, instead, asking for an implementation of both `OrderRepositoryInterface` and `InvoiceRepositoryInterface`. If those dependencies are satisfied with concrete implementations, the Service works correctly finding orders to invoice and generating those invoices.

This code is powerful, but dead simple. It's coupled to nothing but the rest of our Domain Model and Domain Services layer. It is 100% decoupled from any specific database implementation.

Database Infrastructure / Persistence

So far we've discussed taking the database from the core of the application, and replacing it with a robust Domain Model and Domain Services core that encompass the primary functionality of our application. Now the heavy parts of our application are decoupled, well-written, and quite testable. At some point, however, we have to bring the database back into the picture. A data-centric application without some sort of data storage implementation is going to be a failure, regardless of how clean and testable the code base is. So how do we get the database back in the picture?

Now that we've fleshed out our Domain Model and Domain Services layers, we can start to build out our Persistence layer, which is part of the infrastructure layer of our onion. The **persistence layer** is responsible for retrieve and persisting data to our data storage, whatever that may be. In our case, it's probably going to be a relational database system, such as MySQL or PostgreSQL. Or maybe it might be a NoSQL variant or an API services layer that provides our data.

For instance, if we were using a relational database and using the Doctrine ORM library to provide persistence, we might implement our `CustomerRepositoryInterface` like so:

```php
class CustomerRepository implements CustomerRepositoryInterface {
  protected $entityManager;
  protected $entityClass = 'MyVendor\Domain\Entity\Customer';

  public function __construct(EntityManager $entityManager) {
    $this->entityManager = $entityManager;
  }

  public function getAll() {
    return $this->entityManager
      ->getRepository($this->entityClass)->getAll();
  }

  public function getById($id) {
    return $this->entityManager->find(
      $this->entityClass,
      $id
    );
  }
}
```

This is a really simple implementation of a repository using Doctrine ORM. Of course, to use Doctrine, we also have various mapping files and configurations we need to setup to get things to work, but this is our basic repository. We're also missing the definition of a couple methods above, but have simply omitted them for brevity. Were we actually trying to run this code, we'd get an error that we didn't implement all methods of the interface.

It implements our `CustomerRepositoryInterface`, such that anything requesting an instance of this interface would be fully satisfied by using this concrete class.

Of course, this functionality looks pretty generic and unspecific to our `CustomerRepository`. We could easily break this out into an abstract class so that we can prevent duplicate functionality being littered about all of our repositories:

```php
abstract class AbstractRepository {
  protected $entityManager;
  protected $entityClass = '';

  public function __construct(EntityManager $entityManager) {
    $this->entityManager = $entityManager;

    if (empty($this->entityClass)) {
      throw new \RuntimeException(
        'entityClass not specified for ' . __CLASS__
      );
    }
  }
  public function getAll()
  {
    return $this->entityManager
      ->getRepository($this->entityClass)->getAll();
  }

  public function getById($id)
  {
    return $this->entityManager->find(
      $this->entityClass,
      $id
    );
  }
}
```

Our `CustomerRepository` then simply becomes:

```
class CustomerRepository extends AbstractRepository
  implements CustomerRepositoryInterface {

  protected $entityClass = 'MyVendor\Domain\Entity\Customer';
}
```

Now we only need to add customer-specific logic to this repository, as needed.

Utilizing Persistence

The Persistence layer is meant to sit on one of the outer layers of the onion. It is not central to the application. Again, our Domain Model and Domain Services are. The Persistence layer simply provides meaning to that Domain Services layer by implementing the repository interfaces that we setup. **Nothing** should be dependent upon this Persistence layer. In fact, we should be able to swap out Persistence layers with other implementations, and all of the code that ends up using this layer (through using the interfaces) should be none the wiser, and continue to function properly.

We'll experiment with this concept once we start working on our Case Study.

Using Dependency Injection to Fulfill Contracts

So we've defined plenty of interfaces in our Domain Services layer and implemented them with concrete classes in our Persistence layer. We've discussed that nothing can directly depend on this Persistence layer, so just how exactly do we use it in our application?

We previously discussed Dependency Injection as a means of preventing coupling, and that's just what we'd use here. Any time any class needs a concrete implementation of a repository, it should declare a dependency to the interface, instead. For instance, when we discussed our Factory that needed to find the next available account manager, remember, we only asked for an interface:

```php
class CustomerFactory {
  protected $managerRepository;

  public function __construct(AccountManagerRepositoryInterface $repo) {
    $this->managerRepository = $repo;
  }

  // ...
}
```

The `CustomerFactory` is deeper in the onion than that persistence layer, so it can't be dependent upon a concrete `CustomerRepository`, nor would we want it to be. Instead, it declares, via the constructor, that it needs *something* that implements `AccountManagerRepositoryInterface`. It doesn't care what you give it, so long as the what is passed in adheres to the interface. We call this **programming by contract**.

So how exactly does a `CustomerRepository` get into the `CustomerFactory`? Whatever means is responsible for instantiating this Factory would take care of passing in the correct implementation:

```php
$customerFactory = new CustomerFactory(
  new CustomerRepository($entityManager)
);
```

How this works is largely dependent upon your framework and how it handles Dependency Injection Containers or Service Locators. We'll discuss this more when we talk about the framework, and explore how it works in our Case Study. The important thing to note here is that our code should only be dependent upon interfaces, not concrete implementations of the Persistence layer.

Organizing the Code

We've discussed Entities, Services, Factories, Repository interfaces and concrete Repositories. Where do we put all this stuff?

It makes sense to logically separate the layers of your application to their own root fielders, even if the shared parent folder is the same. We have essentially talked about two layers, Domain and Persistence. So starting there might be a good idea, and then grouping each component type under those two folders:

```
src/
    Domain/
        Entity/
        Factory/
        Repository/
        Service/
    Persistence/
        Repository/
```

Following the PSR-4 autoloading standard[28], we'd also have similar namespaces, for instance:

```
MyVendor\Project\Domain\Entity\Customer
```

I've taken this one step further in a series of applications that actually share the same Domain (and ultimately, database). Domain and Persistence have both been broken up into separate code repositories, all loaded into several parent projects using Composer. It's the ultimate in separation of concerns: the code bases are literally separate. Granted, that doesn't mean they're decoupled.

How you want to organize your code is up to you. My ultimate recommendation is to keep the layers segregated out into at least their own folders.

Wrapping it Up

The purpose for taking the database away from being the center of our application is so that we are no longer dependent and coupled to that database. This gives us the freedom to swap out database flavors, or even add in a middle layer such as an API in between our application and the database, without having to rewrite our entire application. It also gives us the flexibility in switching out database libraries, say going from Doctrine to Laravel's Eloquent ORM.

We'll next look at applying some similar principals to the developer's framework of choice, and explore how we can limit decoupling in Framework Independence.

[28]http://www.php-fig.org/psr/psr-4/

External Agency Independence

With the arrival of Composer[29], the PHP scene suddenly exploded with a myriad of packages, libraries, components, and tools that could easily be dropped into any project, and even autoloaded magically through the Composer autoloader. It is now easier than it has ever been to load third party libraries into your source code and solve challenges easily and quickly.

A simple glance at the instructions for many of these projects on GitHub shows how painfully easy it is to get them installed and integrated. Usually, installation instructions are followed by a quick little snippet or two of example usage.

Not so fast.

Every time we pull in one of these third party libraries and use it directly within our client code, we're tightly coupling ourselves to it.

How long is this code going to be around? Is it going to be actively maintained? What if our needs outgrow what it provides? It's likely that at some point, we may need to abandon this library for another solution. The more tightly coupled we are to this library, the harder it's going to be to switch to something else.

Using Interfaces, Adapters and Dependency Injection

Of course, we've already explored the solution to this problem when we discussed how to handle framework services. The solution was pretty simple:

1. Create an interface defining the functionality we need
2. Write an adapter that wraps the third party code, making it conform to our interface
3. Require an implementation of that interface to be injected into whatever client code needs it

Let's look at another example of this.

The Geocoder PHP[30] library provides a great library for geocoding services, itself with several adapters to use a variety of different services to provide the geocoding. Let's say our app simply needs to be able to get the longitude + latitude for any given address.

We first define an interface for this need:

[29]https://getcomposer.org/

[30]https://github.com/geocoder-php/Geocoder

```
interface GeocoderInterface {
  public function geocodeAddress($address);
}
```

This is fairly straight forward. We need this in a controller which interacts with some mapping:

```
class AddressController {
  protected $geocoder;

  public function __construct(
    GeocoderInterface $geocoder
  ) {
    $this->geocoder = $geocoder;
  }

  public function geocode() {
    return $this->geocoder->geocodeAddress(
      $this->params('address')
    );
  }
}
```

Now all we need to do to make this code work is provide something that implements `GeocoderInterface`, and inject that into the controller when it is instantiated. Our adapter provides the needed concrete implementation:

```
class GeocoderPhpAdapter {
  protected $geocoder;

  public function __construct(Geocoder $geocoder) {
    $this->geocoder = $geocoder;
  }

  public function geocodeAddress($address) {
    $results = $this->geocoder->geocode($address);

    return [
      'longitude' => $results['longitude'],
      'latitude' => $results['latitude']
    ];
  }
}
```

We're also injecting the `$geocoder` into this adapter as we may want to have different configurations for different circumstances:

```
$geocoder = new GeocoderPhpAdapter(
  new Geocoder(
    new GoogleMapsProvider(new CurlHttpAdapter())
  )
);
```

That’s a lot of dependency injection!

Benefits

We want to make sure we have flexibility and freedom in our applications. Specifically, we need the ability to switch out third party libraries, whenever necessary, for whatever reason, easily and quickly.

Not only does it safeguard us against a library going away or no longer providing what we need, but it can make testing easier through mocking, and also makes refactoring easier, as it encapsulates all functionality of the third party library into one place.

We’ll use this strategy extensively whenever we use third party libraries throughout the Case Study.

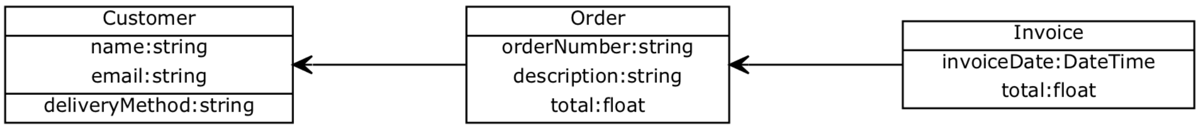
A Case Study in Clean Architecture

The topics covered in the first few sections of this book provided a solid foundation for building quality applications in PHP. Let’s see these practices at work by building a small application and applying these principles as we go. When we’re done, we’ll switch services and frameworks a few times to see how our choices have made it easier to do so.

The Billing System

Our Case Study revolves around our client, SimpleTech, who wants a simple system to keep track of their customer orders, and generate invoices for those orders.

A simple UML diagram of these relationships would look like:



A Customer has multiple Orders, each of which has an Invoice.

If this application seems incredibly simple, it's because it is. You might be asking yourself whether it is worth learning a new architecture and carefully crafting the code for such a simple application that would be quite easy to rewrite if needed.

That's a good observation and leads to a good general rule: if an application is so small, or so simple that it can be rewritten quickly, then the value of a tool such as the Clean Architecture is diminished. That's not to say that it shouldn't be done, especially if you've become quite comfortable writing applications in this manner; it might even be faster!

Let's pretend that our small case study is part of a much larger system that will grow over time to include procurement, manufacturing modules, distribution, accounting, etc; a full blown Enterprise Resource Planning (ERP) system for SimpleTech. Knowing this, it very much makes sense to consider the Clean Architecture as a foundation for such a large system.

Application Workflow

We're going to fulfill the following requirements for SimpleTech:

1. Ability to add new Customers
2. Ability to create simple Orders for a Customer
3. Ability to run an Invoicing process that turns Orders into Invoices when it's time to bill the customer.

Again, this application will be pretty simple, but will allow us to show off some of the things we've learned in terms of layers of applications, design patterns, and architecture.

Prerequisites

Before we begin, you'll have to make sure you have a machine with PHP installed. If this comes as a shock, you may have purchased the wrong book. For more information on setting up PHP, checkout PHP The Right Way[31] and their section on Getting Started.

You'll also need to have some kind of database installed. We'll use sqlite[32] for simplicity, but any relational database, such as MySQL, PostgreSQL, or Oracle, should suffice. You could even setup this simple project using a NoSQL variant, but we won't cover that here.

Everything else we need will installed via Composer.

Setting up Composer

Our next step is to setup Composer[33] so that we can utilize its autoloader for our unit tests. If you don't already have it set up, I *highly* suggest you install it globally so that you can use it from anywhere.

On a *nix based system, you can do so easily:

```
# for a home directory install:
curl -sS https://getcomposer.org/installer | \
    php -- --install-dir=bin --filename=composer

# for a global system install:
curl -sS https://getcomposer.org/installer | \
    sudo php -- --install-dir=/usr/local/bin --filename=composer
```

When done, assuming your paths are set up properly, you can simply run `composer` to verify it is installed correctly:

```
composer
```

You'll either get an error about command not found, or you'll get some output. You might need to double check that `~/bin` or `/usr/local/bin` is in your `$PATH` variable. You can check this easily by running:

```
echo $PATH
```

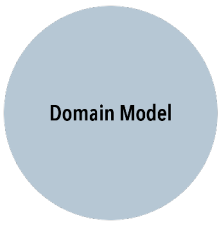
And looking for those directories in the output. If they're not in there, try adding them in your `~/.bashrc`, `~/.zshrc`, or similar file:

[31]http://www.phptherightway.com/
[32]http://www.sqlite.org/
[33]https://getcomposer.org/

```
export $PATH='/home/yourname/bin:$PATH'
```

Make sure you include `:$PATH` on the end, or you'll overwrite everything else stored in your `$PATH` variable!

Building Our Domain



As the domain is central to our application, it makes perfect sense for us to start there. At the core, we have a Domain Model layer, which is composed of models, and models only. We're going to call these models entities. Our entities are going to be plain PHP objects that represent something in our application. These are: `Customer`, `Order`, and `Invoice`.

Remember: the Domain Model layer can have no dependencies whatsoever. It is completely uncoupled from everything but PHP and itself.

Branching out from there, we're going to have the Domain Services layer, which builds on top of the Domain Model layer. This layer can be fully dependent upon the Domain Model layer and itself, but nothing else.

In addition to services, this layer also contains factories, responsible for building objects, and repositories, although usually just interfaces that define a contract for another layer to implement.

Setting up the Project

The first thing we need is a directory structure:

```
mkdir -p cleanphp/src/Domain/Entity
```

In this directory, all of our Entities will live.

With Composer installed (see the previous chapter), we can configure our `composer.json` file with a simple `autoload` section to autoload our resources. This file should go in the root `cleanphp/` directory:

```
{
  "autoload": {
    "psr-4": {
      "CleanPhp\\Invoicer\\": ["src/"]
    }
  }
}
```

This configuration tells Composer that we want to use the PSR-4 autoloading standard to load the `CleanPhp\Invoicer` namespace, and that the root directory for that namespace is located at `src/`. This lets Composer's autoloader find classes of that namespace within the `src/` directory.

Finally, run the `dump-autoload` command to instruct Composer to build its autoload files (which are located at `vendor/composer`):

```
composer dump-autoload
```

If you take a peak, you should now see a `vendor/composer` directory with our autoload configuration set up in `autoload_psr4.php`.

Now we're ready to create the entities.

Creating the Entities

These entity classes are all going to use a unique identifier that represents them. That `$id` attribute will require a `getId()` and a `setId()` method. To keep from repeating ourselves, and as a way to identify all entities, let's go ahead and create an abstract `AbstractEntity` that all of these entities can inherit from:

```
// src/Domain/Entity/AbstractEntity.php

namespace CleanPhp\Invoicer\Domain\Entity;

abstract class AbstractEntity {
  protected $id;

  public function getId() {
    return $this->id;
  }

  public function setId($id) {
    $this->id = $id;
    return $this;
  }
}
```

Now let's define our `Customer` entity, which as a Name, Email Address, and Invoice Delivery Method:

```php
// src/Domain/Entity/Customer.php

namespace CleanPhp\Invoicer\Domain\Entity;

class Customer extends AbstractEntity {
  protected $name;
  protected $email;

  public function getName() {
    return $this->name;
  }

  public function setName($name) {
    $this->name = $name;
    return $this;
  }

  public function getEmail() {
    return $this->emailAddress;
  }

  public function setEmail($email) {
    $this->email = $email;
    return $this;
  }
}
```

Next, let's define our Order entity, which has a Many to One relationship with `Customer`, as well as an Order Number, Description, and Total Order Amount:

```php
// src/Domain/Entity/Order.php

namespace CleanPhp\Invoicer\Domain\Entity;

class Order {
  protected $customer;
  protected $orderNumber;
  protected $description;
  protected $total;

  public function getCustomer() {
    return $this->customer;
  }

  public function setCustomer($customer) {
    $this->customer = $customer;
    return $this;
  }

  public function getOrderNumber() {
    return $this->orderNumber;
  }

  public function setOrderNumber($orderNumber) {
    $this->orderNumber = $orderNumber;
    return $this;
  }

  public function getDescription() {
    return $this->description;
  }

  public function setDescription($description) {
    $this->description = $description;
    return $this;
  }

  public function getTotal() {
    return $this->total;
  }

  public function setTotal($total) {
    $this->total = $total;
    return $this;
  }
}
```

Finally, our `Invoice` entity, which has a Many to One relationship with an `Order`, as well as an Invoice Date and Total Invoice Amount:

```
// src/Domain/Entity/Invoice.php

namespace CleanPhp\Invoicer\Domain\Entity;

class Invoice {
  protected $order;
  protected $invoiceDate;
  protected $total;

  public function getOrder() {
    return $this->order;
  }
  public function setOrder(Order $order) {
    $this->order = $order;
    return $this;
  }

  public function getInvoiceDate() {
    return $this->invoiceDate;
  }

  public function setInvoiceDate(\DateTime $invoiceDate) {
    $this->invoiceDate = $invoiceDate;
    return $this;
  }

  public function getTotal() {
    return $this->total;
  }

  public function setTotal($total) {
    $this->total = $total;
    return $this;
  }
}
```

These three classes complete our small Domain Model layer.

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 01-domain-models:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 01-domain-models
```

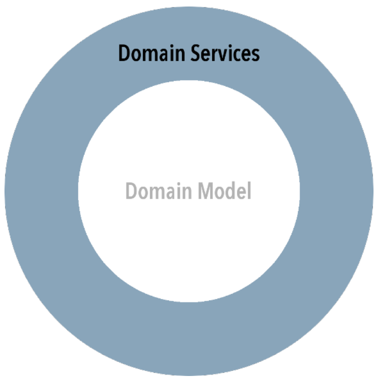
Testing Our Domain Models

I'm going to make an executive decision at this point and decide not to write any tests for these models as they stand right now. Writing tests for simple getter and setter methods is pretty tedious, and if something is going to go wrong, it's likely not going to happen here.

If you want to write these tests, go for it! You'll likely be at least somewhat better off for doing so.

We'll start writing tests next when we start building out our domain services for this application.

Domain Services



Remember that the Domain Services layer was next as we move outward from the Domain Model layer within the Onion/Clean Architecture. This layer will hold all the services and service contracts that the application will use. As the only thing deeper in the onion is the Domain Model layer, the Domain Services layer can **only** depend on the Domain Model layer (as well as PHP itself).

Our domain services layer will comprise of: * **Repository Interfaces** These interfaces will define how the real repositories will work. Since we can't rely on any infrastructure at this point, we can't actually code any concrete repositories yet. * **Factories** These factories will be responsible for creating domain objects based on our business rules. * **Services** These services will be responsible for implementing the rest of our business rules. They may rely on both the repositories and the factories to complete their work.

Let's get started!

Setting up Repositories

We're going to need to retrieve and persist data from our database, and we'll do that by using repositories. Repositories live in the infrastructure layer of our application, but we'll define them in the domain services layer. The infrastructure layer is meant to be highly swappable, so we'll want to define some contracts for that layer to follow within our domain services layer.

Normally, we'd start by writing some tests that define the functionality of these repositories, but since we're just going to have interfaces at this point, there would literally be nothing to test.

Generally, we're going to want to be able to do the following operations for Customers, Orders and Invoices:

- Get by ID
- Get All
- Persist (Save)
- Begin
- Commit

Since this is common functionality, let's go ahead and create a base `RepositoryInterface` to define this functionality:

```
// src/Domain/Repository/RepositoryInterface.php

namespace CleanPhp\Invoicer\Domain\Repository;

interface RepositoryInterface {
  public function getById($id);
  public function getAll();
  public function persist($entity);
  public function begin();
  public function commit();
}
```

Now let's create some interfaces that represent the actual entities, that extend and inherit the functionality of `RepositoryInterface`.

Customers

```
// src/Domain/Repository/CustomerRepositoryInterface.php

namespace CleanPhp\Invoicer\Domain\Repository;

interface CustomerRepositoryInterface
  extends RepositoryInterface {

}
```

Orders

```
// src/Domain/Repository/OrderRepositoryInterface.php

namespace CleanPhp\Invoicer\Domain\Repository;

interface OrderRepositoryInterface
  extends RepositoryInterface {

}
```

Invoices

```php
// src/Domain/Repository/InvoiceRepositoryInterface.php

namespace CleanPhp\Invoicer\Domain\Repository;

interface InvoiceRepositoryInterface
  extends RepositoryInterface {

}
```

These repositories each represent an entity, and define a contract that each concrete repository must follow. Additionally, we'll use these interfaces to type-hint dependency injection in each instance where we need them, so that we can ensure our classes will get the correct functionality they need.

As part of our invoicing process, we need to find all orders that have not yet been invoiced. We can define this need by adding a method to the `OrderRepositoryInterface`:

```php
// src/Domain/Repository/OrderRepositoryInterface.php

namespace CleanPhp\Invoicer\Domain\Repository;

interface OrderRepositoryInterface
  extends RepositoryInterface {

  public function getUninvoicedOrders();
}
```

This `getUninvoicedOrders()` method can be used to get all the orders when our invoicing service runs.

Invoice Factory

Invoices are created for Orders, and inherit some of their data, so it makes sense that we would encapsulate the creation of these Invoices into a factory service.

This simple factory should accept an `Order` object, and return an `Invoice` object to the caller:

```php
public function createFromOrder(Order $order);
```

Let's start by wring a test that's going to define this service. We'll use the awesome Peridot[34] testing framework that follows a Behavior-Driven Development (BDD) approach to testing.

Let's install the latest stable version as a development dependency via Composer:

[34]http://peridot-php.github.io

```
composer require --dev peridot-php/peridot
```

Once it's installed, we can run it with the command:

```
./vendor/bin/peridot
```

If all goes well, you'll see Peridot run it's own tests. That's great, but we want to run our tests. But before we do that, we'll have to write them. Let's start by creating a root level `specs/` directory for our test specs to live in.

Let's write our first test:

```php
// specs/domain/service/invoice-factory.spec.php

use CleanPhp\Invoicer\Domain\Model\Invoice;
use CleanPhp\Invoicer\Domain\Model\Order;
use CleanPhp\Invoicer\Domain\Factory\InvoiceFactory;

describe('InvoiceFactory', function () {
  describe('->createFromOrder()', function () {
    it('should return an order object', function () {
      $order = new Order();
      $factory = new InvoiceFactory();
      $invoice = $factory->createFromOrder($order);

      expect($invoice)->to->be->instanceof(
        'CleanPhp\Invoicer\Domain\Entity\Invoice'
      );
    });
  });
});
```

This simple test just makes sure that our `InvoiceFactory` is returning an instance of an `Invoice` object.

If we run Peridot again, our test will be failing. So let's go ahead and write the basic `InvoiceFactory` class and make this test pass!

We'll start with the basic structure of the `InvoiceFactory`:

```
// src/Domain/Factory/InvoiceFactory.php

namespace CleanPhp\Invoicer\Domain\Factory;

use CleanPhp\Invoicer\Domain\Entity\Invoice;
use CleanPhp\Invoicer\Domain\Entity\Order;

class InvoiceFactory {
  public function createFromOrder(Order $order) {
    return new Invoice();
  }
}
```

This is the minimal work needed to get our Peridot tests to pass, but our class obviously still isn't work the way we want it to as it's just returning an empty `Invoice` object. Let's add a few more expectations to our test to define the behavior of this factory:

```
// specs/domain/factory/invoice-factory.spec.php

use CleanPhp\Invoicer\Domain\Entity\Invoice;
use CleanPhp\Invoicer\Domain\Entity\Order;
use CleanPhp\Invoicer\Domain\Factory\InvoiceFactory;

describe('InvoiceFactory', function () {
  describe('->createFromOrder()', function () {
    it('should return an order object', function () {
      $order = new Order();
      $factory = new InvoiceFactory();
      $invoice = $factory->createFromOrder($order);

      expect($invoice)->to->be->instanceof(
        'CleanPhp\Invoicer\Domain\Entity\Invoice'
      );
    });

    it('should set the total of the invoice', function () {
      $order = new Order();
      $order->setTotal(500);

      $factory = new InvoiceFactory();
      $invoice = $factory->createFromOrder($order);

      expect($invoice->getTotal())->to->equal(500);
    });

    it('should associate the Order to the Invoice', function () {
      $order = new Order();

      $factory = new InvoiceFactory();
      $invoice = $factory->createFromOrder($order);

      expect($invoice->getOrder())->to->equal($order);
    });

    it('should set the date of the Invoice', function () {
      $order = new Order();

      $factory = new InvoiceFactory();
      $invoice = $factory->createFromOrder($order);

      expect($invoice->getInvoiceDate())
        ->to->loosely->equal(new \DateTime());
    });
  });
});
```

Not only do we want our `InvoiceFactory` to return an instance of an `Invoice` object, but it should also have its `$total` property set to the `$total` of the `Order`, as well as have the `Order` that it was generated for assigned to it, and finally, today's date should be set as the `$invoiceDate` of the `Invoice`.

Now we have some pretty robust tests for what we want this factory to do! Let's make these tests pass now by filling out the rest of the `createFromOrder()` method:

```
public function createFromOrder(Order $order) {
  $invoice = new Invoice();
  $invoice->setOrder($order);
  $invoice->setInvoiceDate(new \DateTime());
  $invoice->setTotal($order->getTotal());

  return $invoice;
}
```

And with that, our `InvoiceFactory` is now complete and its tests passing.

Writing tests in this manner allows us to quickly define the behavior of a class and flesh out how it will work and relate to other objects. This falls into the realm of Behavior-Driven Development (BDD)[35], which you can look into in more detail if it interests you. It goes hand-in-hand quite well with Domain-Driven Development and Test-Driven Development.

[35]http://dannorth.net/introducing-bdd/

Peridot has a handy watcher plugin that allows for continuously running the tests as you make changes. We can install it by running:

```
composer require --dev peridot-php/peridot-watcher-plugin
```

After that, we'll create a `peridot.php` file in the root directory that looks like:

```
// peridot.php

use Evenement\EventEmitterInterface;
use Peridot\Plugin\Watcher\WatcherPlugin;

return function(EventEmitterInterface $emitter) {
  $watcher = new WatcherPlugin($emitter);
  $watcher->track(__DIR__ . '/src');
};
```

Now we can use the `--watch` flag to run the tests continuously!

```
./vendor/bin/peridot specs/ --watch
```

Invoicing Service

The biggest piece of our domain logic in this application is the invoicing process. We're going to go collect all uninvoiced orders, once a month, and generate invoices for them. It's our goal to do this independently of any framework, library, or other external service. This way, we can ensure that the core application is completely uncoupled from anything but itself, and it can easily be dropped into any framework and perform it's function.

This service is going to use the `OrderRepositoryInterface` to collect the orders to invoice, and then use our `InvoiceFactory` to create the invoices for those orders. Let's again start by writing some tests to define these behaviors:

```
// specs/domain/service/invoice-factory.spec.php

describe('InvoicingService', function () {
  describe('->generateInvoices()', function () {
    it('should query the repository for uninvoiced Orders');
    it('should return an Invoice for each uninvoiced Order');
  });
});
```

Things are a little trickier this time. Since we haven't written any concrete repositories yet, we can't use one to perform this test. Instead of writing and using a concrete repository, we're going to mock one using the Prophecy library.

Luckily, Peridot also comes with a plugin to make integrating those a piece of cake. Let's install it:

```
composer require --dev peridot-php/peridot-prophecy-plugin
```

And then add it to our `peridot.php` file:

```
// peridot.php

use Evenement\EventEmitterInterface;
use Peridot\Plugin\Prophecy\ProphecyPlugin;
use Peridot\Plugin\Watcher\WatcherPlugin;

return function(EventEmitterInterface $emitter) {
  $watcher = new WatcherPlugin($emitter);
  $watcher->track(__DIR__ . '/src');

  new ProphecyPlugin($emitter);
};
```

Now, we can write a `beforeEach()` block that will get executed before each test to build us a mocked `OrderRepositoryInterface`:

```
// specs/domain/service/invoice-factory.spec.php

$repo = 'CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface';

describe('InvoicingService', function () {
  describe('->generateInvoices()', function () {
    beforeEach(function () {
      $this->repository = $this->getProphet()->prophesize($repo);
    });
    // ...
  });
});
```

When we need it, `$this->repository` will hold an instance of a mocked repository. Now we can finish our first test:

```php
it('should query the repository for uninvoiced Orders', function () {
  $this->repository->getUninvoicedOrders()->shouldBeCalled();
  $service = new InvoicingService($this->repository->reveal());
  $service->generateInvoices();
});
```

We're testing here that when we call the `generateInvoices()` method, we should expect that the `InvoicingService` will make a call to the `getUninvoicedOrders()` method of `OrderRepositoryInterface`.

We'll also want to add an `afterEach()` block to tell Prophecy to check the assertions it makes, like `shouldBeCalled()` as otherwise it won't know exactly when it's safe to check that assertion. We can do it in an `afterEach()` just to make it easy on us, but really we could add it anywhere within the test:

```php
afterEach(function () {
  $this->getProphet()->checkPrediections();
});
```

Of course, without any code, these tests should be failing, so let's go fix that:

```php
// src/Domain/Service/InvoicingService.php

namespace CleanPhp\Invoicer\Domain\Service;

use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;

class InvoicingService {
  protected $orderRepository;

  public function __construct(OrderRepositoryInterface $orderRepository) {
    $this->orderRepository = $orderRepository;
  }

  public function generateInvoices() {
    $orders = $this->orderRepository->getUninvoicedOrders();
  }
}
```

We're simply doing exactly what our test expected: injecting an instance of `OrderRepositoryInterface` and calling its `getUninvoicedOrders()` method when calling the `generateUninvoicedOrders()` method.

Our tests should now pass, so let's dig deeper into the functionality of this service:

```
it('should return an Invoice for each uninvoiced Order', function () {
  $orders = [(new Order())->setTotal(400)];
  $invoices = [(new Invoice())->setTotal(400)];

  $this->repository->getUninvoicedOrders()->willReturn($orders);
  $this->factory->createFromOrder($orders[0])->willReturn($invoices[0]);

  $service = new InvoicingService(
    $this->repository->reveal(),
    $this->factory->reveal()
  );
  $results = $service->generateInvoices();

  expect($results)->to->be->a('array');
  expect($results)->to->have->length(count($orders));
});
```

We've now brought the `InvoiceFactory` into the picture, and are testing that its `createFromOrder` method is called with the results of `getUninvoicedOrders`, meaning that each `Order` returned should be run through the `InvoiceFactory` to generate an `Invoice`.

We're using the `willReturn()` method of Prophecy to instruct the Mock to return `$orders` whenever `getUninvoicedOrders()` is called, and that `$invoices[0]` should be returned when `createFromOrder()` is called with `$orders[0]` as an argument.

Finally, we're doing some expectations after instantiating our object and calling the `generateInvoices()` method to ensure that it is returning the proper data.

Since we're now utilizing the `InvoiceFactory`, which we'll need to mock as well, as we want to be able to test the `InvoicingService` in isolation without having to test the `InvoiceFactory` as well, so let's add that mock to the `beforeEach()`:

```
beforeEach(function () {
  $repo = 'CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface';
  $this->repository = $this->getProphet()->prophesize($repo);
  $this->factory = $this->getProphet()
    ->prophesize('CleanPhp\Invoicer\Domain\Factory\InvoiceFactory');
});
```

And we'll have to update the other test to inject it as well, otherwise it will throw errors:

```php
it('should query the repository for uninvoiced Orders', function () {
  $this->repository->getUninvoicedOrders()->shouldBeCalled();
  $service = new InvoicingService(
    $this->repository->reveal(),
    $this->factory->reveal()
  );
  $service->generateInvoices();
});
```

Our tests are, of course, failing as the code isn't setup to meet the expectations of the tests, so let's go finalize the `InvoicingService`:

```php
// src/Domain/Service/InvoicingService.php

namespace CleanPhp\Invoicer\Domain\Service;

use CleanPhp\Invoicer\Domain\Factory\InvoiceFactory;
use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;

class InvoicingService{
  protected $orderRepository;
  protected $invoiceFactory;

  public function __construct(
    OrderRepositoryInterface $orderRepository,
    InvoiceFactory $invoiceFactory
  ) {
    $this->orderRepository = $orderRepository;
    $this->invoiceFactory = $invoiceFactory;
  }

  public function generateInvoices() {
    $orders = $this->orderRepository->getUninvoicedOrders();

    $invoices = [];

    foreach ($orders as $order) {
      $invoices[] = $this->invoiceFactory->createFromOrder($order);
    }

    return $invoices;
  }
}
```

We're now accepting an instance of `InvoiceFactory` in the constructor, and using it while looping the results of `getUninvoicedOrders()` to create an Invoice for each Order. When we're done, we return this collection. Exactly as our behavior was defined in the test.

Our tests are passing, and our service is now complete.

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 02-domain-services:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 02-domain-services
```

Wrapping it Up

This concludes the domain of our application. The domain model and domain services layers of our application are central to everything else. They contain our business logic, which is the portion of our application that will not change depending on which libraries or frameworks we decide to use, or how we choose to persist our database.

Everything that exists outside of these two layers will utilize them to complete the goals of the application.

We're now ready to start building out the front-end portion of the application!

Zend Framework 2 Setup

We're going to start building our project using Zend Framework 2. ZF2 is the second iteration of Zend Framework, and is a modular, event-based framework with an extensive set of libraries. It is, at least out of the box using the sample application, an MVC framework with support for various database systems.

While ZF2 isn't my first choice in frameworks, it might be one someone lands upon when first looking for a framework, especially due to the incorrect assumption that it might actually be endorsed by or affiliated with PHP itself. This misconception likely stems from the fact that the current runtime for PHP is called the Zend Engine, and the two people who started it, Andi Gutmans and Zeev Suraski (Zeev + Andi = Zend), later started a company, Zend Technologies, which is responsible for Zend Framework.

Regardless, Zend Framework is not directly affiliated with nor endorsed by PHP.

Installing with Composer

We installed Composer in the previous chapter to setup our autoloader. Now we're going to use it to pull down and configure the ZF Skeleton Application. Since you can't clone a git repo into an existing, non-empty directory, we're going to have to get silly for a minute in order to get this to work.

We'll create our ZF Skeleton Application in a separate directory from our previous `cleanphp/` directory:

```
cd /path/to/your/preferred/www
composer create-project \
  --repository-url="http://packages.zendframework.com" \
  -sdev zendframework/skeleton-application \
  cleanphp-skeleton
```

This command will ask you if you want to "remove the existing VCS history." When it does, enter "Y" to get rid of the `.git` directory.

You should now have a working copy of the ZF2 Skeleton application. Head to the `public/` directory and fire up PHP's built-in web server:

```
cd cleanphp-skeleton
php -S localhost:1337 -t public
```

Head to *http://localhost:1337/* in your web browser, and you should see the results!

Combining the Code Bases

Now we have two separate projects, so we'll want to move over the ZF2 specific code into our `cleanphp/` directory. Something like this from the parent directory of both `cleanphp*` directories:

```
cp -R cleanphp-skeleton/config \
  cleanphp-skeleton/data \
  cleanphp-skeleton/public \
  cleanphp-skeleton/module \
  cleanphp-skeleton/init_autoloader.php \
  cleanphp/
```

We'll also want to make sure that Zend Framework is installed via Composer in this project:

```
composer require zendframework/zendframework
```

Now we can remove the `cleanphp-skeleton/` directory.

```
rm -rf cleanphp-skeleton/
```

That was uncomfortable and awkward, so let's get going with ZF2!

Cleaning up the Skeleton

Now it's time to bend the ZF2 skeleton to our will. We're just going to do some cosmetic stuff real quick to get the ZF2 branding out of the way.

First, let's replace the `module/Application/view/layout/layout.phtml` file with:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>CleanPhp</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <link href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.1/css/bootstrap.css"
    media="screen" rel="stylesheet" type="text/css">
  <link href="/css/application.css" media="screen"
    rel="stylesheet" type="text/css">
</head>
<body>
<nav class="navbar navbar-default navbar-fixed-top" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <a class="navbar-brand" href="/">CleanPhp</a>
    </div>
    <div class="collapse navbar-collapse">
      <ul class="nav navbar-nav">
```

```
        <li>
          <a href="/customers">Customers</a>
        </li>
        <li>
          <a href="/orders">Orders</a>
        </li>
        <li>
          <a href="/invoices">Invoices</a>
        </li>
      </ul>
    </div>
  </div>
</nav>
<div class="container">
  <?= $this->content; ?>
  <hr>
  <footer>
    <p>I'm the footer.</p>
  </footer>
</div>
</body>
</html>
```

Here, we're just ditching a lot of the ZF2 view helpers and layout and opting to use a CDN supplied version of Bootstrap. We can go ahead and entirely delete the `public/css`, `public/fonts`, `public/img`, and `public/js` folders.

We defined some links to some future pages in the header. Let's go ahead and setup the routes for those in Zend Framework:

```
// module/Application/config/module.config.php

return [
  // ...
  'router' => [
    'routes' => [
      'home' => [
        'type' => 'Zend\Mvc\Router\Http\Literal',
        'options' => [
          'route'    => '/',
          'defaults' => [
            'controller' => 'Application\Controller\Index',
            'action'     => 'index',
          ],
        ],
      ],
      'customers' => [
        'type' => 'Segment',
        'options' => [
          'route'    => '/customers',
          'defaults' => [
            'controller' => 'Application\Controller\Customers',
            'action'     => 'index',
          ],
        ],
      ],
      'orders' => [
        'type' => 'Segment',
        'options' => [
          'route'    => '/orders',
          'defaults' => [
            'controller' => 'Application\Controller\Orders',
            'action'     => 'index',
          ],
        ],
      ],
      'invoices' => [
        'type' => 'Segment',
        'options' => [
          'route'    => '/invoices',
          'defaults' => [
            'controller' => 'Application\Controller\Invoices',
            'action'     => 'index',
          ],
        ],
      ],
    ],
  ],
  // ...
];
```

Now let's replace the `module/Application/views/application/index/index.phtml` file with something generic, and not a ZF2 advertisement:

```
<div class="jumbotron">
  <h1>Welcome to CleanPhp Invoicer!</h1>
  <p>
    This is the case study project for The Clean Architecture in PHP,
    a book about writing excellent PHP code.
  </p>
  <p>
    <a href="https://leanpub.com/cleanphp" class="btn btn-primary">
      Check out the Book</a>
  </p>
</div>
```

Now it's an advertisement for this book. How nice! Things look a little off, though, so let's add our `public/css/application.css` file to fix that:

```
body {padding-top: 70px; padding-bottom: 40px}
.navbar-brand {font-weight: bold}
div.page-header {margin-top: 0; padding-top: 0}
div.page-header h2 {margin-top: 0; padding-top: 0}
```

Now we're ready to start configuring our database with Zend Framework.

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 03-base-zf2:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 03-base-zf2
```

Setting up Our Database

To setup our database and use it within Zend Framework, we're going to follow the ZF2 Getting Started[36] guide to ensure we do things the Zend way. I'm going to be brisk on my explanations of what we're doing, so refer to this guide for more details if you are interested.

Let's get started by creating our database. I'm going to use a sqlite3 database in these examples, as it's painfully easy to setup (at least on a Unix/Linux environment), but if you're a fan of MySQL or PostgreSQL and want to use one of them, that's perfect.

[36]http://framework.zend.com/manual/current/en/user-guide/database-and-models.html

If you're using Debian/Ubuntu, installing sqlite is as simple as:

```
sudo apt-get install sqlite3 php5-sqlite
```

On Mac OS X, you can use Homebrew[37] to install sqlite.

Let's quickly create our database, which we'll create at `data/database.db`, via command line:

```
sqlite3 data/database.db
```

We're now in the command line sqlite3 application. We can easily drop SQL queries in here and run them. Let's create our tables:

```
CREATE TABLE customers (
  id integer PRIMARY KEY,
  name varchar(100) NOT NULL,
  email varchar(100) NOT NULL
);

CREATE TABLE orders (
  id integer PRIMARY KEY,
  customer_id int REFERENCES customers(id) NOT NULL,
  order_number varchar(20) NOT NULL,
  description text NOT NULL,
  total float NOT NULL
);

CREATE TABLE invoices (
  id integer PRIMARY KEY,
  order_id int REFERENCES orders(id) NOT NULL,
  invoice_date date NOT NULL,
  total float NOT NULL
);
```

You can run the `.tables` command to see the newly created tables, or `.schema` to see the schema definition.

Now let's populate our `customers` table with a couple rows for test data:

```
INSERT INTO customers(name, email) VALUES('Acme Corp', 'ap@acme.com');
INSERT INTO customers(name, email) VALUES('ABC Company', 'invoices@abc.com');
```

[37]http://brew.sh/

Connecting to the Database

We want our ZF2 application to be able to connect to this database. Zend Framework has a set of configuration files located within `config/autoload` that get loaded automatically when the application is run. If the file ends with `local.php`, it is specific to that local environment. If the file ends with `global.php`, it is application specific, instead of environment specific.

Let's create a `db.local.php` file in `config/autoload` to hold our database configuration:

```
return [
  'db' => [
    'driver' => 'Pdo_Sqlite',
    'database' => __DIR__ . '/../../data/database.db',
  ],
];
```

This tells ZF2 that for our database, we want to use the `Pdo_Sqlite` driver, and that our database file is located at `data/database.db`, after doing some back tracking from the current file's directory to get there.

Any `*.local.php` file is not supposed to be committed to source control. Instead, you should commit a `*.local.php.dist` explaining how the configuration file should be set up. This keeps secrets, such as database passwords, from being committed to source control and potentially leaked or exposed.

Since we don't have any secrets here, and in the interest of committing a workable app, I'm going to put this file in source control anyway.

We've now done everything we need to do to tell ZF2 how to talk to our database. Now we just have to write some code to do it.

Table Data Gateway Pattern

Zend Framework 2 uses the Table Data Gateway Pattern, which we very briefly mentioned in Design Patterns, A Primer. In the **Table Data Gateway Pattern**, a single object acts as a gateway to a database table, handling the retrieving and persisting of all rows for that table. [38] This pattern is described in great detail in Martin Fowler's Patterns of Enterprise Application Architecture[39].

Essentially, we're going to have one object, a Data Table, which represents all operations on one of our Entity classes. We're going to go ahead and make these classes implement our Repository Interfaces, so that they can fulfill the needed contract in our code.

We'll place all these files within the `src/Persistence/Zend` directory as our Zend Persistence layer. Let's start with an `AbstractDataTable` class nested under the `DataTable/` directory that will define our generic database operations that the rest of our DataTable classes can inherit from:

[38]http://martinfowler.com/eaaCatalog/tableDataGateway.html

[39]http://martinfowler.com/books/eaa.html

```php
// src/Persistence/Zend/DataTable/AbstractDataTable.php

namespace CleanPhp\Invoicer\Persistence\Zend\DataTable;

use CleanPhp\Invoicer\Domain\Entity\AbstractEntity;
use CleanPhp\Invoicer\Domain\Repository\RepositoryInterface;
use Zend\Db\TableGateway\TableGateway;
use Zend\Stdlib\Hydrator\HydratorInterface;

abstract class AbstractDataTable implements RepositoryInterface {
  protected $gateway;
  protected $hydrator;

  public function __construct(
    TableGateway $gateway,
    HydratorInterface $hydrator
  ) {
    $this->gateway = $gateway;
    $this->hydrator = $hydrator;
  }

  public function getById($id) {
    $result = $this->gateway
      ->select(['id' => intval($id)])
      ->current();

      return $result ? $result : false;
  }

  public function getAll() {
    $resultSet = $this->gateway->select();
    return $resultSet;
  }

  public function persist(AbstractEntity $entity) {
    $data = $this->hydrator->extract($entity);

    if ($this->hasIdentity($entity)) {
      $this->gateway->update($data, ['id' => $entity->getId()]);
    } else {
      $this->gateway->insert($data);
      $entity->setId($this->gateway->getLastInsertValue());
    }

    return $this;
  }
  public function begin() {
    $this->gateway->getAdapter()
      ->getDriver()->getConnection()->beginTransaction();
    return $this;
  }

  public function commit() {
    $this->gateway->getAdapter()
      ->getDriver()->getConnection()->commit();
    return $this;
  }

  protected function hasIdentity(AbstractEntity $entity) {
    return !empty($entity->getId());
  }
}
```

We're defining our basic database operations - the ones required by our `RepositoryInterface` that all other repositories inherit from. These methods are mostly just wrappers around Zend's `TableGateway` (that we'll take a look at in just a minute).

The only interesting piece we have here is the `hasIdentity()` method, which just (loosely) determines if our entity had already been persisted, so that we know whether we're doing an `insert()` or `update()` operation. We're relying on the presence of an ID here, which might not always work. It's good enough for now.

TableGateway

The first thing that our `AbstractDataTable` requires is an instance of `TableGateway`. The `TableGateway` is Zend's workhorse that does all the database heavy lifting. As you can see by looking at `AbstractDataTable`, all of our operations live off one of it's methods.

We're essentially going to use Zend's concrete implementation, just configured to work with our own tables. We'll define those when we worry about actually instantiating a `DataTable`.

Hydrators

The second thing that wants to be injected into the `AbstractDataTable` is an instance of Zend's `HydratorInterface`. A hydrator is responsible for hydrating an object, meaning, filling out it's attributes with values. In our case, we're going from an array of data to a hydrated entity (think posted form data).

Zend's hydrators are also responsible for data extraction, which is the opposite of hydrating: we take data from a hydrated object and store it in an array representation, which is necessary for Zend's database update operations. You can see how it's used in the `persist()` method above.

For the most part, we'll use a hydrator provided by Zend called the `ClassMethods` hydrator. This hydrator scans the object for set and get methods, and uses them to determine how to hydrate or extract that object.

For instance, if an object has a `setAmount()` method, the hydrator will look for an `amount` key in the array and, if found, pass the value at that key to the `setAmount()` method to hydrate that information to the object.

Likewise, if an object has a `getAmount()` method, the hydrator calls it to get the value and adds an element to the resulting array with the key of `amount` and the value returned from `getAmount()`.

In some instances, we'll use the `ClassMethods` hydrator directly. In others, we'll wrap this hydrator to provide some additional functionality to it.

Customer DataTable

Let's define our `CustomerTable` implementation:

```
// src/Persistence/Zend/DataTable/CustomerTable.php

namespace CleanPhp\Invoicer\Persistence\Zend\DataTable;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;

class CustomerTable extends AbstractDataTable
  implements CustomerRepositoryInterface
{
}
```

The `CustomerTable` class simply implements the `AbstractDataTable` class. Since the `CustomerRepositoryInterface` defines no additional functionality, we can just use the `AbstractDataTable` as is.

Order DataTable

Our `OrderTable` will look pretty much the same as our `CustomerTable`:

```
// src/Persistence/Zend/DataTable/OrderTable.php

namespace CleanPhp\Invoicer\Persistence\Zend\DataTable;

use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;

class OrderTable extends AbstractDataTable
    implements OrderRepositoryInterface
{
    public function getUninvoicedOrders()
    {
        return [];
    }
}
```

Our `OrderRepositoryInterface` defines an extra method that none of the other interfaces have: `getUninvoicedOrders()`. We'll worry about defining this functionality later once we start using it.

Invoice DataTable

Finally, or `InvoiceTable`, much the same:

```
// src/Persistence/Zend/DataTable/InvoiceTable.php

namespace CleanPhp\Invoicer\Persistence\Zend\DataTable;

use CleanPhp\Invoicer\Domain\Repository\InvoiceRepositoryInterface;

class InvoiceTable extends AbstractDataTable
  implements InvoiceRepositoryInterface
{
}
```

Table Gateway Factory

Our Data Tables need to be injected with an instance of a `TableGateway` configured for that particular model. In the ZF Getting Started docs, they define a `TableGateway` for each Data Table being defined. We'll create one dynamically by writing a factory to do so:

```
// src/Persistence/Zend/TableGateway/TableGatewayFactory.php

namespace CleanPhp\Invoicer\Persistence\Zend\TableGateway;

use Zend\Db\Adapter\Adapter;
use Zend\Db\ResultSet\HydratingResultSet;
use Zend\Db\TableGateway\TableGateway;
use Zend\Stdlib\Hydrator\HydratorInterface;

class TableGatewayFactory {
  public function createGateway(
     Adapter $dbAdapter,
     HydratorInterface $hydrator,
     $object,
     $table
  ) {
    $resultSet = new HydratingResultSet($hydrator, $object);
    return new TableGateway($table, $dbAdapter, null, $resultSet);
  }
}
```

Our factory requires an instance of the Zend Database Adapter, which we'll configure in just a bit, as well as an instance of the Hydrator to use. Finally, it accepts an instance of the object that represents the data table, and the name of the database table where the data is stored.

For more information on how this works, see the Zend Getting Started Guide[40].

Configuring Zend Framework

Our last step of setting up the database is to configure Zend Framework to use these new Data Tables. Let's start by defining the `CustomerTable` in the service manager. We'll define this in the `global.php` config file, although in a real application, we'd probably find a much better place to put this:

```
// config/autoload/global.php

use CleanPhp\Invoicer\Domain\Entity\Customer;
use CleanPhp\Invoicer\Persistence\Zend\DataTable\CustomerTable;
use CleanPhp\Invoicer\Persistence\Zend\TableGateway\TableGatewayFactory;
use Zend\Stdlib\Hydrator\ClassMethods;

return [
  'service_manager' => [
    'factories' => [
      'CustomerTable' => function($sm) {
        $factory = new TableGatewayFactory();
        $hydrator = new ClassMethods();

        return new CustomerTable(
          $factory->createGateway(
            $sm->get('Zend\Db\Adapter\Adapter'),
            $hydrator,
            new Customer(),
            'customers'
          ),
          $hydrator
        );
```

[40]http://framework.zend.com/manual/current/en/user-guide/database-and-models.html

```
      },
    ]
  ]
];
```

We use our `TableGatewayFactory` to create a `TableGateway` instance to provide to our `CustomerTable`. We're also passing an instance of the `ClassMethods` hydrator, as well as a `Customer` object and the name of the `customers` table.

Both the `TableGatewayFactory` and the `CustomerTable` need an instance of our hydrator, so we declare that before-hand and provide it as needed to each class.

The only new piece here is the Zend Db Adapter.

We'll need to configure that in the same file:

```
// config/autoload/global.php

// ...

return [
  'service_manager' => [
    'factories' => [
      'Zend\Db\Adapter\Adapter' => 'Zend\Db\Adapter\AdapterServiceFactory',

      // ...
    ],
  ],
];
```

This tells the service manager to use the `AdapterServiceFactory` provided by Zend to give us an instance of `Zend\Db\Adapter\Adapter` when needed. If you want to understand how all this works, take a look at the ZF docs for more information, or dive into Zend's source code if you're feeling extra adventurous.

Finally, we'll setup a nearly identical entry for both the `OrderTable` and `InvoiceTable`:

```
// config/autoload/global.php

use CleanPhp\Invoicer\Domain\Entity\Customer;
use CleanPhp\Invoicer\Domain\Entity\Invoice;
use CleanPhp\Invoicer\Domain\Entity\Order;
use CleanPhp\Invoicer\Persistence\Zend\DataTable\CustomerTable;
use CleanPhp\Invoicer\Persistence\Zend\DataTable\InvoiceTable;
use CleanPhp\Invoicer\Persistence\Zend\DataTable\OrderTable;
use CleanPhp\Invoicer\Persistence\Zend\TableGateway\TableGatewayFactory;
use Zend\Stdlib\Hydrator\ClassMethods;
```

```
return [
  'service_manager' => [
    'factories' => [
      // ...

      'InvoiceTable' => function($sm) {
        $factory = new TableGatewayFactory();
        $hydrator = new ClassMethods();

        return new InvoiceTable(
          $factory->createGateway(
            $sm->get('Zend\Db\Adapter\Adapter'),
            $hydrator,
            new Invoice(),
            'invoices'
          ),
          $hydrator
        );
      },
      'OrderTable' => function($sm) {
        $factory = new TableGatewayFactory();
        $hydrator = new ClassMethods();

        return new OrderTable(
          $factory->createGateway(
            $sm->get('Zend\Db\Adapter\Adapter'),
            $hydrator,
            new Order(),
            'orders'
          ),
          $hydrator
        );
      },
    ],
  ],
];
```

Wrapping it Up

We now have all of our database tables configured and ready to use with Zend Framework 2, as well as our database configured, ready, and loaded with some dummy Customer data.

Let's move forward!

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 04-zf2-database-setup:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 04-zf2-database-setup
```

Our Application in Zend Framework 2

Now that we have Zend Framework configured and ready to rock, as well as our database setup and configured, we can start actually using it.

Let's start with customer management. We stubbed out a route, but when we navigate to that route, we're going to get a sad error message from ZF2:

> A 404 error occurred Page not found. The requested controller could not be mapped to an existing controller class.
>
> Controller: ApplicationControllerCustomers (resolves to invalid controller class or alias: ApplicationControllerCustomers) No Exception available

This makes sense as we defined a route to point to a Customers controller, but didn't bother creating that controller. So let's do that.

Customer Management

Let's start building out our `CustomersController::indexAction()`, which will display a grid of all of our customers.

> ☹ I spent a lot of time trying to figure out how to unit test controllers in Zend Framework. I'm going to call it: it's impossible. Depending on the action, you need to either bootstrap or mock four to forty-four different services, plugins, etc.
>
> Zend provides a great tutorial[41] on testing their controllers. They call it unit testing, but that can only be true if they mean the whole ZF2 ecosystem as a unit.
>
> As such, I'm going to disregard tests for these controllers. If this were real life, I'd bite the bullet and write the integration tests (which are important too). For the sake of this book, that's just too much to bother.

Let's being our `indexAction()`:

[41]http://framework.zend.com/manual/2.0/en/user-guide/unit-testing.html

```
// modules/Application/src/Application/Controller/CustomersController.php

namespace Application\Controller;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use Zend\Mvc\Controller\AbstractActionController;

class CustomersController extends AbstractActionController {
  public $customerRepository;

  public function __construct(
    CustomerRepositoryInterface $customers
  ) {
    $this->customerRepository = $customers;
  }

  public function indexAction() {
    return [
      'customers' => $this->customerRepository->getAll()
    ];
  }
}
```

We have a new `CustomersController` class with an `indexAction()` method. An instance of `CustomerRepositoryInterface` is injected in, and later used by the action to call the `getAll()` method. We return the result of that method in an array, keyed at `customers`.

Now we need a proper view to represent the `indexAction()`, and we should see our data on the screen. Let's drop that view file in:

```
<!-- module/Application/views/application/customers/index.phtml -->

<div class="page-header clearfix">
  <h2 class="pull-left">Customers</h2>
  <a href="/customers/new" class="btn btn-success pull-right">
    Create Customer</a>
</div>

<table class="table">
  <thead>
    <tr>
      <th>#</th>
      <th>Name</th>
      <th>Email</th>
    </tr>
 </thead>
 <?php foreach ($this->customers as $customer): ?>
    <tr>
      <td>
        <a href="/customers/edit/<?= $customer->getId() ?>">
          <?= $customer->getId() ?></a>
      </td>
      <td><?= $customer->getName() ?></td>
      <td><?= $customer->getEmail() ?></td>
    </tr>
  <?php endforeach; ?>
</table>
```

Lastly, we'll need to configure ZF2 to know that `CustomersController` is the `Customers` controller we referenced in the route. If if we had called it `CustomersController` in the route, ZF2 still wouldn't know what we're talking about as the string here is simply the key within the controller service locator.

In the `controllers` section of the module config file, we'll add an entry for our new controller:

```
// module/Application/config/module.config.php

return [
  // ...
  'controllers' => [
    'invokables' => [
      'Application\Controller\Index' =>
        'Application\Controller\IndexController'
    ],
    'factories' => [
      'Application\Controller\Customers' => function ($sm) {
        return new \Application\Controller\CustomersController(
          $sm->getServiceLocator()->get('CustomerTable')
        );
      },
    ],
  ],
  // ...
];
```

Unlike the main `IndexController`, this `CustomersController` entry will be registered with ZF as a factory, so that it's not just instantiated outright, but allows us to bake in logic about how it's instantiated, which allows us to inject the proper dependencies. We're using the entry we defined in the last chapter for `CustomerTable` to grab our Customer Data Table, which implements the `CustomerRepositoryInterface` and satisfies the type-hint on the constructor of the `CustomersController`.

So now if we navigate to `/customers` in our beloved browser, we should see all of our customers from our sqlite database rendered on to the screen. Success!

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 05-viewing-customers:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 05-viewing-customers
```

Creating Customers

In our HTML, we have a button for creating new customers that brings the user to the route `/customers/new`. At this route, we'll render a form that, when correctly filled out, will then post back to the same route where we'll persist the new information to the database as a new customer.

Let's start building out our `CustomersController->newAction()` to handle simple GET requests.

CustomersController->newAction()

Let's start building out our `newAction()` and the corresponding view file:

```
// module/Application/src/Application/Controllers/CustomersController.php

namespace Application\Controller;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use Zend\Mvc\Controller\AbstractActionController;

class CustomersController extends AbstractActionController {
  // ...

  public function newAction() {

  }
}
```

This simple controller action is all we initially need. Let's build out our view:

```
<!-- module/Application/view/application/customers/new.phtml -->

<div class="page-header clearfix">
  <h2>New Customer</h2>
</div>

<form role="form" action="" method="post">
  <div class="form-group">
    <label for="name">Name:</label>
    <input type="text" class="form-control" name="name" id="name"
      placeholder="Enter Name">
  </div>
  <div class="form-group">
    <label for="email">Email:</label>
    <input type="text" class="form-control" name="email" id="email"
      placeholder="Enter Email">
  </div>
  <button type="submit" class="btn btn-primary">Save</button>
</form>
```

Next, we'll need to update our routing to handle `/customers/new`:

```
// module/Application/config/module.config.php

return [
  'router' => [
    'routes' => [
      // ...
      'customers' => [
        // ...
        'may_terminate' => true,
        'child_routes' => [
          'create' => [
            'type' => 'Segment',
            'options' => [
              'route' => '/new',
              'defaults' => [
                'action' => 'new',
              ],
            ]
          ],
        ]
      ],
      // ...
    ],
  ],
  // ...
];
```

This simple child route will combine `/customers` of the parent route, with `/new` of the child route to give us our `/customers/new` route, which will point to the `newAction()` of our `CustomersController()`.

Now if we click on the *Create Customer* link, we should see our new form rendered. Now we just have to make this form do something.

CustomerInputFilter

We're going to use Zend's `InputFilter` to validate and sanitize our input. You can read more about Zend's Input Filters in their documentation[42], but essentially, they give us a set of classes to validate and sanitize input data.

We're going to drop our InputFilters into the `src/` directory, as we'll want to use them when we decide to switch away from Zend Framework. Otherwise, we'd have to build a whole new solution for validating input data, which would be fine, but it's nice not to have to do that at the same time.

We'll start by writing a spec to describe the behavior we want. First, we'll need an instance of our soon-to-be new `CustomerInputFilter` for testing:

```php
// specs/input-filter/customer.spec.php

use CleanPhp\Invoicer\Service\InputFilter\CustomerInputFilter;

describe('InputFilter\Customer', function () {
  beforeEach(function () {
      $this->inputFilter = new CustomerInputFilter();
  });

  describe('->isValid()', function () {
    // ...
  });
});
```

We'll be interested in testing the `isValid()` method, which Zend provides to determine whether an `InputFilter`'s data is valid. We'll also use the `setData()` method to supply the `InputFilter` with some data to test.

Let's start with testing validity of the customer name:

[42]http://framework.zend.com/manual/current/en/modules/zend.input-filter.intro.html

```
it('should require a name', function () {
  $isValid = $this->inputFilter->isValid();

  $error = [
    'isEmpty' => 'Value is required and can\'t be empty'
  ];

  $messages = $this->inputFilter
    ->getMessages()['name'];

  expect($isValid)->to->equal(false);
  expect($messages)->to->equal($error);
});
```

Last, we'll test the validity of the email address. Here, we're not particularly worried about the exact messages ZF2 returns when we have invalid data, just that we get some kind of array of errors back, rather than `null`:

```
it('should require an email', function () {
  $isValid = $this->inputFilter->isValid();

  $error = [
    'isEmpty' => 'Value is required and can\'t be empty'
  ];

  $messages = $this->inputFilter
    ->getMessages()['email'];

  expect($isValid)->to->equal(false);
  expect($messages)->to->equal($error);
});

it('should require a valid email', function () {
  $scenarios = [
    [
      'value' => 'bob',
      'errors' => []
    ],
    [
      'value' => 'bob@bob',
      'errors' => []
    ],
    [
      'value' => 'bob@bob.com',
      'errors' => null
    ]
  ];

  foreach ($scenarios as $scenario) {
    $this->inputFilter->setData([
      'email' => $scenario['value']
    ])->isValid();

    $messages = $this->inputFilter
      ->getMessages()['email'];

    if (is_array($messages)) {
      expect($messages)->to->be->a('array');
      expect($messages)->to->not->be->empty();
    } else {
      expect($messages)->to->be->null();
    }
  }
});
```

We can add some more robust data to the list of tested `$scenarios` if we want to more fully test the email RFC for valid emails, but we can also trust that ZF2 handles all the cases pretty well. We just want to make sure that our `CustomerInputFilter` is setting up the validation rules correctly.

Now let's write a new `InputFilter` class for Customer data:

```
// src/Service/InputFilter/CustomerInputFilter.php

namespace CleanPhp\Invoicer\Service\InputFilter;

use Zend\InputFilter\Input;
use Zend\InputFilter\InputFilter;
use Zend\Validator\EmailAddress;

class CustomerInputFilter extends InputFilter {
  public function __construct() {
    $name = (new Input('name'))
      ->setRequired(true);

    $email = (new Input('email'))
      ->setRequired(true);
    $email->getValidatorChain()->attach(
      new EmailAddress()
    );
    $this->add($name);
    $this->add($email);
  }
}
```

Posting Customer Data

Our next step is to utilize this `CustomerInputFilter` in our `CustomersController`. We'll want to do this when we receive a POST request only, and if we receive validation errors, we should report those back to the user. Let's start by writing a spec of our intended behavior.

First, we'll need to inject an instance of the `CustomerInputFilter` into the `CustomersController` as part of the test:

```php
// module/Application/src/Application/Controller/CustomersController.php

namespace Application\Controller;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use CleanPhp\Invoicer\Service\InputFilter\CustomerInputFilter;
use Zend\Mvc\Controller\AbstractActionController;

class CustomersController extends AbstractActionController {
  protected $customerRepository;
  protected $inputFilter;

  public function __construct(
    CustomerRepositoryInterface $customers,
    CustomerInputFilter $inputFilter
  ) {
    $this->customerRepository = $customers;
    $this->inputFilter = $inputFilter;
  }

  // ...
}
```

Now we can update the `newAction()` to handle a POST request:

```php
// module/Application/src/Application/Controller/CustomersController.php

namespace Application\Controller;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use CleanPhp\Invoicer\Service\InputFilter\CustomerInputFilter;
use Zend\Mvc\Controller\AbstractActionController;

class CustomersController extends AbstractActionController {
  // ...

  public function newAction() {
    if ($this->getRequest()->isPost()) {
      $this->inputFilter->setData($this->params()->fromPost());
      if ($this->inputFilter->isValid()) {

      } else {

      }
    }
  }
}
```

First, we determine if the request is a `POST` request. If it is, we supply our InputFilter with the posted form data, then check to see if the InputFilter is valid, given that data.

We have two remaining paths to implement:

1. The data is valid
2. The data is invalid

When the data is valid, we want to persist it to our repository. However, the data coming in from the POST is a giant array. We need to be able to persist an instance of `Customer`. The best way to handle this is to hydrate a `Customer` object with the POST data. To do that, we'll need to inject an instance of a `HydratorInterface` into the controller:

```php
// module/Application/src/Application/Controller/CustomersController.php

namespace Application\Controller;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use CleanPhp\Invoicer\Service\InputFilter\CustomerInputFilter;
use Zend\Mvc\Controller\AbstractActionController;
use Zend\StdLib\Hydrator\HydratorInterface;

class CustomersController extends AbstractActionController {
  protected $customerRepository;
  protected $inputFilter;

  public function __construct(
    CustomerRepositoryInterface $customers,
    CustomerInputFilter $inputFilter,
    HydratorInterface $hydrator
  ) {
    $this->customerRepository = $customers;
    $this->inputFilter = $inputFilter;
    $this->hydrator = $hydrator;
  }

  // ...
}
```

We'll also want to update our controller config to inject these two new objects that the `CustomersController` needs:

```
// module/Application/config/module.config.php

use CleanPhp\Invoicer\Service\InputFilter\CustomerInputFilter;
use Zend\Stdlib\Hydrator\ClassMethods;

return [
  // ...
  'controllers' => [
    'invokables' => [
      'Application\Controller\Index' =>
        'Application\Controller\IndexController'
    ],
    'factories' => [
      'Application\Controller\Customers' => function ($sm) {
        return new \Application\Controller\CustomersController(
          $sm->getServiceLocator()->get('CustomerTable'),
          new CustomerInputFilter(),
          new ClassMethods()
        );
      },
    ],
  ],

  // ...
];
```

Next, we're going to use the hydrator to build a `Customer` object, and then persist that customer object using our `CustomerRepository`:

```php
public function newAction() {
  if ($this->getRequest()->isPost()) {
    $this->inputFilter->setData($this->params()->fromPost());
    if ($this->inputFilter->isValid()) {
      $customer = $this->hydrator->hydrate(
        $this->inputFilter->getValues(),
        new Customer()
      );

      $this->customerRepository->begin()
        ->persist($customer)
        ->commit();
    } else {

    }
  }
}
```

We'll also need a `use` statement for the `Customer` class at the top of the file.

At this point, we can enter a new Customer in the browser and have it persisted to the database. But afterward, the user is dumped back to the New Customer page with no indication that their save was successful.

Let's add a redirect to the `/customers` page, as well as a flash message alerting them that the save was successful:

```php
public function newAction() {
  if ($this->getRequest()->isPost()) {
    $this->inputFilter->setData($this->params()->fromPost());
    if ($this->inputFilter->isValid()) {
      // ...

      $this->flashMessenger()->addSuccessMessage('Customer Saved');
      $this->redirect()->toUrl('/customers');
    } else {

    }
  }
}
```

If you give it a shot in the browser, you should now be redirected to the `/customers` page. In order to get the flash message to show up, we'll need to setup our `layout.phtml` file to render flash messages. Zend provides a helper[43] to easily display these flash messages, but it looks terrible. We'll create our own partial file to render them, and then include that in our `layout.phtml` file.

[43]http://framework.zend.com/manual/current/en/modules/zend.view.helpers.flash-messenger.html

```
<!-- view/application/partials/flash-messages.phtml -->
<?php
$flash = $this->flashMessenger();
$flash->setMessageOpenFormat('<div%s role="alert">
   <button type="button" class="close"
     data-dismiss="alert" aria-label="Close">
     <span aria-hidden="true">&times;</span>
   </button>
   <div>')
  ->setMessageSeparatorString('</div><div>')
  ->setMessageCloseString('</div></div>');
?>

<?= $this->flashMessenger()->render(
  'success',
  ['alert', 'alert-dismissible', 'alert-success']
) ?>
```

This is a bunch of bootstrapping to style the existing Zend helper, then using that helper to generate the messages.

Let's include it in the `layout.phtml` file:

```
<!-- ... ->
<div class="container">
  <?= $this->partial('application/partials/flash-messages') ?>
  <?= $this->content; ?>
  <hr>
  <footer>
    <p>I'm the footer.</p>
  </footer>
</div>
<!-- ... -->
```

Now our flash message should be rendered when we create new customers.

Handling Validation Errors

On `InputFilter->isValid()` failure, we'll want to do two things: hydrate and return a `Customer` object with the submitted data, so we can persist it to the form, and return the validation error messages so we can show them to the user.

We'll use the already injected `HydratorInterface`, but this time, instead of hydrating sanitized data from the InputFilter, we're going to hydrate the data directly posted:

```php
public function newAction() {
  $viewModel = new ViewModel();
  $customer = new Customer();

  if ($this->getRequest()->isPost()) {
    $this->inputFilter->setData($this->params()->fromPost());

    if ($this->inputFilter->isValid()) {
      $this->hydrator->hydrate(
        $this->inputFilter->getValues(),
        $customer
      );

      $this->customerRepository->begin()
        ->persist($customer)
        ->commit();

      $this->flashMessenger()->addSuccessMessage('Customer Saved');
      $this->redirect()->toUrl('/customers');
    } else {
      $this->hydrator->hydrate(
        $this->params()->fromPost(),
        $customer
      );
    }
  }

  $viewModel->setVariable('customer', $customer);

  return $viewModel;
}
```

Don't forget to drop a `use` statement for `Zend\View\Model\ViewModel` at the top of the file.

We've started by declaring a new `Customer` object that gets passed along to the view. We've updated our valid clause to use this customer, rather than instantiating it's own. We've also updated our `else` condition to hydrate this object with data directly from the `POST`.

Since we're now passing off customer details to the view, we'll need to update our view file to use these values when generating the form, so that they'll show the bad data when we fail validation:

```
<!-- module/Application/view/application/customers/new.phtml -->
<div class="page-header clearfix">
  <h2>New Customer</h2>
</div>

<form role="form" action="" method="post">
  <div class="form-group">
    <label for="name">Name:</label>
    <input type="text" class="form-control" name="name" id="name"
      placeholder="Enter Name" value="<?= $this->customer->getName() ?>">
  </div>
  <div class="form-group">
    <label for="email">Email:</label>
    <input type="text" class="form-control" name="email" id="email"
      placeholder="Enter Email" value="<?= $this->customer->getEmail() ?>">
  </div>
  <button type="submit" class="btn btn-primary">Save</button>
</form>
```

We're now using the supplied `$customer` to set the `value` for each input. On `GET`, these values will be empty, but on a failed `POST`, they'll contain the user submitted data.

Now, let's take care of showing the validation messages. First, we'll start by making sure they get passed off to the view in the `newAction()`:

```
public function newAction() {
  $viewModel = new ViewModel();
  $customer = new Customer();

  if ($this->getRequest()->isPost()) {
    // ...

    if ($this->inputFilter->isValid()) {
      // ...
    } else {
      $this->hydrator->hydrate(
        $this->params()->fromPost(),
        $customer
      );
      $viewModel->setVariable('errors', $this->inputFilter->getMessages());
    }
  }

  // ...

  return $viewModel;
}
```

Now we can render these errors in the view file. To do that, we're going to make a custom View Helper to render the error messages, if present, for any given input field.

View Helpers View Helpers in Zend Framework are reusable classes that can accept data and generate HTML. As long as they are configured properly within the service manager, ZF2 takes care of instantiating them for you when you use them in a view.

Using View Helpers involves invoking their name from the `$this` object variable within the view:

```
<?= $this->helperName('some data') ?>
```

We'll create a View Helper to help us display validation messages returned to the view. Our View Helper will live in the `module/Application/src/View/Helper` directory, and we'll call it `ValidationErrors.php`:

```
namespace Application\View\Helper;

use Zend\View\Helper\AbstractHelper;

class ValidationErrors extends AbstractHelper {
  public function __invoke($element) {
    if ($errors = $this->getErrors($element)) {
      return '<div class="alert alert-danger">' .
        implode('. ', $errors) .
      '</div>';
    }

    return '';
  }

  protected function getErrors($element) {
    if (!isset($this->getView()->errors)) {
      return false;
    }

    $errors = $this->getView()->errors;

    if (isset($errors[$element])) {
      return $errors[$element];
    }

    return false;
  }
}
```

This view helper will accept an element, which we use to lookup errors with. If we find some, we return them rendered in pretty HTML. The errors for each element are an array (to allow for multiple errors), so we'll simply implode them and separate them with a period.

Next, we need to let Zend know about this view helper using it's service locator config in `module.config.php`:

```
return [
  // ...
  'view_helpers' => [
    'invokables' => [
      'validationErrors' => 'Application\View\Helper\ValidationErrors',
    ]
  ],
  // ...
];
```

Finally, we can update the view file to use this new helper and display any validation error messages for each field:

```
<!-- module/Application/view/application/customers/new.phtml -->
<div class="page-header clearfix">
  <h2>New Customer</h2>
</div>

<form role="form" action="" method="post">
  <div class="form-group">
    <label for="name">Name:</label>
    <input type="text" class="form-control" name="name" id="name"
      placeholder="Enter Name" value="<?= $this->customer->getName() ?>">
    <?= $this->validationErrors('name') ?>
  </div>
  <div class="form-group">
    <label for="email">Email:</label>
    <input type="text" class="form-control" name="email" id="email"
      placeholder="Enter Email" value="<?= $this->customer->getEmail() ?>">
    <?= $this->validationErrors('email') ?>
  </div>
  <button type="submit" class="btn btn-primary">Save</button>
</form>
```

If we submit the form without any data now, or with data that doesn't meet our validation requirements, such as an invalid email, we should get validation error messages rendered under each field. Any data we do enter should also be preserved in the input field.

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 06-creating-customers:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 06-creating-customers
```

Editing Customers

Our next step is to implement the editing of existing customers. Since this code is going to be very similar to our create customers code, we'll use the same action and modify it slightly to handle both new and existing Customers.

Let's first start by refactoring our `newAction()` to be `newOrEditAction()`, and make sure it still works before continuing. Let's start with the `CustomersController`:

```
// module/Application/src/Application/Controller/CustomersController.php

// ...

class CustomersController extends AbstractActionController {
  // ...

  public function newOrEditAction() {
    // ...
  }
}
```

Next, we'll update the routing config to point to this new action, and also add the edit action while we're at it:

```
// module/Application/config/module.config.php

return [
  'router' => [
    'routes' => [
      // ...

      'customers' => [
        'type' => 'Segment',
        'options' => [/* ... */],
        'may_terminate' => true,
        'child_routes' => [
          'new' => [
            'type' => 'Segment',
            'options' => [
              'route' => '/new',
              'constraints' => [
                'id' => '[0-9]+',
              ],
              'defaults' => [
                'action' => 'new-or-edit',
              ],
            ]
          ],
          'edit' => [
            'type' => 'Segment',
            'options' => [
              'route' => '/edit/:id',
              'constraints' => [
                'id' => '[0-9]+',
              ],
              'defaults' => [
                'action' => 'new-or-edit',
              ],
            ]
          ],
        ]
      ],

      // ...
    ],
  ],
];
```

Finally, let's rename our `view/application/customers/new-or-edit.phtml` file to `module/Application/view/application/customers/new-or-edit.phtml`. At this point, our *Create Customer* button and action should still work. If we click on the id of a row in the `indexAction()`, we should also get a form in the browser, just missing our data. Let's fix that.

The first thing we'll want to do is check for an ID passed via the URL. If we have one, we should get a `Customer` object from the `CustomerRepository`. If there is no ID, we should instantiate a new `Customer` object just like we currently are:

```
public function newOrEditAction() {
  $id = $this->params()->fromRoute('id');
  $customer = $id ? $this->customerRepository->getById($id) : new Customer();

  // ...
}
```

This simple change should be all we need to support editing Customers. Give it a try. Sweet, huh?

We want to do two more things:

1. Link to the Edit Customer page after a successful save.
2. Show Edit Customer as the title instead of New Customer when editing

The first one is easy; we change the redirect line in `newOrEditAction()` to:

```
$this->redirect()->toUrl('/customers/edit/' . $customer->getId());
```

And changing the title in the view is pretty easy, too:

```
<div class="page-header clearfix">
  <h2>
    <?= !empty($this->customer->getId()) ? 'Edit' : 'New' ?>
    Customer
  </h2>
</div>
```

We simply check to see if the `$customer` has an ID to determine if it is an edit or add operation.

Customer Management is now complete!

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 06-editing-customers:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 06-editing-customers
```

Order Management

Let's move on to orders. We're going to start by hand-crafting a List Orders view, much the same way we created a List Customers view. We already defined our basic route for `/orders` earlier in this chapter, so let's continue that by creating our controller that will be served by this route.

For our `indexAction()`, we simply want to get an a collection of all Orders stored within the database. The controller will use an implementation of the `OrderRepositoryInterface`, injected via the constructor, and it's `getAll()` method to get the orders.

```php
// module/Application/src/Application/Controller/OrdersController.php

namespace Application\Controller;

use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;
use Zend\Mvc\Controller\AbstractActionController;

class OrdersController extends AbstractActionController {
  protected $orderRepository;

  public function __construct(OrderRepositoryInterface $orders) {
    $this->orderRepository = $orders;
  }

  public function indexAction() {
    return [
      'orders' => $this->orderRepository->getAll()
    ];
  }
}
```

To use this controller, we'll need to configure it in the controller service config. We can do so right after the `Customers` controller definition:

```php
// module/Application/config/module.config.php

return [
  // ...
  'controllers' => [
    // ...
    'Application\Controller\Orders' => function ($sm) {
      return new \Application\Controller\OrdersController(
        $sm->getServiceLocator()->get('OrderTable')
      );
    },
  ],
  // ...
];
```

Finally, let's drop in a view file to render our list of orders:

```
<!-- module/Application/views/application/orders/index.php -->

<div class="page-header clearfix">
  <h2 class="pull-left">Orders</h2>
  <a href="/orders/new" class="btn btn-success pull-right">
    Create Order</a>
</div>

<table class="table table-striped clearfix">
  <thead>
    <tr>
      <th>#</th>
      <th>Order Number</th>
      <th>Customer</th>
      <th>Description</th>
      <th class="text-right">Total</th>
    </tr>
  </thead>
  <?php foreach ($this->orders as $order): ?>
    <tr>
      <td>
        <a href="/orders/view/<?= $this->escapeHtmlAttr($order->getId()) ?>">
          <?= $this->escapeHtml($order->getId()) ?></a>
      </td>
      <td><?= $this->escapeHtml($order->getOrderNumber()) ?></td>
      <td>
        <a href="/customers/edit/<?=
          $this->escapeHtmlAttr($order->getCustomer()->getId()) ?>">
          <?= $this->escapeHtml($order->getCustomer()->getName()) ?></a>
      </td>
      <td><?= $this->escapeHtml($order->getDescription()) ?></td>
      <td class="text-right">
        $ <?= number_format($order->getTotal(), 2) ?>
      </td>
    </tr>
  <?php endforeach; ?>
</table>
```

If you refresh, you should see an empty grid! If we manually drop a couple orders in the database, we should see some data show up. And a big fat error, because we're trying to access the Customer associated to the Order, but we haven't actually hydrated one.

This is where we start to see the pitfalls of ZF2's Data Table Gateway. But for the sake of getting something done, let's continue on.

Hydrating the Related Customer

In order to hydrate the Customer related to an Order, we'll have to build a custom hydrator. To do so, we'll simply wrap Zend's `ClassMethods` hydrator that we're already using, and add some additional functionality to it.

We'll start by writing a spec to describe the functionality we need:

```php
// specs/hydrator/order.spec.php

use CleanPhp\Invoicer\Domain\Entity\Order;
use CleanPhp\Invoicer\Persistence\Hydrator\OrderHydrator;
use Zend\Stdlib\Hydrator\ClassMethods;

describe('Persistence\Hydrator\OrderHydrator', function () {
  beforeEach(function() {
    $this->hydrator = new OrderHydrator(new ClassMethods());
  });

  describe('->hydrate()', function () {
    it('should perform basic hydration of attributes', function () {
      $data = [
        'id' => 100,
        'order_number' => '20150101-019',
        'description' => 'simple order',
        'total' => 5000
      ];

      $order = new Order();
      $this->hydrator->hydrate($data, $order);

      expect($order->getId())->to->equal(100);
      expect($order->getOrderNumber())->to->equal('20150101-019');
      expect($order->getDescription())->to->equal('simple order');
      expect($order->getTotal())->to->equal(5000);
    });
  });
});
```

If first test case is to make sure that our `OrderHydrator` performs basic hydration of our scalar type values. We'll be passing off this work to the `ClassMethods` hydrator since it's pretty good at it. Next, we'll need to handle our use case for persisting a `Customer` object on the `Order`:

```
use CleanPhp\Invoicer\Domain\Entity\Customer;
use CleanPhp\Invoicer\Domain\Entity\Order;
use CleanPhp\Invoicer\Persistence\Hydrator\OrderHydrator;
use Zend\Stdlib\Hydrator\ClassMethods;

describe('Persistence\Hydrator\OrderHydrator', function () {
  beforeEach(function() {
    $this->repository = $this->getProphet()->prophesize(
      'CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface'
    );
    $this->hydrator = new OrderHydrator(
      new ClassMethods(),
      $this->repository->reveal()
    );
  });

  describe('->hydrate()', function () {
    // ...

    it('should hydrate a Customer entity on the Order', function () {
      $data = [
        'customer_id' => 500
      ];

      $customer = (new Customer())->setId(500);
      $order = new Order();

      $this->repository->getById(500)
        ->shouldBeCalled()
        ->willReturn($customer);

      $this->hydrator->hydrate($data, $order);

      expect($order->getCustomer())->to->equal($customer);

      $this->getProphet()->checkPredictions();
    });
  });
});
```

We've added a dependency to our hydrator for an instance of `CustomerRepositoryInterface`. We'll use this to query for the customer record when we find a `customer_id` value in the data being hydrated. This is now mocked and injected into the constructor.

Our test verifies that this properly occurs by checking the value of `$order->getCustomer()` and making sure that its the same customer that we mocked `CustomerRepository` to return.

Now let’s build this class and make our tests work!

```php
// src/Persistence/Hydrator/OrderHydrator.php

namespace CleanPhp\Invoicer\Persistence\Hydrator;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use Zend\Stdlib\Hydrator\HydratorInterface;

class OrderHydrator implements HydratorInterface {
  protected $wrappedHydrator;
  protected $customerRepository;

  public function __construct(
    HydratorInterface $wrappedHydrator,
    CustomerRepositoryInterface $customerRepository
  )
  {
    $this->wrappedHydrator = $wrappedHydrator;
    $this->customerRepository = $customerRepository;
  }

  public function extract($object) { }

  public function hydrate(array $data, $order) {
    $this->wrappedHydrator->hydrate($data, $order);

    if (isset($data['customer_id'])) {
      $order->setCustomer(
        $this->customerRepository->getById($data['customer_id'])
      );
    }

    return $order;
  }
}
```

This simple functionality works just like we drew it up in our tests. We use the `ClassMethods()` hydrator to do most of the work for us. When a `customer_id` value is present, we query our `CustomerRepository` for that customer and set it on the `Order` object. Our tests should now pass!

This eagerly loading of relationships wouldn’t always be ideal. What if we don’t need the `Customer` for the current use case? One great way to implement some lazy loading of these resources, which would only load the `Customer` if we requested it in client code, would be to use

Marco Pivetta's[44] awesome ProxyManager[45] library, which allows us to use Proxy classes instead of Entities directly, and lazily load related resources. Check out his library if you're interested. Another solution, which we'll explore later, is to just use a better persistence library, such as Doctrine ORM.

For now, however, we can use this new `OrderHydrator` in our `OrderTable` by modifying the service locator definition for `OrderTable` to use `OrderHydrator` instead of class methods:

```php
// config/autoload/global.php

return [
  // ...
  'service_manager' => [
    'factories' => [
      // ...
      'OrderHydrator' => function ($sm) {
        return new OrderHydrator(
          new ClassMethods(),
          $sm->get('CustomerTable')
        );
      },

      // ...

      'OrderTable' => function($sm) {
        $factory = new TableGatewayFactory();
        $hydrator = $sm->get('OrderHydrator');

        return new OrderTable(
          $factory->createGateway(
            $sm->get('Zend\Db\Adapter\Adapter'),
            $hydrator,
            new Order(),
            'orders'
          ),
          $hydrator
        );
      },
    ],
    // ...
  ],
  // ...
];
```

[44]https://github.com/Ocramius

[45]https://github.com/Ocramius/ProxyManager

We declare a new entry in the service locator for `OrderHydrator`, so that we can use it wherever we need it. Our first use of it is in the definition for `OrderTable` in the service locator which now, instead of `ClassMethods` as it was previously using, it now uses `OrderHydrator`.

If we refresh our `/orders` page, we should now see our test Order with the associated Customer rendered to the page, error free.

To recap: when call `OrderTable->getAll()`, we're hydrating all Orders in the database, as well as eagerly loading the associated Customer. When we render these to the page, and call `Order->getCustomer()`, we're using that eagerly loaded Customer object to render the name and ID of the customer to the page.

Our last step is to implement the `extract()` method of our hydrator that we left blank. For this, we're simply going to pass off work to the `ClassMethods->extract()` method as we don't have a specific use case for anything else right now.

```php
// src/Persistence/Hydrator/OrderHydrator.php

namespace CleanPhp\Invoicer\Persistence\Hydrator;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use Zend\Stdlib\Hydrator\HydratorInterface;

class OrderHydrator implements HydratorInterface {
  // ...

  public function extract($object) {
    return $this->wrappedHydrator->extract($object);
  }

  // ...
}
```

If you want to write a spec for this, feel free. It's simple enough that I'll assume the Zend developers have tested it enough for us and that we didn't mess up such a simple call.

Viewing Orders

Let's work on the view page for Orders now. First thing we'll want to do is setup a route. We'll adapt our existing `/orders` route to optionally allow an `/action` and `/:id` to handle all this magic in one.

Another option would be to add an explicit child route for `/view/:id`. Either way will result in a 404 if a user navigates to a sub route that doesn't exist, such as `/orders/steal-all-the-gold`.

```
// module/Application/config/module.config.php

return [
  // ...
  'router' =>
    'routes' => [
      // ...
      'orders' => [
        'type' => 'Segment',
        'options' => [
          'route' => '/orders[/:action[/:id]]',
          'defaults' => [
            'controller' => 'Application\Controller\Orders',
            'action' => 'index',
          ],
        ],
      ],
      // ...
    ],
  ]
];
```

Now, we need a controller action to serve this route in our `OrdersController`:

```
public function viewAction() {
  $id = $this->params()->fromRoute('id');
  $order = $this->orderRepository->getById($id);

  return [
    'order' => $order
  ];
}
```

The `viewAction()` is pretty simple: we grab the ID from the route params, query for that Order from the `$orderRepository` that was injected into the controller, and return it to the view.

Finally, we'll need a view file to display our Order data:

```
<!-- module/Application/views/application/orders/view.phtml -->
<div class="page-header clearfix">
  <h2>Order #<?= $this->escapeHtml($this->order->getOrderNumber()) ?></h2>
</div>

<table class="table table-striped">
  <thead>
    <tr>
      <th colspan="2">Order Details</th>
    </tr>
  </thead>
  <tr>
    <th>Customer:</th>
    <td>
      <a href="/customers/edit/<?=
        $this->escapeHtmlAttr($this->order->getCustomer()->getId()) ?>">
        <?= $this->escapeHtml($this->order->getCustomer()->getName()) ?></a>
    </td>
  </tr>
  <tr>
    <th>Description:</th>
    <td><?= $this->escapeHtml($this->order->getDescription()) ?></td>
  </tr>
  <tr>
    <th>Total:</th>
    <td>$ <?= number_format($this->order->getTotal(), 2) ?></td>
  </tr>
</table>
```

This simple view is just using a table to dump out details about our Order, and provides a link back to the Customer record. It's super simple. But what happens when we navigate to an ID that doesn't exist in the database? We'll get a giant error.

Let's handle this case by throwing a 404:

```
$order = $this->orderRepository->getById($id);

if (!$order) {
  $this->getResponse()->setStatusCode(404);
  return null;
}
```

If we don't get an `Order` object back (and instead get `null`), we simply grab the response stored on the object, set it's status to 404, and return (to halt further processing).

We can check this in the browser by navigation to an Order ID that doesn't exist.

Creating Orders

Creating an Order will be very similar to creating a Customer.

We'll start by writing a spec for our `OrderFilter`, which we'll use to validate our form data:

```php
// specs/input-filter/order.spec.php

use CleanPhp\Invoicer\Service\InputFilter\OrderInputFilter;

describe('InputFilter\Order', function () {
  beforeEach(function () {
    $this->inputFilter = new OrderInputFilter();
  });

  describe('->isValid()', function () {
    // ...
  });
});
```

We're simply setting up an instance of our new `OrderInputFilter` so it's available to specs. We'll have to test each form element within the `->isValid()` block, so let's start by testing the `customer_id`:

```php
it('should require a customer.id', function () {
  $isValid = $this->inputFilter->isValid();

  $error = [
    'id' => [
      'isEmpty' => 'Value is required and can\'t be empty'
    ]
  ];

  $customer = $this->inputFilter
    ->getMessages()['customer'];

  expect($isValid)->to->equal(false);
  expect($customer)->to->equal($error);
});
```

For `customer_id`, we're just interested in making sure it was provided. In the future, we could, and should, also validate that the provided value is actually a customer in the database.

When we build our form, instead of using the database value (`customer_id`), we're going to use our entity relationships, which means that we'll be looking for `customer[id]` via the POST data, so our input filter is going to treat `customer` as an array.

We're testing here using the logic of the `Required` validator in Zend framework. Obviously, if we ever switch our validation library, we'll have to update the specs to the format they provide.

On to `orderNumber`:

```
it('should require an order number', function () {
  $isValid = $this->inputFilter->isValid();

  $error = [
    'isEmpty' => 'Value is required and can\'t be empty'
  ];

  $orderNo = $this->inputFilter
    ->getMessages()['orderNumber'];

  expect($isValid)->to->equal(false);
  expect($orderNo)->to->equal($error);
});
```

Here, we're simply validating that the `orderNumber` value was provided. Again, we're using the language of the domain, `orderNumber`, instead of the database column of `order_number`.

We also want to make sure it falls within our constraints of being exactly 13 characters in length:

```
it('should require order numbers be 13 chars long', function () {
  $scenarios = [
    [
      'value' => '124',
      'errors' => [
        'stringLengthTooShort' =>
          'The input is less than 13 characters long'
      ]
    ],
    [
      'value' => '20001020-0123XR',
      'errors' => [
        'stringLengthTooLong' =>
          'The input is more than 13 characters long'
      ]
    ],
    [
      'value' => '20040717-1841',
      'errors' => null
    ]
  ];

  foreach ($scenarios as $scenario) {
    $this->inputFilter = new OrderInputFilter();
    $this->inputFilter->setData([
      'orderNumber' => $scenario['value']
    ])->isValid();

    $messages = $this->inputFilter
      ->getMessages()['orderNumber'];

    expect($messages)->to->equal($scenario['errors']);
  }
});
```

The `$scenarios` variable lists several different scenarios to test, and which errors we would expect in each scenario.

Next, we'll test the `description`:

```
it('should require a description', function () {
  $isValid = $this->inputFilter->isValid();

  $error = [
    'isEmpty' => 'Value is required and can\'t be empty'
  ];

  $messages = $this->inputFilter
    ->getMessages()['description'];

  expect($isValid)->to->equal(false);
  expect($messages)->to->equal($error);
});
```

Finally, we'll test the total, and ensure that it's a floating point number value:

```
it('should require a total', function () {
  $isValid = $this->inputFilter->isValid();

  $error = [
    'isEmpty' => 'Value is required and can\'t be empty'
  ];

  $messages = $this->inputFilter
    ->getMessages()['total'];

  expect($isValid)->to->equal(false);
  expect($messages)->to->equal($error);
});
```

```
it('should require total to be a float value', function () {
  $scenarios = [
    [
      'value' => 124,
      'errors' => null
    ],
    [
      'value' => 'asdf',
      'errors' => [
          'notFloat'
            => 'The input does not appear to be a float'
      ]
    ],
    [
      'value' => 99.99,
      'errors' => null
    ]
  ];

  foreach ($scenarios as $scenario) {
    $this->inputFilter = new OrderInputFilter();
    $this->inputFilter->setData([
      'total' => $scenario['value']
    ])->isValid();

    $messages = $this->inputFilter
      ->getMessages()['total'];

    expect($messages)->to->equal($scenario['errors']);
  }
});
```

We provide a list of scenarios again, and check each one of them to make sure we get the expected error messages, or no error messages in the case of valid input.

Now that we've defined our spec, let's go ahead and write the Input Filter:

```
// src/Service/InputFilter/OrderInputFilter.php

namespace CleanPhp\Invoicer\Service\InputFilter;

use Zend\I18n\Validator\IsFloat;
use Zend\InputFilter\Input;
use Zend\InputFilter\InputFilter;
use Zend\Validator\StringLength;

class OrderInputFilter extends InputFilter {
  public function __construct() {
    $customer = (new InputFilter());
    $id = (new Input('id'))
      ->setRequired(true);
    $customer->add($id);

    $orderNumber = (new Input('orderNumber'))
      ->setRequired(true);
    $orderNumber->getValidatorChain()->attach(
      new StringLength(['min' => 13, 'max' => 13])
    );

    $description = (new Input('description'))
      ->setRequired(true);

    $total = (new Input('total'))
       ->setRequired(true);
    $total->getValidatorChain()->attach(new IsFloat());

    $this->add($customer, 'customer');
    $this->add($orderNumber);
    $this->add($description);
    $this->add($total);
  }
}
```

These validation rules match what we specified in our tests, which should be passing with this code in place.

The only thing special of note here is that we're nesting an `InputFilter` for `customer[id]` and adding the `$customer` filter as a named `InputFilter` of `customer`, so that Zend understands the nested data we're returning in the POST and validates it properly.

OrdersController::newAction()

Now that we have a `OrderInputFilter`, we can start work on our `newAction()` for the `OrdersController`.

When creating a new order, we'll need to supply a list of Customers to the view to allow the user to select which Customer the Order belongs to. To do so, we'll have an inject an instance of `CustomerRepositoryInterface` into the controller.

```php
// module/Application/src/Application/Controller/OrdersController.php

namespace Application\Controller;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;
use Zend\Mvc\Controller\AbstractActionController;

class OrdersController extends AbstractActionController {
  protected $orderRepository;
  protected $customerRepository;

  public function __construct(
    OrderRepositoryInterface $orderRepository,
    CustomerRepositoryInterface $customerRepository
  ) {
    $this->orderRepository = $orderRepository;
    $this->customerRepository = $customerRepository;
  }

  // ...
}
```

We'll need to update our controller config to pass it an instance of `CustomerRepositoryInterface`:

```php
// module/Application/config/module.config.php

return [
  // ...
  'controllers' => [
    // ...

    'factories' => [
      // ...
      'Application\Controller\Orders' => function ($sm) {
        return new \Application\Controller\OrdersController(
          $sm->getServiceLocator()->get('OrderTable'),
          $sm->getServiceLocator()->get('CustomerTable')
        );
      },
    ],
  ],
  // ...
];
```

Now that we have an instance of `CustomerRepositoryInterface`, we can use it to supply the view with a list of Customers to select and create an Order for:

```
public function newAction() {
  $viewModel = new ViewModel();
  $order = new Order();

  $viewModel->setVariable(
    'customers',
    $this->customerRepository->getAll()
  );
  $viewModel->setVariable('order', $order);

  return $viewModel;
}
```

This should be the minimal code we need for the GET action to work. Don't forget to add some `use` statements for the `Order` and `Zend\View\Model\ViewModel` class.

Let's create the `order.phtml` file:

```
<!-- module/Application/view/application/orders/new.phtml -->
<div class="page-header clearfix">
  <h2>Create Order</h2>
</div>

<form role="form" action="" method="post">
  <div class="form-group">
    <label for="customer_id">Customer:</label>
    <select class="form-control" name="customer[id]" id="customer_id">
      <option value=""></option>
      <?php foreach ($this->customers as $customer): ?>
      <option value="<?= $this->escapeHtmlAttr($customer->getId()) ?>"<?=
        !is_null($this->order->getCustomer()) &&
          $this->order->getCustomer()->getId() == $customer->getId() ?
            ' selected="selected"' : '' ?>>
        <?= $this->escapeHtml($customer->getName()) ?>
      </option>
      <?php endforeach; ?>
    </select>
    <?= $this->validationErrors('customer.id') ?>
  </div>
  <div class="form-group">
    <label for="orderNumber">Order Number:</label>
    <input type="text" class="form-control" name="orderNumber"
      id="order_number" placeholder="Enter Order Number"
      value="<?= $this->escapeHtmlAttr($this->order->getOrderNumber()) ?>">
    <?= $this->validationErrors('orderNumber') ?>
  </div>
  <div class="form-group">
    <label for="description">Description:</label>
    <input type="text" class="form-control" name="description"
      id="description" placeholder="Enter Description"
      value="<?= $this->escapeHtmlAttr($this->order->getDescription()) ?>">
    <?= $this->validationErrors('description') ?>
  </div>
  <div class="form-group">
    <label for="total">Total:</label>
    <input type="text" class="form-control" name="total"
      id="total" placeholder="Enter Total"
      value="<?= $this->escapeHtmlAttr($this->order->getTotal()) ?>">
    <?= $this->validationErrors('total') ?>
  </div>
  <button type="submit" class="btn btn-primary">Save</button>
</form>
```

We use the `$order` object supplied by the `ViewModel` to populate the form. Of course, at this point we don't have any data to populate it with. We use the `$customers` to provide the select options for the Customer drop down.

If we hit up `/orders/new` in our browser, we'll see our hard work. When we click the `Save` button, the page looks the same. It's probably time to handle the POST data.

Handling POST Data

To facilitate saving the posted data, we'll need an instance of our `OrderInputFilter` injected to the controller so we can use it to validate the POST data, and an instance of our `OrderHydrator` to hydrate the data:

```
// module/Application/src/Application/Controller/OrdersController.php

namespace Application\Controller;

use CleanPhp\Invoicer\Domain\Entity\Order;
use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;
use CleanPhp\Invoicer\Persistence\Hydrator\OrderHydrator;
use CleanPhp\Invoicer\Service\InputFilter\OrderInputFilter;
use Zend\Mvc\Controller\AbstractActionController;
use Zend\View\Model\ViewModel;

class OrdersController extends AbstractActionController {
  protected $orderRepository;
  protected $customerRepository;
  protected $inputFilter;
  protected $hydrator;

  public function __construct(
    OrderRepositoryInterface $orderRepository,
    CustomerRepositoryInterface $customerRepository,
    OrderInputFilter $inputFilter,
    OrderHydrator $hydrator
  ) {
    $this->orderRepository = $orderRepository;
    $this->customerRepository = $customerRepository;
    $this->inputFilter = $inputFilter;
    $this->hydrator = $hydrator;
  }

  // ...
}
```

If this seems like a lot to inject into the controller, it just might be. A solution might be to break this controller up into multiple controllers with smaller responsibilities/less actions. Or you could investigate a different pattern, like Paul Jones' Action-Domain-Responder[46]. For now, I'll stick with this one controller.

Next, let's update our `controllers` config to inject in an `OrderInputFilter`:

```
// module/Application/config/module.config.php

use CleanPhp\Invoicer\Service\InputFilter\OrderInputFilter;

return [
  // ...
  'controllers' => [
    'invokables' => [
      'Application\Controller\Index' =>
        'Application\Controller\IndexController'
    ],
    'factories' => [
      // ...
      'Application\Controller\Orders' => function ($sm) {
```

[46]https://github.com/pmjones/adr

```
        return new \Application\Controller\OrdersController(
          $sm->getServiceLocator()->get('OrderTable'),
          $sm->getServiceLocator()->get('CustomerTable'),
          new OrderInputFilter(),
          $sm->getServiceLocator()->get('OrderHydrator')
        );
      },
    ],
  ],
  // ...
];
```

Now our `OrdersController` should have everything it needs to operate. Let's update the `newAction()` to utilize these components to posted data:

```
public function newAction() {
  $viewModel = new ViewModel();
  $order = new Order();

  if ($this->getRequest()->isPost()) {
      $this->inputFilter
        ->setData($this->params()->fromPost());

      if ($this->inputFilter->isValid()) {
        $order = $this->hydrator->hydrate(
          $this->inputFilter->getValues(),
          $order
        );

        $this->orderRepository->begin()
            ->persist($order)
            ->commit();

        $this->flashMessenger()->addSuccessMessage('Order Created');
        $this->redirect()->toUrl('/orders/view/' . $order->getId());
      } else {
        $this->hydrator->hydrate(
          $this->params()->fromPost(),
          $order
        );
        $viewModel->setVariable(
          'errors',
          $this->inputFilter->getMessages()
        );
      }
  }
  $viewModel->setVariable(
    'customers',
    $this->customerRepository->getAll()
  );
  $viewModel->setVariable('order', $order);

  return $viewModel;
}
```

This is very similar to how the `newOrEditAction()` worked in our `CustomersController`.

We check to see if the request was a POST. If it was, we load up our `OrderInputFilter` with the data from the post, then check to see if that data is valid. If it is, we hydrate an `Order` object with the filtered values, persist them to the `OrderRepository`, then store a flash message and redirect to the View Order page.

If the data is not valid, we again hydrate a customer object, but this time with the raw POST data, and store the validation errors on the view model to render in the view.

Next, we need to update our `OrderHydrator` to be able to handle `customer[id]` being sent in via POST data. When it encounters this data, we'll want to instantiate a new `Customer` object and set the ID.

Let's update our spec to handle this case:

```
// specs/hydrator/order-hydrator.spec.php

// ...

describe('Persistence\Hydrator\OrderHydrator', function () {
  // ...

  describe('->hydrate()', function () {
    // ...

    it('should hydrate the embedded customer data', function () {
      $data = ['customer' => ['id' => 20]];
      $order = new Order();

      $this->hydrator->hydrate($data, $order);

      assert(
        $data['customer']['id'] === $order->getCustomer()->getId(),
        'id does not match'
      );
    });
  });
});
```

Let’s update our `OrderHydrator` to meet this spec:

```
// src/Persistence/Hydrator/OrderHydrator.php

namespace CleanPhp\Invoicer\Persistence\Hydrator;

use CleanPhp\Invoicer\Domain\Entity\Customer;
use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use Zend\Stdlib\Hydrator\HydratorInterface;

class OrderHydrator implements HydratorInterface {
  protected $wrappedHydrator;
  protected $customerRepository;

  public function __construct(
    HydratorInterface $wrappedHydrator,
    CustomerRepositoryInterface $customerRepository
  ) {
    $this->wrappedHydrator = $wrappedHydrator;
    $this->customerRepository = $customerRepository;
  }

  public function extract($order) {
    return $this->wrappedHydrator->extract($order);
  }

  public function hydrate(array $data, $order) {
    $customer = null;

    if (isset($data['customer'])) {
      $customer = $this->wrappedHydrator->hydrate(
        $data['customer'],
        new Customer()
      );
      unset($data['customer']);
    }

    if (isset($data['customer_id'])) {
      $customer = $this->customerRepository->getById($data['customer_id']);
    }

    $this->wrappedHydrator->hydrate($data, $order);

    if ($customer) {
      $order->setCustomer($customer);
    }
    return $order;
  }
}
```

We'll also need to convert the embedded `Customer` to a `customer_id` when extracting so that it properly saves to the database, so let's add a spec for that, too:

```
// specs/hydrator/order-hydrator.spec.php

// ...

describe('Persistence\Hydrator\OrderHydrator', function () {
  describe('->extract()', function () {
    it('should extract the customer object', function () {
        $order = new Order();
        $order->setCustomer((new Customer())->setId(14));

        $data = $this->hydrator->extract($order);

        assert(
          $order->getCustomer()->getId() === $data['customer_id'],
          'customer_id is not correct'
        );
    });
  });
]);
```

And we'll make the corresponding changes to the `OrderHydrator`:

```
public function extract($object) {
  $data = $this->wrappedHydrator->extract($object);

  if (array_key_exists('customer', $data) &&
    !empty($data['customer'])) {

    $data['customer_id'] = $data['customer']->getId();
    unset($data['customer']);
  }

  return $data;
}
```

If, when we extract the data using the `ClassMethods` hydrator, we find a `customer` key, we'll extract the ID of the Customer object and save it as `customer_id` in the returned data.

If we test out the Order creation now, we should find that *most* everything works. Except we never get a validation message when we don't select a Customer.

Since we have a nested InputFilter situation going on with `customers[id]`, our simple check for errors on a single element won't work.

We're going to use a nice little library called Keyper[47] that will allow us to grab a value nested within an array:

```
composer require vnn/keyper
```

Now let's modify the `ValidationErrors` ViewHelper:

```
// module/Application/src/Application/View/Helper/ValidationErrors.php

protected function getErrors($element) {
  if (!isset($this->getView()->errors)) {
    return false;
  }

  $errors = Keyper::create($this->getView()->errors);

  return $errors->get($element) ?: false;
}
```

Keyper will take care of getting our nested value `customer.id` from the multidimensional array for us. If you submit an empty form again, you should now get a validation error message.

And with this, order management is now complete!

git

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 08-managing-orders:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 08-managing-orders
```

Invoice Management

Our last module of our sample project is Invoice management. We're going to setup an index page for listing Invoices, just like the ones we setup for Customers and Orders. Let's start with our `InvoiceController`, which will initially look just like our other two controllers:

[47] https://github.com/varsitynewsnetwork/keyper

```php
// module/Application/src/Application/Controller/InvoicesController.php

namespace Application\Controller;

use CleanPhp\Invoicer\Domain\Repository\InvoiceRepositoryInterface;
use Zend\Mvc\Controller\AbstractActionController;

class InvoicesController extends AbstractActionController {
  protected $invoiceRepository;

  public function __construct(InvoiceRepositoryInterface $invoices) {
    $this->invoiceRepository = $invoices;
  }

  public function indexAction() {
    $invoices = $this->invoiceRepository->getAll();

    return [
      'invoices' => $invoices
    ];
  }
}
```

Next, we'll need to register the `InvoiceController` with the service locator:

```php
// module/Application/config/module.config.php

return [
  // ..
  'controllers' => [
    // ...
    'factories' => [
      // ...
      'Application\Controller\Invoices' => function ($sm) {
        return new \Application\Controller\InvoicesController   (
          $sm->getServiceLocator()->get('InvoiceTable')
        );
      }
    ],
  ],
];
```

Pretty standard stuff. We'll also need to define the `InvoiceTable` now for our view file:

Let's complete the `indexAction()` by providing the view file:

```
<div class="page-header clearfix">
  <h2 class="pull-left">Invoices</h2>
  <a href="/invoices/generate"
    class="btn btn-success pull-right">
    Generate Invoices</a>
</div>

<table class="table table-striped clearfix">
  <thead>
    <tr>
      <th>#</th>
      <th>Order Number</th>
      <th>Invoice Date</th>
      <th>Customer</th>
      <th>Description</th>
      <th class="text-right">Total</th>
    </tr>
  </thead>
  <?php foreach ($this->invoices as $invoice): ?>
    <tr>
      <td>
        <a href="/invoices/view/<?=
          $this->escapeHtmlAttr($invoice->getId()) ?>">
          <?= $this->escapeHtml($invoice->getId()) ?></a>
      </td>
      <td>
        <?= $invoice->getInvoiceDate()->format('m/d/Y') ?>
      </td>
      <td>
        <?= $this->escapeHtml($invoice->getOrder()->getOrderNumber()) ?>
      </td>
      <td>
        <a href="/customers/edit/<?=
          $this->escapeHtmlAttr(
            $invoice->getOrder()->getCustomer()->getId()
          ) ?>">
          <?= $this->escapeHtml(
            $invoice->getOrder()->getCustomer()->getName()
          ) ?></a>
      </td>
      <td>
        <?= $this->escapeHtml($invoice->getOrder()->getDescription()) ?>
      </td>
      <td class="text-right">
        $ <?= number_format($invoice->getTotal(), 2) ?>
      </td>
    </tr>
  <?php endforeach; ?>
</table>
```

If we visit `/invoices` in our browser, we should see our new, lovely, empty grid. If we manually create some invoices in the database and refresh again, we get a giant error. Zend returns a simple string representation of the `invoice_date`, instead of a hydrated `DateTime` object, so we now need to create an `InvoiceHydrator`

Invoice Hydration

We have two special needs for invoice hydration that fall out of the realm of Zend's standard `ClassMethods` hydrator:

1. We need to be able to hydrate a `DateTime` object, as we setup our `Invoice::$invoiceDate` attribute to be an instance of `DateTime`. Unfortunately, Zend doesn't provide any magical way for us to hydrate to these built-in objects.
2. We need to be able to hydrate the associated `Invoice->$order` Order relationship, so that we can do cool things on the View Invoice page, like displaying which Order the Invoice is for.

To handle the `DateTime` issue, we're going to write a hydration strategy[48] class to deal with it. Zend's Hydrators have a concept of strategies built in. These strategies can be added to instances of hydrators to tell the hydrator how to handle a specific properties.

Going this route also affords us the luxury of reusing this for any other `DateTime` properties we need to handle.

We'll extend the `Zend\Stdlib\Hydrator\Strategy\DefaultStrategy` class, which provides extensible `extract()` and `hydrate()` methods.

Let's first define a spec for this strategy functionality:

```
// specs/hydrator/strategy/date.spec.php

use CleanPhp\Invoicer\Persistence\Hydrator\Strategy\DateStrategy;

describe('Peristence\Hydrator\Strategy\DateStrategy', function () {
  beforeEach(function () {
    $this->strategy = new DateStrategy();
  });

  describe('->hydrate()', function () {
    it('should turn the string date into a DateTime object', function () {
      $value = '2014-12-26';
```

[48]http://framework.zend.com/manual/current/en/modules/zend.stdlib.hydrator.strategy.html

```
      $obj = $this->strategy->hydrate($value);

      assert($obj->format('Y-m-d') === $value, 'incorrect datetime');
    });
  });

  describe('->extract()', function () {
    it('should turn the DateTime object into a string', function () {
      $value = new DateTime('2014-12-28');
      $string = $this->strategy->extract($value);

      assert($string === $value->format('Y-m-d'));
    });
  });
});
```

We're expecting that our `hydrate()` method will accept a string date/time representation and turn it into a proper `DateTime` object, and expecting that `extract()` will do the opposite, and turn a `DateTime` object into a string date/time representation.

Now let's write the actual class:

```
// src/Persistence/Hydrator/Strategy/DateStrategy.php

namespace CleanPhp\Invoicer\Persistence\Hydrator\Strategy;

use DateTime;
use Zend\Stdlib\Hydrator\Strategy\DefaultStrategy;

class DateStrategy extends DefaultStrategy {
  public function hydrate($value) {
    if (is_string($value)) {
      $value = new DateTime($value);
    }

    return $value;
  }

  public function extract($value) {
    if ($value instanceof DateTime) {
      $value = $value->format('Y-m-d');
    }

    return $value;
  }
}
```

And so our spec requires, so our code does. This is pretty simple.

Now let's spec out our actual `InvoiceHydrator` and figure out how it should work. We'll start with the `extract()` method:

```php
// specs/hydrator/invoice.spec.php

use CleanPhp\Invoicer\Domain\Entity\Invoice;

describe('Persistence\Hydrator\InvoiceHydrator', function () {
  describe('->extract()', function () {
    it('should perform simple extraction on the object', function () {
      $invoice = new Invoice();
      $invoice->setTotal(300.14);

      $data = $this->hydrator->extract($invoice);

      expect($data['total'])->to->equal($invoice->getTotal());
    });

    it('should extract a DateTime object to a string', function () {
      $invoiceDate = new \DateTime();
      $invoice = new Invoice();
      $invoice->setInvoiceDate($invoiceDate);

      $data = $this->hydrator->extract($invoice);

      expect($data['invoice_date'])
        ->to->equal($invoice->getInvoiceDate()->format('Y-m-d'));
    });
  });

  describe('->hydrate()', function () {
    it('should perform simple hydration on the object', function () {
      $data = ['total' => 300.14];
      $invoice = $this->hydrator->hydrate($data, new Invoice());

      expect($invoice->getTotal())->to->equal($data['total']);
    });

    it('should hydrate a DateTime object', function () {
      $data = ['invoice_date' => '2014-12-13'];
      $invoice = $this->hydrator->hydrate($data, new Invoice());

      expect($invoice->getInvoiceDate()->format('Y-m-d'))
        ->to->equal($data['invoice_date']);
    });
  });
});
```

We're doing four tests that the hydrator should do:

1. simple extraction on all properties
2. DateTime extraction on the `invoice_date` property
3. simple hydration on all attributes
4. DateTime hydration on the `invoice_date` attribute

To do this, we'll rely on wrapping an instance of Zend's `ClassMethods` hydrator, just like we did with the `OrderHydrator`, so let's setup a `beforeEach()` condition to setup an instance of the hydrator for us:

```
// specs/hydrator/invoice.spec.php

use CleanPhp\Invoicer\Domain\Entity\Invoice;
use CleanPhp\Invoicer\Persistence\Hydrator\InvoiceHydrator;
use Zend\Stdlib\Hydrator\ClassMethods;

describe('Persistence\Hydrator\InvoiceHydrator', function () {
  beforeEach(function () {
    $this->hydrator = new InvoiceHydrator(new ClassMethods());
  });

  // ...
});
```

This satisfies the specs requirement to have an instance variable of `$hydrator` and injects our `InvoiceHydrator` with an instance of `ClassMethods`. Let's give it a try and see if our spec passes:

```
// src/Persistence/Hydrator/InvoiceHydrator.php

namespace CleanPhp\Invoicer\Persistence\Hydrator;

use CleanPhp\Invoicer\Persistence\Hydrator\Strategy\DateStrategy;
use Zend\Stdlib\Hydrator\ClassMethods;
use Zend\Stdlib\Hydrator\HydratorInterface;

class InvoiceHydrator implements HydratorInterface {
  protected $wrappedHydrator;

  public function __construct(ClassMethods $wrappedHydrator) {
    $this->wrappedHydrator = $wrappedHydrator;
    $this->wrappedHydrator->addStrategy(
      'invoice_date',
      new DateStrategy()
    );
  }

  public function extract($object) {
    return $this->wrappedHydrator->extract($object);
  }

  public function hydrate(array $data, $object) {
    return $this->wrappedHydrator->hydrate($data, $object);
  }
}
```

First, in the constructor, we attach the `DateStrategy` to our `$wrappedHydrator` for the `invoice_-date` column, so that the hydrator will use that strategy when it encounters the `invoice_date` property.

For both `extract()` and `hydrate()`, we're simply passing off the work to Zend's `ClassMethods` hydrator and returning the result, knowing that it will use our `DateStrategy` when appropriate.

Now, let's handle `Order` hydration. Let's start by writing specs:

```
// specs/hydrator/invoice.spec.php

// ...
use CleanPhp\Invoicer\Domain\Entity\Order;
// ...

describe('Persistence\Hydrator\InvoiceHydrator', function () {
  beforeEach(function () {
    $this->repository = $this->getProphet()
      ->prophesize(
        'CleanPhp\Invoicer\Domain\Repository\\' .
        'OrderRepositoryInterface'
      );
    $this->hydrator = new InvoiceHydrator(
      new ClassMethods(),
      $this->repository->reveal()
    );
  });

  describe('->extract()', function () {
    // ...

    it('should extract the order object', function () {
      $invoice = new Invoice();
      $invoice->setOrder((new Order())->setId(14));

      $data = $this->hydrator->extract($invoice);

      expect($data['order_id'])
        ->to->equal($invoice->getOrder()->getId());
    });
  });


  describe('->hydrate()', function () {
    // ...

    it('should hydrate an Order entity on the Invoice', function () {
      $data = ['order_id' => 500];

      $order = (new Order())->setId(500);
      $invoice = new Invoice();

      $this->repository->getById(500)
        ->shouldBeCalled()
        ->willReturn($order);

      $this->hydrator->hydrate($data, $invoice);

      expect($invoice->getOrder())->to->equal($order);

      $this->getProphet()->checkPredictions();
    });

    it('should hydrate the embedded order data', function () {
      $data = ['order' => ['id' => 20]];
      $invoice = new Invoice();

      $this->hydrator->hydrate($data, $invoice);

      expect($invoice->getOrder()->getId())->to->equal($data['order']['id']);
    });
  });
});
```

The first thing we're doing is injecting an instance of `OrderRepositoryInterface` into the `InvoiceHydrator` so that it can query for the necessary Order when hydrating. Next, we add a couple scenarios to `extract()` and `hydrate()`.

For `extract()`, we want to make sure that, if there's an `Order` on the `Invoice`, our extracted data should contain a key for `order_id` with the value of the `Order` object's `$id`.

For `hydrate()`, we test two things:

1. If there is an `order_id`, we should query the database for that `Order` and assign it to the `Invoice`
2. If there is a nested `order[id]`, we should hydrate an `Order` object with that data and assign it to the `Invoice`.

Let's update our hydrator:

```php
// src/Persistence/Hydrator/InvoiceHydrator.php

namespace CleanPhp\Invoicer\Persistence\Hydrator;

use CleanPhp\Invoicer\Domain\Entity\Order;
use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;
use CleanPhp\Invoicer\Persistence\Hydrator\Strategy\DateStrategy;
use Zend\Stdlib\Hydrator\HydratorInterface;

class InvoiceHydrator implements HydratorInterface {
  protected $wrappedHydrator;
  private $orderRepository;

  public function __construct(
    HydratorInterface $wrappedHydrator,
    OrderRepositoryInterface $orderRepository
  ) {
    $this->wrappedHydrator = $wrappedHydrator;
    $this->wrappedHydrator->addStrategy(
      'invoice_date',
      new DateStrategy()
    );
    $this->orderRepository = $orderRepository;
  }

  public function extract($object) {
    $data = $this->wrappedHydrator->extract($object);

    if (array_key_exists('order', $data) &&
      !empty($data['order'])) {

      $data['order_id'] = $data['order']->getId();
      unset($data['order']);
    }
    return $data;
  }

  public function hydrate(array $data, $invoice) {
    $order = null;

    if (isset($data['order'])) {
      $order = $this->wrappedHydrator->hydrate(
        $data['order'],
        new Order()
      );
      unset($data['order']);
    }

    if (isset($data['order_id'])) {
      $order = $this->orderRepository->getById($data['order_id']);
    }

    $invoice = $this->wrappedHydrator->hydrate($data, $invoice);

    if ($order) {
      $invoice->setOrder($order);
    }

    return $invoice;
  }
}
```

Last, we need to update the definition of our `InvoiceTable` in the service manager to use this new `InvoiceHydrator`:

```
// config/autoload/global.php

return [
  // ...
  'InvoiceHydrator' => function ($sm) {
    return new InvoiceHydrator(
      new ClassMethods(),
      $sm->get('OrderTable')
    );
  },
  // ...
  'InvoiceTable' =>  function($sm) {
    $factory = new TableGatewayFactory();
    $hydrator = $sm->get('InvoiceHydrator');
    return new InvoiceTable(
      $factory->createGateway(
         $sm->get('Zend\Db\Adapter\Adapter'),
         $hydrator,
         new Invoice(),
         'invoices'
      ),
      $hydrator
    );
  },
  // ...
];
```

So we define a new entry of `InvoiceHydrator` to get our new hydrator, and then update the entry for `InvoiceTable` to use this new hydrator.

If we refresh our Invoice index page, we should now see data in the grid (assuming you manually entered some into sqlite). All of our specs should be passing, too.

We did it!

Generating Invoices

Creating Invoices is going to work a bit different than creating Customers and creating Orders. Instead of providing the user with a form to enter Invoice data, we're going to look for uninvoiced Orders, using the `OrderRepository::getUninvoicedOrders()` method.

For our UI, when clicking on the *Generate Invoices* button, we're going to display a page showing all Orders available for invoicing. At the bottom, we'll include another *Generate Invoices* button which will take us to another action to actually generate those invoices.

Finally, when we're done generating the invoices, we'll drop the user on view that shows them the invoices were generated.

The first thing we'll want to do is give our `invoices` route some more liberty to serve up any action we drop in the controller, just like we did for `orders`:

```
// module/Application/config/module.config.php

return [
  // ...
  'router' => [
    'routes' => [
      // ...
      'invoices' => [
        'type' => 'Segment',
        'options' => [
          'route' => '/invoices[/:action[/:id]]',
          'defaults' => [
            'controller' => 'Application\Controller\Invoices',
            'action' => 'index',
          ],
        ],
      ],
    ],
  ],
  // ...
];
```

In order for our `InvoicesController` to get the list of uninvoiced Orders, it will need an instance of the `OrderRepositoryInterface`, so let's update the controller to specify that:

```
// module/Application/src/Application/Controller/InvoicesController.php

namespace Application\Controller;

use CleanPhp\Invoicer\Domain\Repository\InvoiceRepositoryInterface;
use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;
use Zend\Mvc\Controller\AbstractActionController;

class InvoicesController extends AbstractActionController {
  protected $invoiceRepository;
  protected $orderRepository;

  public function __construct(
    InvoiceRepositoryInterface $invoices,
    OrderRepositoryInterface $orders
  ) {
    $this->invoiceRepository = $invoices;
    $this->orderRepository = $orders;
  }

  // ...
}
```

Of course, to do this, we'll need to update the controller configuration to pass in an instance of the interface:

```
// ...

return [
  // ...
  'controllers' => [
    // ...
    'factories' => [
      // ...
      'Application\Controller\Invoices' => function ($sm) {
        return new \Application\Controller\InvoicesController(
          $sm->getServiceLocator()->get('InvoiceTable'),
          $sm->getServiceLocator()->get('OrderTable')
        );
      },
      // ...
    ],
  ],
  // ...
];
```

Let’s now work on our initial action that will present the list of uninvoiced orders to the user:

```
public function generateAction() {
  return [
    'orders' => $this->orderRepository->getUninvoicedOrders()
  ];
}
```

This simple action simply returns the uninvoiced orders. Nothing more; nothing less.

Let’s build our view:

```
<!-- view/application/invoices/generate.phtml -->

<h2>Generate New Invoices</h2>

<p>
  The following orders are available to be invoiced.
</p>

<?php if (empty($this->orders)): ?>
<p class="alert alert-info">
  There are no orders available for invoice.
</p>
<?php else: ?>
<table class="table table-striped clearfix">
```

```
  <thead>
    <tr>
      <th>#</th>
      <th>Order Number</th>
      <th>Customer</th>
      <th>Description</th>
      <th class="text-right">Total</th>
    </tr>
  </thead>
  <?php foreach ($this->orders as $order): ?>
    <tr>
      <td>
        <a href="/orders/view/<?= $this->escapeHtmlAttr($order->getId()) ?>">
          <?= $this->escapeHtml($order->getId()) ?></a>
      </td>
      <td><?= $this->escapeHtml($order->getOrderNumber()) ?></td>
      <td>
        <a href="/customers/edit/<?=
          $this->escapeHtmlAttr($order->getCustomer()->getId()) ?>">
          <?= $this->escapeHtml($order->getCustomer()->getName()) ?></a>
      </td>
      <td><?= $this->escapeHtml($order->getDescription()) ?></td>
      <td class="text-right">
        $ <?= number_format($order->getTotal(), 2) ?>
      </td>
    </tr>
  <?php endforeach; ?>
</table>

<form action="/invoices/generate-process" method="post" class="text-center">
    <button type="submit" class="btn btn-primary">Generate Invoices</button>
</form>
<?php endif; ?>
```

This pretty simple view first checks to see if we have any orders, and displays a helpful message if we don't. If we do have orders, we loop them and display them in a table, much like we do in our index views.

Our table of orders is identical to the table in the orders index view. A better solution would be to store this code in a separate view file and load it using Zend's partial view helper[49]. Give it a shot.

Next, we'll want to implement the `getUninvoicedOrders()` method of the `OrderRepository`:

[49]http://framework.zend.com/manual/current/en/modules/zend.view.helpers.partial.html

```php
// src/Persistence/Zend/DataTable/OrderTable.php

namespace CleanPhp\Invoicer\Persistence\Zend\DataTable;

use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;

class OrderTable extends AbstractDataTable
  implements OrderRepositoryInterface
{
  public function getUninvoicedOrders() {
    return $this->gateway->select(
      'id NOT IN(SELECT order_id FROM invoices)'
    );
  }
}
```

This method simply returns all orders where the ID is not in the `invoices` table.

If we visit `/invoices/generate` in our browser, we should see our view listing all the orders available for invoicing.

Next, we'll see about implementing our second *Generate Invoices* button. We'll first need to update the `InvoiceController` to accept an instance of the `InvoicingService` we wrote a couple chapters ago:

```php
namespace Application\Controller;

use CleanPhp\Invoicer\Domain\Repository\InvoiceRepositoryInterface;
use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;
use CleanPhp\Invoicer\Domain\Service\InvoicingService;
use Zend\Mvc\Controller\AbstractActionController;

class InvoicesController extends AbstractActionController {
  protected $invoiceRepository;
  protected $orderRepository;
  protected $invoicing;

  public function __construct(
    InvoiceRepositoryInterface $invoices,
    OrderRepositoryInterface $orders,
    InvoicingService $invoicing
  ) {
    $this->invoiceRepository = $invoices;
    $this->orderRepository = $orders;
    $this->invoicing = $invoicing;
  }
  // ...
}
```

Of course, we need to modify our controller config for this to work:

```
return [
  // ...
  'controllers' => [
    // ...
    'factories' => [
      // ...
      'Application\Controller\Invoices' => function ($sm) {
        return new \Application\Controller\InvoicesController(
          $sm->getServiceLocator()->get('InvoiceTable'),
          $sm->getServiceLocator()->get('OrderTable'),
          new InvoicingService(
            $sm->getServiceLocator()->get('OrderTable'),
            new InvoiceFactory()
          )
        );
      },
      // ...
    ],
  ],
  // ...
];
```

We'll use this `InvoicingService` in our `generateAction()`:

```
public function generateProcessAction() {
  $invoices = $this->invoicing->generateInvoices();

  $this->invoiceRepository->begin();

  foreach ($invoices as $invoice) {
    $this->invoiceRepository->persist($invoice);
  }

  $this->invoiceRepository->commit();

  return [
    'invoices' => $invoices
  ];
}
```

Here, we loop through the invoices generated by the `InvoicingService` and persist them to the `InvoiceRepository`. Finally, we'll return the list of generated invoices to the view.

Speaking of the view, let's create that:

```
<div class="page-header">
  <h2>Generated Invoices</h2>
</div>

<?php if (empty($this->invoices)): ?>
<p class="text-center">
  <em>No invoices were generated.</em>
</p>
<?php else: ?>
<table class="table table-striped clearfix">
  <thead>
    <tr>
      <th>#</th>
      <th>Order Number</th>
      <th>Invoice Date</th>
      <th>Customer</th>
      <th>Description</th>
      <th class="text-right">Total</th>
    </tr>
  </thead>
  <?php foreach ($this->invoices as $invoice): ?>
    <tr>
      <td>
        <a href="/invoices/view/<?=
          $this->escapeHtmlAttr($invoice->getId()) ?>">
          <?= $this->escapeHtml($invoice->getId()) ?></a>
      </td>
      <td>
        <?= $invoice->getInvoiceDate()->format('m/d/Y') ?>
      </td>
      <td>
        <?= $this->escapeHtml($invoice->getOrder()->getOrderNumber()) ?>
      </td>
      <td>
        <a href="/customers/edit/<?=
          $this->escapeHtmlAttr(
            $invoice->getOrder()->getCustomer()->getId()
          ) ?>">
          <?= $this->escapeHtml(
            $invoice->getOrder()->getCustomer()->getName()
          ) ?></a>
      </td>
      <td>
        <?= $this->escapeHtml($invoice->getOrder()->getDescription()) ?>
      </td>
      <td class="text-right">
        $ <?= number_format($invoice->getTotal(), 2) ?>
      </td>
    </tr>
  <?php endforeach; ?>
</table>
<?php endif; ?>
```

This view will show the user all the invoices generated on an invoicing run. It looks just like our Invoice index view.

Our table of invoices is identical to the table in the invoice index view. A better solution would be to store this code in a separate view file and load it using Zend's partial view helper[50]. Give it a shot.

The last thing we have to do is setup our View Invoice action.

Viewing Invoices

Let's create a `viewAction()`. We'll steal from the `OrderController` and modify it:

```
// module/Application/src/Application/Controller/InvoicesController.php

public function viewAction() {
  $id = $this->params()->fromRoute('id');
  $invoice = $this->invoiceRepository->getById($id);

  if (!$invoice) {
    $this->getResponse()->setStatusCode(404);
    return null;
  }

  return [
    'invoice' => $invoice,
    'order' => $invoice->getOrder()
  ];
}
```

And a simple view:

[50] http://framework.zend.com/manual/current/en/modules/zend.view.helpers.partial.html

```
<!-- module/Application/view/application/invoices/view.phtml -->

<div class="page-header clearfix">
  <h2>Invoice #<?= $this->escapeHtml($this->invoice->getId()) ?></h2>
</div>

<table class="table table-striped">
  <thead>
    <tr>
      <th colspan="2">Invoice Details</th>
    </tr>
  </thead>
  <tr>
    <th>Customer:</th>
    <td>
      <a href="/customers/edit/<?=
        $this->escapeHtmlAttr($this->order->getCustomer()->getId()) ?>">
        <?= $this->escapeHtml($this->order->getCustomer()->getName()) ?></a>
    </td>
  </tr>
  <tr>
    <th>Order:</th>
    <td>
        <a href="/orders/view/<?=
        $this->escapeHtmlAttr($this->order->getId()) ?>">
            <?= $this->escapeHtml($this->order->getOrderNumber()) ?></a>
    </td>
  </tr>
  <tr>
    <th>Description:</th>
    <td><?= $this->escapeHtml($this->order->getDescription()) ?></td>
  </tr>
  <tr>
    <th>Total:</th>
    <td>$ <?= number_format($this->invoice->getTotal(), 2) ?></td>
  </tr>
</table>
```

And with this, we've completed the sample application.

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 09-invoicing:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 09-invoicing
```

Doctrine 2

We have a working, functional app written in Zend Framework 2 that satisfies our business requirements. Let's say for some reason that we wanted to try a different approach to interacting with the database. Maybe we're about to scale up and add many more features and have decided that we need an easier solution for interacting with the database.

Whatever the reason is, let's say that we have done extensive research and have settled on using Doctrine ORM[51].

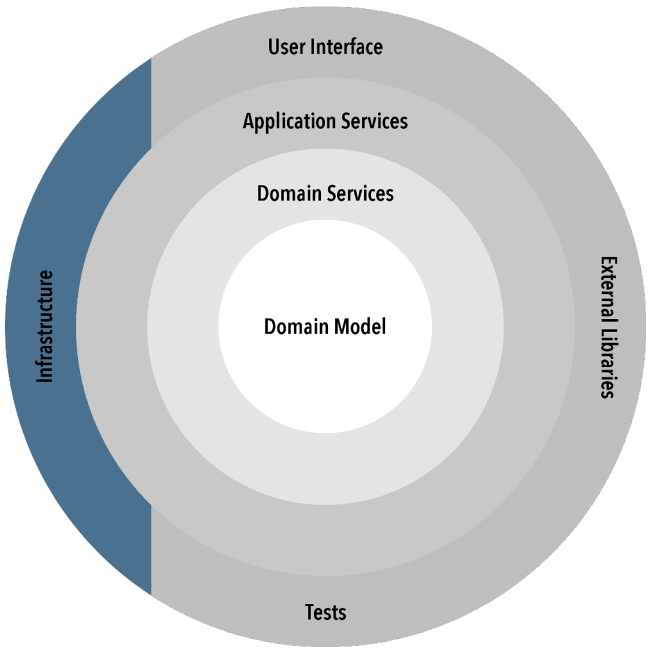
Doctrine is a Data Mapper[52] based Object Relational Mapping (ORM)[53] library. Essentially, Doctrine allows us to map database tables and columns to PHP objects and attributes, and manipulate data as if we were simply manipulating objects.

The Doctrine ORM project is built on top of the Doctrine DBAL[54] (Database Abstraction Layer) project, and utilizes the Doctrine Common[55] project as well. We'll get these tools simply by requiring the `doctrine/orm` library via Composer. We'll also need `symfony/yaml` as we're going to be writing some mapping files in YAML:

```
composer require doctrine/orm symfony/yaml
```

[51]http://www.doctrine-project.org/projects/orm.html
[52]http://martinfowler.com/eaaCatalog/dataMapper.html
[53]http://en.wikipedia.org/wiki/Object-relational_mapping
[54]http://www.doctrine-project.org/projects/dbal.html
[55]http://www.doctrine-project.org/projects/common.html

Rebuilding the Persistence Layer



Our goal is to swap out the persistence library, which is part of our infrastructure layer. Remember that this layer of the onion diagram has nothing dependent upon it, meaning that we should be able to entirely swap out this layer without having to touch any other layers of the application.

Okay, I lied; there is one piece of the ZF2 app we'll need to tweak: the configuration files. These files are the social lubricant that gets the two layers talking. Without it, ZF2 simply wouldn't be able to use Doctrine. These two layers are pretty unique; every other layer just relies on interfaces and dependency injection to make them work together. But config is necessary whenever a database or third party service is involved.

So let's see how well we did earlier when we set up these layers.

Creating Doctrine-based Repositories

We'll need to implement the repository interfaces in our domain services layer using Doctrine. These concrete repositories will sit on top of and utilize Doctrine's `EntityManager` object, which, along with the `UnitofWork` object, are the workhorses of Doctrine. In many cases, these repositories methods will simply be a proxy to the `EntityManager` itself.

Let's start by creating an `AbstractDoctrineRepository` that will encapsulate the vast majority of the functionality that each individual repository can inherit from, much like we did with the Zend Data Tables:

```php
// src/Persistence/Doctrine/Repository/AbstractDoctrineRepository.php

namespace CleanPhp\Invoicer\Persistence\Doctrine\Repository;

use CleanPhp\Invoicer\Domain\Entity\AbstractEntity;
use CleanPhp\Invoicer\Domain\Repository\RepositoryInterface;
use Doctrine\ORM\EntityManager;

abstract class AbstractDoctrineRepository implements RepositoryInterface {
  protected $entityManager;
  protected $entityClass;

  public function __construct(EntityManager $em) {
    if (empty($this->entityClass)) {
      throw new \RuntimeException(
        get_class($this) . '::$entityClass is not defined'
      );
    }

    $this->entityManager = $em;
  }

  public function getById($id) {
    return $this->entityManager->find($this->entityClass, $id);
  }

  public function getAll() {
    return $this->entityManager->getRepository($this->entityClass)
      ->findAll();
  }

  public function getBy(
    $conditions = [],
    $order = [],
    $limit = null,
    $offset = null
  ) {
    $repository = $this->entityManager->getRepository(
      $this->entityClass
    );

    $results = $repository->findBy(
      $conditions,
      $order,
      $limit,
      $offset
    );

    return $results;
  }

  public function persist(AbstractEntity $entity) {
    $this->entityManager->persist($entity);
    return $this;
  }

  public function begin() {
    $this->entityManager->beginTransaction();
    return $this;
  }

  public function commit() {
    $this->entityManager->flush();
    $this->entityManager->commit();
    return $this;
  }
}
```

This class has a member named `$entityClass`, which must be supplied with the fully qualified name of the Entity class that the repository will be managing. To ensure this requirement is met, we check it in the constructor and throw an exception if no string is provided. Each subclass will be required to provide a value for this member variable.

Additionally, the constructor accepts an instance of Doctrine's `EntityManager`, which we use extensively in the rest of the methods to get the work done.

Each of the additional methods simply implements a method of the interface, and uses the `EntityManager` to retrieve and persist data as necessary.

With this abstract class in place, we can start to implement some concrete repositories.

CustomerRepository

```
// src/Persistence/Doctrine/Repository/CustomerRepository.php

namespace CleanPhp\Invoicer\Persistence\Doctrine\Repository;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;

class CustomerRepository extends AbstractDoctrineRepository
  implements CustomerRepositoryInterface {

  protected $entityClass = 'CleanPhp\Invoicer\Domain\Entity\Customer';
}
```

Since we don't have any custom methods outside of what `AbstractDoctrineRepository` defines, we're done!

OrderRepository

```php
// src/Persistence/Doctrine/Repository/OrderRepository.php

namespace CleanPhp\Invoicer\Persistence\Doctrine\Repository;

use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;
use Doctrine\ORM\Query\Expr\Join;

class OrderRepository extends AbstractDoctrineRepository
  implements OrderRepositoryInterface {

  protected $entityClass = 'CleanPhp\Invoicer\Domain\Entity\Order';

  public function getUninvoicedOrders() {
    $builder = $this->entityManager->createQueryBuilder()
      ->select('o')
      ->from($this->entityClass, 'o')
      ->leftJoin(
        'CleanPhp\Invoicer\Domain\Entity\Invoice',
        'i',
        Join::WITH,
        'i.order = o'
      )
      ->where('i.id IS NULL');

    return $builder->getQuery()->getResult();
  }
}
```

The `OrderRepositoryInterface` specifies an additional method to retrieve all orders without invoices, so we have implemented that functionality using Doctrine's `QueryBuilder` class, which allows us to define queries against entity objects. Note that this is a bit different from doing queries against a database schema: we're using the language of the domain, not the database. This query language is called Doctrine Query Language (DQL)[56].

Doctrine provides extensive documentation[57] for the `QueryBuilder` class if you are interested more details.

Invoice Repository

The `InvoiceRepository` is another simple subclass that requires no further customization:

[56]http://doctrine-orm.readthedocs.org/en/latest/reference/dql-doctrine-query-language.html
[57]http://doctrine-orm.readthedocs.org/en/latest/reference/query-builder.html

```php
// src/Persistence/Doctrine/Repository/InvoiceRepository.php

namespace CleanPhp\Invoicer\Persistence\Doctrine\Repository;

use CleanPhp\Invoicer\Domain\Repository\InvoiceRepositoryInterface;

class InvoiceRepository extends AbstractDoctrineRepository
  implements InvoiceRepositoryInterface {

  protected $entityClass = 'CleanPhp\Invoicer\Domain\Entity\Invoice';
}
```

Entity Mapping

Doctrine relies on mapping files[58] to gain information about the database schema and how to map that database schema to the entity objects. Doctrine is an example over configuration over convention. Other ORMs, especially ones that use the ActiveRecord[59] pattern, require way less configuration, but also make deep assumptions about the database.

There are three methods we can use to generate these mappings:

1. DocBlock class annotations on the entities themselves
2. XML Mapping Files
3. YAML Mapping Files

The first option is by far the most common, and is even a recommended standard by Symfony[60]. The second option is for super enterprise people, and the third option is the one we're going to go with.

I have two reasons for not using DocBlocks:

- it leaks persistence information into the domain layer, and the domain layer should not be concerned with how the data is persisted
- it makes the code rely on comments to function properly. I'm a strong believer that comments should not have a direct effect on how code behaves at runtime.

The second complaint mostly goes away if PHP ever implements true annotations in the language, but for now, it's a giant code smell for me.

I have one reason for not using XML: it's XML.

Really, though, go with whatever method you think is best. It's small potatoes in the grand scheme of things, and none of the three methods will affect how the repositories work.

We'll store these mapping files in in the `Persistence/Doctrine/Mapping` folder.

[58]http://doctrine-orm.readthedocs.org/en/latest/reference/basic-mapping.html
[59]http://www.martinfowler.com/eaaCatalog/activeRecord.html
[60]http://symfony.com/doc/current/best_practices/business-logic.html#doctrine-mapping-information

Customers.dcm.yml

This file, `CleanPhp.Invoicer.Domain.Entity.Customer.dcm.yml` dictates mapping information for the `Customer` entity:

```
CleanPhp\Invoicer\Domain\Entity\Customer:
  type: entity
  table: customers
  id:
    id:
      type: bigint
      generator:
        strategy: IDENTITY
  fields:
    name:
      length: 100
    email:
      length: 50
```

Order.dcm.yml

This file, `CleanPhp.Invoicer.Domain.Entity.Order.dcm.yml` dictates mapping information for the `Order` entity:

```
CleanPhp\Invoicer\Domain\Entity\Order:
  type: entity
  table: orders
  id:
    id:
      type: bigint
      generator:
        strategy: IDENTITY
  fields:
    orderNumber:
      column: order_number
      length: 20
    description:
    total:
      type: decimal
      precision: 10
      scale: 2
  manyToOne:
    customer:
      targetEntity: CleanPhp\Invoicer\Domain\Entity\Customer
      inversedBy: orders
```

Invoice.dcm.yml

This file, `CleanPhp.Invoicer.Domain.Entity.Invoice.dcm.yml` dictates mapping information for the `Invoice` entity:

```
CleanPhp\Invoicer\Domain\Entity\Invoice:
  type: entity
  table: invoices
  id:
    id:
      type: bigint
      generator:
        strategy: IDENTITY
  fields:
    invoiceDate:
      column: invoice_date
      type: date
    total:
      type: decimal
      precision: 10
      scale: 2
  manyToOne:
    order:
      targetEntity: CleanPhp\Invoicer\Domain\Entity\Order
      inversedBy: invoices
```

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 10-doctrine-repo-mappings:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 10-doctrine-repo-mappings
```

Integrating Zend Framework and Doctrine

The next step is to wire up Zend Framework and Doctrine to work together. This turns out to be pretty simple thanks to DoctrineOrmModule[61], a ZF2 module written by the Doctrine team to integrate the two projects.

We'll grab the latest version with Composer:

[61]https://github.com/doctrine/DoctrineORMModule

```
composer require doctrine/doctrine-orm-module
```

DoctrineOrmModule will bring all the necessary Doctrine dependencies along with it. Now we just have to configure this module within Zend Framework and we're ready to use it.

The first thing we'll want to do is enable the DoctrineORMModule and DoctrineModule (a dependency of DoctrineORMModule):

```
// config/application.config.php

return [
  'modules' => [
    'DoctrineModule',
    'DoctrineORMModule',
    'Application',
  ],
  // ...
];
```

Next, we'll drop in a global configuration file to define the basics of the Doctrine setup, including which mapping driver to use, where to find the entities, and where to find the mapping files:

```
// config/autoload/db.global.php

return [
  'doctrine' => [
    'driver' => [
      'orm_driver' => [
        'class' => 'Doctrine\ORM\Mapping\Driver\YamlDriver',
        'cache' => 'array',
        'paths' => [
          realpath(__DIR__ . '/../../src/Domain/Entity'),
          realpath(__DIR__ . '/../../src/Persistence/Doctrine/Mapping')
        ],
      ],
      'orm_default' => [
        'drivers' => ['CleanPhp\Invoicer\Domain\Entity' => 'orm_driver']
      ]
    ],
  ],
];
```

We'll also modify `db.local.php` file to setup the database connection information for DoctrineModule. This is the same information we had previously, just modified slightly to fit the format expected by DoctrineModule:

```php
// config/autoload/db.local.php

return [
  'doctrine' => [
    'connection' => [
      'orm_default' => [
        'driverClass' => 'Doctrine\DBAL\Driver\PDOSqlite\Driver',
        'params' => [
          'path' => __DIR__ . '/../../data/database.db',
        ]
      ]
    ],
  ],
];
```

And that's it! Zend Framework is now configured to use Doctrine, so let's start using it.

Injecting the New Repositories

The only thing left to do is to start using Doctrine. We'll want to inject these new repositories into the controllers so they can start using them.

Rather than put a bunch of duplicated code in the service config, we're going to create a Zend `ServiceManager` Factory that will be responsible for instantiating all of the Repositories.

```php
// src/Persistence/Doctrine/Repository/RepositoryFactory;

namespace CleanPhp\Invoicer\Persistence\Doctrine\Repository;

use RuntimeException;
use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class RepositoryFactory implements FactoryInterface {
  public function createService(ServiceLocatorInterface $sl) {
    $class = func_get_arg(2);
    $class = 'CleanPhp\Invoicer\Persistence\Doctrine\Repository\\' . $class;

    if (class_exists($class, true)) {
      return new $class(
        $sl->get('Doctrine\ORM\EntityManager')
      );
    }

    throw new RuntimeException(
      'Unknown Repository requested: ' . $class
    );
  }
}
```

This factory simply does a string replace on the passed service locator key (which thanks to ZF2 is hidden within `func_get_args()`) to normalize it into the fully qualified namespace for the requested repository, instantiates the resulting class, and returns the instantiated object.

If the factory can't find the class for the requested repository, it throws a `RuntimeException`.

Next, we'll make a couple entries in the service config to utilize this `RepositoryFactory` when any of the new Doctrine-base repositories are requested:

```
// config/autoload/global.php

return [
  'service_config' => [
    'factories' => [
      'OrderHydrator' => function ($sm) {
        return new OrderHydrator(
          new ClassMethods(),
          $sm->get('CustomerRepository')
        );
      },
      'CustomerRepository' =>
        'CleanPhp\Invoicer\Persistence\Doctrine\Repository\RepositoryFactory',
      'InvoiceRepository' =>
        'CleanPhp\Invoicer\Persistence\Doctrine\Repository\RepositoryFactory',
      'OrderRepository' =>
        'CleanPhp\Invoicer\Persistence\Doctrine\Repository\RepositoryFactory',
    ]
  ]
];
```

Finally, we'll update the controller config to inject these new repositories into the controllers:

```
// module/Application/config/module.config.php

return [
  // ...
  'controllers' => [
     'invokables' => [
       'Application\Controller\Index' =>
         'Application\Controller\IndexController'
     ],
     'factories' => [
       'Application\Controller\Customers' => function ($sm) {
         return new \Application\Controller\CustomersController(
           $sm->getServiceLocator()->get('CustomerRepository'),
           new CustomerInputFilter(),
           new ClassMethods()
         );
       },
       'Application\Controller\Invoices' => function ($sm) {
         return new \Application\Controller\InvoicesController(
           $sm->getServiceLocator()->get('InvoiceRepository'),
           $sm->getServiceLocator()->get('OrderRepository'),
           new InvoicingService(
             $sm->getServiceLocator()->get('OrderRepository'),
             new InvoiceFactory()
           )
         );
       },
       'Application\Controller\Orders' => function ($sm) {
         return new \Application\Controller\OrdersController(
           $sm->getServiceLocator()->get('OrderRepository'),
           $sm->getServiceLocator()->get('CustomerRepository'),
           new OrderInputFilter(),
           $sm->getServiceLocator()->get('OrderHydrator')
         );
       },
     ],
  ],
  // ...
];
```

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 11-doctrine-integration:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 11-doctrine-integration
```

Updating the Hydrators

The last update we need to make to the persistence layer is updating the hydrators to work with Doctrine. Doctrine handles the actual hydration pretty well. In fact, we don't need the

`InvoiceHydrator` anymore. We can go ahead and delete that, which means we can also delete the `DateStrategy` strategy we built.

We'll also make some updates to the `OrderHydrator`, first starting with the specs:

```
// specs/hydrator/order.spec.php

use CleanPhp\Invoicer\Domain\Entity\Customer;
use CleanPhp\Invoicer\Domain\Entity\Order;
use CleanPhp\Invoicer\Persistence\Hydrator\OrderHydrator;
use Zend\Stdlib\Hydrator\ClassMethods;

describe('Persistence\Hydrator\OrderHydrator', function () {
  beforeEach(function() {
    $this->repository = $this->getProphet()->prophesize(
      'CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface'
    );
    $this->hydrator = new OrderHydrator(
      new ClassMethods(),
      $this->repository->reveal()
    );
  });

  describe('->hydrate()', function () {
    it('should perform basic hydration of attributes', function () {
      $data = [
        'id' => 100,
        'order_number' => '20150101-019',
        'description' => 'simple order',
        'total' => 5000
      ];

      $order = new Order();
      $this->hydrator->hydrate($data, $order);

      expect($order->getId())->to->equal(100);
      expect($order->getOrderNumber())->to->equal('20150101-019');
      expect($order->getDescription())->to->equal('simple order');
      expect($order->getTotal())->to->equal(5000);
    });

    it('should hydrate the embedded customer data', function () {
      $data = ['customer' => ['id' => 20]];
      $order = new Order();

      $this->repository->getById(20)->willReturn((new Customer())->setId(20));
      $this->hydrator->hydrate($data, $order);
      assert(
        $data['customer']['id'] === $order->getCustomer()->getId(),
        'id does not match'
      );
    });
  });

  describe('->extract()', function () {
    // ...
  });
});
```

We've removed the test case *should hydrate a Customer entity on the Order* and updated the *should hydrate the embedded customer data* test case to expect the repository to be used to query for the `Customer`.

We don't need the hydration of the customer via `customer_id` as that's not the way Doctrine provides the data, and Doctrine takes care of the hydration for us.

As Doctrine's `UnitOfWork` needs to know about all existing entities, otherwise it tries to re-INSERT them, we've updated the hydrator to query for the customer by ID if it encounters an ID. Another way we could solve this problem is to use the `EntityManager::merge()` method to make Doctrine aware of the entity.

Let's make the corresponding changes to the `OrderHydrator`:

```
// src/Persistence/Hydrator/OrderHydrator.php

namespace CleanPhp\Invoicer\Persistence\Hydrator;

use CleanPhp\Invoicer\Domain\Entity\Order;
use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use Zend\Stdlib\Hydrator\HydratorInterface;

class OrderHydrator implements HydratorInterface {
  // ...

  public function hydrate(array $data, $order) {
    if (isset($data['customer'])
    && isset($data['customer']['id'])) {
      $data['customer'] = $this->customerRepository->getById(
        $data['customer']['id']
      );
    }
    return $this->wrappedHydrator->hydrate(
      $data,
      $order
    );
  }
}
```

Now if we start clicking around the application, it should continue to work! It looks, from a UI and interaction perspective, like nothing has changed. Under the hood, however, we're using an entirely different database architecture.

That's very powerful.

git This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 12-doctrine-hydrators:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 12-doctrine-hydrators
```

Summary

The application is now using Doctrine for its underlying database abstraction layer. We didn't have to change any application code in the process, as the application code relied only on interfaces and dependency injection. Since what is injected works as it should, the application continues to function without further modifications.

We also have the benefit of having to write much less code to get database interaction to work. In most cases, we're simply proxying off to Doctrine's `EntityManager` to do the work for us.

Switching to Laravel

We're going to try a much larger undertaking for our next experiment. Up until now, we've used Zend Framework 2 for our controller services + view layer of the application. Zend Framework 2 is great, if not over complicated, but everyone seems to love Laravel, so let's stop fighting against the current.

The application layer is the biggest part of our application. In a real application, it would house dozens upon dozens of controllers, hundreds of views, and many, many routes and configuration.

Our goal this entire time has been to lean on the framework as little as possible. This was the purpose of our domain model and domain services layers.

While we will have to rewrite, or at least tweak, all of our controllers and views, we should not have to touch any of our domain, persistence, and business logic. If we find ourselves dipping into those layers as part of switching frameworks, then we've done something very wrong.

Let's get started.

Setting up Laravel

Let's get started by setting up a new Laravel project. Once we have the source code downloaded, we'll go ahead and move our existing core (located in `src/`), to the Laravel project.

For the most part, we'll follow the official Laravel docs[62].

Let's start by creating a new Laravel project at `cleanphp-laravel` and then running the `artisan fresh` command to purge some of the scaffolding we don't need:

```
composer create-project laravel/laravel --prefer-dist cleanphp-laravel
cd cleanphp-laravel
php artisan fresh
```

Next, we'll copy over some of our code from our original project, starting with the `.git` directory to retain our source control history, as well as the `src/` and `specs/` directories, and the Peridot configuration file, `peridot.php`.

We're going to rename `src/` to `core/` (for no real good reason other than `src` is short for "source" and we have a lot of source code outside of this directory):

[62]http://laravel.com/docs

```
cp -R ../cleanphp-example/.git .
cp -R ../cleanphp-example/src ../cleanphp-example/specs \
  ../cleanphp-example/peridot.php .
git mv src core
```

Let's also add an entry to the `autoload` section of `composer.json` to autoload code in our `core/` directory:

```
"autoload": {
  "classmap": [
    "database"
  ],
  "psr-4": {
    "App\\": "app/",
    "CleanPhp\\Invoicer\\": "core/"
  }
},
```

This requires us to reload the Composer autoloader so that it contains the autoload specification for this new entry:

```
composer dump-autoload
```

We'll also want to bring our database along with us:

```
cp -R ../cleanphp-example/data/database.db storage/database.db
git mv data/database.db storage/database.db
```

Next, let's grab our `application.css` file from the old project and move it over, and remove some of the CSS and fonts that come default with Laravel:

```
mkdir public/css
cp ../cleanphp-example/public/css/application.css public/css/
rm -rf public/css/app.css public/favicon.ico public/fonts
```

If we run `git status`, we'll see that we created a mess of our source control by moving all of these things around. Let's go add some of the important Laravel code, and remove some of the ZF2 stuff we no longer need:

```
git add .gitignore app/ artisan bootstrap/ \
  composer.* config/ public/index.php resources/ \
  server.php storage/

git rm -rf data/ init_autoloader.php module/
```

This Laravel application should now be ready for us to start porting over some of the code from our ZF2 project. Let's fire up the development server and have a look in the browser:

```
php artisan serve
```

If we visit *http://localhost:8000* in the browser, we should see the Laravel welcome page.

If you run `git status` again, you'll notice we still have some uncommitted files. We can ignore those for now, or go ahead and get rid of them.

Our last step is to get Peridot installed again and verify that our specs are all still passing:

```
composer require --dev peridot-php/peridot peridot-php/leo \
  peridot-php/peridot-watcher-plugin peridot-php/peridot-prophecy-plugin
```

And run Peridot:

```
./vendor/bin/peridot specs/
```

Whoops! Looks like our specs are failing due to missing ZF2 dependencies. We'll fix that in a bit.

git

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 13-laravel-setup:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 13-laravel-setup
```

Configuring Doctrine

Next up, we're going to configure Doctrine. By default, Laravel uses Eloquent ORM[63]. We'll opt to keep using Doctrine for now to limit how much code we'll need to rewrite.

To use Doctrine, we'll need to install a third party provider to make the two projects talk. In the course of writing this book, I reviewed several of these "bridge" libraries. Unfortunately, all of

[63]http://laravel.com/docs/eloquent

them either only worked for Laravel 4, depended on broken or non existent versions or commits of Doctrine, or only supported XML or Annotation based mapping files.

So I quickly wrote a very minimal library to integrate the two projects. We'll use this in our examples, but please don't use it in production. Bad things may very well happen.

To hopefully prevent public usage, this library was not published on Packagist, so we'll have to manually add a repository to the `composer.json` file:

```
"repositories": [
  {
    "type": "vcs",
    "url": "https://github.com/mrkrstphr/laravel-indoctrinated.git"
  }
],
"require": {
  "laravel/framework": "5.0.*",
  "mrkrstphr/laravel-indoctrinated": "dev-master"
},
```

Run `composer update` to install this new repository, which will bring along with it `doctrine/orm` and its dependencies.

Once that's installed, we'll need to add the new provider to the array of providers located at `config/app.php`:

```
return [
  'providers' => [
    // ...
    'Mrkrstphr\LaravelIndoctrinated\DoctrineOrmServiceProvider'
  ],
  // ...
];
```

Next, we'll run a command to publish this provider, which generates a sample config file for us:

```
php artisan vendor:publish \
  --provider "Mrkrstphr\LaravelIndoctrinated\DoctrineOrmServiceProvider"
```

Now let's modify our config file located at `config/doctrine.php`:

```
return [
  // database connection information is managed in Laravel's
  // config/database.php file

  'mappings' => [
    'type' => 'yaml',
    'paths' => [__DIR__ . '/../core/Persistence/Doctrine/Mapping']
  ],
];
```

As the config file says, our database connection information will be managed by Laravel. Let's change the `config/database.php` file as follows:

```
return [
  'default' => 'sqlite',
  'connections' => [
    'sqlite' => [
      'driver' => 'sqlite',
      'database' => storage_path() . '/database.db',
    ]
  ]
];
```

We now have Laravel configured to talk to Doctrine and setup to use our `database.db` sqlite database.

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 14-laravel-doctrine:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 14-laravel-doctrine
```

Setting up the Dashboard

By default, Laravel sets up a `WelcomeController` located at `app/Http/Controllers/WelcomeController.php`. Let's rename this to `DashboardController.php` and modify its contents:

```
// app/Http/Controllers/DashboardController.php

namespace App\Http\Controllers;

class DashboardController extends Controller {
  public function indexAction() {
    return view('dashboard');
  }
}
```

Let's also move the `resources/views/welcome.blade.php` template file to `dashboard.blade.php` and copy over our `dashboard.phtml` from our previous code:

```
@extends('layouts.layout')

@section('content')
<div class="jumbotron">
  <h1>Welcome to CleanPhp Invoicer!</h1>
  <p>
    This is the case study project for The Clean Architecture
    in PHP, a book about writing excellent PHP code.
  </p>
  <p>
    <a href="https://leanpub.com/cleanphp" class="btn btn-primary">
      Check out the Book</a>
  </p>
</div>
@stop
```

Laravel ships with a templating engine called Blade[64]. In the example above, we have a Blade template that extends another template named `layouts.layout`, which will be our overall layout. Next, we define a section named `content` with the actual content of our template.

Let's create the layout now so that we can see how Blade merges the two. We'll place this at `resources/views/layouts/layout.blade.php`:

[64]http://laravel.com/docs/templates

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>CleanPhp</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <link href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.1/css/bootstrap.min.css"
    media="screen" rel="stylesheet" type="text/css">
  <link href="/css/application.css" media="screen"
    rel="stylesheet" type="text/css">
</head>
<body>
<nav class="navbar navbar-default navbar-fixed-top" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <a class="navbar-brand" href="/">CleanPhp</a>
    </div>
    <div class="collapse navbar-collapse">
      <ul class="nav navbar-nav">
        <li>
          <a href="/customers">Customers</a>
        </li>
        <li>
          <a href="/orders">Orders</a>
        </li>
        <li>
          <a href="/invoices">Invoices</a>
        </li>
      </ul>
    </div>
  </div>
</nav>
<div class="container">
  <?php if (Session::has('success')): ?>
  <div class="alert alert-success"><?= Session::get('success') ?></div>
  <?php endif; ?>
  @yield('content')
  <hr>
  <footer>
    <p>I'm the footer.</p>
  </footer>
</div>
</body>
</html>
```

This layout file is largely the same as our layout in ZF2. We're using Blade to render the area named *content*, which was defined in our `dashboard.blade.php` template file, at a specific spot. Each of our templates going forward will define this `content` section.

We're also using Laravel's `Session` facade to grab data from the session, namely our flash message for when we need to alert the user to something awesome happening (we'll get to that in a bit when we start working on the other controllers).

The last thing we need to update is our route for the dashboard. Laravel routes are stored in the `app/Http/routes.php` file. Currently, the only route defined is still pointing at the defunct `WelcomeController`. Let's fix that:

```
Route::get('/', 'DashboardController@indexAction');
```

Now we're instructing Laravel to render the `DashboardController::indexAction()` when a GET request is made to `/`.

Try it out in your browser. You should see our lovely dashboard, back in action!

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 15-laravel-dashboard:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 15-laravel-dashboard
```

Customer Management

Now let's move on to managing customers. We'll start this time by defining the routes we need for the `CustomersController`:

```
// app/Http/routes.php

// ...

Route::get('/customers', 'CustomersController@indexAction');
Route::match(
  ['get', 'post'],
  '/customers/new',
  'CustomersController@newOrEditAction'
);
Route::match(
  ['get', 'post'],
  '/customers/edit/{id}',
  'CustomersController@newOrEditAction'
);
```

These new routes are set up to the actions we laid out in Zend Framework 2. So let's copy over our `CustomersController` with the necessary changes to make it work for Laravel.

We'll start by defining our basic controller:

```
// app/Http/Controllers/CustomersController.php

namespace App\Http\Controllers;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;

class CustomersController extends Controller {
  private $customerRepository;

  public function __construct(
    CustomerRepositoryInterface $customerRepository
  ) {
    $this->customerRepository = $customerRepository;
  }

  // ...
}
```

This basic controller looks almost exactly like it did in ZF2, although missing some dependencies (for now), and has been moved to a new namespace `App\Http\Controllers`.

Laravel ships with a very powerful Service Container[65] that handles dependency injection very well. When `CustomersController` is instantiated, it will automatically search the container for `CustomerRepositoryInterface` and load it if found. If not found, it will try to instantiate the object, although in our case, it will not try to instantiate an interface.

So the next thing we need to do is get an entry for `CustomerRepositoryInterface` into the service container. We'll do so in the `app/Providers/AppServiceProvider.php` file:

[65]http://laravel.com/docs/container

```
// app/Providers/AppServiceProvider.php

namespace App\Providers;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use CleanPhp\Invoicer\Persistence\Doctrine\Repository\CustomerRepository;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider {
  public function register() {
    $this->app->bind(
      CustomerRepositoryInterface::class,
      function ($app) {
        return new CustomerRepository(
          $app['Doctrine\ORM\EntityManagerInterface']
        );
      }
    );
  }
}
```

We bind an entry into the service manager with the fully qualified namespace name of `CustomerRepositoryInterface`, using the `::class` keyword off that object (if we refactor this class and rename it, we won't have to worry about missing some string-based references). When invoked, this entry will return an instantiated `CustomerRepository`, passing along an instance of Doctrine's `EntityManager`, which comes from the Doctrine provider we setup earlier.

This is all we need to do to take care of the instantiate of the controller. Laravel takes care of the rest!

Let's move on to creating the Customer listing.

Customer Listing

We'll start with the listing of customers (the `/customers` route, which translates to `indexAction()`

Let's add our `indexAction()` to `CustomersController`:

```
public function indexAction() {
  $customers = $this->customerRepository->getAll();
  return view('customers/index', ['customers' => $customers]);
}
```

Just like in ZF2, we're pulling all the customers out of the `CustomersRepository`, and passing them along to the view. We're using Laravel's `view()` helper to define the template to render and the data to provide to it.

Let's create our view `resources/views/customers/index.blade.php`:

```
@extends('layouts.layout')

@section('content')
<div class="page-header clearfix">
  <h2 class="pull-left">Customers</h2>
  <a href="/customers/new" class="btn btn-success pull-right">
    Create Customer</a>
</div>

<table class="table">
  <thead>
  <tr>
    <th>#</th>
    <th>Name</th>
    <th>Email</th>
  </tr>
  </thead>
  <?php foreach ($customers as $customer): ?>
    <tr>
      <td>
        <a href="/customers/edit/{{{ $customer->getId() }}}">
          {{{ $customer->getId() }}}</a>
      </td>
      <td>{{{ $customer->getName() }}}</td>
      <td>{{{ $customer->getEmail() }}}</td>
    </tr>
  <?php endforeach; ?>
</table>
@stop
```

Just like in our `dashboard.blade.php`, we're stating that this template extends `layouts/layout.blade.php` and we're defining a section named "content" with our actual view.

New here is the usage of Blades templating syntax. Instead of using `echo` statements, we're wrapping the variable we want printed with `{{{ }}}`, which is Laravel's escape and output syntax. This protects us against XSS injections by filtering user input data.

Now if we navigate to the *Customers* page, or manually visit `/customers`, we should see a populated grid of our customers.

Adding and Editing Customers

Next, we'll recreate our ability to add and edit customers. We'll need a few more dependencies to make this happen, namely our `CustomerInputFilter` and `Customer` object hydrator.

These classes exist in our `core/` directory, but they are dependent upon some Zend Framework libraries. We purposefully put these classes in the `src/` directory when working in ZF2, instead of putting them within the `module/` directory structure so that we could reuse them.

ZF2 is organized as a collection of components. Namely, we'll need the Zend InputFilter and Zend StdLib (which houses the hydration classes) components. Unfortunately, ZF2 isn't as modular as it sets itself up to be. These two libraries actually depend on the Zend ServiceManager and Zend I18N libraries, but they don't state it in their `composer.json` file's `require` block.

This is quite a bit of hogwash. We'll need to require all these libs:

```
composer require zendframework/zend-inputfilter \
  zendframework/zend-servicemanager \
  zendframework/zend-i18n \
  zendframework/zend-stdlib
```

Once we have our Laravel application up and running, we'll probably want to refactor away these components to use their Laravel counterparts.

Another benefit at this point is that our Peridot tests should pass now with these dependencies in place:

```
./vendor/bin/peridot specs
```

Now that we have these dependencies, we can inject them into the `CustomersController` constructor:

```
// app/Http/Controllers/CustomersController.php

namespace App\Http\Controllers;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use CleanPhp\Invoicer\Service\InputFilter\CustomerInputFilter;
use Zend\Stdlib\Hydrator\HydratorInterface;

class CustomersController extends Controller {
  protected $customerRepository;
  protected $inputFilter;
  protected $hydrator;

  public function __construct(
    CustomerRepositoryInterface $customerRepository,
    CustomerInputFilter $inputFilter,
    HydratorInterface $hydrator
  ) {
    $this->customerRepository = $customerRepository;
    $this->inputFilter = $inputFilter;
    $this->hydrator = $hydrator;
  }

  // ...
}
```

Laravel knows to simply instantiate the `CustomerInputFilter`, but it can't instantiate an interface, so we need to tell it what to do with `HydratorInterface`. Let's instruct it to instantiate Zend's `ClassMethods` hydrator to meet this dependency:

```php
use Zend\Stdlib\Hydrator\ClassMethods;
use Zend\Stdlib\Hydrator\HydratorInterface;

class AppServiceProvider extends ServiceProvider {
  public function register() {
    $this->app->bind(HydratorInterface::class, function ($app) {
      return new ClassMethods();
    });

    // ...
  }
}
```

Now when faced with a request for a `HydratorInterface`, Laravel will simply instantiate `ClassMethods`.

Let's implement the `newOrEditAction()`:

```php
// app/Http/Controllers/CustomersController.php

public function newOrEditAction(Request $request, $id = '') {
  $viewModel = [];

  $customer = $id ? $this->customerRepository->getById($id) : new Customer();

  if ($request->getMethod() == 'POST') {
    $this->inputFilter->setData($request->request->all());

    if ($this->inputFilter->isValid()) {
      $this->hydrator->hydrate(
        $this->inputFilter->getValues(),
        $customer
      );

      $this->customerRepository
        ->begin()
        ->persist($customer)
        ->commit();

      Session::flash('success', 'Customer Saved');

      return new RedirectResponse(
        '/customers/edit/' . $customer->getId()
      );
    } else {
      $this->hydrator->hydrate(
        $request->request->all(),
        $customer
      );
      $viewModel['error'] = $this->inputFilter->getMessages();
    }
  }

  $viewModel['customer'] = $customer;

  return view('customers/new-or-edit', $viewModel);
}
```

So there's a lot going on here, however it's nearly identical to our ZF2 code for the same action. Let's step through it:

1. As we have a variable `{id}` in our route, Laravel is kind enough to pass that along as an argument to the action. We're also asking for an instance of `Illuminate\Http\Request`, and Laravel is happy to oblige.
2. We setup an empty "ViewModel" array variable and use that throughout the method to collect data to pass along to the view, which we do at the very end using Laravel's `view()` helper.
3. If we have an ID, we utilize the `CustomerRepository` to retrieve that Customer from the database, otherwise we instantiate an empty `Customer` object.
4. If the request is a `GET` request, we simply add the `$customer` to the `$viewModel` and carry on.
5. If the request is a `POST`, we populate the `CustomerInputFilter` with the posted data, and then check to see if the input filter is valid.
6. If the input is valid, we hydrate the `$customer` object with the posted values, persist it to the repository, setup a flash success message using Laravel's `Session` facade, and redirect to the edit page for the Customer.
7. If the input is not valid, we hydrate the customer with the raw posted data, store the validation error messages in the `$viewModel`, and carry on.

For this to work, we'll need to add a couple more `use` statements to the top of the controller (I recommend alphabetizing them):

```
use CleanPhp\Invoicer\Domain\Entity\Customer;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Session;
use Symfony\Component\HttpFoundation\RedirectResponse;
```

Now, of course, we need to setup the template file `customers/new-or-edit` that's referenced in the action:

```
<!-- resources/views/customers/new-or-edit.blade.php -->

@extends('layouts.layout')

@section('content')
<div class="page-header clearfix">
  <h2>
    <?= !empty($customer->getId()) ? 'Edit' : 'New' ?>
    Customer
  </h2>
</div>

<form role="form" action="" method="post">
  <input type="hidden" name="_token" value="<?= csrf_token(); ?>">

  <div class="form-group">
    <label for="name">Name:</label>
    <input type="text" class="form-control" name="name" id="name"
      placeholder="Enter Name" value="<?= $customer->getName() ?>">
    @include(
      'validation-errors',
      ['name' => 'name', 'errors' => isset($error) ? $error : []]
    )
  </div>
  <div class="form-group">
    <label for="email">Email:</label>
    <input type="text" class="form-control" name="email" id="email"
      placeholder="Enter Email" value="<?= $customer->getEmail() ?>">
    @include(
      'validation-errors',
      ['name' => 'email', 'errors' => isset($error) ? $error : []]
    )
  </div>
  <button type="submit" class="btn btn-primary">Save</button>
</form>
@stop
```

Again, we're using the Blade templating language, but otherwise this template is nearly verbatim from our ZF2 project.

Some differences:

1. We're setting up a CSRF[66] token so that Laravel can validate the POST as authentic.
2. We're using a partial instead of a view helper to show the validation error messages for a particular element.

We'll need to setup that partial in order for this to work, so let's do that now:

```
<!-- resources/views/validation-errors.blade.php -->

<?php if ($errors): $errors = \Vnn\Keyper\Keyper::create($errors) ?>
<?php if ($errors->get($name)): ?>
<div class="alert alert-danger">
    <?= implode('. ', $errors->get($name)) ?>
</div>
<?php endif; ?>
<?php endif; ?>
```

As we did with our ZF2 ViewHelper, we're using Keyper to easily check for nested error messages. If we have any error messages for the input, we implode them with a period, then format them nicely with some Bootstrap styles.

Let's make sure we have Keyper installed for this to work:

```
composer require vnn/keyper
```

With this in place, we now have the ability to add and edit Customers again!

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 16-laravel-customers:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 16-laravel-customers
```

Order Management

Let's keep rolling right into Orders. Again, we'll start by defining our routes, which are taken and adapted from our ZF2 application:

[66] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29

```
// app/Http/routes.php

// ...

Route::get('/orders', 'OrdersController@indexAction');
Route::match(['get', 'post'], '/orders/new', 'OrdersController@newAction');
Route::get('/orders/view/{id}', 'OrdersController@viewAction');
```

We'll start with our `indexAction()`.

Listing Orders

Let's create our `OrdersController` with its `indexAction()`:

```
// app/Http/Controllers/OrdersController.php

namespace App\Http\Controllers;

use CleanPhp\Invoicer\Domain\Repository\CustomerRepositoryInterface;
use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;
use CleanPhp\Invoicer\Persistence\Hydrator\OrderHydrator;
use CleanPhp\Invoicer\Service\InputFilter\OrderInputFilter;

class OrdersController extends Controller {
  protected $orderRepository;
  protected $customerRepository;
  protected $inputFilter;
  protected $hydrator;

  public function __construct(
    OrderRepositoryInterface $orderRepository,
    CustomerRepositoryInterface $customerRepository,
    OrderInputFilter $inputFilter,
    OrderHydrator $hydrator
  ) {
    $this->orderRepository = $orderRepository;
    $this->customerRepository = $customerRepository;
    $this->inputFilter = $inputFilter;
    $this->hydrator = $hydrator;
  }

  public function indexAction() {
    $orders = $this->orderRepository->getAll();
    return view('orders/index', ['orders' => $orders]);
  }
}
```

Again, we'll have to inform Laravel of what concrete class to instantiate for `CustomerRepositoryInterface`:

```
// app/Providers/AppServiceProvider.php

public function register() {
  // ...

  $this->app->bind(
    OrderRepositoryInterface::class,
    function ($app) {
      return new OrderRepository(
        $app['Doctrine\ORM\EntityManagerInterface']
      );
    }
  );
}
```

Also make sure to drop the required `use` statements at the top:

```
use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;
use CleanPhp\Invoicer\Persistence\Doctrine\Repository\OrderRepository;
```

Last, let's add our order index file

```
<!-- resources/views/orders/index.blade.php -->

@extends('layouts.layout')

@section('content')
<div class="page-header clearfix">
  <h2 class="pull-left">Orders</h2>
  <a href="/orders/new" class="btn btn-success pull-right">
    Create Order</a>
</div>

<table class="table table-striped clearfix">
  <thead>
  <tr>
    <th>#</th>
    <th>Order Number</th>
    <th>Customer</th>
    <th>Description</th>
    <th class="text-right">Total</th>
  </tr>
  </thead>
  <?php foreach ($orders as $order): ?>
  <tr>
    <td>
      <a href="/orders/view/{{{ $order->getId() }}}">
        {{{ $order->getId() }}}</a>
    </td>
    <td>{{{ $order->getOrderNumber() }}}</td>
    <td>
      <a href="/customers/edit/{{{ $order->getCustomer()->getId() }}}">
        {{{ $order->getCustomer()->getName() }}}</a>
    </td>
    <td>{{{ $order->getDescription() }}}</td>
    <td class="text-right">
      $ {{ number_format($order->getTotal(), 2) }}
    </td>
  </tr>
  <?php endforeach; ?>
</table>
@stop
```

And now we have the ability to list orders.

Viewing Orders

Listing orders was easy, and viewing orders should be just as easy. Let's start with the controller action:

```
// app/Http/Controllers/OrdersController.php

public function viewAction($id) {
  $order = $this->orderRepository->getById($id);

  if (!$order) {
    return new Response('', 404);
  }

  return view('orders/view', ['order' => $order]);
}
```

We've introduced a new object `Response`, so let's make sure we add a `use` statement for it at the top of the file:

```
use Illuminate\Http\Response;
```

And then swiftly on to the template:

```
<!-- resources/views/orders/view.blade.php -->

@extends('layouts.layout')

@section('content')
<div class="page-header clearfix">
  <h2>Order #{{{ $order->getOrderNumber() }}}</h2>
</div>

<table class="table table-striped">
  <thead>
  <tr>
    <th colspan="2">Order Details</th>
  </tr>
  </thead>
  <tr>
    <th>Customer:</th>
    <td>
      <a href="/customers/edit/{{{ $order->getCustomer()->getId() }}}">
        {{{ $order->getCustomer()->getName() }}}</a>
    </td>
  </tr>
  <tr>
    <th>Description:</th>
    <td>{{{ $order->getDescription() }}}</td>
  </tr>
  <tr>
    <th>Total:</th>
    <td>$ {{{ number_format($order->getTotal(), 2) }}}</td>
  </tr>
</table>
@stop
```

Pretty much the same stuff we've been doing. And it works!

Adding Orders

Let's work on adding orders, now. The controller action:

```
// app/Http/Controllers/OrdersController.php

public function newAction(Request $request) {
  $viewModel = [];
  $order = new Order();

  if ($request->getMethod() == 'POST') {
    $this->inputFilter
      ->setData($request->request->all());

    if ($this->inputFilter->isValid()) {
      $order = $this->hydrator->hydrate(
        $this->inputFilter->getValues(),
        $order
      );

      $this->orderRepository
        ->begin()
        ->persist($order)
        ->commit();

      Session::flash('success', 'Order Saved');
      return new RedirectResponse(
        '/orders/view/' . $order->getId()
      );
    } else {
      $this->hydrator->hydrate(
        $request->request->all(),
        $order
      );
      $viewModel['error'] = $this->inputFilter->getMessages();
    }
  }

  $viewModel['customers'] = $this->customerRepository->getAll();
  $viewModel['order'] = $order;

  return view('orders/new', $viewModel);
}
```

This action is nearly identical to the `CustomersController::newOrEditAction()`, so if you want an explanation of what is going on, go check out that section. The only new thing we've added is querying for the list of Customers so that the user can select which Customer the Order is for.

We've added quite a few new objects here, so let's add them to the `use` statement block:

```
use CleanPhp\Invoicer\Domain\Entity\Order;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Session;
use Symfony\Component\HttpFoundation\RedirectResponse;
```

Next, the template:

```
<!-- resources/views/orders/new.blade.php -->

@extends('layouts.layout')

@section('content')
<div class="page-header clearfix">
    <h2>Create Order</h2>
</div>

<form role="form" action="" method="post">
  <input type="hidden" name="_token" value="<?= csrf_token(); ?>">

  <div class="form-group">
    <label for="customer_id">Customer:</label>
    <select class="form-control" name="customer[id]" id="customer_id">
      <option value=""></option>
      <?php foreach ($customers as $customer): ?>
        <option value="{{{ $customer->getId() }}}"<?=
        !is_null($order->getCustomer()) &&
        $order->getCustomer()->getId() == $customer->getId() ?
          ' selected="selected"' : '' ?>>
          {{{ $customer->getName() }}}
        </option>
      <?php endforeach; ?>
    </select>
    @include(
      'validation-errors',
      ['name' => 'customer.id', 'errors' => isset($error) ? $error : []]
    )
  </div>
  <div class="form-group">
    <label for="orderNumber">Order Number:</label>
    <input type="text" class="form-control" name="orderNumber"
      id="order_number" placeholder="Enter Order Number"
      value="{{{ $order->getOrderNumber() }}}">
    @include(
      'validation-errors',
      ['name' => 'orderNumber', 'errors' => isset($error) ? $error : []]
    )
  </div>
  <div class="form-group">
    <label for="description">Description:</label>
    <input type="text" class="form-control" name="description"
      id="description" placeholder="Enter Description"
      value="{{{ $order->getDescription() }}}">
    @include(
      'validation-errors',
      ['name' => 'description', 'errors' => isset($error) ? $error : []]
    )
  </div>
  <div class="form-group">
    <label for="total">Total:</label>
    <input type="text" class="form-control" name="total"
      id="total" placeholder="Enter Total"
      value="{{{ $order->getTotal() }}}">
    @include(
      'validation-errors',
      ['name' => 'total', 'errors' => isset($error) ? $error : []]
    )
  </div>
  <button type="submit" class="btn btn-primary">Save</button>
</form>
@stop
```

The new thing here is the select box for selecting the Customer for the Order. We simply loop through the provided array of customers and output an `<option>` for each one, just as we did in the ZF2 application.

This concludes Order Management!

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 17-laravel-orders:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 17-laravel-orders
```

Invoice Management

As before, let's start with the routes:

```
// app/Http/routes.php

// ...

Route::get('/invoices', 'InvoicesController@indexAction');
Route::get('/invoices/view/{id}', 'InvoicesController@viewAction');
Route::get('/invoices/new', 'InvoicesController@newAction');
Route::post('/invoices/generate', 'InvoicesController@generateAction');
```

And stub out our `InvoicesController`:

```
namespace App\Http\Controllers;

use CleanPhp\Invoicer\Domain\Repository\InvoiceRepositoryInterface;
use CleanPhp\Invoicer\Domain\Repository\OrderRepositoryInterface;
use CleanPhp\Invoicer\Domain\Service\InvoicingService;

class InvoicesController extends Controller {
  protected $invoiceRepository;
  protected $orderRepository;
  protected $invoicing;

  public function __construct(
    InvoiceRepositoryInterface $invoices,
    OrderRepositoryInterface $orders,
    InvoicingService $invoicing
  ) {
    $this->invoiceRepository = $invoices;
    $this->orderRepository = $orders;
    $this->invoicing = $invoicing;
  }
}
```

We'll also need to let Laravel know what to instantiate for `InvoiceRepositoryInterface`:

```
// app/Providers/AppServiceProvider.php
public function register() {
  // ...

  $this->app->bind(
    InvoiceRepositoryInterface::class,
    function ($app) {
      return new InvoiceRepository(
        $app['Doctrine\ORM\EntityManagerInterface']
      );
    }
  );
}
```

And let's not forget the two new `use` statements at the top of the file:

```
use CleanPhp\Invoicer\Domain\Repository\InvoiceRepositoryInterface;
use CleanPhp\Invoicer\Persistence\Doctrine\Repository\InvoiceRepository;
```

Listing Invoices

The `indexAction()` will look terribly familiar:

```
// app/Http/Controllers/InvoicesController.php

public function indexAction() {
  $invoices = $this->invoiceRepository->getAll();
  return view('invoices/index', ['invoices' => $invoices]);
}
```

As well as the view:

```
<!-- resources/views/invoices/index.blade.php -->

@extends('layouts.layout')

@section('content')
  <div class="page-header clearfix">
    <h2 class="pull-left">Invoices</h2>
    <a href="/invoices/new" class="btn btn-success pull-right">
      Generate Invoices</a>
  </div>

  <table class="table table-striped clearfix">
    <thead>
    <tr>
      <th>#</th>
      <th>Order Number</th>
      <th>Invoice Date</th>
      <th>Customer</th>
      <th>Description</th>
      <th class="text-right">Total</th>
    </tr>
    </thead>
    <?php foreach ($invoices as $invoice): ?>
    <tr>
      <td>
        <a href="/invoices/view/{{{ $invoice->getId() }}}">
          {{{ $invoice->getId() }}}</a>
      </td>
      <td>
        {{{ $invoice->getInvoiceDate()->format('m/d/Y') }}}
      </td>
      <td>{{{ $invoice->getOrder()->getOrderNumber() }}}</td>
      <td>
        <a href="/customers/edit/{{{ $invoice->getOrder()
            ->getCustomer()->getId() }}}">
          {{{ $invoice->getOrder()->getCustomer()->getName() }}}</a>
      </td>
      <td>{{{ $invoice->getOrder()->getDescription() }}}</td>
      <td class="text-right">
        $ {{{ number_format($invoice->getTotal(), 2) }}}
      </td>
    </tr>
    <?php endforeach; ?>
  </table>
@stop
```

I'm running out of things to say after these code snippets.

Generating Invoices

Our next step is generating new invoices. We'll start with the `/invoices/new` route which resolves to the `newAction()`:

```
// app/Http/Controllers/InvoicesController.php

public function newAction() {
  return view('invoices/new', [
    'orders' => $this->orderRepository->getUninvoicedOrders()
  ]);
}
```

This simple action just grabs all uninvoiced orders and supplies them to the view template:

```
<!-- resources/views/invoices/new.blade.php -->

@extends('layouts.layout')

@section('content')
<h2>Generate New Invoices</h2>

<p>
  The following orders are available to be invoiced.
</p>

<?php if (empty($orders)): ?>
<p class="alert alert-info">
  There are no orders available for invoice.
</p>
<?php else: ?>
<table class="table table-striped clearfix">
  <thead>
  <tr>
    <th>#</th>
    <th>Order Number</th>
    <th>Customer</th>
    <th>Description</th>
    <th class="text-right">Total</th>
  </tr>
  </thead>
  <?php foreach ($orders as $order): ?>
    <tr>
      <td>
        <a href="/orders/view/{{{ $order->getId() }}}">
            {{{ $order->getId() }}}</a>
      </td>
      <td>{{{ $order->getOrderNumber() }}}</td>
      <td>
        <a href="/customers/edit/{{{ $order->getCustomer()->getId() }}}">
            {{{ $order->getCustomer()->getName() }}}</a>
      </td>
      <td>{{{ $order->getDescription() }}}</td>
      <td class="text-right">
        $ {{{ number_format($order->getTotal(), 2) }}}
      </td>
    </tr>
  <?php endforeach; ?>
</table>

<form action="/invoices/generate" method="post" class="text-center">
```

```
  <input type="hidden" name="_token" value="<?= csrf_token(); ?>">

  <button type="submit" class="btn btn-primary">Generate Invoices</button>
</form>
<?php endif; ?>
@stop
```

The view shows the uninvoiced orders, if any, and provides a button to generate invoices for those orders.

So let's work that action:

```
// app/Http/Controllers/InvoicesController.php

public function generateAction() {
  $invoices = $this->invoicing->generateInvoices();

  $this->invoiceRepository->begin();

  foreach ($invoices as $invoice) {
    $this->invoiceRepository->persist($invoice);
  }

  $this->invoiceRepository->commit();

  return view('invoices/generate', ['invoices' => $invoices]);
}
```

This, like all the code in this chapter, is stolen directly from the ZF2 project, and modified slightly for Laravel. Let's finish off with the view, which shows a list of the generated invoices:

```
<!-- resources/views/invoices/generate.blade.php -->

@extends('layouts.layout')

@section('content')
<div class="page-header">
  <h2>Generated Invoices</h2>
</div>

<?php if (empty($invoices)): ?>
<p class="text-center">
  <em>No invoices were generated.</em>
</p>
<?php else: ?>
<table class="table table-striped clearfix">
```

```
  <thead>
    <tr>
      <th>#</th>
      <th>Order Number</th>
      <th>Invoice Date</th>
      <th>Customer</th>
      <th>Description</th>
      <th class="text-right">Total</th>
    </tr>
  </thead>
  <?php foreach ($invoices as $invoice): ?>
  <tr>
    <td>
      <a href="/invoices/view/{{{ $invoice->getId() }}}">
        {{{ $invoice->getId() }}}</a>
    </td>
    <td>
      {{{ $invoice->getInvoiceDate()->format('m/d/Y') }}}
    </td>
    <td>{{{ $invoice->getOrder()->getOrderNumber() }}}</td>
    <td>
      <a href="/customers/edit/{{{ $invoice->getOrder()
        ->getCustomer()->getId() }}}">
        {{{ $invoice->getOrder()->getCustomer()->getName() }}}</a>
    </td>
    <td>{{{ $invoice->getOrder()->getDescription() }}}</td>
    <td class="text-right">
      $ {{{ number_format($invoice->getTotal(), 2) }}}
    </td>
  </tr>
  <?php endforeach; ?>
</table>
<?php endif; ?>
@stop
```

And viola, invoice generation.

Viewing Invoices

The last stop on our Laravel journey is to view an individual invoice. Let’s start with the `viewAction()`:

```
// app/Http/Controllers/InvoicesController.php

public function viewAction($id) {
  $invoice = $this->invoiceRepository->getById($id);

  if (!$invoice) {
    return new Response('', 404);
  }

  return view('invoices/view', [
    'invoice' => $invoice,
    'order' => $invoice->getOrder()
  ]);
}
```

Let's make sure `Response` is part of the `use` statements:

```
use Illuminate\Http\Response;
```

And next, our view:

```
<!-- resources/views/invoices/view.blade.php -->

@extends('layouts.layout')

@section('content')
<div class="page-header clearfix">
  <h2>Invoice #{{{ $invoice->getId() }}}</h2>
</div>

<table class="table table-striped">
  <thead>
    <tr>
      <th colspan="2">Invoice Details</th>
    </tr>
  </thead>
  <tr>
    <th>Customer:</th>
    <td>
      <a href="/customers/edit/{{{ $order->getCustomer()->getId() }}}">
        {{{ $order->getCustomer()->getName() }}}</a>
    </td>
  </tr>
  <tr>
    <th>Order:</th>
    <td>
      <a href="/orders/view/{{{ $order->getId() }}}">
        {{{ $order->getOrderNumber() }}}</a>
    </td>
  </tr>
  <tr>
    <th>Description:</th>
    <td>{{{ $order->getDescription() }}}</td>
  </tr>
  <tr>
    <th>Total:</th>
    <td>$ {{{ number_format($invoice->getTotal(), 2) }}}</td>
  </tr>
</table>
@stop
```

And viewing invoices, and all invoice functionality, is complete.

git

This would make a good place to commit your code to source control.

If you're just reading, but want to see the code in action, you can checkout the tag 18-laravel-invoices:

```
git clone https://github.com/mrkrstphr/cleanphp-example.git
git checkout 18-laravel-invoices
```

Next Steps

If this were a real project, I'd recommend a few things:

1. Test this application layer. Test it through and through. These will likely be integration tests, or full system tests, as unit testing controllers in any framework, except maybe something like Silex, is near impossible.
2. Ditch the ZF2 components and use the Laravel counterparts. ZF2 isn't component based, no matter how hard they try to pretend, and we were forced to bring along the kitchen sink just to use two tiny little slivers of the framework.
3. Fully embrace Blade, or don't. We kind of went half-and-half in the examples. I'd opt for not using blade; it's just weird.

Summary

This was a lot of work. A lot of tedious work. If we were switching from ZF2 to Laravel in a real, large application, this would have been a lot more work.

Is this feasible? It definitely is, as we’ve seen here, but it’s a huge undertaking. One that most certainly will lead to bugs and issues not present in the old system â€“ especially since we didn’t write any tests for this layer. As testing this layer can be quite complicated, I left it out of this book. However, having a full suite of tests for each layer will aid greatly in detecting and fixing issues early in the process.

Switching frameworks is incredibly laborious. Do so sparingly, and spend a lot of time evaluating your framework choice up front, so that hopefully you’ll never have a need to switch. Further, write code intelligently and favor highly decoupled components. It will only make your application better in the long run.