

Laravel Testing Decoded

...with Jeffrey Way

Laravel Testing Decoded

The testing book you've been waiting for.

JeffreyWay

This book is for sale at <http://leanpub.com/laravel-testing-decoded>

This version was published on 2013-05-28



*

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

*

© 2013 JeffreyWay

TABLE OF CONTENTS

[Welcome](#)

[It Has Begun](#)

[Is This Book For Me?](#)

[Why Laravel-Specific?](#)

[Exercises](#)

[Errata](#)

[How to Consume This Book](#)

[Get in Touch](#)

[Into the Great Wide Open](#)

Chapter 1: Test All The Things

[You Already Test](#)

[6 Wins From TDD](#)

[1. Security](#)

[2. Contribution](#)

[3. Big-Boy Pants](#)

[4. Testability Improves Architecture](#)

[5. Documentation](#)

[6. It's Fun](#)

[What Should I Test?](#)

[6 Signs of Untestable Code](#)

[1. New Operators](#)

[2. Control-Freak Constructors](#)

[3. And, And, And](#)

[4 Ways to Spot a Class With Too Many Responsibilities](#)

[4. Too Many Paths? Polymorphism to the Rescue!](#)

[5. Too Many Dependencies](#)

[6. Too Many Bugs](#)

[Test Jargon](#)

[Unit Testing](#)

[Model Testing](#)

[Integration Testing](#)

[Functional \(Controller\) Testing](#)

[Acceptance Testing](#)

[Relax](#)

[Chapter 2: Introducing PHPUnit](#)

[Installation](#)

[Making Packages Available Globally](#)

[Assertions 101](#)

[Decoding A Test Class Structure](#)

[assertTrue](#)

[assertEquals](#)

[assertSame](#)

[assertContains](#)

[assertArrayHasKey](#)

[assertInternalType](#)

[assertInstanceOf](#)

[Asserting Exceptions](#)

[Summary](#)

[Chapter 3: Configuring PHPUnit](#)

[Options](#)

[Technicolor](#)

[Bootstrapping](#)

[Output Formats](#)

[XML Configuration File](#)

[Continuous Testing](#)

[Watching Files](#)

[Triggering Multiple Files](#)

[Some Vim-Specific Advice](#)

[Summary](#)

[Chapter 4: Making PHPUnit Less Verbose](#)

[Importing Assertions as Functions](#)

[Applying the Laravel Style to PHPUnit](#)

Chapter 5: Unit Testing 101

[My Struggles](#)

[Unit Testing](#)

[Arrange, Act, Assert](#)

[Testing in Isolation](#)

[Tests Should Not Be Order-Dependent](#)

[Test-Driven Development](#)

[Behavior-Driven Development](#)

[Testing Functions](#)

[Slime vs. Generalize](#)

[Slime](#)

[Generalize](#)

[Making the Test Pass](#)

[Testing Classes](#)

[Refactoring the Tests](#)

[Refactoring the Production Code](#)

[Polymorphism](#)

[Extensibility](#)

[Mocks](#)

[Project Complete](#)

[Final Source](#)

[Summary](#)

[Chapter 6: Testing Models](#)

[What to Test](#)

[Accessors and Mutators](#)

[Cat Years Example](#)

[Password Hashing Example](#)

[Custom Methods](#)

[Validations](#)

[Helpers](#)

[Factories](#)

[Laravel Test Helpers](#)

[Factories](#)

[Overrides](#)

[Models](#)

[Test Helpers](#)

[assertValid and assertNotValid](#)

[Asserting Relationships](#)

[Summary](#)

[Chapter 7: Easier Testing With Mockery](#)

[Mocking Decoded](#)

[Installation](#)

[The Dilemma](#)

[Dependency Injection](#)

[The Solution](#)

[Simple Mock Objects](#)

[Return Values From Mocked Methods](#)

[Expectations](#)

[Partial Mocks](#)

[Hamcrest](#)

[Summary](#)

[Chapter 8: Test Databases](#)

[Test Databases](#)

[Specifying the Environment](#)

[Calling Artisan From Tests](#)

[Try It Out](#)

[Databases in Memory](#)

[Summary](#)

Chapter 9: Just Swap That Thang

[Mockery](#)

[Testing](#)

[Mocking Events](#)

[Summary](#)

[Chapter 10: Testing Controllers](#)

[What Does a Controller Do?](#)

[3 Steps to Testing Controllers](#)

[The Hello World of Controller Testing](#)

[Overloading is Your Friend](#)

[Calling Controller Actions](#)

[Laravel's Helper Assertions](#)

[Mocking the Database](#)

[Required Refactoring](#)

[The IoC Container](#)

[Redirections](#)

[Paths](#)

[Repositories](#)

[Structure](#)

[Updating the Tests](#)

[Crawling the DOM](#)

[Ensure View Contains Text](#)

[Basic Traversing](#)

[Fetch By Position](#)

[Fetch First or Last](#)

[Fetch Siblings](#)

[Fetch Children](#)

[Capture Text Content](#)

[Forms](#)

[Summary](#)

[Chapter 11: The IoC Container](#)

[Dependency Injection?](#)

[Constructor Injection](#)

[Setter Injection](#)

[Resolving](#)

[Solution 1: Defaults](#)

[Solution 2: Resolving](#)

[App Bindings](#)

[Interfaces](#)

[Example 2: Automatic Resolution](#)

[Extra Credit](#)

[Summary](#)

[Chapter 12: Test-Driving Artisan Commands Exercise](#)

[Commands 101](#)

[Scaffolding](#)

[Arguments](#)

[Options](#)

[Single Responsibility Principle](#)

[Exercise](#)

[Create the Package](#)

[Generating the Command](#)

[Service Providers](#)

[Testing Artisan Commands](#)

[Planning](#)

[Expectations](#)

[Dependency Injection](#)

[Generator Class](#)

[Making the Test Pass](#)

[Testing the Model Generator](#)

[Summary](#)

Chapter 13: Testing APIs

APIs in Laravel

Three Components to Writing an API in Laravel

1. Authentication
2. Route Prefixing
3. Return JSON

Testing APIs

Use a Database in Memory

Migrate the Database for Each Test

Enable Filters

Set the Authenticated User

Test Examples

User Must Be Authenticated

Check For Error

Fetch All Photos For the Authenticated User

Refactoring

Updating a Photo

Factories

Specifying Options

Summary

[Chapter 14: Acceptance Testing With Codeception](#)

[An Amuse Bouche](#)

[Testing Refresher](#)

[Acceptance Testing](#)

[Installation](#)

[Global Installation](#)

[Local Installation](#)

[Bootstrapping](#)

[Configuring Acceptance Tests](#)

[Generate a Test](#)

[Manual Approach](#)

[Generator Approach](#)

[Decoding the Command](#)

[Writing the First Test](#)

[Running All Tests](#)

[Summary](#)

[Chapter 15: Authentication With Codeception Exercise](#)

[The Feature](#)

[Translating the Feature for Codeception](#)

[Register Routes](#)

[Building the Form](#)

[Resources](#)

[Authenticating the User](#)

[Adding a Test Database](#)

[Invalid Credentials](#)

[Summary](#)

[Chapter 16: Functional Testing in Codeception](#)

[The Laravel4 Module](#)

[The DB Module](#)

[Updating TestGuy](#)

[Registering a User](#)

[Summary](#)

[Chapter 17: Continuous Integration With Travis CI](#)

[Hello, Travis](#)

- [1. Connect](#)
- [2. Register Hooks](#)
- [3. Configure](#)

[Build Configuration](#)

[Absolute Basics](#)

[Bootstrapping](#)

[Register Dependencies](#)

[Notifications](#)

[Turn Off Notifications](#)

[Set Recipients](#)

[IRC](#)

[Summary](#)

Frequently Asked Questions (A Living Document)

[1. How Do I Test Global Functions?](#)

[Mock It](#)

[The Namespacing Trick](#)

[2. How Do I Create a SQL Dump for Codeception Tests?](#)

[3. How Do I Test Protected Methods?](#)

[4. Should I Test Getters and Setters?](#)

[5. Why Is PHPUnit Ignoring My Filters?](#)

[6. Can I Mock a Method in the Class I'm Testing?](#)

[Traditional Partial Mocks](#)

[Passive Partial Mocks](#)

Goodbye

Wait a Second...

Welcome

I've seen it way too many times. As your application grows, so does your sloppy, untested codebase. Before long, you begin to drown, as your ability to manually test the application becomes unrealistic, or even impossible! It's at these specific times, when you begin to realize the down-right necessity for testing. Sure, you might have read a TDD book in the past, but, like many things in life, we require real-life experience, before we suddenly - in that wonderful "aha moment" - get it.

The only problem is that testing can be a tricky thing. In fact, it's quite possible that your codebase, as it currently stands, is untestable! What you may not realize is that, while, yes, testing does help to ensure that your code works as expected, following this pattern will also make you a better developer. That messy, untestable spaghetti code that you might have snuck into your project in the past will never happen again. Trust me: as soon as you bring the question "*how could I test this*" to the forefront of every new piece of code, you'll, with a smile on your face, look back to your former self, and laugh at your crazy, cowboy ways.

Welcome to modern software development.



While the principles of testing (and TDD) are language-agnostic, when it comes to execution, there are a variety of tools and techniques at your finger tips. This book is as much an introduction to TDD, as it is a deep analysis of the Laravel way of testing applications.

It Has Begun

When it comes to programming languages, everyone has an opinion. And when the topic of conversation switches to PHP specifically, well, prepare yourself for the vitriol. Despite the fact that the language has matured significantly in the last five years, there are those who, like a father staring at his grown-up daughter, still can't help but see PHP as a baby.

The naysayers don't see version 5.5, or OOP, or modern frameworks like

Laravel, or Composer, or a growing emphasis on test-first development. No, what they see is that old sloppy PHP 4 code, and, worse, poorly made WordPress themes from 2008.

Is PHP as beautiful a language as Ruby? No. Is its API inconsistent from time to time? Definitely. Has its community lead the development world, in terms of innovation and software design? Certainly not. So the question is, why? Why does PHP dominate - to the point of 80% market share - when competing languages are admittedly more elegant? Well, maybe there's something else to its success. Maybe, the fact that you can create a file, echo *hello world*, and immediately see the output in a browser is far more powerful and user friendly than we give it credit for. Maybe its flexibility is a virtue, rather than a vice.

Since when did ease-of-use become something that others mocked?

Or, perhaps the simple truth is that PHP is not *the new hotness*. It's not overly sexy. It's not in beta. But, you know what? We get stuff done. Say as much as you want about PHP 4. While you're doing that, the rest of us will be building things with the latest that the language and surrounding ecosystem have to offer.

The time for hating PHP is over. The PHP renaissance has begun. We use modern object-oriented techniques, we share packages through Composer, we embrace version control and continuous integration, we evangelize modern frameworks, we believe in testing (*you soon will too*), we welcome newcomers (rather than lock the door), and we do it all with a smile.

The best part is that, as a Laravel user, you're at the forefront of this new modern movement! When I first joined Laravel's IRC channel, within minutes, somebody said "*Welcome to the family*." Nothing describes our community more beautifully than that. We're all in this together. This is the PHP community I love.

If you purchased this book, it sounds like you could use a bit of help in the testing department. In Laravel spirit, welcome to the family. Let's figure this out together.

Is This Book For Me?

The difficult thing about writing a technical book is determining where to draw

the line, in terms of which prerequisites are required before reading chapter one. As long as you have a basic understanding of the following technologies, please do continue!

- PHP 5.3
- Laravel 3 (preferably version 4)
- Composer

Why Laravel-Specific?

Sure, many of the techniques outlined in this book can be applied to any language or framework, however, in my experiences, it's best to take your first steps into this new world in as comfortable shoes as possible. Can you learn test-driven development from a Java book? Absolutely! Would it be easier through the lens of the language and framework that you already know? Certainly.

Secondly, the downside to a generic testing book is that I wouldn't be able to demonstrate many of the PHP-specific features and packages that I use in my every day coding. This includes everything from PHPUnit helper packages, to acceptance testing frameworks, like Codeception.

Finally, it's my hope that, as you read through this book, in addition to improving your testing knowledge, you'll also pick up a variety of Laravel-specific tips and tricks.

Exercises

Sporadically throughout this book, *Exercise* chapters will be provided. Think of these as highly in depth tutorials that you are encouraged to work through, as you read the chapter. Theory can only take us so far; it's the actual coding that ultimately commits these patterns and techniques to memory.

So, when you come to an *Exercise* chapter, pull out the computer and join me through each step!

Errata

Please note that, while I've made every attempt to ensure that this book is free of errors and typos, my *human-ness* virtually guarantees that some will sneak in! If you notice any mistakes, please [file an issue on GitHub](#), and I'll update the book as soon as possible. As a reward, a five minute hug will be granted to each bug

filer.

How to Consume This Book

While you can certainly read this book from cover to cover, feel free to flip around to the chapters that interest you most. Each chapter is self-containing. For instance, if you already understand the basics of unit testing, then you clearly don't need to read the obligatory "Unit Testing 101" chapter! Skip it, and move on to the more interesting bits and pieces.

Get in Touch

It sounds like we're going to be spending a lot of time together, as you work your way through this book. If you'd like to attach a face to the author, and, perhaps, ask some questions along the way, be sure to say hello.

- **IRC (#laravel channel):** JeffreyWay
- **Twitter:** [@jeffrey_way](https://twitter.com/@jeffrey_way)

Into the Great Wide Open

The night was sultry. Wait, no, that's the wrong book. The night was silent, save for the strangely comforting repetition of my ceiling fan blades. My wife and animals had long since abandoned me, in favor of sleep. The dog, as he usually does, held out the longest, but I certainly couldn't blame him; who was I to complain that they weren't awake at three in the morning? No, the setting was just right. I could feel it. As I continued to direct every inch of my mind toward my laptop screen, things were beginning to...click.

Developers know this feeling well: those all too rare moments, when, suddenly, what was once impossible to understand, now, at least slightly, makes sense. We refer to these as "*aha*" moments. The first time that I fully understood what purpose a `<div>` serves was one such moment. Sure, it may sound obvious to you now, but think back to the early days. What the heck does wrapping this HTML in a `<div>` do? The output in the browser looks exactly the same! One day, I was told to think of them as buckets; place your HTML within these buckets, and, then, when you need to move things around, you only need to reposition the `<div>`. Like the snap of a finger, I understood.

I could recite dozens of unique moments like this one, including my slow appreciation for object-oriented programming, coding to an interface, and test-driven development.

Yes, my love for testing was not an immediate thing, I'm sorry to say. Similar to most developers, I'd read an article or book on the subject, think to myself, "Hey, that's interesting," and then continue on my existing path. Regardless of whether I knew it or not, though, the seeds had been planted. As time moved on, that gradual nudging in the back of my mind incrementally grew stronger and louder.

"You should be testing, Jeffrey." "If you had written tests for this, you wouldn't be manually testing this over and over." "They're going to laugh at you, if you don't offer tests for this pull request."

Like most things in life, true change requires us to plant our feet in the sand, and yell, “*No more! I’m done with the old way.*” I did it. Thousands of developers have, as well. Now, it’s your turn.

As [Leeroy Jenkins](#) might say, all right: let’s do this! Here’s the testing book you’ve been waiting for.

Chapter 1: Test All The Things

Every testing book in existence offers the obligatory “*Why Test*” chapter. If you think about it, the simple fact that you purchased this book hints that you’re already sold on the concept. Having said that, it can be beneficial to learn why others advocate it as religiously as they do.

Learning how to properly test your applications requires, unfortunately, a fairly steep learning curve. That may be surprising; it certainly was for me! The basic principle is laughably simple: write tests to prove that your code works as expected.

Find yourself continuously reaching for Google Chrome to test a particular piece of functionality? Close it, and instead write a test.

How could that possibly be confusing? Well, things quickly become tricky when you begin researching *what* to test.

Do I test controllers? What about models? Do views really need tests? What about my framework’s code, or hitting the database, or fetching information from web services? And what about the dozen different kinds of testing that I hear folks on StackOverflow referring to? What about test frameworks? PHPUnit? Rspec? Capybara? Codeception? Mink? The list goes on and on. Where do I start?

I’m not a betting man (*well, I partially am, but mostly on Scrabble games with my wife, where I get to [embarrass her in public](#) if I win*), but there’s little doubt in my mind that, at some point in your career (if not right now), you’ve found yourself asking these very questions. Testing is easy. Understanding what and how to test is another story. Hopefully, this book will help.

You Already Test

The truth is that you’re already a master of testing. If you’ve ever written

`console.log` or submitted a form in your web app to test a particular piece of functionality, then you were testing. Even as babies, we were expert testers. “*If I twist this knob, then the door opens. Success!*”

The only problem is that those tests were performed manually. Why do a job that a computer can handle for you (and much quicker, too)? Our goal, in this book, is to transition from manually testing every piece of functionality, to automating the entire process. This allows for a continuous testing cycle, where one of your test suites is triggered repeatedly as you develop your applications. You’ll be amazed by the level of security that this can provide.

“When developers first discover the wonders of test-driven development, it’s like gaining entrance to a new and better world with less stress and insecurity.” - DHH

6 Wins From TDD

Testing is a deceptive thing. Initially, you might think that the only purpose for it is to ensure that your code works as expected. But, you’d be wrong. In fact, there are multiple advantages to following a test-driven development cycle.

1. Security

Should you accidentally make a mistake or break a piece of existing functionality, the *test robots* will notify you right away. Imagine making an edit, clicking save, and immediately receiving feedback on whether you screwed up. How much better would you sleep at night? Remember that terribly coded class you were too afraid to refactor, because you might break the code? If tests were backing up that code, your fear would have been unwarranted.

2. Contribution

As you begin developing open source software, you’ll likely leverage the convenience and power of social coding through GitHub. Eventually (one of the perks), other members of the community will begin contributing to your projects when they encounter bugs or hope to implement new functionality. However, if your project doesn’t contain a test suite, when developers submit pull requests, how could you (or they) possibly determine if their changes have broken the code? The answer? You can’t - not without manually testing every possible path through the code. Who has the time to do that for every pull request?

Think of a highly tested project as a well-oiled machine. If I want to contribute to your project, I only need to follow a handful of steps:

1. Clone the repository
2. Write a test that describes the bug
(`testThrowsExceptionIfUserNameDoesNotExist`)
3. Make the necessary changes to fix it
4. Run the tests to ensure that everything returns green (*success*)
5. Commit the changes, and submit my pull request

There are even continuous integration services, like [Travis](#), which will automatically trigger a project's tests when a pull request is submitted. If those tests fail, I immediately know that it shouldn't be merged without further tweaking.

The screenshot shows the Travis CI web interface. At the top, there's a search bar with 'laravel' typed into it. Below the search bar, there are tabs for 'Recent' and 'Search'. A list of recent builds is displayed, with the first item being 'laravel/framework 1608' (Duration: 9 min 6 sec, Finished: a...). To the right of this list is a detailed view for the 'laravel/framework' project. The project name is 'laravel/framework' with a GitHub icon. Below the project name are tabs for 'Current', 'Build History', 'Pull Requests', and 'Branch Summary'. The 'Current' tab is selected, showing build details: Build 1608 (Passed), Commit 9914bf0 (master), State Passed, Compare b1bb225cb6d4...9914bf01d2e4, Finished about 2 hours ago, Author Taylor Otwell, Duration 9 min 6 sec, Committer Taylor Otwell, Message Swap dev queue back to iron, Config Php: 5.3, 5.4. Below this is a 'Build Matrix' section with columns for Job, Duration, Finished, and Php. It lists two jobs: 1608.1 (Duration 4 min 56 sec, Finished about 2 hours ago, Php 5.3) and 1608.2 (Duration 4 min 10 sec, Finished about 2 hours ago, Php 5.4).

travis-ci.org

3. Big-Boy Pants

If I may go on a tangent for a moment, when it comes to the PHP community, my view is that WordPress has been a double-edged sword. On one hand, it brought blogging to the masses. This is undeniable, and must be respected. It

also provided an easy-to-use theming framework for developers. Create an `index.php` file, insert a loop to fetch the recent posts, and then style. What could be easier than that?

Well, that's true. However, it also inadvertently nurtured a community of PHP developers who hesitated to reach beyond WordPress for new projects. Consequently, modern practices and patterns, such as test-driven development, MVC, and version control, are largely foreign to them. This unfortunate truth has had two side-effects:

1. Much of the vitriol directed toward the PHP community is the result of PHP 4 and WordPress code.
2. Making the leap from WordPress to a full-stack framework, like Laravel, can prove incredibly difficult. Due to the number of new tools and patterns, the learning curve can be quite steep.

Is WordPress responsible for these side-effects? Yes, and no. One thing's for sure, though: it certainly hasn't pushed the boundaries of software craftsmanship. In fact, testing is ignored for 95% (*made up number*) of the available WordPress plugins.

Eventually, though, we all learn to put on our big-boy pants. We refer to that old-fashioned practice of *coding without thinking* as being a cowboy. Don't plan, don't think, don't test; just start coding, *guns a blazing*, while frantically refreshing the browser to determine if each change has broken the application.

We're better than that. We're developers. Let's not be cowboys.

An interesting transition takes place, when you force yourself to think before coding: it actually improves the quality of the code. Who would have thought? What you'll soon learn is that there's more to testing than simply verifying that a method performs as expected. When we test, we interact with the class or API before it has been written. This forces us to remove all constraints and instead focus on readability. *What would be the most readable way to fetch data from this web service?* Write it as such, watch it fail, and then make it work. It's a beautiful thing!

4. Testability Improves Architecture

One thing that you'll learn throughout this book is to bring the question, *How might I test this?* to the forefront of every new piece of code. This simple question will be your security blanket, keeping you safe from repeating the wreckless, jumbled code of your past. No longer will you get away with, out of laziness, making a method perform too many actions. Doing so isn't testable. Tests encourage structure by making you design before coding.

5. Documentation

A huge, huge bonus to writing tests is that they provide free documentation for the system. Want to know what functionality a particular class offers? Poke around the tests, and, if they were named properly (meaning that they describe the behavior of the SUT, or system under test), then you'll have a full understanding in no time!

6. It's Fun

Let's face it: we're geeks. And what geek doesn't enjoy a good game? A fun side-effect to test-driven development is that it turns your job into a game. How can I take this code from red to green? Follow each step until you get there. It may sound silly at first, but I promise you: it's fun. You'll see for yourself soon enough.

What Should I Test?

The basic rule - one that I will repeat multiple times throughout this book - is to test anything that has the potential to break. Is it possible that one of your routes could *break*, leading to a 404 page? Yes? Then write a test to ensure that you catch any breaks as quickly as possible. What about your custom class that fetches some data from a table and writes it to a file, as a report? Should you test that? Sure. What about a utility that fetches your latest tweets and displays them in the sidebar of your website? If the only alternative is to open Google Chrome and check the sidebar, then, most certainly, the answer is yes.

The downside, at least initially, is that this *test all the things* mantra can quickly become overwhelming. With such a steep learning curve, it's okay if you take one step at a time. Learn how to test your models. Once you become comfortable with the process, then move on. Baby steps is the way to go. There's a reason why we begin by counting on our fingers.

One Note of Caution:

This tip is only partially accurate. Testing anything and everything that has the potential to break is a noble goal, but there *is* such a thing as over-testing (though this is debated heavily). A growing sector in the community would argue that it's best to limit your tests to the areas of your code in which they're most beneficial. In effect, if you rarely make mistakes when writing, say, accessors and mutators, then don't bother writing those tests. Tests are meant to serve you; it's not the other way around.

Your job, reader, is to decide for yourself where this line is drawn for your own applications. Or, in other words, when it comes to testing, where is the point of [diminishing returns](#)?

6 Signs of Untestable Code

Learning how to test code is a bit like moving to a country where no one speaks your language. Eventually, the more you push through, you begin to recognize certain patterns. Before long, you find yourself speaking fluently. It's not rocket science that we're working with here; anyone can learn this stuff. All it takes is pressure...and time (*Use Morgan's Freeman voice when reading that last line*).

As your testing chops improve, you'll begin to instantly recognize coding pitfalls. Like instinct, you'll find yourself silently scanning a piece of code, making note of each anti-pattern.

Here are five easy things to look out for:

1. New Operators

The principles of unit testing dictate that we should test in isolation. We'll cover this concept more in future chapters, but, in short, your goal should be to test the current class, and nothing else. Don't access the database, don't test that your `Filesystem` class fetches some data from a web service. Those should have their own tests, so don't double up.

Once you begin littering the new operator throughout your classes, you break this rule. Remember: testing in isolation requires that the class, itself, does not instantiate other objects.

Anti-pattern:

```
1 public function fetch($url)
2 {
```

```

3     // We can't test this!
4     $file = new Filesystem;
5
6     return $this->data = $file->get($url);
7 }
```

This is one of those situations where PHP isn't quite as flexible as we might hope. While languages like Ruby offer the ability to re-open a class (known as monkey-patching) and override methods (*particularly helpful for testing*), PHP, unfortunately, does not - *at least, not without recompiling PHP with special extensions*. As such, we must make use of dependency injection religiously.

Better:

```

1 protected $file;
2
3 public function __construct(Filesystem $file)
4 {
5     $this->file = $file;
6 }
7
8 public function fetch($url)
9 {
10    return $this->data = $this->file->get($url);
11 }
```

With this modification, a mocked version of the `Filesystem` class can be injected, allowing for complete testability. Don't worry if the code below is foreign to you. You'll learn the inner workings soon! For now, simply try to soak it in.

```

1 public function testFetchesData()
2 {
3     $file = Mockery::mock('Filesystem');
4     $file->shouldReceive('get')->once()->andReturn('foo');
5
6     $someClass = new SomeClass($file);
7     $data = $someClass->fetch('http://example.com');
```

```
8
9     $this->assertEquals('foo', $data);
10 }
```

The only time when it's acceptable to instantiate a class inside of another class is when that object is what we refer to as a value-object, or a simple container with getters and setters that doesn't do any real work.



Tip: Hunt down the new keyword in your classes like a hawk. They're code smells in PHP (at least for 90% of the cases)!

2. Control-Freak Constructors

A constructor's only responsibility should be to assign dependencies. Think of this as your class *asking* for things. *Can I have the Filesystem class, please?*

If you're doing anything beyond that, consider refactoring.

Anti-pattern:

```
1 public function __construct(Filesystem $file, Cache $cache)
2 {
3     $this->file = $file;
4     $this->cache = $cache;
5
6     $data = $this->file->get('http://example.com');
7     $this->write($data);
8 }
```

Better:

```
1 public function __construct(Filesystem $file, Cache $cache)
2 {
3     $this->file = $file;
4     $this->cache = $cache;
5 }
```

The reason why we do this is because when testing you'll repeatedly follow the

~~THE REASON WHY WE DO THIS IS BECAUSE, WHEN TESTING, YOU WILL REPEATEDLY FOLLOW THE SAME PROCESS:~~

1. Arrange
2. Act
3. Assert

If a class' constructor is littered with its own actions and method calls, each test you write must account for these actions.



Tip: Keep it simple: limit your constructors to dependency assignments.

3. And, And, And

Calculating what responsibility a class should have can be a difficult thing at first. Sure, we hear and understand *The Single Responsibility Principle*, but putting that knowledge into practice can be tough at first.

4 Ways to Spot a Class With Too Many Responsibilities

1. The simplest way to determine if your class is doing too much is to speak aloud what the class does. If you find yourself using the word, *and*, too often, then, chances are, refactoring is in order.
2. Train yourself to immediately analyze the number of lines in each method. Ideally, a methods should be limited to just a few (one is preferable, even). If, on the other hand, every method is dozens of lines long, this a clear indication that too much is going on.
3. If you're having trouble choosing a name for a class, this, too, just might be a sign that you've gone off track, and need to restructure.
4. If all else fails, show the class to one of your developer friends. If they don't immediately realize the general purpose of the class (*oh, this class handles the hashing of passwords*), then make some changes.

Here are some examples to get you started:

- A `FileLogger` class is responsible for logging data to a file.
- A `TwitterStream` class fetches and returns tweets from the Twitter API, when given a username.
- A `Validator` class is responsible for validating data against a set of rules.

- A `SQLBuilder` builds a SQL query, given a set of data.
- A `UserAuthenticator` class determines if the provided login credentials are correct.

Notice how, in none of the examples above did the word, *and*, occur. This makes them considerably easier to test, as you're not forced to juggle multiple objects.



Tip: Reduce each class to being responsible for one thing. This is referred to as *The Single Responsibility Principle*.

4. Too Many Paths? Polymorphism to the Rescue!

Truly understanding what polymorphism is requires an [aha moment](#). But, like many things in life, once you understand it, you'll never forget it.



Definition: Polymorphism refers to the act of breaking a complex class into sub-classes which share a common interface, but can have unique functionality.

The easiest possible way to determine if a class could benefit from polymorphism is to hunt down switch statements (or too many repeated conditionals). Imagine a bank account class that should calculate yearly interest differently, based on whether the type of account is checking, savings, or some other type entirely. Here's an incredibly simplified example:

```
1 function addYearlyInterest($balance)
2 {
3     switch ($this->accountType) {
4         case 'checking':
5             $rate = $this->getCheckingInterestRate();
6             break;
7
8         case 'savings':
9             $rate = $this->getSavingsInterestRate();
10            break;
11
12        // other types of accounts here
```

```
13     }
14
15     return $balance + ($balance * $rate);
16 }
```

In situations such as this, a smarter course of action is to extract this similar, but unique logic to sub-classes.

Define an interface to ensure that you have access to a `getRate` method.

```
1 interface BankInterestInterface {
2     public function getRate();
3 }
```

Create a checking implementation of the interface.

```
1 class CheckingInterest implements BankInterestInterface {
2     public function getRate()
3     {
4         return .01;
5     }
6 }
```

Create a savings implementation of the interface.

```
1 class SavingsInterest implements BankInterestInterface {
2     public function getRate()
3     {
4         return .03;
5     }
6 }
```

Now, the original method can be cleaned up considerably. Notice how we've type-hinted the `$interest` variable below. This provides us with some protection, as we don't want to fall into a trap of calling a `getRate` method on a class, if it doesn't exist. This is precisely why we've coded to an interface. By implementing the interface, the class is forced to offer a `getRate` method.

```

1 function addYearlyInterest($balance, BankInterestInterface
2 $interest)
3 {
4     $rate = $interest->getRate();
5
6     return $balance + ($balance * $rate);
7 }
8
9 $bank = new BankAccount;
10 $bank->addYearlyInterest(100, new CheckingInterest); // 101
11 $bank->addYearlyInterest(100, new SavingsInterest); // 103

```

Even better, testing this code is a cinch, now that we no longer need to account for multiple paths through the function. Again, don't sweat over the syntax. We'll cover it soon enough.

```

1 public function testAddYearlyInterest()
2 {
3     $interest = Mockery::mock('BankInterestInterface');
4     $interest->shouldReceive('getRate')->once()->andReturn(.03);
5
6     $bank = new BankAccount;
7     $newBalance = $bank->addYearlyInterest(100, $interest);
8
9     $this->assertEquals(103, $newBalance);
10 }

```



Tip: Polymorphism allows you to split complex classes into small chunks, often referred to as *sub-classes*. Remember: the smaller the class, the easier it is to test.

5. Too Many Dependencies

In the *Control Freak Constructor* tip, I noted that dependencies should be injected through the class' constructor. Having said that, if you find that a particular class requires four or more dependencies, this is, more often than not, a tell-tale sign that your class is asking for too much.

“I usually re-evaluate any classes with more than four [dependencies].” -
[Taylor Otwell](#)

A basic principle of object-oriented programming is that there's a correlation between the number of parameters a class or method accepts, and the degree to which it is flexible (and, in effect testable). Each time that you remove a dependency or parameter, you're improving the code.

If one of your classes lists too many dependencies, consider refactoring.

6. Too Many Bugs

I once heard Ben Orenstein remark that *bugs love company*. Boy, is this a true statement if I've ever heard one. If you notice that they crop up in a particular class too frequently, then the code just might be screaming for refactoring and sub-classes. Think about it: the reason for the bug's existence is because you couldn't understand the code well enough the first time around; it was too complicated! And guess what? Complicated code often signals untestable code. The coupling may simply be too strong, opening the way for sneaky bugs...and all their pals, too.



Definition: Coupling refers to the degree in which two components in your system are dependent upon one another. If removing one affects the other, then you've unfortunately written tightly coupled code that isn't easy to change.

As Ben beautifully put it, if there was a bug on line seven, then, chances are, there's also a bug on line eleven. Nip that in the bud as early as possible.



Tip: When presented with such bugs, begin asking yourself how you can split the logic up into smaller (easier to test) classes. In addition to improved testability, one perk to this pattern is that it allows for significantly more readable production code.

Bugs love company. Pull out the bug spray.

Test Jargon

I'm not so self-consumed to think that this book will serve as your sole source of testing education (I certainly hope it isn't). If you're anything like myself you'll

find yourself scouring the web late at night for every last fragment of education to fill in those missing pieces in your understanding.

In the process, you'll come across incredibly confusing jargon. Worse, this terminology is inconsistent from language to language! Yikes! In this book - and in the spirit of simplicity - we'll break things down into their simplest terms. Scan the following definitions, but don't feel that you must commit them to memory all at once. In truth, in many ways, I'm very much against all this confusing jargon. If a term doesn't immediately make sense, then it should be changed. The development community should take its cues from astrophysics.

“The most accessible field in science, from the point of view of language, is astrophysics. What do you call spots on the sun? Sunspots. Regions of space you fall into and you don’t come out of? Black holes. Big red stars? Red giants. So I take my fellow scientists to task. He’ll use his word, and if I understand it, I’ll say, “Oh, does that mean da-da-da-de-da?” - Neil Degrasse Tyson

Unit Testing

Think of unit testing as going over your classes and methods with a fine-tooth comb, ensuring that each piece of code works exactly like you expect. Unit tests should be executed in isolation, to make the process of debugging as easy as possible. 80% of your tests will be in this style.

If it helps, when you think of unit testing, think *one object, and one object only*. If a test fails, you know exactly where to look.

Model Testing

Some members of the Ruby on Rails community associate model testing (even when these tests touch the database) with unit testing. This unfortunately can be a bit misleading. Unit tests should be isolated from all external dependencies. Once you ignore this basic rule, you're no longer unit testing. You're writing integration tests (more on that shortly).

In this book, when we test our models, we'll stay true to the traditional definition of unit testing, unless specified otherwise.

Imagine that a method in your model is responsible for sending an email. If following good design patterns, you'll likely have a class that is dedicated to sending email (*single responsibility principle*). This presents a problem, however: how do we successfully unit test this method, if it calls an external Mailer class? The answer is to use mocks, which we'll cover extensively in this book. A mock allows us to fake the Mailer class, and write an expectation to ensure that the proper method is called. This way, even if the Mailer component is currently broken (*it will have its own tests*), we can still verify whether or not the model method is working as expected.

Integration Testing

If a unit test verifies that code works correctly in isolation, then an integration test will fall on the other end of the spectrum. These tests will flex multiple parts of your application, and typically won't rely on mocks or stubs. As such, be sure to create a special test database.

As an example, think of a car. Sure, the engine and fuel injection system might individually work as expected (each passes its own set of unit tests), but will they work when grouped together? Integration testing verifies this.

Functional (Controller) Testing

What do you call the process of testing a controller? Some frameworks may refer to this as functional testing (and it is), but we'll simply stick with - wait for it - *controller testing!*

More traditionally, think of functional testing as a way for you and your team to ensure that the code does what you expect. While unit tests verify each *unit* of a class, functional testing is a bit broader and can trigger multiple pieces of your application. Most frequently, these tests will be triggered from the outside in, which is why they're also commonly referred to as *System Tests*. One important note is that functional tests typically won't require a server to be running.

Acceptance Testing

You've already learned that functional testing ensures that the code meets the requirements of the development team. However, there will be cases when, even though the tests return green, the final implemented feature will not meet the requirements of the client. This is what we refer to as acceptance testing. In other words, does this code meet the requirements of the client? Your software can

pass all unit, functional, and integration tests, but still fail the acceptance tests, if the client or customer realizes that the feature doesn't work as they expected.



Tip: If functional tests meet a developer's assumptions and requirements, acceptance tests are intended to verify the client's expectations.

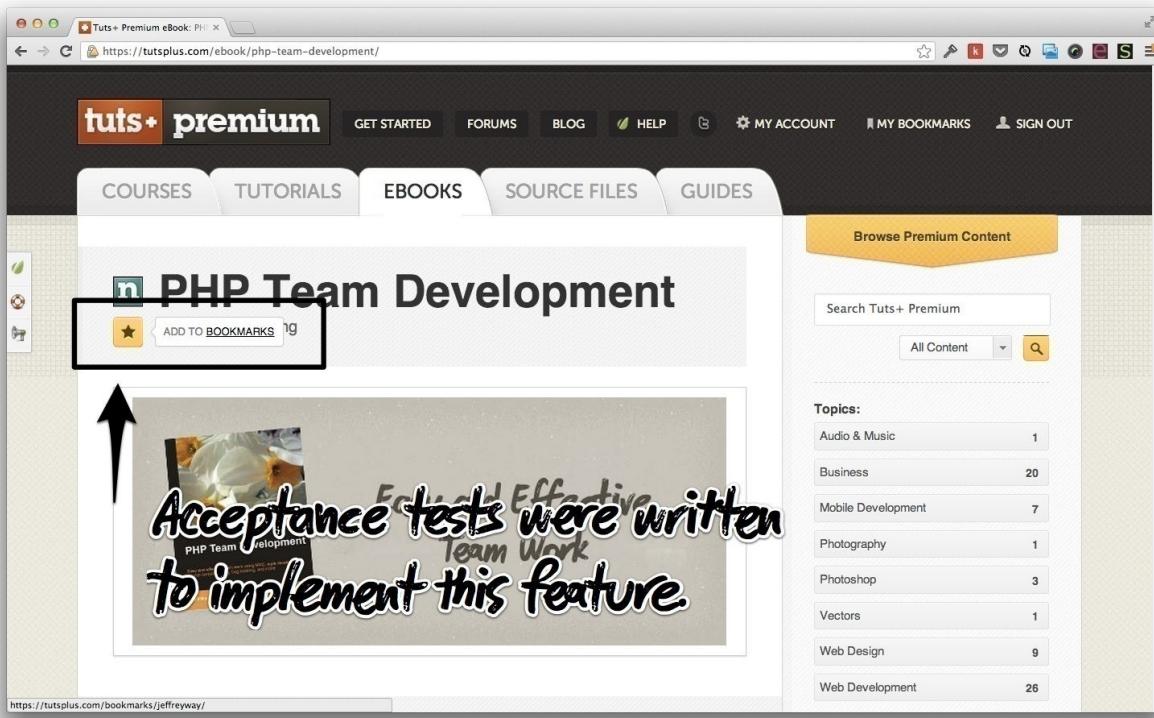
Think of [Tuts+ Premium](#), the subscription-based technical education service that I work for. Recently, we added a new bookmarking feature that allows you to save courses and eBooks that you want to read later. Before the developers can begin writing a single line of code, they first need to understand what the content team's expectations are - they're the ones requesting the feature. This requires an acceptance test.

¹ In order to keep track of what to learn next

² As a member

³ I want to bookmark content

Once this acceptance test passes, it may be assumed that the feature has fully been implemented, and meets the client's (the content team, in this case) expectations.



The bookmarking feature on Tuts+ Premium required acceptance tests.

In the final section of this book, you'll learn how to write acceptance tests using the Codeception framework. You'll find that that this allows us to use *human speak* to define how we want to interact with our applications.

Are Tests Too Expensive?

A common testing myth is that, though they might be beneficial, when it comes to the real world, a client's budget factors into the equation. As they put it, no client is willing to double your budget for the sole purpose of providing you with more security.

Is there merit to this argument? No, no there's not. In fact, plenty of studies have found that a test-driven development cycle reduces the length of time it takes to complete a project.

Relax

I'll be the first one to tell you that these definitions took me a very long time to learn and appreciate. I certainly don't expect this to stick after the first reading. Sheesh; we haven't even gotten to the "*Intro to PHPUnit*" chapter! For now, simply keep in mind that, as you develop applications, you'll make use of

multiple styles of testing.

Having said that, the unfortunate truth is that the development community, as a whole, can't seem to agree on terminology to save their lives. You'll also come across terms, like system testing, request specs, medium tests, and more. In most cases, there's close overlap between these terms and the ones referenced earlier. Don't worry about this too much; the most important thing is to get you testing. You'll develop your own style in time.

As the saying goes, it doesn't matter how you test...just as long as you do test.

The dissonance continues beyond terminology. While it's fair to say that most developers these days agree that writing tests is vital, in what order those tests are written is a different story. Some evangelists, like Bob Martin (Uncle Bob), recommend strict adherence to the TDD philosophy: do not write a single line of production code until you've first written a test.

“It has become infeasible for a software developer to consider himself professional if he does not practice test-driven development.” - [Bob Martin](#)

But, other equally influential developers, like DHH (creator of Ruby on Rails), freely admit that they write the tests after the production code - roughly 80% of the time.

“Don’t force yourself to test-first every controller, model, and view (my ratio is typically 20% test-first, 80% test-after).” - [David Heinemeier Hansson](#)

It's your job to take in all of the input and advice around the web, and mold that into a style that you (or your development team) can embrace. As such, view this book less as a Bible, and more as one person's adaptation of testing, that you can then morph to your style. *There is no spoon.*

Chapter 2: Introducing PHPUnit

I get it: you're a cowboy and want to dive into testing your Laravel applications right now. Whoa, hold on there, pony. Before we can dig into Laravel 4's excellent testing facilities, we first need to install a required prerequisite: PHPUnit.



[PHPUnit](#) is a unit testing framework, created by Sebastian Bergmann. Similar to other frameworks, PHPUnit uses assertions to verify that the code under test (or SUT, for *system under test*) behaves as expected.

Installation

At the time of this writing, there are a few different ways to install PHPUnit on your system, including:

- **Pear** - No one on the planet has ever enjoyed Pear.
- **Composer** - Installs PHPUnit on a per-project basis.
- **PHAR** - Think of this one as a one-click archive.

You're certainly free to choose the method you wish, however, as we'll heavily be leveraging Laravel and Composer in this book, we'll take the simplest route, and add PHPUnit as a development dependency to the `composer.json` file of our Laravel application.



Wait, what's this [Composer](#) thing? It's the PHP community's preferred tool for dependency management. It provides an easy way to declare a project's dependencies, and pull them in with a single command. As a Laravel 4 developer, it's vital that, before moving forward, you have a basic understanding of what Composer is, and how to use it.

Before we dive into a Laravel 4 project, let's begin with the absolute basics. When learning new technologies, libraries, or frameworks, always strip the examples down to the bare essentials. This is the best way to learn as quickly as possible.

Create a new folder, called `lesson`. Within it, add a new `composer.json` file, and append the following:

```
1  {
2      "require-dev": {
3          "phpunit/phpunit": "3.7.*"
4      }
5 }
```

Notice how we're not adding the dependency to the `require` object. As PHPUnit is only necessary during development, we don't need to worry about the production server pulling in this package.

The next step is to install it and any other dependencies that you might have registered in your application. From the command line, assuming that Composer is installed globally on your system, run:

```
1 composer install --dev
```

Give that command a moment to process, and, if all goes according to plan, you should now have access to both `vendor/bin/phpunit` and `vendor/phpunit/`. Go ahead and try it out by viewing a list of PHPUnit's commands. From the root of your project, run:

```
1 vendor/bin/phpunit -h
```

```
● laravel-app vendor/bin/phpunit -h
PHPUnit 3.7.18 by Sebastian Bergmann.

Usage: phpunit [switches] UnitTest [UnitTest.php]
       phpunit [switches] <directory>

--log-junit <file>      Log test execution in JUnit XML format to file.
--log-tap <file>        Log test execution in TAP format to file.
--log-json <file>        Log test execution in JSON format.

--coverage-clover <file> Generate code coverage report in Clover XML format.
--coverage-html <dir>    Generate code coverage report in HTML format.
--coverage-php <file>    Serialize PHP_CodeCoverage object to file.
--coverage-text=<file>  Generate code coverage report in text format.
                         Default to writing to the standard output.

--testdox-html <file>   Write agile documentation in HTML format to file.
--testdox-text <file>   Write agile documentation in Text format to file.

--filter <pattern>     Filter which tests to run.
--testsuite <pattern>  Filter which testsuite to run.
--group ...            Only runs tests from the specified group(s).
--exclude-group ...   Exclude tests from the specified group(s).
--list-groups          List available test groups.
--test-suffix ...     Only search for test in files with specified
                     suffix(es). Default: Test.php,.phpt

--loader <loader>     TestSuiteLoader implementation to use.
```

If that output feels overwhelming, don't worry. Like Git, a basic understanding of these various commands can carry you a long way.



Tip: You certainly don't want to type `vendor/bin/phpunit` every time you need to run your tests. Add `vendor/bin` to your path, so that you can simply type `phpunit`.

Don't forget that aliases are your friend. To call PHPUnit with a single key, on a Mac, try: `alias t="vendor/bin/phpunit"`. This is a temporary alias, but can easily be added to `~/.bash_profile`.

Making Packages Available Globally

The method outlined in the previous section requires PHPUnit to be installed through Composer for each new project. If you prefer, you can alternatively create a *master* composer project for packages that should be available globally. To allow for this, within your computer's *home* directory (or anywhere, really), create a new folder, `composer-packages`, add its companion `composer.json` file, and list your desired global dependencies within the `require-dev` object.

```
1 {
2     "require": {
3         "phpunit/phpunit": "3.7.*"
4     }
5 }
```

Next, add the full path to the project's vendor/bin directory to your path. Here's an example for my Unix-based system. This can be added to your `~/.bash_profile` or `~/.zshrc`, if you use [oh-my-zsh](#).

```
1 export PATH=/Users/Jeffrey/composer-packages/vendor/bin
```

With this technique, PHPUnit will always be available. Even better, updating these *global* packages only requires a simple `composer update`.



While installing PHPUnit directly through Composer is certainly a valid route - and will do just fine for the purposes of this book - some developers prefer to stick with Pear (*I have no idea why*). If you fit that description, feel free to do so. It makes zero difference.

Assertions 101

It's time for the "*hello world*" of testing. In this section, you'll be introduced to your first PHPUnit assertion, `assertTrue`.

Add a new file, called `PracticeTest.php`, to a `tests` folder within your project. Here's some boilerplate to get you started.

```
1 <?php // lesson/tests/PracticeTest.php
2
3 class PracticeTest extends PHPUnit_Framework_TestCase {
4     public function testHelloWorld()
5     {
6         $greeting = 'Hello, World.';
7
8         $this->assertTrue($greeting === 'Hello, World.');
9     }
10 }
```

Before we decode it, run the test to ensure that it works as expected.

```
1 $ phpunit tests
2 PHPUnit 3.7.18 by Sebastian Bergmann.
3
4 .
5
6 Time: 0 seconds, Memory: 2.50Mb
7
8 OK (1 test, 1 assertion)
```

Because we have not yet applied any configuration options, we must specify a path to where the tests for our project are located. PHPUnit will tunnel through this folder recursively, searching for test files. This means that you're free to organize the tests folder structure however you wish. For now, though, keep things simple, and store all test files in the same space.

First, notice the final line, `OK (1 test, 1 assertion)`. This, to state the obvious, reveals that one test passed successfully. Did you notice the single period in the output? That's the representation of a single test. When you add a second test, then two periods will be displayed... if they both pass, of course. In the event that the test fails, instead of a period, you'll find a big fat `F`, for *failure*.

```
1 $ phpunit --colors tests
2 PHPUnit 3.7.18 by Sebastian Bergmann.
3
4 F
5
6 Time: 0 seconds, Memory: 2.75Mb
7
8 There was 1 failure:
9
10 1) PracticeTest::testHelloWorld
11 Hello, World.
12 Failed asserting that true is false.
13
14 /Users/Jeffrey/Desktop/chapter-1/tests/PracticeTest.php:8
```

The majority of your days will be spent staring at failing tests, so get used to it! Luckily, the output will prove incredibly helpful. Above, we can see that there was one failure from the `testHelloWorld` test, and specifically on line eight. Helpful!



Where Are The Colors? In the next chapter, we'll dig more deeply into PHPUnit's configuration options. Until then, use the `--colors` option to request colored output: `vendor/bin/phpunit --colors tests`.

Decoding A Test Class Structure

Even though this is a simple class, there are a handful of important bits and pieces to be aware of. To refresh your memory, here's the code once again.

```
1 <?php // lesson/tests/PracticeTest.php
2
3 class PracticeTest extends PHPUnit_Framework_TestCase {
4     public function testHelloWorld()
5     {
6         $greeting = 'Hello, World.';
7
8         $this->assertTrue($greeting === 'Hello, World.',
9             $greeting);
10    }
```

- **File Naming** - The file name is important. Notice that we're following a `FooTest.php` convention. While this can be modified, let's stick with the defaults for now. Convention over configuration, right?
- **Matching** - The name of the class is the same as the file name.
- **Inheritance** - The class extends `PHPUnit_Framework_TestCase`. This class was made available when we installed PHPUnit through Composer. You'll soon find, however, that, when testing in Laravel, we typically extend Laravel's `TestCase` (the other file within the `tests/` folder). If you move up the inheritance chain, you'll find that a parent class does extend `PHPUnit_Framework_TestCase`. We simply inherit from `TestCase` to set up some Laravel-specific processes, as well as pull in a few helper methods for testing our applications.

- **Method Naming** - Every test should be contained within a method that has a descriptive name, and begins with the word, *test*.

Take a few moments, and commit some of these techniques to memory. Let's next investigate the test logic, itself.

assertTrue

```
1 $greeting = 'Hello, World.';  
2 $this->assertTrue($greeting == 'Hello, World.', $greeting);
```

Hopefully, you'll find that PHPUnit's test assertions are quite readable! Even without understanding the nuts and bolts, it's easy to decipher what's happening here. All assertions are available on the test class instance. In this case, we want to assert that the value of the `$greeting` variable is, in fact, equal to "Hello, World."

The `assertTrue` method accepts two methods.

```
1 $this->assertTrue(ACTUAL, OPTIONAL MESSAGE);
```

As you'll find in PHPUnit, inverse methods are nearly always available. Should you need to assert that a value is, not true, but false, then `assertFalse` has you covered.

```
1 $this->assertFalse(ACTUAL, OPTIONAL MESSAGE);
```

assertEquals

In the previous example, our only goal was to assert that a variable's value was equal to a specified string. While `assertTrue` will get the job done, it's not the most readable option in this case. Never ignore test readability. It'll eventually bite you, if you do.

Let's introduce another assertion that is a better fit for this task: `assertEquals`.

```
1 $greeting = 'Hello, World.';  
2 $this->assertEquals('Hello, World.', $greeting);
```

That's better, isn't it? Similar to most of PHPUnit's assertions, `assertEquals` accepts three arguments:

```
1 $this->assertEquals(EXPECTED, ACTUAL, OPTIONAL MESSAGE);
```

`assertNotEquals` is the inverse of this assertion, and has the same signature.

If you wish to prove that two values are equal to one another, then, clearly, `assertEquals` is a better choice than `assertTrue`, even though both will work. Run the tests by returning to the command line and calling PHPUnit:

```
1 $ phunit
```

```
● laravel-app phunit
PHPUnit 3.7.18-1-g2c67e52 by Sebastian Bergmann.

Configuration read from /Users/Jeffrey/Desktop/laravel-app/phpunit.xml

.

Time: 0 seconds, Memory: 3.50Mb

OK (1 test, 1 assertion)
● laravel-app
```

Success! Do a happy dance, spin three times, and continue on.



Your Turn: Make a variable, `$sum`, equal to $2 + 2$. Next, write an assertion to prove that `$sum` does, in fact, equal 4. For bonus points, write the assertion first.

assertSame

`assertEquals` will break down as soon as you need to compare with strict equality. For instance, how might you assert that a variable is equal to 0? You might try:

```
1 $val = 0;  
2 $this->assertEquals(0, $val);
```

This will work; however, should you substitute any other falsy value, the tests will still return green.

```
1 $val = null;  
2 $this->assertEquals(0, $val); // true
```

In situations, when you require strict comparison (or effectively `==`), reach for `assertSame`, as demonstrated below:

```
1 $val = null;  
2 $this->assertSame(0, $val); // false  
3  
4 $val = 0;  
5 $this->assertSame(0, $val); // true
```

assertContains

The goal of this book isn't to teach every assertion, however, it's important to cover the essentials. While PHPUnit offers dozens of assertions, you'll likely find that even a handful of them will serve the huge majority of your testing needs. In the instances where a piece of code requires a different form of assertion, PHPUnit is heavily documented.

Imagine that you have a list of names, and need to prove that the array contains a specific value. Again, while `assertTrue` can handle this task, it's better to opt for the more readable option: `assertContains`.

```
1 public function testLaravelDevsIncludesDayle()  
2 {  
3     $names = ['Taylor', 'Shawn', 'Dayle'];
```

```
4     $this->assertContains('Dayle', $names);  
5 }
```

The signature of `assertContains` is:

```
1 $this->assertContains(NEEDLE, HAYSTACK, OPTIONAL MESSAGE);
```



An easy way to remember the argument order for these various assertions is to read them aloud. “Assert contains Dayle, given this HAYSTACK” is more readable than “Assert contains HAYSTACK, and the item, Dayle.”

As you are now aware, the inverse assertion is also available. Let’s ensure that a troll is not included in our list of influential Laravel developers.

```
1 $names = ['Taylor', 'Shawn', 'Dayle'];  
2 $this->assertNotContains('Troll', $names); // true
```

assertArrayHasKey

There will be times when you need to assert that a provided array contains, not a specific value, but a key. For instance:

```
1 $family = [  
2     'parents' => 'Joe',  
3     'children' => ['Timmy', 'Suzy']  
4 ];
```

In this pseudo-example, let’s assume that we require a `$family` to contain a parent. In situations such as this, `arrayContains` is not the right tool for the job.

```
1 $this->assertContains('parents', $family); // Failed asserting  
that an arra\  
2 y contains 'parents'.
```

Instead, we need to assert that a key exists within the specified array. The solution is `assertArrayHasKey`.

```

1 public function testFamilyRequiresParent()
2 {
3     $family = [
4         'parents' => 'Joe',
5         'children' => ['Timmy', 'Suzy']
6     ];
7
8     $this->assertArrayHasKey('parents', $family); // true
9 }
```

To take things a step further, what if we expect the parents key to contain an array of one or more parents? How might we do that?

assertInternalType

`assertInternalType` can be used to verify the type of the supplied variable.

```
1 $this->assertInternalType(EXPECTED, ACTUAL, MESSAGE);
```

Continuing on with the family example, to assert that a family's parents key is equal to an array of one or more items, we might do:

```

1 public function testFamilyRequiresParent()
2 {
3     $family = [
4         'parents' => 'Joe',
5         'children' => ['Timmy', 'Suzy']
6     ];
7
8     $this->assertInternalType('array', $family['parents']); //
false
9 }
```

Or, to assert that a variable is of type integer:

```

1 $age = 25;
2 $this->assertInternalType('integer', $age); // true
```

assertInstanceOf

Often, you will need to ensure that a variable is an instance of some class. This is easy in PHPUnit, via the `assertInstanceOf` method.

```
1 $this->assertInstanceOf(EXPECTED, ACTUAL, MESSAGE);
```

For a usage example, given the following code:

```
1 class DateFormatter {
2     protected $stamp;
3
4     public function __construct(DateTime $stamp)
5     {
6         $this->stamp = $stamp;
7     }
8
9     public function getStamp()
10    {
11        return $this->stamp;
12    }
13 }
```

To ensure that `$stamp` is an instance of PHP's `DateTime` class (other than the fact that we've provided type hints), we could use the following test:

```
1 public function testStampMustBeInstanceOfDateTime()
2 {
3     $date = new DateFormatter(new DateTime);
4
5     $this->assertInstanceOf('DateTime', $date->getStamp()); // true
6 }
```

Asserting Exceptions

When unit testing, it's important to test every possible path through your code. This is one of the core reasons why, if you use too many conditionals, it can make a class or method difficult to test.

An example of such a path might be one that throws an exception. In PHPUnit,

-----, we use doc-blocks to assert exceptions, like so:

```
1 /**
2  * @expectedException EXCEPTION_NAME
3 */
```

This doc-block declares that, given the contents of the method that it corresponds to, PHPUnit should expect an exception to be thrown. If one is not, then the test will fail.

Let's say that a method should throw an exception if a non-numeric value is passed. Here's how we might assert that:

```
1 /**
2  * @expectedException InvalidArgumentException
3 */
4 public function testCalculatesCommission()
5 {
6     $commission = new Commission;
7     $commission->setSalePrice('fifteen dollars');
8 }
```

Notice how, in this case, there is no `assertX` call. Instead, we've merely added the necessary code that should make the class throw an exception.

Summary

While we could continue on our way, reviewing every available PHPUnit assertion, this would be unwise. Our brains are leaky little things; fill them up too quickly, and they burst. The handful of assertions covered in this chapter should be enough to get you on your way. When you do need additional functionality, the documentation is only a click away.

Next, before we dive into Laravel-specific testing, we should take some time to configure PHPUnit more fully to our needs.

Chapter 3: Configuring PHPUnit

PHPUnit ships with a vast array of configuration options - many of which you'll never use. In this chapter, we'll discuss a selection of the most helpful options, and then review the process of creating a custom configuration XML file to store these settings.

Options

Technicolor

Though you're free to live life in black and white, it's recommended that you instead enable colored output when running tests. Similar to a traffic stop light, a simple green versus red output can help to speed up your development workflow. We respond to color more quickly than text.

¹ `phpunit --colors`

```
1. Jeffrey@Jeffreys-MacBook-Air: ~/Desktop/chapter-1 (zsh)

● chapter-1 vendor/bin/phpunit --colors tests
PHPUnit 3.7.18 by Sebastian Bergmann.

.

Time: 0 seconds, Memory: 2.50Mb

OK (1 test, 1 assertion)
● chapter-1
```

```
1. Jeffrey@Jeffreys-MacBook-Air: ~/Desktop/chapter-1 (zsh)
  chapter-1 vendor/bin/phpunit --colors tests
PHPUnit 3.7.18 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 2.75Mb

There was 1 failure:

1) PracticeTest::testHelloWorld
Hello, World.
Failed asserting that true is false.

/Users/Jeffrey/Desktop/chapter-1/tests/PracticeTest.php:8

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
  chapter-1
```

Bootstrapping

There will be times when you need to include certain files before executing your tests. A perfect example of this is ensuring that Composer's autoload script (located at `vendor/autoload.php`) is included before any tests run. Though you could manually require this file either at the beginning of every test class or within a master file, it's better to leverage PHPUnit's bootstrap switch, like so:

¹ `phpunit --bootstrap="vendor/autoload.php"`

Output Formats

In addition to the default format for test reporting, PHPUnit offers two alternative flags: `--tap` and `--testdox`.

TAP, an acronym for *Test Anything Protocol*, is Perl's text-based interface between testing modules.

```
1 $ phpunit --tap tests
2 TAP version 13
3 ok 1 - PracticeTest::testHelloWorld
4 ok 2 - PracticeTest::testHelloUniverse
5 1..2
```

The TestDox format takes a slightly different approach. It will read your test methods and convert them from *camelCase* names to readable sentences. As an example, `testRedirectsToHomePageOnSave` will be converted to *Redirects To Home Page On Save*. See below:

```
1 $ phpunit --testdox tests
2 PHPUnit 3.7.18 by Sebastian Bergmann.
3
4 Practice
5 [x] Reloads current page if save fails
6 [x] Redirects to home page on save
```

This can be helpful for a birds-eye view of which tests were successful, as well as providing documentation. Remember, that's one of the significant advantages to writing tests: free documentation!

XML Configuration File

If you find yourself passing multiple switches with every call to `phpunit`, then you're doing it wrong. Don't do this:

```
1 phpunit --bootstrap="vendor/autoload.php" --testdox --colors
tests/
```

Instead, leverage PHPUnit's ability to read a configuration file. Let's review the

simplest possible example. Within the root of your project, create a `phpunit.xml` file and insert:

```
1 <phpunit colors="true"></phpunit>
```

With that single line, you may now exclude the `--colors` flag from your tests. However, let's flesh this file out a bit more, using some options that you haven't yet seen.

```
1 <phpunit bootstrap="vendor/autoload.php"
2           colors="true"
3           convertErrorsToExceptions="true"
4           convertNoticesToExceptions="true"
5           convertWarningsToExceptions="true"
6           stopOnFailure="true">
7 </phpunit>
```

The first two attributes should already be familiar to you. The remainder are optional, and boil down to personal choice. I tend to prefer that all errors, notices, and warnings be converted to exceptions. Also, if a test fails, then I want to cancel any remaining tests. Think of it as *control + c on failure*. Though some may disagree, as I see it, there's no use in executing potentially hundreds of tests if you're only concerned with fixing the first failure.

As things currently stand, we still must manually specify the path to the `tests/` directory. That's no good! To set a custom directory, use the `<testsuites>` element along with any number of child `<testsuite>` elements, like so:

```
1 <phpunit bootstrap="vendor/autoload.php"
2           colors="true"
3           convertErrorsToExceptions="true"
4           convertNoticesToExceptions="true"
5           convertWarningsToExceptions="true"
6           stopOnFailure="true">
7
8     <testsuites>
9       <testsuite name="Test Suite">
10         <directory>tests</directory>
11       </testsuite>
```

```
12      </testsuites>
13  </phpunit>
```

Splitting your test suites in this way can add a great deal of flexibility down the line, when you only wish to test a particular subset of your application. Let's add another test suite that is specifically for, say, integration tests.

```
1 <testsuites>
2   <testsuite name="Test Suite">
3     <directory>tests</directory>
4   </testsuite>
5
6   <testsuite name="Integration">
7     <directory>tests/integration</directory>
8   </testsuite>
9 </testsuites>
```

To execute only the tests referenced in the "Integration" test suite, run:

```
1 phpunit --testsuite="Integration"
```

For most chapters in this book, we'll stick with one master test suite, however, for real-world projects, be sure to organize your tests. On save, you don't want to be executing acceptance tests, in addition to the unit tests.

Nonetheless, to execute all tests within the `tests/` folder while passing the options listed above, simply run `phpunit`. Success!



Did You Know? Laravel ships with its own `phpunit.xml` file (located in the root of your application). Though it may freely be modified, most of the time, you'll find that the default settings are perfect for your needs. As such, manually creating this file is only necessary for non-Laravel projects.

Continuous Testing

An excellent way to get into the testing groove is to automate the process of running your tests each time an applicable file is saved. While light-weight editors, like Sublime Text, don't offer this functionality out of the box (most

IDEs do), it's a fairly simple process to install a few command-line tools to get the job done.

Please note that, to leverage the following tools, you'll need to install Ruby and Rubygems. Mac users can [follow this guide](#), while Windows faithfuls can use the simple [Ruby Installer](#).



[RubyGems](#) is the Ruby community's gem hosting service, not dissimilar to Composer's Packagist companion.

If you like the idea of automated testing, I recommend using [Guard](#) to handle your continuous testing needs, along with the notification tool that best suits your needs and OS.



Definition: Guard is a command line tool to easily handle events on file system modifications.

To get setup, begin by installing Guard and the PHPUnit plugin through the command line.

```
1 gem install guard
2 gem install guard-phpunit
```

Pay close attention to the output upon running these commands. It's possible that you'll be advised to install additional packages, such as rb-fsevent. If so, follow orders!

```
1 gem install rb-fsevent
```

Lastly, if you desire notifications (essentially a PASS/FAIL banner on save), Mac users should install the [Terminal Notifier Guard](#) gem.

```
1 gem install terminal-notifier-guard
```

With that, you're all set to go! Initialize Guard for your application by running:

```
1 guard init
2
3 19:15:57 - INFO - Writing new Guardfile to
4 /Users/Jeffrey/Desktop/demo/Guardfile
5 19:15:57 - INFO - phpunit guard added to Guardfile, feel free
6 to edit it
```

Assuming that `guard-phpunit` is the only Guard plugin that you have installed, running this command will generate a file, called `Guardfile`, written in Ruby, containing the following boilerplate:

```
1 # A sample Guardfile
2 # More info at https://github.com/guard/guard#readme
3
4 guard 'phpunit', :cli => '--colors' do
5   watch(%r{^.+Test.php$})
6 end
```

If you're new to Ruby, don't let this code scare you; it's really quite simple. It specifies that we want Guard to watch all files that match the regular expression, `^.+Test\.php$`, or, in human speak, any file in the root of the project that ends with `Test.php`.

Let's adjust this somewhat to reflect a more realistic folder structure.

```
1 guard 'phpunit', :cli => '--colors', :tests_path => 'tests' do
2   watch(%r{^.+Test.php$})
3 end
```

Using the `tests_path` option, we've specified that the `tests/` path should be used when triggering all tests. Give it a try: create a simple `tests/ExampleTest.php` file, and add:

```
1 <?php // tests/ExampleTest.php
2
3 class ExampleTest extends PHPUnit_Framework_TestCase {
4   public function testContinuousTesting()
```

```
5     {
6         $this->assertTrue(false);
7     }
8 }
```

Now Guard doesn't magically run automatically. You must tell it when to begin watching files, using the guard command.

```
1 $ guard
2 19:41:15 - INFO - Guard uses TerminalNotifier to send
3 notifications.
4 19:41:15 - INFO - Guard uses TerminalTitle to send
5 notifications.
6 19:41:15 - INFO - Running all tests
7 19:41:15 - INFO - F
8 > [^C5AD6455F02F]
9 > [^C5AD6455F02F] Failures:
10 > [^C5AD6455F02F]      1) ExampleTest::testContinuousTesting
11 > [^C5AD6455F02F]          Failed asserting that false is true.
12 > [^C5AD6455F02F]          #
13 /Users/Jeffrey/Desktop/demo/tests/ExampleTest.php: \
14 6
15 > [^C5AD6455F02F]
16 > [^C5AD6455F02F] Finished in 0 seconds
17 > [^C5AD6455F02F] 1 test, 1 assertion, 1 failures
```

If everything went according to plan, you'll see something along the lines of Figure 2.3. Continuous testing for the win!

```
1 <?php
2
3 class ExampleTest extends PHPUnit_Framework_TestCase
4 {
5     public function testContinuousTesting()
6     {
7         $this->assertTrue(true);
8     }
}
```

As long as a file within the `tests/` folder that matches the regular expression, `^.+Test\.php$`, is modified, the tests will run again. Make a few changes to `tests/ExampleTest.php`, save, and see for yourself.

If you return to the command line, to re-run all tests, hit `Enter`. Alternatively, to cancel Guard, type `exit`, or simply `e`.

Watching Files

Chances are high that you want Guard to monitor more than merely test files. If following a TDD cycle, you'll surely need a way to trigger a class' associated test upon save. Here's an updated `Guardfile` that creates a link between `calculator.php` and `calculatorTest.php`.

```
1 guard 'phpunit', :cli => '--colors', :tests_path => 'tests' do
2   watch(%r{^.+Test.php$})
3
4   watch('app/libraries/Calculator.php') {
5     'tests/libraries/CalculatorTest.p\
6     hp'}
7 end
```

To verbalize this new line of code: when `calculator.php` is modified, then run, not all tests, but specifically the one at `tests/libraries/CalculatorTest.php`. However, we clearly don't want to repeat this step for every single file in our project! Let's instead use regular expressions to make the matching more

dynamic.

```
1 guard 'phpunit', :cli => '--colors', :tests_path => 'tests' do
2   watch(%r{^.+Test.php$})
3
4   watch(%r{app/libraries/( .+)\.php}) { |m| "tests/libraries/#"
5     {m[1]}Test.php" \
6   }
7 end
```

With this modification, any file within the `app/libraries` directory will trigger its associated test.



Tip: Notice how the folder structure for our tests mirror the app structure. This is widely considered to be a best practice.

This takes care of our library tests, however, to extend it even further, we might update the `Guardfile` to:

```
1 guard 'phpunit', :cli => '--colors', :tests_path => 'tests' do
2   watch(%r{^.+Test.php$})
3
4   watch(%r{app/( .+)/(.+)\.php}) { |m| "tests/#{m[1]}/#"
5     {m[2]}Test.php" }
6 end
```

Now, the subfolder matching will be dynamic as well. When the file, `app/models/User.php`, is saved, Guard will now attempt to run the test located in `tests/models/UserTest.php`.

Triggering Multiple Files

There will be times when you wish to execute all tests for your app, if a particular file is saved. For example, if `app/routes.php` is saved, there likely won't be any dedicated test file to execute. In these situations, it's easiest to instead trigger all tests within a given folder.

```
1 guard 'phpunit', :cli => '--colors', :tests_path => 'tests' do
```

```
2   watch(%r{^.+Test.php$})
3
4   watch(%r{^app/views/.+$}) { Dir.glob('tests/**/*Test.php') }
5 end
```

With this last modification, if any file within the `app/views` folder is saved, Guard will trigger all `*Test.php` files within the `tests/` directory. Above, we're using `Dir.glob()` to return an array of all appropriate files. It's admittedly not the most elegant solution, but, until the Guard PHPUnit plugin offers a better solution, this is the best option.



Tip: Remember, Guard isn't limited to PHPUnit by any stretch. It can additionally be used to compile Sass and CoffeeScript, reload the browser, and much more. [Refer to the Guard Wiki](#) for a full list of available plugins.

Some Vim-Specific Advice

Though plenty of developers prefer automated testing *on save*, I take a different approach and manually trigger my tests. Unless you're a Vim user, feel free to continue on to the next section.

As a Vim user, I use a few mappings when doing TDD.

```
1 nmap ,t :!phpunit<cr>
```

This specifies that, when I type `,t`, the `phpunit` command should be executed (`<cr>` represents a carriage return). I've found that, for my workflow, this makes for the cleanest TDD cycle - one that I'm in complete control over.

While the previous command is excellent for triggering the full testsuite, when testing in isolation, it's better to exclusively target the applicable class. In these cases, I manually create mappings on the fly to override the default behavior of `,t`. A first option might be:

```
1 nmap ,t :!phpunit %<cr>
```

The only difference with this mapping is that PHPUnit will exclusively test the

current class (represented by the percentage sign). Though this works, it assumes that the mapping is triggered from the test class. Chances are, though, that, as a Vim user, you'll be using *split panes* (multiple windows). A better approach is to hardcode the file that you're currently working on.

```
1 nmap ,t :!phpunit app/tests/MyClassTest.php<cr>
```

Now, regardless of which split pane you might be in, running `,t` will correctly trigger the tests located in `MyClassTest.php`. Success! You now have a simple key stroke for testing the class. Don't forget that you can press the *up* arrow key to cycle through your previous commands. This way, you don't have to manually type the same sequence over and over.

Summary

In this chapter, you were introduced to the basics of configuring PHPUnit. For language-agnostic projects, an understanding of these various options will prove essential. However, as noted earlier, when testing Laravel applications, don't forget that the framework provides its own `phpunit.xml` file out of the box.

This file is by no means *read-only*. If you need to add a new test suite (perhaps for your acceptance tests), edit the file as needed. And, with that, we're on to the next chapter. Go have some celebratory raisins; you're doing great.

Chapter 4: Making PHPUnit Less Verbose

Importing Assertions as Functions

There's no denying that `$this->assertEquals` can get verbose, when you have to write it repeatedly. Though we won't be using this technique throughout the book, if you wish, you can import all of the assertions as simple functions. This, in effect, allows you to remove the `$this` portion.

```
1 $name = 'Doug';
2 assertEquals('Doug', $name);
```

If you installed PHPUnit through Composer, then the list of functions will be stored in:

```
1 vendor/phpunit/phpunit/PHPUnit/Framework/Assert/Functions.php
```

Simply require that file at the top of your test class (or at the top of Laravel's `TestCase`), and you may now reference these assertions as global functions.

```
1 <?php
2
3 require_once
'vendor/phpunit/phpunit/PHPUnit/Framework/Assert/Functions.php\
4 ';
5
6 class PracticeTest extends TestCase {
7     public function testAdd()
8     {
9         $sum = add(10, 5);
10
11         assertEquals(15, $sum);
12     }
13 }
```

Applying the Laravel Style to PHPUnit

Chances are, Laravel appeals to you so much because it can be read like simple English. Coding is complicated enough; we don't need to introduce unnecessary jargon to make the process even more confusing! Though the misinformed will criticize Laravel for using too many statics, if they would take a closer look, they would realize that facades allow for a static-like API, while instantiating the applicable classes and injecting any dependencies behind the scenes. This allows for easy-to-use code, such as:

```
1 Redirect::route();
2 Form::open();
3 Validator::attempt();
4 View::make();
```

The only problem is that, when it comes to testing, we're still limited to PHPUnit's verbose syntax that doesn't blend with Laravel as elegantly as we might hope. This is why I recently released [two wrappers classes](#) around PHPUnit's assertion library that bring the Laravel *feel* to your testing: Should and Assert. With these two wrappers, which can be downloaded through Composer, your assertions will take the form of:

```
1 Should::equal(4, 2 + 2);
2 Should::beTrue(true);
3
4 Assert::greaterThan(20, 21);
```

Behind the scenes, the applicable PHPUnit assertions are still being called. These wrappers simply provide a more expressive interface for writing tests. Though I won't be using this package in this book, if you'd like to use it in your own projects, update your composer.json to reference the `phpunit-wrappers` package, like so:

```
1 {
2     "require-dev": {
3         "way/phpunit-wrappers": "dev-master"
4     }
5 }
```

Next, run `composer install --dev` or `composer update --dev` to download the package from GitHub into your vendor directory. Finally, import the `Should` or `Assert` class (or both) into your test file, and have fun!

```
1 <?php
2
3 use Way\Tests\Assert;
4 use Way\Tests\Should;
5
6 class PracticeTest extends TestCase
7 {
8     public function testItWorks()
9     {
10         $name = 'Joe';
11
12         Should::equal('Joe', $name);
13         Assert::equals('Joe', $name);
14     }
15 }
```

If you want to create a custom alias for a PHPUnit assertion, doing so is a cinch.

```
1 Should::getInstance()->registerAliases([
2     'eq' => 'assertEquals'
3 ]);
```

Now, when you call `Should::eq()`, behind the scenes, the arguments that you pass to it will be sent through to PHPUnit's native `assertEquals` method. Nifty!



Please note that, while I favor many of the tricks outlined in this chapter, for the purposes of the book, they won't be used. Let's stick with the standard API.

Chapter 5: Unit Testing 101

In Chapter 1, we took the first steps toward “slow and steady wins the race” testing. Let’s continue on this path.

My Struggles

One unfortunate truth of learning how to test is that so much of the information on the web and in books contradicts one another. As a student, I’d read one article, think I had a decent grasp on the terminology and rules, and then move on to the next article that, in many ways, preached an entirely different set of guidelines. Imagine learning what two plus two equals, when all the resources provide different answers?

What’s the reason for this dissonance in the community? Well, it’s really quite simple: we’re still figuring this stuff out! Even though the concept of testing software goes as far back as the seventies - during the days of [Smalltalk](#) - the truth is that, even today, the development community remains divided on many issues, including such basic things as whether testing is beneficial. As a result, as you continue learning beyond this book, be prepared to discover different methodologies and terminologies, dependent upon a number of factors, including the article’s publish date, which language it was written for, and what belief system its writer has.

The differing view points were only the beginning of my struggles, though. As I continued digging in, I didn’t yet understand or appreciate the various types of testing. As such, I’d learn from one source that testing in isolation was paramount, and then move on to find countless other articles, where the authors seemed to ignore this rule, in favor of touching every part of the system. Of course, now, I realize that these articles focused on different types of testing (acceptance or functional vs. unit), but, back then, I was dumbfounded and overwhelmed.

I suppose what I want you to understand and appreciate is that we’re still in the early days of test-driven development, and testing in general. Even worse, while the Ruby on Rails community is a culture that has widely embraced and

evangelized testing, the same unfortunately can't be said for the PHP world. But, we're fixing that, right? The tides are turning; I hope you're on board!

In this chapter, we'll focus exclusively on unit testing with test-driven development. If you've come across other types of testing in your web travels - perhaps ones that do hit databases and query APIs - rest assure that they absolutely have their place. In fact, in real world projects, you'll have a number of test suites, each which exercises your application in slightly different ways. That being said, this chapter is not concerned with those other styles of testing; we'll get to them in due time. For now, though, and as a first step into the testing waters, let's focus exclusively on one object at a time.

Unit Testing

Okay, it's time to dig into the process of unit testing, not our models or controllers (we'll get to those soon), but simple functions and utility classes. This will give us the perfect excuse to learn more about proper test organization.



Unit testing refers to the smallest testable piece of an application. If I feed a method this set of data, then, in return, I expect *that*. Nothing beyond that class should be touched (*if we're following the London School of TDD*). Not the database, not a single dependency. This is referred to as testing in isolation.

Arrange, Act, Assert

Preparing a unit test can be reduced to three simple actions.

1. **Arrange:** Set the stage, instantiate objects, pass in mocks.
2. **Act:** Execute the *thing* that you wish to test.
3. **Assert:** Compare your expected output to what was returned.

As you'll find, every test follows this basic flow. Here's an example:

```
1 public function testFetchesItemsInArrayUntilKey()
2 {
3     // Arrange
4     $names = ['Taylor', 'Dayle', 'Matthew', 'Shawn', 'Neil'];
5
6     // Act
```

```

7     $result = array_until('Matthew', $names);
8
9     // Assert
10    $expected = ['Taylor', 'Dayle'];
11    $this->assertEquals($expected, $result);
12 }

```

If you pick up a BDD-specific book, however, you might come across slightly different terminology that translates to the same thing: *Given*, *When*, *Then*.

Given this set of data, **when** I perform this action, **then** I expect that response.

Here's a second test for illustration purposes:

```

1 /**
2  * @expectedException InvalidArgumentException
3 */
4 public function testThrowsExceptionIfKeyDoesNotExist()
5 {
6     // Given this set of data
7     $names = ['Taylor', 'Dayle', 'Matthew', 'Shawn', 'Neil'];
8
9     // When I call the until function and
10    // specify a different key
11    $result = array_until('Bob', $names);
12
13    // Then an exception should be thrown (see doc-block)
14 }

```

Though the terminology is different, this Given/When/Then syntax is a bit more readable, and encourages you to describe the behavior of your code. Also, note that each step in the cycle is quite small. If the ratio becomes lopsided, it's a good indication that refactoring is in order.

Finally, if you're curious, here's the code to make both tests pass:

```
1 function array_until($stopPoint, $arr)
```

```

2  {
3      $index = array_search($stopPoint, $arr);
4
5      if ($false === $index)
6      {
7          throw new InvalidArgumentException('Key does not exist in
array');
8      }
9
10     return array_slice($arr, 0, $index);
11 }

```

Testing in Isolation

A core fundamental to successful unit testing is to test in isolation. This means that all outside dependencies which aren't directly related to the *thing* that you're attempting to test should be stubbed or mocked (more on these terms later). For example, when unit testing a model, you shouldn't, in the process, be hitting the database. You shouldn't hit a web service. You shouldn't even reference one of your other classes. Instead, stick with one object at a time. You can flex multiple parts of the system in the future, when you write integration tests.

Tests Should Not Be Order-Dependent

Another rule of unit testing is that your tests should never be dependent upon previous tests. For example, beginning developers will often set state in one test, and then continue to depend upon that same state for assertions in future tests. Don't do this; it's an anti-pattern.

What you want to strive for is tests which can easily be executed independently from one another without causing failure. An easy way to detect this is if a particular test passes when the entire suite is executed, but fails when triggered by itself. In these situations, the failure is due to the test's dependency on some previously set state. Not good.



Tip: Each test should recreate the world in which it acts upon. Or, in other words, never depend on test order.

Test-Driven Development

Test-Driven Development is an agile software pattern in which a developer prepares a test **before** a single line of production code is written. Popularized by Kent Beck, in addition to ensuring that your code works as expected, this methodology forces you to think before coding. While a cowboy approach may once have been the norm in our industry, as a collective, we're rapidly moving away from it, in favor of a mature, intentional development process.

TDD declares three core rules:

1. **Write a Failing Test:** You may not write a single line of production code unless a failing test is present.
2. **Make it Pass:** Once a test has been defined, you may only write the minimum amount of code to make the test pass - even if this means *faking* a method's return value. Approach each failing test from the perspective of, "*what is the simplest way to make this test pass?*"
3. **Refactor:** Only when the tests have passed (they return green) may you refactor your code.

Behavior-Driven Development

In addition to TDD, or test-driven development, you'll often hear about a seemingly *different* form of testing, referred to as *behavior-driven development*. As it turns out, if you write TDD well - by describing the behavior of your code - then chances are, you're following the rules of BDD. With this style of coding, you describe how the SUT (system under test) should behave. Once you've written the minimum amount of code to make that behavior work, then the feature has been implemented. In this book, when I refer to TDD, I'm also referring to BDD. The two are not mutually exclusive. One is merely the other in its best form.

Testing Functions

Before we tackle the calculator, first, let's test-drive a simple function.

In real-world applications, you'll be testing logic from external sources, such as a function, class, or web service. To take a first step into these waters, we'll test a helper function for a Laravel that should generate an anchor tag for, called `link_to`.

In fact, Laravel 4 recently implemented this very function. Nonetheless, let's rebuild it for education purposes.

Additionally, we'll embrace a TDD (test-driven development) approach, and write the test first.

```
1 public function testGeneratesAnchorTag()
2 {
3     $actual = link_to('dogs/1', 'Show Dog');
4     $expect = "<a href='http://:dogs1'>Show Dog</a>";
5
6     $this->assertEquals($expect, $actual);
7 }
```

Hopefully, this is beginning to make sense. Notice how we interact with the function before writing any production code. This way, we're never constrained by existing code. Or, in other words, how would you want to achieve this functionality in a perfect world? Write it as such, and then it's your responsibility to organize the necessary production code to make that possible.

In this case, we need a `link_to(URL, BODY)` function that should generate the HTML string that is stored in the `$expect` variable above.

```
1 $expect = "<a href='http://:dogs1'>Show Dog</a>";
```

Remember that we shouldn't be searching for “`localhost:8000`,” when running command line tests.

Then, we write the assertion and run the test.

```
1 $expect = "<a href='http://:dogs1'>Show Dog</a>";
2 $actual = link_to('dogs/1', 'Show Dog');
3
4 $this->assertEquals($expect, $actual);
```

Return to the command line, and execute the test.

```
1 phpunit
```

```

● laravel-app [master vendor/bin/phpunit
PHPUnit 3.7.18-1-g2c67e52 by Sebastian Bergmann.

Configuration read from /Users/Jeffrey/Desktop/laravel-app/phpunit.xml

PHP Fatal error: Call to undefined function link_to() in /Users/Jeffrey/Desktop/laravel-
app/app/tests/PracticeTest.php on line 6
PHP Stack trace:
PHP 1. {main}() /Users/Jeffrey/Desktop/laravel-app/vendor/phpunit/phpunit/composer/bin/
phpunit:0
PHP 2. PHPUnit_TextUI_Command::main() /Users/Jeffrey/Desktop/laravel-app/vendor/phpunit/
phpunit/composer/bin/phpunit:63
PHP 3. PHPUnit_TextUI_Command->run() /Users/Jeffrey/Desktop/laravel-app/vendor/phpunit/
phpunit/PHPUnit/TextUI/Command.php:129
PHP 4. PHPUnit_TextUI_TestRunner->doRun() /Users/Jeffrey/Desktop/laravel-app/vendor/php
unit/phpunit/PHPUnit/TextUI/Command.php:176
PHP 5. PHPUnit_Framework_TestSuite->run() /Users/Jeffrey/Desktop/laravel-app/vendor/php
unit/phpunit/PHPUnit/TestRunner.php:346

```

Fig 1.3 - The first failing test.

The wonderful thing about TDD is that it turns your coding process into a game. Not sure what to do next? Run the tests, and PHPUnit will tell you! TDD is about tiny, tiny steps. In this case, an error is displayed: *Call to undefined function link_to()*. There's our next step!

Often, you'll find that it's helpful to write the class or function under test just above the test class, like so:

```

1 function link_to() {}
2
3 class FunctionsTest extends PHPUnit_Framework_TestCase {
4     public function testBuildsAnchorTag()
5     {
6         $actual = link_to('dogs/1', 'Show Dog');
7         $expect = "<a href='http://:dogs1'>Show
Dog</a>";
8
9         $this->assertEquals($expect, $actual);
10    }
11 }
```

Once finished, you can always copy it to the proper file.



Tip: In the early testing stages, make simplicity a priority.

Having said that, to mimic a real-world scenario, this time, we'll take the latter approach. Let's assume that we have a `helpers.php` file within the `app/` folder. Think of this file as one that contains a variety of simple global functions, which can be used throughout your application.

Begin by creating the file, `app/helpers.php`.

```
1 // app/helpers.php
2 function link_to($url, $body)
3 {
4
5 }
```

Though we could simply require this file within the test class (and probably should at this early stage), let's instead try to embrace autoloading. Composer isn't a mind reader, though; we need to instruct it to load this new file. Luckily, that's a cinch. Return to `composer.json` and specify a new `files` array within the `autoload` object. Here's how your Composer file should look:

```
1 // composer.json
2 {
3     "require": {
4         "laravel/framework": "4.0.*"
5     },
6     "require-dev": {
7         "phpunit/phpunit": "3.7.*"
8     },
9     "autoload": {
10        "classmap": [
11            "app/commands",
12            "app/controllers",
13            "app/models",
14            "app/database/migrations",
```

```
15           "app/database/seeds",
16           "app/tests/TestCase.php"
17     ],
18     "files": [
19       "app/helpers.php"
20     ]
21   },
22   "minimum-stability": "dev"
23 }
```

Remember: we can't add `app/helpers.php` to the `classmap`, as it's not a class. In these situations, the best choice is to instead use the `files` array, as illustrated above.

Next, as should be done whenever this file is updated, instruct Composer to dump a new list of all the files that should be autoloaded.

```
1 composer dumpautoload -o
```



Tip: When running the `dump-autoload` command, always pass the `-o` (optimize) flag.

If we execute `phpunit` again, we'll receive the next step.

```
1 phpunit
2
3 There was 1 failure:
4
5 1) PracticeTest::testBuildsAnchorTag
6 Failed asserting that null matches expected '<a
7 href='http://dogs1'>Show\Dog</a>'.
```

Aha; so it seems that the test expected the anchor tag string, however, instead, only `null` was returned. Let's slime it as the first step.

```
1 // app/helpers.php
```

```
2
3 function link_to($url, $body)
4 {
5     return "<a href='http://:dogs1'>Show Dog</a>";
6 }
```

Slime vs. Generalize

Throughout this book, you'll frequently see references to the terms, *slime* and *generalize*.

Slime



Definition: *Slime* is nothing more than a bit of jargon that refers to returning a dummy value for the sole purpose of making a test pass. More traditionally, this is referred to as *faking it*. Feel free to use these terms interchangeably.

The principles of test-driven development dictate that tiny steps should be taken when coding. In other words, what is the simplest way to make a test pass? Initially, *sliming it* just might be the answer. Though hardcoding a returned value might seem foolish to you, the ultimate goal is to ensure that your code never becomes more complicated than is necessary to make the tests pass.

Generalize

Clearly, sliming is temporary. The key is to determine when it's appropriate to generalize.



Definition: Think of the term, *generalize*, as a way to specify when it's necessary to remove slime, in favor of real production code.

Perhaps an example is in order. Imagine a basic add function. A first assertion might be to ensure that, if you pass two and two, then the returned value from the function will, in fact, be four. What is the easiest possible solution to this *assertion*? Anyone, anyone?

The solution is to slime it, and hardcode the returned value, like so:

```
1 function add() {
2     return 4;
3 }
```

Once you write a second test - perhaps, passing three and three - a hardcoded value will no longer suffice, at which point generalizing will become a necessity. See how that works?

Calculating (excuse the pun) when to generalize is something that each developer must determine on their own.

Making the Test Pass

Returning to the `link_to` test, at this point we have two courses of action when our tests return red (or fail):

1. Change the message
2. Make it green

Let's remove the slime, and fetch the URL dynamically with Laravel's native `url` function.

```
1 // app/helpers.php
2
3 function link_to($url, $body)
4 {
5     $url = url($url);
6     return "<a href='{$url}'>{$body}</a>";
7 }
```

Run the tests, and it passes!

```
1 phpunit
2
3 OK (1 test, 1 assertion)
```



Did You Know? Laravel, itself, offers a helpers file. In addition to the helpful `url()` function used above (simply a convenience that maps to `URL::to()`), it offers a number of particularly useful array operation commands.

Let's take things one step further. It might be nice if the `link_to` function offered a way to specify attributes for the anchor tag as well, such as a `class` or `id`. To make this a reality, first add a new test and specify the desired end result.

```
1 <?php
2
3 class PracticeTest extends TestCase {
4
5     public function testBuildsAnchorTag()
6     {
7         $actual = link_to('dogs1', 'Show Dog');
8         $expect = "<a href='http://:dogs1'>Show
Dog</a>";
9
10        $this->assertEquals($expect, $actual);
11    }
12
13    public function testAppliesAttributesUsingArray()
14    {
15        $actual = link_to('dogs1', 'Show Dog',
['class' => 'button']);
16        $expect = "<a href='http://:dogs1'
class='button'>Show Dog</a>";
17
18        $this->assertEquals($expect, $actual);
19    }
20 }
```



As a basic rule of thumb, each test should represent but one path through your code.

What's the next step? Run the tests and find out, fool.

```
1 phpunit
2
3 1) PracticeTest::testBuildsAnchorTag
```

```
4 Failed asserting that two strings are equal.
5 --- Expected
6 +++ Actual
7 @@ @@
8 - '<a href="http://dogs1' class="button">Show Dog</a>'
9 +'<a href="http://dogs1'>Show Dog</a>'
```

Got it? Let's make this new test pass. We'll skip the sliming this time.

```
1 <?php // app/helpers.php
2
3 function link_to($url, $body, $parameters = null)
4 {
5     $url = url($url);
6     $attributes = '';
7
8     // If the user specified any parameters, then
9     // parse them and append to returned string
10    if ($parameters)
11    {
12        foreach($parameters as $attribute => $value)
13        {
14            $attributes .= "
15                {$attribute}='{$value}'";
16        }
17
18    return "<a href='{$url}'{$attributes}>{$body}</a>";
19 }
```

Run the tests, and...

```
1 OK (2 tests, 2 assertions)
```

It passes! Now that we have a passing test, we can move on to any necessary refactoring, as allowed by the TDD pattern.

In this case, I happen to know that Laravel's `HtmlBuilder` class (you know it as

the `Html` facade) offers an `attributes` static method that will handle the parsing. Let's refrain from reinventing the wheel by instead leveraging that implementation.

```
1 function link_to($url, $body, $parameters = null)
2 {
3     $url = url($url);
4
5     // If the user specified any parameters, then
6     // parse them and append to returned string
7     $attributes = $parameters ?
8
9     Html::attributes($parameters) : '';
10 }
```

Running the tests should return one small quirk.

```
1 1) PracticeTest::testGeneratesAnchorTagAcceptsAttributesArray
2 Failed asserting that two strings are equal.
3 --- Expected
4 +++ Actual
5 @@ @@
6 - '<a href='http://dogs1' class='button'>Show Dog</a>'
7 + '<a href='http://dogs1' class="button">Show Dog</a>'
```

Luckily, in this case, the `Html::attributes()` method simply uses double quotes, when our test expected singles. That's a non-issue and an easy fix: update the test's `$expect` variable, like so:

```
1 $expect = "<a href='http://dogs1' class=\"button\">Show
Dog</a>";
```

And that should be enough to give you a passing test! Congratulations! You just used test-driven development to write a helpful function that can be used within your views. That's all TDD is. Don't complicate it any more than that. Once again, the process is:

- Write a test that defines how you want to interact with your code
- Watch it fail
- Write the necessary production code to make it pass
- Refactor



Important: This form of testing conforms more to the Chicago style of TDD. There are those who would argue that the `ur1` function that was triggered inside `link_to` should have been mocked; we're testing `link_to`, not `ur1`. There absolutely is value to this approach, and you'll learn more about it (London school of TDD) soon. But, for now, just as TDD requires small steps, so does the learning process.

Testing Classes

The most common introduction to unit testing generally revolves around the creation of a calculator. I know what you're thinking: "*I hate calculator testing tutorials!*" I get it, trust me; nobody builds basic calculators in real life. Nonetheless, there's a reason why they're used so frequently (along with car examples). A calculator consists of basic logic that every one of us knows by heart. Two plus two will always equal four. By removing the barrier to entry, you can dedicate your entire focus to learning the assertions, rather than worrying about production code logic. Like I said, though, I feel your pain. Rest assured that we'll dig into lots of fancy stuff over the course of this book. But one step at a time, padawan!

Unit testing a class is no different than simple functions: instantiate the class, call the desired method, and write an assertion. To finish up this chapter, let's use TDD to write the obligatory calculator class. We'll tackle this in two sections:

1. What is the simplest way to handle basic arithmetic?
2. How can we leverage polymorphism to allow for a more extensible calculator?

This section will be framework-agnostic, so, if working along, create an empty directory on your desktop, and add a `calculatorTest.php` file with the first test.

```

1 <?php // CalculatorTest.php
2
3 class CalculatorTest extends PHPUnit_Framework_TestCase {

```

```
4     public function testInstance()
5     {
6         $calculator = new Calculator;
7     }
8 }
```

Immediately upon running the test - `phpunit CalculatorTest.php` - we'll see:

```
1 Class 'Calculator' not found
```

Of course; create `calculator.php` within the project root. We're not worried about folder organization for this simple demo.

```
1 <?php // Calculator.php
2
3 class Calculator {
4
5 }
```

Next, while the easiest way to include this class is to require it at the top like we did earlier, let's stay in the habit of embracing Composer for autoloading classes.

Create a new `composer.json` file, and specify that `calculator.php` should be autoloaded.

```
1 {
2     "autoload": {
3         "classmap": [
4             "calculator.php"
5         ]
6     }
7 }
```

A simple `composer install` and `composer dump` should add this file to the autoloader. Now, pass the `--bootstrap="vendor/autoload.php` option when calling `phpunit`, or create a configuration file. To keep things simple, we'll stick with the former.

```
1 phpunit --bootstrap="vendor/autoload.php" CalculatorTest.php
```

And we're green! What do we require from this class next? Well, I'd like it to keep track of the current result at all times; it should default to 0 upon instantiation. How might we represent that as a test?

```
1 <?php // CalculatorTest.php
2
3 class CalculatorTest extends PHPUnit_Framework_TestCase {
4     public function testResultDefaultsToZero()
5     {
6         $calc = new Calculator;
7         $this->assertSame(0, $calc->getResult());
8     }
9 }
```



Tip: When you require strict comparison, always opt for `assertSame` over `assertEquals`.

PHPUnit will inform us that the `getResult` method does not yet exist, so let's create it.

```
1 <?php // Calculator.php
2
3 class Calculator {
4     public function getResult()
5     {
6         return 0;
7     }
8 }
```

Why hardcode 0? Have you forgotten already? We're *slimin' it old school*. Don't worry; we'll fix it soon. Until then, though, I'd like to test an `add` method. When I call this method, the value specified should be added to a property that keeps a tally of the current total.

```
1 <?php // CalculatorTest.php
```

```

2
3 class CalculatorTest extends PHPUnit_Framework_TestCase {
4     public function testResultDefaultsToZero()
5     {
6         $calc = new Calculator;
7         $this->assertSame(0, $calc->getResult());
8     }
9
10    public function testAddsNumbers()
11    {
12        $calc = new Calculator;
13        $calc->add(5);
14        $this->assertEquals(5, $calc->getResult());
15    }
16 }
```

To make this second test pass, we'll need to introduce a `$result` property to keep track of the current total. We'll also want to [remove the slime](#).

```

1 // Calculator.php
2
3 class Calculator {
4     protected $result = 0;
5
6     public function add($num)
7     {
8         $this->result += $num;
9     }
10
11    public function getResult()
12    {
13        return $this->result;
14    }
15 }
```

Already, though, I'm seeing a potential snag. What if you pass a non-numeric value to the `add` method, such as an array or string? In those situations, I want to ensure that an exception is thrown.



Tip: Think of exceptions as your way, as the developer, of saying, “*I take exception to this action.*”

To throw an exception, if following the rules of TDD, we can’t write any production code until a failing test is written.

```
1 // CalculatorTest.php
2
3 /**
4  * @expectedException InvalidArgumentException
5 */
6 public function testRequiresNumericValue()
7 {
8     $calc = new Calculator;
9     $calc->add('five');
10}
```

Above, we’re using doc-blocks to inform PHPUnit that, given the provided code, we expect an exception to be thrown. Without altering any production code, of course, no exception will be thrown, which will cause this test to fail.

```
1) CalculatorTest::testRequiresNumericValue
2 Failed asserting that exception of type
"InvalidArgumentException" is thrown
3 n.
```

Let’s fix that.

```
1 // Calculator.php
2
3 public function add($number)
4 {
5     if ( ! is_numeric($number))
6         throw new InvalidArgumentException;
7
8     $this->result += $number;
9 }
```

Excellent - that was easy! Next, it would be helpful if I could pass a variable number of arguments to the add method, such as add(2, 2, 5).

```
1 // CalculatorTest.php
2
3 public function testAcceptsMultipleArgs()
4 {
5     $calc = new Calculator;
6     $calc->add(1, 2, 3, 4);
7
8     $this->assertEquals(10, $calc->getResult());
9     $this->assertNotEquals(
10         'Snoop Doggy Dogg and Dr. Dre is at the door',
11         $calc->getResult()
12     ); // jokes, jokes, jokes
13 }
```

Is this process starting to make sense, now? Write a test that defines what you wish you could do, watch the test fail, and then write the necessary production code to make it pass. I'm trying to drill this into your head, if you hadn't realized! Even better, it's fun (at least for us).

Here's the modified code (or at least a first draft) to allow for passing multiple numbers to add.

```
1 <?php // Calculator.php
2
3 class Calculator {
4     protected $result = 0;
5
6     public function add()
7     {
8         foreach (func_get_args() as $num)
9         {
10             if ( ! is_numeric($num))
11                 throw new InvalidArgumentException;
12
13             $this->result += $num;
14     }
15 }
```

```

15     }
16
17     public function getResult()
18     {
19         return $this->result;
20     }
21 }
```

This time, we filter through all of the passed arguments, and, as long as the value is numeric, we add it to the `$result` property. This will return the tests to green, but I'm beginning to worry a bit. What about when we implement the subtract, multiply, and divide methods? This may result in a lot of repeated code, breaking the *don't repeat yourself* (DRY) pattern. For example, let's add the necessary functionality to subtract numbers.

First, the test:

```

1 // CalculatorTest.php
2
3 public function testSubtract()
4 {
5     $calc = new Calculator;
6     $calc->subtract(4);
7
8     $this->assertEquals(-4, $calc->getResult());
9 }
```

And then the production code to make it pass.

```

1 // Calculator.php
2
3 public function subtract()
4 {
5     foreach(func_get_args() as $number)
6     {
7         if (! is_numeric($number))
8             throw new InvalidArgumentException;
```

```
10             $this->result -= $number;
11         }
12     }
```



Tip: When you find yourself copying and pasting code from one method to another, this is a tell-tale sign that refactoring is required.

Let's take a break, while the tests are green, and DRY up both the tests and production code.

Refactoring the Tests

Did you notice how, for each test, the calculator was instantiated? Things like this are better extracted to the `setUp` method, which will fire before each test runs.

```
1 public function setUp()
2 {
3     $this->calc = new Calculator;
4 }
```

At this point, our test class should look like:

```
1 <?php // CalculatorTest.php
2
3 class CalculatorTest extends PHPUnit_Framework_TestCase {
4
5     public function setUp()
6     {
7         $this->calc = new Calculator;
8     }
9
10    public function testResultDefaultsToZero()
11    {
12        $this->assertSame(0, $this->calc->getResult());
13    }
14}
```

```

15     /**
16      * @expectedException InvalidArgumentException
17     */
18     public function testRequiresNumericValue()
19     {
20         $this->calc->add('five');
21     }
22
23     public function testAcceptsMultipleArgs()
24     {
25         $this->calc->add(1, 2, 3, 4);
26
27         $this->assertEquals(10, $this->calc->getResult());
28         $this->assertNotEquals(
29             'Snoop Doggy Dogg and Dr. Dre is at the door',
30             $this->calc->getResult()
31         ); // jokes, jokes, jokes
32     }
33
34     public function testAddsNumbers()
35     {
36         $this->calc->add(5);
37
38         $this->assertEquals(5, $this->calc->getResult());
39     }
40
41     public function testSubtractsNumbers()
42     {
43         $this->calc->subtract(4);
44
45         $this->assertEquals(-4, $this->calc->getResult());
46     }
47
48 }

```

Refactoring the Production Code

Now, on to the production code. Rather than filtering through all arguments, determining whether they are numeric, and performing the arithmetic, as a first

step, we'll extract this logic away to a couple of dedicated methods: calculateAll and calculate.

```
1 <?php // Calculator.php
2
3 class Calculator {
4
5     protected $result = 0;
6
7     public function getResult()
8     {
9         return $this->result;
10    }
11
12    public function add()
13    {
14        $this->calculateAll(func_get_args(), '+');
15    }
16
17    public function subtract()
18    {
19        $this->calculateAll(func_get_args(), '-');
20    }
21
22    protected function calculateAll(array $nums, $symbol)
23    {
24        foreach ($nums as $num)
25        {
26            $this->calculate($num, $symbol);
27        }
28    }
29
30    protected function calculate($num, $symbol)
31    {
32        if ( ! is_numeric($num))
33            throw new InvalidArgumentException;
34
35        switch ($symbol)
36        {
```

```

37         case '+':
38             $this->result += $num;
39             break;
40
41         case '-':
42             $this->result -= $num;
43             break;
44     }
45 }
46
47 }

```

Though the code above does introduce a couple more methods, it lays the structure for further complexity in the class. Now, the calculate method handles all logic for performing the arithmetic.

Polymorphism

If that switch statement doesn't feel right to you, have a celebratory Fig Newton and pat yourself on the back. Just about any time that you use a switch statement, your class is secretly screaming for polymorphism. Of course, there are exceptions to every rule, but use this as a guideline. Too many conditionals are a cause for concern.

Let's redesign. The problem with the current setup is that every time we want to add a new operation type (add, subtract, multiply), we must add another method to the class and update the switch statement. Very quickly, this class could become unwieldy. It would be better if the class could call a method that performs the arithmetic, without having to be concerned over how that operation is performed. This is where polymorphism comes into play.

It looks like we'll need to design our calculator a bit differently. This is specifically why it's important to think before you begin coding.

Create an interface, Operation, that contains a single method.

```

1 interface Operation {
2     /**
3      * Perform the arithmetic

```

```

4      *
5      * @param integer $num
6      * @param integer $current
7      * @return integer
8      */
9      public function run($num, $current);
10 }

```

The reason why coding to an interface is good practice is because the calculator class will be calling methods on injected objects. We need to be sure that this `run()` method is available. By requesting an implementation of the Operation class, we can have complete assurance that such a method will be available at run-time.

Next, we'd create a new class for each type of operation. Here's one for addition:

```

1 class Addition implements Operation {
2     public function run($num, $current)
3     {
4         return $current + $num;
5     }
6 }

```

Notice how simple that is to read? The best part is that it's a cinch to test. That's the idea!

Now that we have one implementation of the Operation interface, we next need to update the tests to reflect this design change.

```

1 public function testAddsNumbers()
2 {
3     $this->calc->setOperands(5);
4     $this->calc->setOperation(new Addition);
5     $result = $this->calc->calculate();
6
7     $this->assertEquals(5, $result);
8 }

```

To make this test pass, the production code, also, needs to be adjusted, like so:

```
1 <?php
2
3 class Calculator {
4
5     protected $result = 0;
6     protected $operands = [];
7     protected $operation;
8
9     public function getResult()
10    {
11        return $this->result;
12    }
13
14     public function setOperands()
15    {
16        $this->operands = func_get_args();
17    }
18
19     public function setOperation(Operation $operation)
20    {
21        $this->operation = $operation;
22    }
23
24     public function calculate()
25    {
26        foreach ($this->operands as $num)
27        {
28            if ( ! is_numeric($num))
29                throw new InvalidArgumentException();
30
31            $this->result = $this->operation->run($num,
32            $this->result);
33        }
34
35        return $this->result;
36    }
37
```

```
37 }
```

Admittedly, this is slightly more complex, but it'll pay off in spades when it comes time to extend the functionality.

Extensibility

To multiply numbers as well, the `calculator` doesn't even need to be touched. Simply create a new `Multiplication` class that implements the interface that we created earlier, and pass it through. `calculator` will then call its method without knowing how it behaves. This is the beauty of polymorphism.

Of course, we start with a test to verify when our new multiplication *feature* is finished.

```
1 public function testMultipliesNumbers()
2 {
3     $this->calc->setOperands(3, 3, 3);
4     $this->calc->setOperation(new Multiplication());
5     $result = $this->calc->calculate();
6
7     $this->assertEquals(27, $result);
8 }
```

Only then, when we have a failing test, would we work on the production code. Right?

```
1 class Multiplication implements Operation {
2     public function run($num, $current)
3     {
4         // Any number times 0 is 0
5         if ($current === 0) return $num;
6
7         return $current * $num;
8     }
9 }
```

That's it! A simple bit of logic, and the tests pass.

Mocks

If you've been watching closely, you might have noticed that we broke one of our rules. If unit testing requires isolation, we're cheating when the calculate method calls those other classes. It's important to realize that there are two schools of thought on this issue (often labeled Chicago vs. London TDD, as noted earlier in this chapter). My personal recommendation is to push for as much isolation as possible. We've been somewhat lax in this chapter, only because I'm trying to ease you into the process of testing. We haven't yet gotten to stubs and mocks.

Having said that, if you want a quick demonstration for how you might test this class in true isolation, see below. Please note that this code leverages a popular PHP mocking framework, called Mockery. An entire chapter of the book is dedicated to it, so you'll come to know it well!

```
1 public function testAddsNumbers()
2 {
3     // Mock all outside objects.
4     // We're not interested in testing those.
5     // They should have their own tests.
6     $mock = Mockery::mock('Addition');
7
8     // All we care about is verifying that
9     // the proper method was called.
10    $mock->shouldReceive('run')
11        ->once()
12        ->with(5, 0)
13        ->andReturn(5);
14
15    $this->calc->setOperands(5);
16
17    // Rather than new Addition, we
18    // pass in the mock object
19    $this->calc->setOperation($mock);
20
21    // And then it's business per usual!
22    $result = $this->calc->calculate();
23
24    $this->assertEquals(5, $result);
```

```
25 }
```

Remember: this methodology is only effective if the class that has been mocked, too, has tests! If we can prove that the `Addition` class works perfectly, then there's absolutely no reason to test it twice.

```
1 <?php
2
3 class AdditionTest extends PHPUnit_Framework_TestCase {
4
5     public function testFindsTheSumOfNumbers()
6     {
7         $addition = new Addition;
8         $sum = $addition->run(5, 0);
9
10        $this->assertEquals(5, $sum);
11    }
12
13 }
```

Or, another way of thinking about this is, if the `Addition` class does happen to break at some point, because we used mocks, the `Calculator` class will still return green. That's what we want.



Tip: An added bonus to using mocks in situations such as this is that they allow you to drastically reduce the number of tests. When mocking the operation class, we no longer need tests in `CalculatorTest` for verifying addition, subtraction, multiplication, *etc.* Each of those implementations will have its own tests.

Project Complete

With just a bit of effort, we have a well-tested class for performing basic arithmetic. You'd certainly want to add more to this, but I'll leave that part in your hands as home-work.

So what was this all for? Why dedicate so much of a unit testing chapter to understanding polymorphism? When building testable applications, I want you

to break each class down to its core responsibility. An `Addition` class is responsible for adding numbers; a `Subtract` class is responsible for subtracting. Not only does this approach allow for cleaner and more extensible code, but it also makes the process of testing significantly easier. Memorize this:

The smaller the class, the easier it is to test.

Hopefully, even though we used a generic calculator as the example, you were able to learn a variety of new techniques, including unit testing, the single responsibility principle, polymorphism, and a touch of Mockery. But, there's much more to learn. As [Happy](#) would say, we've only just begun.

Final Source

First, the `CalculatorTest`. Don't forget: once you learn more about mocks and stubs, the tests below can be improved, refactored, and reduced. Refer to the **Mocks** section earlier in this chapter for a teaser.

```
1 <?php // CalculatorTest.php
2
3 class CalculatorTest extends PHPUnit_Framework_TestCase {
4
5     public function setUp()
6     {
7         $this->calc = new Calculator;
8     }
9
10    public function testResultDefaultsToZero()
11    {
12        $this->assertSame(0, $this->calc->getResult());
13    }
14
15    public function testAddsNumbers()
16    {
17        $this->calc->setOperands(5);
18        $this->calc->setOperation(new Addition);
19        $result = $this->calc->calculate();
20
21        $this->assertEquals(5, $result);
```

```

22     }
23
24     /**
25      * @expectedException InvalidArgumentException
26     */
27     public function testRequiresNumericValue()
28     {
29         $this->calc->setOperands('five');
30         $this->calc->setOperation(new Addition);
31         $this->calc->calculate();
32     }
33
34     public function testAcceptsMultipleArgs()
35     {
36         $this->calc->setOperands(1, 2, 3, 4);
37         $this->calc->setOperation(new Addition);
38         $result = $this->calc->calculate();
39
40         $this->assertEquals(10, $result);
41         $this->assertNotEquals(
42             'Snoop Doggy Dogg and Dr. Dre is at the door',
43             $result
44         ); // jokes, jokes, jokes
45     }
46
47     // Notice how these next two tests are redundant, now that
48     // we're using polymorphism. They're unnecessary.
49     public function testSubtractsNumbers()
50     {
51         $this->calc->setOperands(4);
52         $this->calc->setOperation(new Subtraction);
53         $result = $this->calc->calculate();
54
55         $this->assertEquals(-4, $result);
56     }
57
58     public function testMultipliesNumbers()
59     {
60         $this->calc->setOperands(3, 3, 3);

```

```

61         $this->calc->setOperation(new Multiplication);
62         $result = $this->calc->calculate();
63
64     $this->assertEquals(27, $result);
65 }
66
67 }

```

And the calculator class, itself. Please note that the various classes have been grouped for convenience. In your projects, each class should be contained within its own file. The only exception to this rule, in this author's opinion, is when you have a class that is only applicable within the context of another. This can be applied to exception classes and value objects.

```

1 <?php // Calculator.php
2
3 interface Operation {
4     public function run($num, $current);
5 }
6
7 class Addition implements Operation {
8     public function run($num, $current)
9     {
10         return $current + $num;
11     }
12 }
13
14 class Multiplication implements Operation {
15     public function run($num, $current)
16     {
17         // Any number times 0 is 0
18         if ($current === 0) return $num;
19
20         return $current * $num;
21     }
22 }
23
24 class Subtraction implements Operation {

```

```
25
26     public function run($num, $current)
27     {
28         return $current - $num;
29     }
30 }
31
32 class Calculator {
33
34     protected $result = 0;
35
36     public function getResult()
37     {
38         return $this->result;
39     }
40
41     public function setOperands()
42     {
43         $this->operands = func_get_args();
44     }
45
46     public function setOperation(Operation $operation)
47     {
48         $this->operation = $operation;
49     }
50
51     public function calculate()
52     {
53         foreach ($this->operands as $num)
54         {
55             if ( ! is_numeric($num))
56                 throw new InvalidArgumentException();
57
58             $this->result = $this->operation->run($num,
59 $this->result);
60         }
61
62         return $this->result;
63     }
64 }
```

63

64 }

Summary

Plenty of unit testing introductions will overwhelm you with theory and jargon. But who does this serve? It's not dissimilar to teaching basic Math by beginning with Geometry - it's simply not the right way to go about things. I'm reminded of a quote from the movie, Contact.

“This was just the first step; in time you’ll take another.”

Yes, folks, the movie quotes will continue. This subject matter is just too dry not to sporadically throw in some geeky movie references.

Chapter 6: Testing Models

You've surely heard the phrase, "*fat model, skinny controller*" before. If not, this is simply a pattern which dictates that it's better to keep your controllers as minimal as possible: handle responses, send messages to the domain, load views. I often compare controllers to the cops who direct traffic, when the power goes out - because that's what they do: direct traffic. Calculations or operations of any real complexity should be placed within a model or service. For this reason, it's crucial that your models be fully tested.

What to Test

I've drilled this into your head multiple times now. The basic rule is that everything that has an opportunity to break should be tested. More directly, plan on, at the very least, testing:

- Validations
- Scopes
- Accessors and Mutators
- Associations/Relationships
- Custom Methods

Accessors and Mutators

Though opinions vary, my recommendation is to omit tests for basic things, such as getters and setters. However, there's one exception to this rule: if any logic is performed - beyond "*set this property on the object*" - then, yes, do prepare tests to verify that it works as expected.

In Laravel, to transform a model's attributes when getting or setting, we use the following syntax (using age as the name of the attribute):

```
1 // Accessor
2 public function getAgeAttribute($age) {}
3
4 // Mutator
```

```
5 public function setAgeAttribute($age) {}
```

This can be particularly helpful when a value needs to be manipulated in some way. Here are a couple of examples.

Cat Years Example

What if, for some ridiculous reason, you had a `Cat` model, and wanted to automatically translate a provided age (in *human years*) into *cat years* before saving to the database? This is a perfect use case for mutators.

```
1 <?php
2
3 class Cat extends Eloquent {
4     public function setAgeAttribute($age)
5     {
6         $this->attributes['age'] = $age * 7;
7     }
8 }
```

Now, when age is set on the model, it will first be filtered through this method, which multiplies the age by seven (*the ratio of cat to human years*). In effect:

```
1 $cat = new Cat;
2 $cat->age = 6;
3
4 echo $cat->age; // 42
```

If you're thinking that this would be easy to test, you'd be right!

```
1 public function testAutomaticallyCompensatesForCatYears()
2 {
3     $cat = new Cat;
4     $cat->age = 6;
5
6     $this->assertEquals(42, $cat->age); // true
7 }
```

Password Hashing Example

As another example, let's say that we want to remove the need to manually hash a user's password within the controller. Well, once again, a mutator will handle this task quite nicely.

```
1 <?php
2
3 class User extends Eloquent {
4     public function setPasswordAttribute($password)
5     {
6         $this->attributes['password'] = Hash::make($password);
7     }
8 }
```

Testing this method is only slightly more tricky. Because the mutator calls `Hash::make`, if we want to test in isolation, then that class needs to be mocked. Luckily, because it's a facade, doing so is trivial.

```
1 public function testHashesPasswordWhenSet()
2 {
3     Hash::shouldReceive('make')->once()->andReturn('hashed');
4
5     $author = new Author;
6     $author->password = 'foo';
7
8     $this->assertEquals('hashed', $author->password);
9 }
```

This test uses a mock as a way of saying:

"I expect the `make` method on the `Hash` class to be triggered one time. When this occurs, rather than calling the original method, I'll just return the string, `hashed`.

This technique offers three advantages:

1. We continue testing in complete isolation from external objects.
2. If the mutator does not call `Hash::make`, the expectation will fail, and we'll

be notified immediately upon running `phpunit`.

3. Testing `Hash::make` can be tricky, because its return value will be somewhat random. Though you could, perhaps, ensure that the length of the mutated password is sixty characters (matching a hash's length), why bother when you can instead mock the dependency entirely?

Custom Methods

Of course, there will also be various custom methods within your model that should be tested. Approach these in the same way that you would any other test. For example, let's assume that an `Article` model should offer a way to retrieve meta information in readable form. This time, let's write the test first.

```
1 <?php
2
3 class ArticleTest extends TestCase {
4     public function testGetsReadableMetaData()
5     {
6         $article = new Article;
7         $article->title = 'My First Article';
8         $article->author = 'Perd Hapley';
9
10        $this->assertEquals(
11            '"My First Article" was written by Perd Hapley.',
12            $article->meta()
13        );
14    }
15 }
```

So, when a `meta` method is called for the given model, we expect the string, “*My First Article*” was written by *Perd Hapley*. to be returned.

The production code to make this test pass simply needs to fetch a couple attribute values and format them as a sentence. Rather than dealing with a bunch of concatenation, let's stick with PHP's native `sprintf` function.

```
1 <?php
2
3 class Article extends Eloquent {
```

```
4     public function meta()
5     {
6         return sprintf(
7             '"%s" was written by %s.',
8             $this->title,
9             $this->author
10        );
11    }
12 }
```

Testing is easy!

Validations

Another relatively simple way to ease into the process of testing models is to test validations. As an example, to create a new author, a name field or property must be present. Otherwise, it's invalid, right? How might we verify that in a test?

```
1 // app/tests/models/AuthorTest.php
2
3 public function testIsInvalidWithoutAName()
4 {
5     $author = new Author;
6
7     $this->assertFalse($author->validate());
8 }
```

Notice how the name of the test describes our desired behavior for the model.

Running `phpunit` will, of course, fail, alerting you that an `Author` model does not exist.

```
1 PHP Fatal error:  Class 'Author' not found
```

There's the next step: create the model.

```
1 <?php // app/models/Author.php
2
```

```
3 class Author extends Eloquent {}
```



Tip: When using my [generators tool](#), I prefer to create bash aliases, such as `g:m` to generate a model, `g:mig` for a migration, etc. As such, to create a new Author model, I would simply run `g:m author` from the Terminal.

Should you run the tests again, you'll be notified that a `validate` method does not exist on the model. Let's add this to a `BaseModel` class that all models can inherit from. But, before continuing on with this current test, we need to ensure that this new `validate` method works correctly. Because we haven't yet fully gone over mocks and stubs, don't worry if some of the following code appears foreign to you.

```
1 <?php // app/tests/BaseModelTest.php
2
3 class BaseModelTest extends TestCase {
4
5     protected $model;
6
7     public function setUp()
8     {
9         parent::setUp();
10
11         $this->model = $model = new BaseModel;
12         $model::$rules = ['title' => 'required'];
13     }
14
15     public function testReturnsTrueIfValidationPasses()
16     {
17         Validator::shouldReceive('make')->once()->andReturn(
18             Mockery::mock(['passes' => true])
19         );
20
21         $this->model->title = 'Foo Title';
22         $result = $this->model->validate();
23
24         $this->assertEquals(true, $result);
```

```

25     }
26
27     public function testSetsErrorsOnObjectIfValidationFails()
28     {
29         Validator::shouldReceive('make')->once()->andReturn(
30             Mockery::mock(['passes' => false, 'messages' =>
31             'messages'])
32         );
33
34         $result = $this->model->validate();
35
36         $this->assertEquals(false, $result);
37         $this->assertEquals('messages', $this->model->errors);
38     }

```

Again, notice how the names of the tests describe how the code should behave. This is preferred over `testValidate`. These two tests account for both paths through the method: successful and failed validation. Because we're testing in isolation, it's paramount that we refrain from triggering Laravel's native `Validator` class. Or, in other words, we don't need proof that that class works correctly. Taylor has already tested it. As such, we can mock it. You'll learn all of this in the Mockery-specific chapter shortly.

Here's the associated code to make those two tests pass:

```

1 <?php // app/models/BaseModel.php
2
3 class BaseModel extends Eloquent
4 {
5     public $errors;
6
7     public function validate()
8     {
9         $v = Validator::make($this->attributes,
10 static::$rules);
11
12         if ($v->passes()) return true;
13

```

```
13     $this->errors = $v->messages();
14
15     return false;
16 }
17 }
```

This validate method consists of fairly boilerplate code to validate the current set of attributes against a static rules property that should exist on each model.

Now that a validate method is available to all models, we can return to the former test. Here it is again for reference.

```
1 // app/tests/models/AuthorTest.php
2
3 public function testIsInvalidWithoutAName()
4 {
5     $author = new Author;
6
7     $this->assertFalse($author->validate());
8 }
```

And its related production code, which now inherits from the newly added BaseModel.

```
1 // app/tests/models/AuthorTest.php
2
3 class Author extends BaseModel {
4     public static $rules = [];
5 }
```

At this point, PHPUnit should, as expected, fail.

```
1 1) AuthorTest::testIsInvalidWithoutAName
2 Failed asserting that true is false.
```

When making use of the generic assertTrue and assertFalse methods, always provide a custom message to avoid meaningless failures, like *Failed asserting that true is false*.

```
1 $this->assertFalse($author->validate(), 'Expected validation to fail.');
```

Although we expected the validation to fail, because no rules have been set, it did not. To fix that, update the `$rules` property on the `Author` model, as needed.

```
1 <?php // app/models/Author.php
2
3 class Author extends BaseModel {
4     protected static $rules = [
5         'name' => 'required'
6     ];
7 }
```

This should return us to green! Let's try a couple more. An email address should be required, too.

```
1 // app/tests/models/AuthorTest.php
2
3 public function testIsInvalidWithoutAValidEmail()
4 {
5     // Set fixture
6     $author = new Author;
7     $author->name = 'Joe';
8     $author->email = 'foo';
9
10    $this->assertFalse($author->validate(), 'Expected validation to fail.')\
11 ;
12 }
```

Once again, we're back to a failing test. Not only should an email address be set, but it should adhere to the format of a real email address.

```
1 <?php // app/models/Author.php
2
3 class Author extends BaseModel {
4     protected static $rules = [
```

```
5     'name'    => 'required',
6     'email'   => 'required|email'
7 ];
8 }
```



Tip: Refer to the [Laravel documentation](#) for a full list of available validation rules.

If you think about it, a valid email is not enough; we should also ensure that it's a unique one. We don't want multiple authors with the same email address. Write a test (which touches the database once) to verify that this never happens.

```
1 // app/tests/models/AuthorTest.php
2
3 public function testIsValidWithoutUniqueEmail()
4 {
5     // Set fixture
6     $author = new Author;
7     $author->name = 'Joe';
8     $author->email = 'joe@example.com';
9     $author->save();
10
11    // Now, try to insert a new author
12    // with the same email. We don't want that.
13    $author = new Author;
14    $author->name = 'Frank';
15    $author->email = 'joe@example.com';
16
17    // Ensure that validation fails, because there already is
18    // an author with that email registered.
19    $this->assertFalse($author->validate(), 'Expected
validation to fail.')\
20 ;
21 }
```

Fixtures: Think of a fixture as a dummy record that can be inserted into a table for the purposes of testing.



Laravel offers a `unique:TABLENAME` validation rule that will serve our needs nicely here.

```
1 <?php // app/models/Author.php
2
3 class Author extends BaseModel {
4     protected static $rules = [
5         'name' => 'required',
6         'email' => 'required|email|unique:authors'
7     ];
8 }
```

That should do it. Back to green.

```
1 OK (3 tests, 3 assertions)
```



Warning: One glaring problem with this approach is that, as you add additional fields, it can lead to false positives in your tests. It's better to begin with a complete model, and then update individual fields, as needed. More on that later in this chapter.

Helpers

Because, as you might imagine, testing validations is so common, it makes sense to abstract this away to a custom assertion, such as `assertValid` and `assertNotValid`. If working with PHP 5.4 or higher, traits are a helpful choice for storing these sorts of mixins.

```
1 <?php // app/tests/helpers/ModelHelpers.php
2
3 trait ModelHelpers {
4     public function assertValid($model)
5     {
6         $this->assertTrue(
7             $model->validate(),
```

```

8             'Model did not pass validation.'
9         );
10    }
11
12    public function assertNotValid($model)
13    {
14        $this->assertFalse(
15            $model->validate(),
16            'Did not expect model to pass validation.'
17        );
18    }
19 }

```

We've introduced a new `app/tests/helpers` directory, so don't forget to update composer, and then `composer dump-autoload`.

```

1 // composer.json
2
3 "autoload": {
4     "classmap": [
5         // ...
6         "app/tests/TestCase.php",
7         "app/tests/helpers"
8     ]
9 }

```

The wonderful thing about traits is that they can easily be imported into existing classes, via the `use` keyword. To pull these two mixins into `AuthorTest`, it's as easy as doing:

```

1 class AuthorTest extends TestCase {
2     use ModelHelpers;
3
4     // ...
5 }

```

Now, this more readable assertion syntax may be used for testing validations.

```
1 // app/tests/models/AuthorTest.php
2
3 public function testIsInvalidWithoutAName()
4 {
5     $author = new Author;
6
7     $this->assertNotValid($author);
8 }
9
10 public function testIsInvalidWithoutAValidEmail()
11 {
12     $author = new Author;
13     $author->name = 'Joe';
14
15     $this->assertNotValid($author);
16 }
```

Factories

Up until this point, we've resorted to the somewhat cumbersome task of manually building test objects.

```
1 $author = new Author;
2 $author->name = 'Joe';
3 $author->email = 'joe@example.com';
```

Surely, there must be an easier way. Ideally, we could leverage a factory, which will handle the process of dynamically building the attributes for a model. Further, it should follow the clean Laravel style, such as:

```
1 $author = Factory::author();
```

Or, to create an object with all of the necessary fields, and insert it into the DB:

```
1 Factory::create('author');
```

Finally, we also need to have the option of overriding one or more fields' default value.

```
1 Factory::create('author', ['email' => 'foo@foo.com']);
```

Though building a tool like this from scratch is beyond the scope of this book, I've already done the work for you! [It's available on Packagist](#). Update your composer.json file as detailed on the Packagist page, pull in the dependencies, and, now, the tests may be updated to:

```
1 <?php // app/tests/models/AuthorTest.php
2
3 use Way\Tests\Factory;
4
5 class AuthorTest extends TestCase {
6     use ModelHelpers;
7
8     public function testIsInvalidWithoutAName()
9     {
10         $author = Factory::author(['name' => null]);
11
12         $this->assertNotValid($author);
13     }
14
15     public function testIsInvalidWithoutAValidEmail()
16     {
17         $author = Factory::author(['email' => 'foo']);
18
19         $this->assertNotValid($author);
20     }
21
22     public function testIsInvalidWithoutUniqueEmail()
23     {
24         $author = Factory::create('author', ['email' =>
25             'joe@example.com'])\;
26
27         // Now try to insert a new author with the same email.
28         $author = Factory::author(['email' =>
29             'joe@example.com']);
```

```
30         // Let's make sure that it fails.
31         $this->assertNotValid($author);
32     }
33 }
34 }
```

Much cleaner!

Laravel Test Helpers

The package that you just installed eases the process of writing tests for Laravel applications, by offering:

- A Factory utility (quickly create and populate models)
- Various Model test helpers (`assertValid`, `assertBelongsTo`, etc.)
- Assert and Should PHPUnit wrappers. We reviewed these in Chapter 4.

Factories

Ever found yourself repeatedly creating test models over and over?

```
1 $user = new User;
2 $user->email = 'foo@example.com'
3 $user->name = 'Joe';
```

This can very quickly flood your test classes. Instead, use a factory!

```
1 <?php
2
3 use Way\Tests\Factory;
4
5 class UserTest extends TestCase {
6
7     public function testBasicExample()
8     {
9         $user = Factory::attributesFor('User');
10    }
11 }
```

Now, the `$user` variable will be equal to random data that fits the data types for each field. Something like:

```
1 .array(6) {
2   'id' =>
3   NULL
4   'name' =>
5   string(3) "Kim"
6   'email' =>
7   string(15) "kim@example.com"
8   'age' =>
9   int(26)
10  'created_at' =>
11  string(19) "2013-05-01 02:21:49"
12  'updated_at' =>
13  string(19) "2013-05-01 02:21:49"
14 }
```

Overrides

There will be times, though, when you need to specify values for some fields. This can be particularly helpful for validation, where, say, a model should be invalid, unless an email is provided.

```
1 $user = Factory::attributesFor('User', ['email' => null]);
```

Any fields specified in the second argument will override the random defaults.

Models

The static `attributesFor` method is great for fetching an array of attributes. If you want the full Laravel collection, then you have two options:

```
1 $user = Factory::user();
```

This technique uses `callStatic` to allow for the readable syntax shown above. This works, if the model is in the global namespace, but if it's not, you'll want to use the `make` method.

```
1 $user = Factory::make('Models\User');
```

If you happen to be working with a real test database, you can also use the `create` method, which will instantiate the model, fill it with dummy data, and save it to the database.

```
1 $user = Factory::create('User');
```

Test Helpers

This package also includes a growing list of test helpers.

`assertValid` and `assertNotValid`

```
1 <?php
2
3 use Way\Tests\Factory;
4
5 class UserTest extends TestCase {
6     use Way\Tests\ModelHelpers;
7
8     public function testIsInvalidWithoutName()
9     {
10         $user = Factory::user(['name' => null]);
11
12         $this->assertNotValid($user);
13     }
14
15 }
```

All model test helpers are stored as traits. This makes them super easy to import into our test class. Simply add `use Way\Tests\ModelHelpers;` to the top of the class, and you should be good to go.

In the example above, we are asserting that a `User` model should be invalid, unless its `name` field is not empty.

Currently, the assertion will look for a `validate` method on the model.

Asserting Relationships

There are also various assertions for Laravel relationships. Let's assert that a User model has many Posts.

```
1 public function testHasManyPosts()
2 {
3     $this->assertHasMany('posts', 'User');
4 }
```

Running phpunit will return:

```
bash 3) UserTest::testHasManyPosts Expected the 'User' class to
have method, 'posts'. Failed asserting that false is true.
```

Go ahead and add a posts method to User.

```
1 public function posts()
2 {
3     $this->hasMany('Post');
4 }
```

And now we're back to green.

Currently, you can assert:

- assertBelongsTo
- assertHasOne
- assertHasMany

Summary

Arguments can be made for not testing controllers, where not a huge amount of logic is being performed, but, when it comes to models, it's vital that you not cut any corners.

Chapter 7: Easier Testing With Mockery

It's an unfortunate truth that, while the basic principle behind testing is quite simple, fully introducing this process into your day-to-day coding workflow is more difficult than you might hope. The various jargon alone can prove overwhelming! Luckily, a variety of tools have your back, and help to make the process as simple as it can be. [Mockery](#), the premier mock object framework for PHP, is one such tool!

In this article, we'll dig into what test doubles are, why they're useful, and how to integrate Mockery into your testing workflow.

Mocking Decoded

A mock object is nothing more than a bit of test jargon that refers to simulating the behavior of real objects. In simpler terms, often, when testing, you won't want to execute a particular method. Instead, you simply need to ensure that it was, in fact, called.

Perhaps an example is in order. Imagine that your code triggers a method that will log a bit of data to a file. When testing this logic, you certainly don't want to physically touch the file system. This has the potential to drastically decrease the speed of your tests. In these situations, it's best to mock your file system class, and, rather than manually read the file to prove that it was updated, merely ensure that the applicable method on the class was, in fact, called. This is mocking! There's nothing more to it than that; simulate the behavior of objects.

Remember: jargon is just jargon. Never allow an initially confusing piece of terminology to deter you from learning a new skill.

Particularly as your development process matures - including embracing the single responsibility principle and leveraging dependency injection - a familiarity with mocking will quickly become essential.

Mocks vs Stubs:

MOCKS VS. STUBS

You'll frequently hear the terms, *mock* and *stub*, thrown about interchangably. In fact, the two serve different purposes. The former refers to the process of defining expectations and ensuring desired behavior. In other words, a mock can potentially lead to a failed test. A stub, on the other hand, is simply a dummy set of data that can be passed around to meet certain criteria.

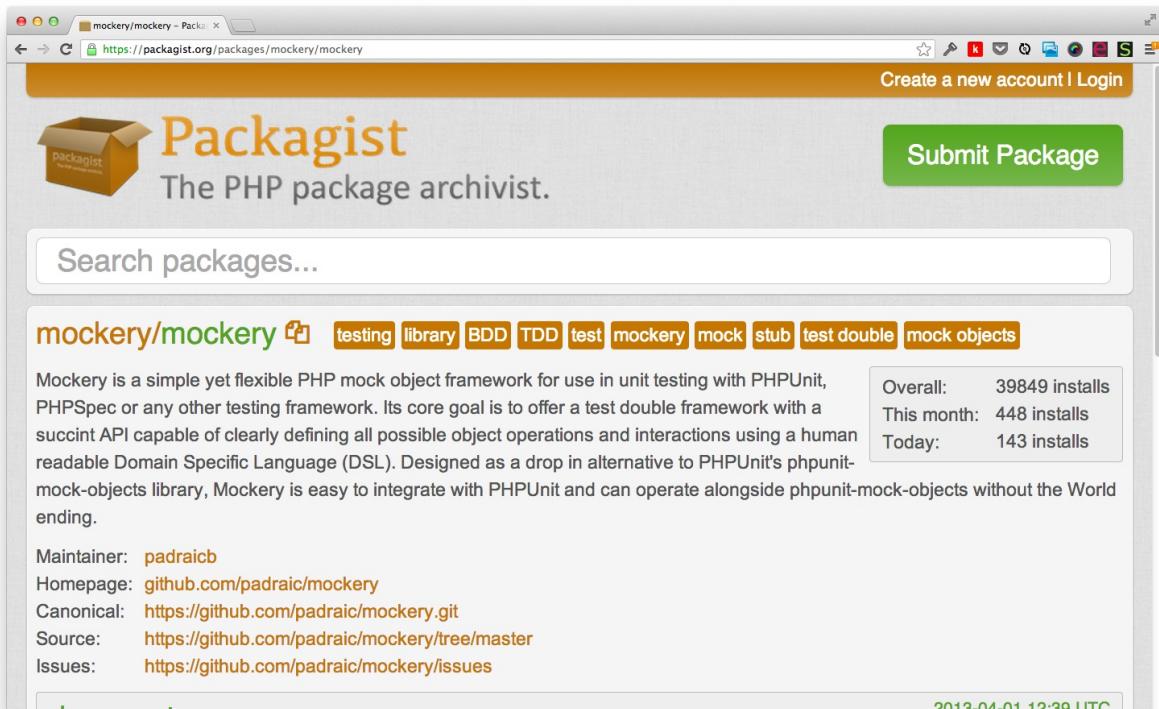
As you'll find, there are multiple ways to *fake* data, each of which has its advantages, dependent upon the scenario. As a group, they are referred to as test doubles.

The defacto testing library for PHP, PHPUnit, ships with its own API for mocking objects; however, unfortunately, it can prove cumbersome to work with. As you're surely aware, the more difficult testing is, the more likely it is that the developer simply (and sadly) won't.

Luckily, a variety of third-party solutions are available through Packagist (Composer's package repository), which allow for increased readability, and, more importantly, *writability*. Among these solutions - and most notable of the set - is Mockery, a framework-agnostic mock object framework.

Designed as a drop-in alternative for those who are overwhelmed by PHPUnit's mocking verbosity, Mockery is a simple, but powerful utility. As you'll surely find, in fact, it's the industry standard for modern PHP development.

Installation



Like most modern PHP tools, Mockery may be installed with Composer.

Like most PHP tools these days, the recommended method to install Mockery is through Composer (though it's available through Pear too).

Wait, what's this [Composer](#) thing? It's the PHP community's preferred tool for dependency management. It provides an easy way to declare a project's dependencies, and pull them in with a single command. As a modern PHP developer, it's vital that you have a basic understanding of what Composer is, and how to use it.

If working along, for learning purposes, add a new composer.json file to an empty project and append:

```
1      {
2          "require-dev": {
3              "mockery/mockery": "dev-master"
4          }
5      }
```

This bit of JSON specifies that, for development, your application requires the Mockery library. From the command-line, a `composer install --dev` will pull in the package.

```
1 $ composer install --dev
2 Loading composer repositories with package information
3 Installing dependencies (including require-dev)
4   - Installing mockery/mockery (dev-master 5a71299)
5     Cloning 5a712994e1e3ee604b0d355d1af342172c6f475f
6
7 Writing lock file
8 Generating autoload files
```

As an added bonus, Composer ships with its own autoloader for free! Either specify a classmap of directories and `composer dump-autoload`, or follow the PSR-0 standard and adjust your directory structure to match. [Refer to Nettuts+](#) to learn more. If you're still manually requiring countless files in each PHP file, well, you just might be doing it wrong.

The Dilemma

Before we can implement a solution, it's best to first review the problem. Imagine that you need to implement a system for handling the process of generating content and writing it to a file. Perhaps the generator compiles various data, either from local file stubs, or a web service, and then that data is written to the file system.

If following the single responsibility principle - *which dictates that each class should be responsible for exactly one thing* - then it stands to reason that we should split this logic into two classes: one for generating the necessary content, and another for physically writing the data to a file. A Generator and File class, respectively, should do the trick.

Tip: Why not use `file_put_contents` directly from the Generator class? Well, ask yourself: “*How could I test this?*” There are techniques, such as monkey patching, which can allow you to overload these sorts of things, but, as a best practice, it's better to instead wrap such functionality up, so that it may easily be mocked with tools, like Mockery!

Here's a basic structure (with a healthy dose of pseudo code) for our Generator class.

```
1 <?php // src/Generator.php
2
3 class Generator {
4     protected $file;
5
6     public function __construct(File $file)
7     {
8         $this->file = $file;
9     }
10
11    protected function getContent()
12    {
13        // simplified for demo
14        return 'foo bar';
15    }
16
17    public function fire()
18    {
19        $content = $this->getContent();
20
21        $this->file->put('foo.txt', $content);
22    }
23 }
```

Dependency Injection

This code leverages what we refer to as dependency injection. Once again, this is simply developer jargon for injecting a class's dependencies through its constructor method, rather than hard-coding them.

Why is this beneficial? Because, otherwise, we wouldn't be able to mock the File class! Sure, we could mock the File class, but if its instantiation is hard-coded into the class that we're testing, there's no easy way to replace that instance with the mocked version.

```
1 public function __construct()
```

```
2 {
3     // anti-pattern
4     $this->file = new File;
5 }
```

The best way to build testable application is to approach each new method call with the question, “*How might I test this?*” While there are tricks for getting around this hard-coding, doing so is widely considered to be a bad practice. Instead, always inject a class’s dependencies through the constructor, or via setter injection.

Setter injection is more or less identical to constructor injection. The principle is exactly the same; the only difference is that, rather injecting the class’s dependencies through its constructor method, they’re instead done so through a setter method, like so:

```
1 public function setFile(File $file)
2 {
3     $this->file = $file;
4 }
```

A common criticism of dependency injection is that it introduces additional complexity into an application, all for the sake of making it more testable. Though the complexity argument is debatable in this author’s opinion, if you should prefer, you can allow for dependency injection, while still specifying fallback defaults. Here’s an example:

```
1 class Generator {
2     public function __construct(File $file = null)
3     {
4         $this->file = $file ?: new File;
5     }
6 }
```

Now, if an instance of `File` is passed through to the constructor, that object will be used in the class. On the other hand, if nothing is passed, the `Generator` will *fall back* to manually instantiating the applicable class. This allows for such variations as:

```
1 # Class instantiates File
2 new Generator;
3
4 # Inject File
5 new Generator(new File);
6
7 # Inject a mock of File for testing
8 new Generator($mockedFile);
```

Continuing on, for the purposes of this tutorial, the `File` class will be nothing more than a simple wrapper around PHP's `file_put_contents` function.

```
1 <?php // src/File.php
2
3 class File
4 {
5     /**
6      * Write data to a given file
7      *
8      * @param string $path
9      * @param string $content
10     * @return mixed
11    */
12    public function put($path, $content)
13    {
14        return file_put_contents($path, $content);
15    }
16 }
```

Rather simple, eh? Let's write a test to see, first-hand, what the problem is.

```
1 <?php // tests/GeneratorTest.php
2
3 class GeneratorTest extends PHPUnit_Framework_TestCase {
4     public function testItWorks()
5     {
6         $file = new File;
7         $generator = new Generator($file);
```

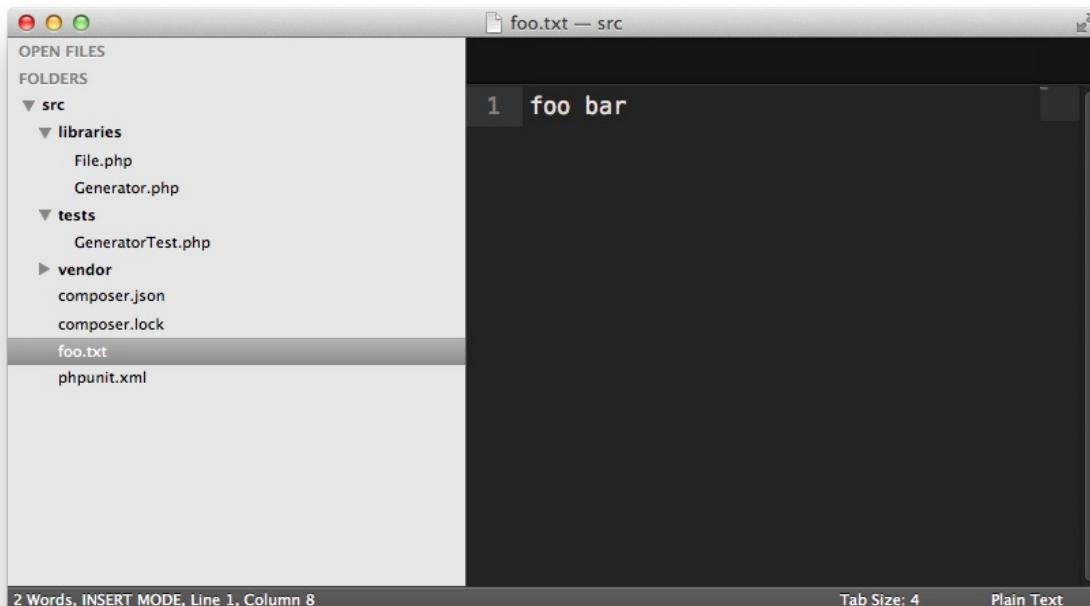
```
8
9         $generator->fire();
10    }
11 }
```

Please note that these examples assume that the necessary classes are being autoloaded with Composer. Your `composer.json` file optionally accepts an `autoload` object, where you may specify which directories or classes to autoload. No more messy `require` statements!

If working along, running `phpunit` will return:

```
1 OK (1 test, 0 assertions)
```

It's green; that means we can move on to the next task, right? Well, not exactly. While it's true that the code does, indeed, work, each time this test is run, a `foo.txt` file will be created on the file system. What about when you've written dozens more tests? As you can imagine, very quickly, your test's speed of execution will stutter.



A screenshot of a terminal window titled "foo.txt — src". The window shows the file structure of a project. On the left, the "OPEN FILES" and "FOLDERS" panes are visible, showing a directory tree with "src" containing "libraries" (with "File.php" and "Generator.php") and "tests" (with "GeneratorTest.php"). Below "src" is a "vendor" folder containing "composer.json" and "composer.lock". On the right, the main pane displays the contents of "foo.txt", which contains the text "1 foo bar". At the bottom of the terminal window, status information is shown: "2 Words, INSERT MODE, Line 1, Column 8", "Tab Size: 4", and "Plain Text".

image

“Though the tests pass, they’re incorrectly touching the filesystem.”

Still not convinced? If reduced testing speed won’t sway you, then consider common sense. Think about it: we’re testing the Generator class; why do we have any interest in executing code from the File class? It should have its own tests! Why the heck would we double up?

The Solution

Hopefully, the previous section provided the perfect illustration for why mocking is essential. As was noted earlier, though we could make use of PHPUnit native API to serve our mocking requirements, it’s not overly enjoyable to work with. To illustrate this truth, here’s an example for asserting that a mocked object should receive a method, getName and return John Doe.

```
1 public function testNativeMocks()
2 {
3     $mock = $this->getMock('SomeClass');
4     $mock->expects($this->once())
5         ->method('getName')
6         ->will($this->returnValue('John Doe'));
7 }
```

While it gets the job done - asserting that a getName method is called once, and returns *John Doe* - PHPUnit’s implementation is confusing and verbose. With Mockery, we can drastically improve its readability.

```
1 public function testMockery()
2 {
3     $mock = Mockery::mock('SomeClass');
4     $mock->shouldReceive('getName')
5         ->once()
6         ->andReturn('John Doe');
7 }
```

Notice how the latter example reads (and speaks) better.

Continuing with the example from the previous “*Dilemma* section, this time, within the GeneratorTest class, let’s instead mock - or simulate the behavior of

- the File class with Mockery. Here's the updated code:

```
1 <?php
2
3 class GeneratorTest extends PHPUnit_Framework_TestCase {
4     public function tearDown()
5     {
6         Mockery::close();
7     }
8
9     public function testItWorks()
10    {
11        $mockedFile = Mockery::mock('File');
12
13        $mockedFile->shouldReceive('put')
14            ->with('foo.txt', 'foo bar')
15            ->once();
16
17        $generator = new Generator($mockedFile);
18        $generator->fire();
19    }
20 }
```

Confused by the `Mockery::close()` reference within the `tearDown` method? This static call cleans up the Mockery container used by the current test, and run any verification tasks needed for your expectations.

A class may be mocked using the readable `Mockery::mock()` method. Next, you'll typically need to specify which methods on this mock object you expect to be called, along with any applicable arguments. This may be accomplished, via the `shouldReceive(METHOD)` and `with(ARG)` methods.

In this case, when we call `$generator->fire()`, we're asserting that it should call the `put` method on the `File` instance, and send it the path, `foo.txt`, and the data, `foo bar`.

```
1 // libraries/Generator.php
2
```

```
3 public function fire()
4 {
5     $content = $this->getContent();
6
7     $this->file->put('foo.txt', $content);
8 }
```

Because we're using dependency injection, it's now a cinch to instead inject the mocked `File` object.

```
1 $generator = new Generator($mockedFile);
```

If we run the tests again, they'll still return green, however, the `File` class - and, consequently, the file system - will never be touched! Again, there's no need to touch `File`. It should have its own tests! Mocking for the win!

Simple Mock Objects

Mock objects needn't always reference a class. If you only require a simple object, perhaps for a user, you might pass an array to the `mock` method - where, for each item, the key and value correspond to the method name and return value, respectively.

```
1 public function testSimpleMocks()
2 {
3     $user = Mockery::mock(['getFullName' => 'Jeffrey Way']);
4     $user->getFullName(); // Jeffrey Way
5 }
```

Return Values From Mocked Methods

There will surely be times, when a mocked class method needs to return a value. Continuing on with our Generator/File example, what if we need to ensure that, if the file already exists, it shouldn't be overwritten? How might we accomplish that?

The key is to use the `andReturn()` method on your mocked object to simulate different *states*. Here's an updated example:

```

1 public function testDoesNotOverwriteFile()
2 {
3     $mockedFile = Mockery::mock('File');
4
5     $mockedFile->shouldReceive('exists')
6         ->once()
7         ->andReturn(true);
8
9     $mockedFile->shouldReceive('put')
10        ->never();
11
12     $generator = new Generator($mockedFile);
13     $generator->fire();
14 }
```

This updated code now asserts that an exists method should be triggered on the mocked File class, and it should, for the purposes of this test's path, return true, signaling that the file already exists and shouldn't be overwritten. We next ensure that, in situations such as this, the put method on the File class is never triggered. With Mockery, this is easy, thanks to the never() expectation.

```

1 $mockedFile->shouldReceive('put')
2     ->never();
```

Should we run the tests again, an error will be returned:

```

1 Method exists() from File should be called
2 exactly 1 times but called 0 times.
```

Aha; so the test expected that \$this->file->exists() should be called, but that never happened. As a result, it failed. Let's fix it!

```

1 <?php
2
3 class Generator {
4     protected $file;
5
6     public function __construct(File $file)
```

```

7      {
8          $this->file = $file;
9      }
10
11     protected function getContent()
12     {
13         // simplified for demo
14         return 'foo bar';
15     }
16
17     public function fire()
18     {
19         $content = $this->getContent();
20         $file = 'foo.txt';
21
22         if (! $this->file->exists($file))
23         {
24             $this->file->put($file, $content);
25         }
26     }
27 }
```

That's all there is to it! Not only have we followed a TDD (test-driven development) cycle, but the tests are back to green!

It's important to remember that this style of testing is only effective if you do, in fact, test the dependencies of your class as well! Otherwise, though the tests may show green, for production, the code will break. Our demo this far has only ensured that Generator works as expected. Don't forget to test File as well!

Expectations

Let's dig a bit more deeply into Mockery's expectation declarations. You're already familiar with `shouldReceive`. Be careful with this, though; its name is a bit misleading. When left on its own, it does not require that the method should be triggered; the default is zero or more times (`zeroOrMoreTimes()`). To assert that you require the method to be called once, or potentially more times, a

handful of options are available:

```
1 $mock->shouldReceive('method')->once();
2 $mock->shouldReceive('method')->times(1);
3 $mock->shouldReceive('method')->atLeast()->times(1);
```

There will be times when additional constraints are necessary. As demonstrated earlier, this can be particularly helpful when you need to ensure that a particular method is triggered with the necessary arguments. It's important to keep in mind that the expectation will only apply if a method is called with these exact arguments.

Here's a few examples.

```
1 $mock->shouldReceive('get')->withAnyArgs()->once(); // the
default
2 $mock->shouldReceive('get')->with('foo.txt')->once();
3 $mock->shouldReceive('put')->with('foo.txt', 'foo bar')->once();
```

This can be extended even further to allow for the argument values to be dynamic in nature, as long as they meet a certain criteria. Perhaps we only wish to ensure that a string is passed to a method:

```
1 $mock->shouldReceive('get')->with(Mockery::type('string'))-
>once();
```

Or, maybe the argument needs to match a regular expression. Let's assert that any file name that ends with .txt should be matched.

```
1 $mockedFile->shouldReceive('put')
2         ->with('.txt$', Mockery::any())
3         ->once();
```

And as a final (but not limited to) example, let's allow for an array of acceptable values, using the anyOf matcher.

```
1 $mockedFile->shouldReceive('get')
2         ->with(Mockery::anyOf('log.txt', 'cache.txt'))
```

```
3     ->once();
```

With this code, the expectation will only apply if the first argument to the get method is log.txt or cache.txt. Otherwise, a Mockery exception will be thrown when the tests are run.

```
1 Mockery\Exception\NoMatchingExpectationException: No matching
handler found\
2 ...
```



Tip: Don't forget, you can always alias Mockery as `m` at the top of your class to make things a bit more succinct: use `Mockery` as `m`;. This allows for the more succinct, `m::mock()`.

Lastly, we have a variety of options for specifying what the mocked method should do or return. Perhaps we only need it to return a boolean. Easy:

```
1 $mock->shouldReceive('method')
2     ->once()
3     ->andReturn(false);
```

Partial Mocks

You may find that there are situations when you only need to mock a single method, rather than the entire object. Let's imagine, for the purposes of this example, that a method on your class references a custom global function (gasp) to fetch a value from a configuration file.

```
1 <?php
2
3 class MyClass {
4     public function getOption($option)
5     {
6         return config($option);
7     }
8
9     public function fire()
10    {
```

```
11     $timeout = $this->getOption('timeout');
12     // do something with $timeout
13 }
14 }
```

While there are a few different techniques for mocking global functions, nonetheless, it's best to avoid this method call all together. This is precisely when partial mocks come into play.

```
1 public function testPartialMockExample()
2 {
3     $mock = Mockery::mock('MyClass[getOption]');
4     $mock->shouldReceive('getOption')
5         ->once()
6         ->andReturn(10000);
7
8     $mock->fire();
9 }
```

Notice how we've placed the method to mock within brackets. Should you have multiple methods, simply separate them by a comma, like so:

```
1 $mock = Mockery::mock('MyClass[method1, method2]');
```

With this technique, the remainder of the methods on the object will trigger and behave as they normally would. Keep in mind that you must always declare the behavior of your mocked methods, as we've done above. In this case, when `getOption` is called, rather than executing the code within it, we simply return `10000`.

An alternative option is to make use of passive partial mocks, which you can think of as setting a default state for the mock object: all methods defer to the main parent class, unless an expectation is specified.

The previous code snippet may be rewritten as:

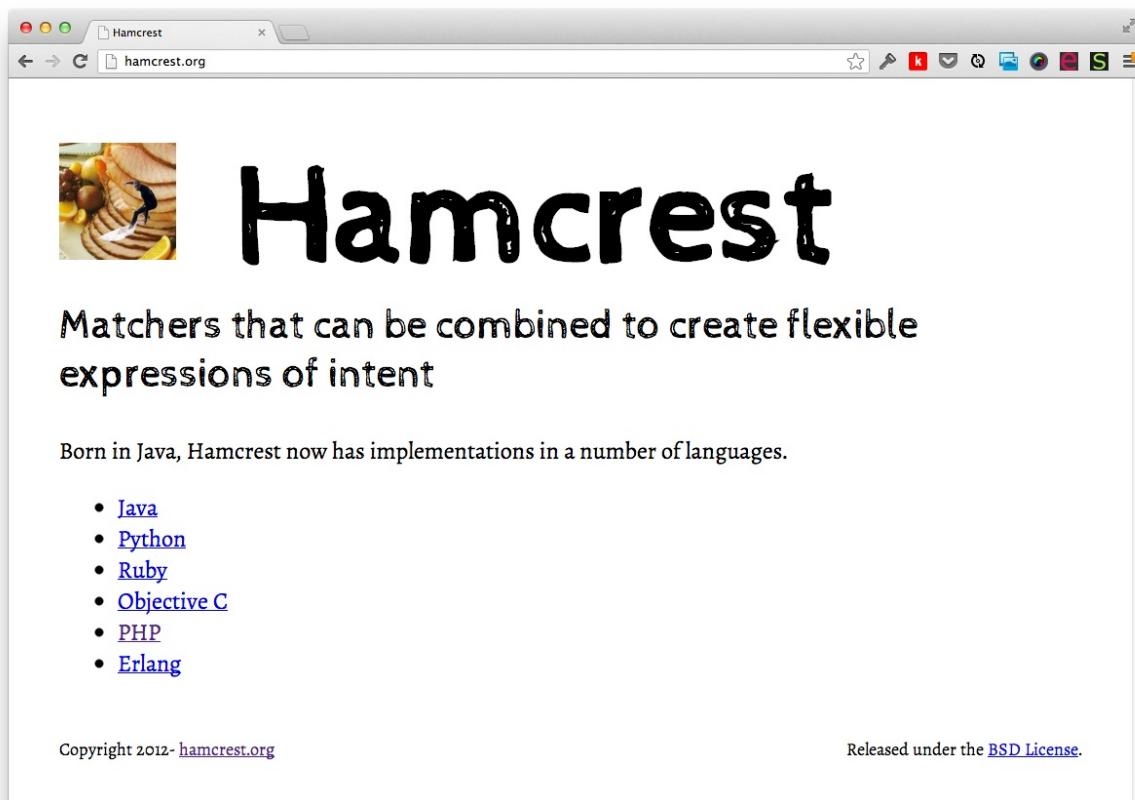
```
1 public function testPassiveMockExample()
2 {
```

```

3     $mock = Mockery::mock('MyClass')->makePartial();
4     $mock->shouldReceive('getOption')
5         ->once()
6         ->andReturn(10000);
7
8     $mock->fire();
9 }
```

In this example, all methods on MyClass will behave as they normally would, excluding getOption, which will be mocked and return 10000`.

Hamcrest



The Hamcrest library provides an additional set of matchers for defining expectations.

Once you've familiarized yourself with the Mockery API, it's recommended that you also leverage the Hamcrest library, which provides an additional set of matchers for defining readable expectations. Like Mockery, it may be installed through Composer.

through Composer.

```
1 "require-dev": {
2     "mockery/mockery": "dev-master",
3     "davedevelopment/hamcrest-php": "dev-master"
4 }
```

Once installed, you may use a more human-readable notation to define your tests. Below are a handful of examples, including slight variations that achieve the same end result.

```
1 <?php
2
3 class HamcrestTest extends PHPUnit_Framework_TestCase {
4     public function testHamcrestMatchers()
5     {
6         $name = 'Jeffrey';
7         $age = 28;
8         $hobbies = ['coding', 'guitar', 'chess'];
9
10        assertThat($name, is('Jeffrey'));
11        assertThat($name, is(not('Joe')));
12
13        assertThat($age, is(greaterThan(20)));
14        assertThat($age, greaterThan(20));
15
16        assertThat($age, is(integerValue()));
17
18        assertThat(new Foo, is(anInstanceOf('Foo')));
19
20        assertThat($hobbies, is(arrayValue()));
21        assertThat($hobbies, arrayValue());
22
23        assertThat($hobbies, hasKey('coding'));
24    }
25 }
```

Notice how Hamcrest allows you to write your assertions in as readable or terse a way as you desire. The use of the `is()` function is nothing more than syntactic

sugar to aid in readability.

You'll find that Mockery blends quite nicely with Hamcrest. For instance, with Mockery alone, to specify that a mocked method should be called with a single argument of type, `string`, you might write:

```
1 $mock->shouldReceive('method')
2     ->with(Mockery::type('string'))
3     ->once();
```

If using Hamcrest, `Mockery::type` may be replaced with `stringValue()`, like so:

```
1 $mock->shouldReceive('method')
2     ->with(stringValue())
3     ->once();
```

Hamcrest follows the `resourceValue` naming convention for matching the type of a value.

- `nullValue`
- `integerValue`
- `arrayValue`
- *rinse and repeat*

Alternatively, to match any argument, `Mockery::any()` may become `anything()`.

```
1 $file->shouldReceive('put')
2     ->with('foo.txt', anything())
3     ->once();
```

Summary

The biggest hurdle to using Mockery is, ironically, not the API, itself, but understanding why and when to use mocks in your testing.

The key is to learn and respect the single responsibility principle in your coding workflow. Coined by Bob Martin, the SRP dictates that a class “*should have one*,

and only one, reason to change." In other words, a class shouldn't need to be updated in response to multiple, unrelated changes to your application, such as modifying business logic, or how output is formatted, or how data may be persisted. In its simplest form, just like a method, a class should do one thing.

The `File` class manages file system interactions. A `MysqlDb` repository persists data. An `Email` class prepares and sends emails. Notice how, in none of these examples was the word, *and* used.

Once this is understood, testing becomes considerably easier. Dependency injection should be used for all operations that do not fall under the class's *umbrella*. When testing, focus on one class at a time, and mock all of its dependencies. You're not interested in testing them anyways; they have their own tests!

Though nothing prevents you from making use of PHPUnit's native mocking implementation, why bother when Mockery's improved readability is only a `composer update` away?

Chapter 8: Test Databases

There are those who will tell you that under no circumstances should your unit tests touch the database. In fact, I'm a strong proponent for this very thing. If a test can be executed adequately without hitting the database, then absolutely do it. This has the potential to drastically increase the speed of your tests, and is truer to what unit testing is.

Having said that, there may be times when this simply isn't possible for certain tests. Maybe you're working with a bit of legacy code, or maybe the simple fact of the matter is that you'll sleep better at night with a handful of tricky unit tests that run against the database. The world won't end if you break a few rules every once in a while. And, of course, the techniques outlined in this chapter will be useful for other styles of testing, such as integration tests.



Before moving forward, a word of caution: if you frequently come to the conclusion that a database call is necessary for your unit tests, then perhaps you should first consider redesigning. Is the class too tightly coupled to the database layer?

Test Databases

For the instances when it's necessary to work with a fixture (an object with faked values), you certainly don't want to be using the production database. Luckily, Laravel can exist and function in multiple *environments*.

For example, when triggering PHPUnit tests, the framework will automatically set the environment to *testing*. To specify custom configuration options on a per-environment basis, simply create a new folder within `app/config/` that has the same name as the environment that its contents should correspond to. Notice that Laravel already includes an `app/config/testing` folder. As such, any configuration file that is placed here will take precedence over its production sibling. In translation, the contents of `app/config/testing/database.php` will override `app/config/database.php`.

To specify a unique MySQL database for testing, you might write:

```
1 <?php // app/config/testing/database.php
2
3 return array(
4     'connections' => array(
5
6         'mysql' => array(
7             'driver'      => 'mysql',
8             'host'        => 'localhost',
9             'database'    => 'TEST_DB_NAME',
10            'username'   => 'USERNAME',
11            'password'   => 'PASSWORD',
12            'charset'    => 'utf8',
13            'collation'  => 'utf8_unicode_ci',
14            'prefix'     => '',
15        )
16
17    )
18 );
```

Don't forget that, to save time, you can copy and paste the applicable bits from `app/config/database.php`. In this case, it's only necessary to update the `mysql` connection to point to the new test database.

That's all it should take!

Specifying the Environment

As noted earlier, within the context of a PHPUnit test, the environment will automatically be set to testing. You can confirm this by viewing `app/tests/TestCase.php`.

```
1 // app/tests/TestCase.php
2
3 public function createApplication()
4 {
5     $unitTesting = true;
6
7     $testEnvironment = 'testing';
```

```
8
9     return require __DIR__.'....bootstrap/start.php';
10 }
```

However, the same will not be true when running Artisan commands.

For instance, imagine that you've added a new migration to create an authors table with a few fields. If using my popular [Laravel 4 Generators tool](#), this can be accomplished using a single command.

```
1 php artisan generate:migration create_posts_table --
2   fields="name:string, em\
3   ail:string:unique"
```

This command will generate the migration, as well as the necessary schema. However, upon migrating the database with `php artisan migrate`, you'll find that this still executes the migration on the production database. Of course it does! How was Laravel supposed to know that you expected it to be applied to the testing DB?

The `--env` option may be used to declare your desired environment.

```
1 php artisan migrate --env="testing"
```

Chances are, though, that you'll want to refresh the migrations for each test. How might we accomplish that?

Calling Artisan From Tests

If touching the database, it's important that each test executes with a clean slate, so to speak. Otherwise, there's the potential to fall into traps where a particular expectation fails, due to the previous test modifying the database table in a way that you didn't expect.

The solution is to, for each test, roll back the migrations and re-run them. Artisan offers the `migrate:refresh` command, which will handle this exact task. To trigger these commands from your code (try not to ever do this in your production code), use the `Artisan::call()` method, like so:

```
1 <?php // app/tests/models/AuthorTest.php
2
3 class AuthorTest extends TestCase {
4     public function setUp()
5     {
6         parent::setUp();
7
8         Artisan::call('migrate:refresh');
9     }
10 }
```

Simply pass the name of an Artisan command as the first argument to `Artisan::call()`, and it'll achieve the same end-result as:

```
1 php artisan COMMAND --env="testing"
```

You also may need to seed the database as well.

```
1 $this->seed();
```

Try It Out

Step 1: Create a book-testing database.

Step 2: Install the migrations table (only necessary once):

```
1 php artisan migrate:install --env="testing"
```

Step 3: Add a migration to create a new authors table.

```
1 php artisan generate:migration create_authors_table --
fields="name:string, \
2 email:string:unique".
```

This migration will, of course, be used for production, as well as testing.

Step 4: Add a dummy test to `app/tests/models/AuthorTest.php` to ensure that the `setUp()` method runs at least once.

```
1 <?php # app/tests/models/AuthorTest.php
2
3 class AuthorTest extends TestCase {
4     public function setUp()
5     {
6         parent::setUp();
7
8         Artisan::call('migrate:refresh');
9     }
10
11     public function testDummyToEnsureSetUpRuns()
12     {
13         $this->assertTrue(true);
14     }
15 }
```

Step 5: Run the tests: phpunit

If executed successfully, you'll see that the `CreateMigrationsTable` migration was executed, and the associated table was added to the test database. Success! You may now rest assured that each test will begin with a freshly reset and migrated database.

Databases in Memory

One popular technique for improving the performance of tests which touch the database is to opt for a Sqlite database in memory. Though benchmarks vary, this technique may still be one that you should consider.

To instruct Laravel to use a database in memory, update `app/config/testing/database.php` to set a new default driver for testing.

```
1 <?php // app/config/testing/database.php
2
3 return array(
4     'default' => 'sqlite',
5
6     'connections' => array(
7         'sqlite' => array(
```

```
8         'driver'     => 'sqlite',
9         'database'   => ':memory:',
10        'prefix'      => '',
11    )
12 )
13 );
```

Notice how, rather than setting the database key to a .sqlite file, instead, a DB in memory was specified. With this modification alone, when testing, Laravel will automatically set the environment to *testing* and reference the in-memory DB.

When choosing this method, it's vital that, before each test, you re-migrate and seed the database. Here's an example:

```
1 public function setUp()
2 {
3     parent::setUp();
4
5     Artisan::call('migrate');
6     $this->seed();
7 }
```

These two lines will construct and seed all necessary tables before firing each test.

Summary

When it comes to unit testing specifically, use the techniques outlined in this chapter as a last resort. First, make every effort to test in isolation. If doing so seems impossibly, then it's likely that your code should be decoupled.

In a perfect world, test databases should only be used for outside-in tests.

Chapter 9: Just Swap That Thang

Laravel's use of facades is easily one of the most misunderstood *features* of the framework. In fact, to newcomers, it may appear as though Laravel is a big mess of untestable static method calls.

```
1 Mail::send();  
2 DB::getQueryLog();  
3 Queue::push();
```

Those of us with a moderate amount of experience with object-oriented programming and testing are conditioned to instantly squirm at the site of statics, like these. Sure, you can test a single static method; but, as soon as that method calls another method, you'll quickly find yourself in a world of hurt when it's time to test.

Fortunately, Laravel - rather brilliantly, I might add - allows for this clean syntax, while still, behind the scenes, instantiating the applicable class, allowing for perfect testability. This makes for the best of both worlds, and is what we refer to as the *Facade Pattern*.



Facade: A facade is an object that provides a simplified interface to a larger body of code.

Other than readability, one of the significant advantages to this approach is that the underlying class can easily be swapped out with a mock for testing purposes. In translation, this means that you are free to use these facades within your code (without injecting them), while still allowing for complete testability.

Mockery

Every Laravel facade extends a parent Facade class that offers, among other things, a `shouldReceive` method. When called, Laravel will automatically swap out the registered instance with a mock, using Mockery. See for yourself:

```

1 // 
vendor/laravel/framework/src/Illuminate/Support/Facades/Facade.php
2
3 /**
4  * Initiate a mock expectation on the facade.
5 *
6  * @param  dynamic
7  * @return \Mockery\Expectation
8 */
9 public static function shouldReceive()
10 {
11     $name = static::getFacadeAccessor();
12
13     if (static::isMock())
14     {
15         $mock = static::$resolvedInstance[$name];
16     }
17     else
18     {
19         static::$resolvedInstance[$name] = $mock =
\Mockery::mock(static::g\
20 etMockableClass($name));
21
22         static::$app->instance($name, $mock);
23     }
24
25     return call_user_func_array(array($mock, 'shouldReceive'), func_get_arg\
26 s());
27 }

```

Don't worry if this code is difficult to interpret. You needn't understand every facet. Most importantly, though, recognize that the resolved instance is being swapped out with a mocked version.

```

1 static::$resolvedInstance[$name] = $mock =
\Mockery::mock(static::getMockab\
2 leClass($name));

```

So how might we use this to our advantage?

Testing

For the purposes of simplicity, imagine that, when a particular URL is requested, a file will be created on the server. It's a silly example, but will illustrate this swapping functionality nicely. When unit testing, we certainly don't want to physically create the file for each test. It's a better idea to mock the `File` class, like so:

```
1 public function testCreatesFile()
2 {
3     File::shouldReceive('put')->once();
4
5     $this->call('GET', 'foo');
6 }
```

This code declares that, when `localhost:8000/foo` is requested, we expect that the `put` method on the mocked version of the `Filesystem` class will be called.

Assuming that no `foo` route currently exists, PHPUnit will, of course, fail:

```
1) ExampleTest::testCreatesFile
2 Symfony\Component\HttpKernel\Exception\NotFoundHttpException:
```

Let's create the route.

```
1 // app/routes.php
2
3 Route::get('foo', function()
4 {
5
6 });
```

Somehow, the test will now pass. But how is that possible? We declared that `File::put()` should be called once. Yet, even though the route closure is empty, the test still passes. Huh? As it turns out, you'll fall into this trap often.

Remember: because we're leveraging Mockery, a tearDown method must always be declared in your test file. Among other things, this call will verify your expectations. To avoid repetition, you can place this tearDown method within the parent TestCase class that Laravel provides.

```
1 <?php // app/tests/FooTest.php
2
3 class FooTest extends TestCase {
4
5     public function tearDown()
6     {
7         Mockery::close();
8     }
9
10    public function testCreatesFile()
11    {
12        File::shouldReceive('put')->once();
13
14        $this->call('GET', 'foo');
15    }
16
17 }
```

Now, we're back to a failed test, as expected.

```
1) ExampleTest::testCreatesFile
Mockery\Exception\InvalidArgumentException: Method put() from
Illuminate\Filesystem\
system\Filesytem should be called
exactly 1 times but called 0 times.
```

Here, we see Mockery hard at work. Let's make the test pass.

```
1 // app/routes.php
2
3 Route::get('foo', function()
4 {
    File::put(__DIR__.'/file.txt', 'Lorem ipsum');
```

```
6 } );
```

We're back to green! Admittedly, it's a silly example, but, nonetheless, it's really a very cool (and unique) feature of Laravel! Even though the route's callback calls `File::put()`, we can override the resolved instance of the `Filesystem` class and replace it with a mocked version - all by simply writing `File::shouldReceive()`.

The end result is that the production code continues to be as readable as possible, while still allowing for the tests to run crazy fast.

As an alternative to using `shouldReceive`, if you prefer, you can instead use the facade's `swap` method, which will replace the underlying instance with the object that you pass in.

Mocking Events

Laravel 4 provides an easy way to fire and listen for custom events, via the `Event` class (also a facade). Using [Tuts+ Premium](#) as an example, it might make sense for the app to fire an event when a user cancels their subscription. This way, other parts of the application can respond by sending a cancellation email to the user, updating a report, clearing out a database record...any number of things.

Here's what the `destroy` method on a users controller might look like.

```
1 public function destroy($id)
2 {
3     // Get the user who is canceling
4     $user = $this->user->findOrFail($id);
5
6     // Soft delete the user
7     // To soft delete, set the $softDelete property
8     // on the User model
9     $user->delete();
10
11    // Make announcement, and send the $user object
12    Event::fire('cancellation', ['user' => $user]);
13
14    // Redirect to home page
```

```
15     return Redirect::home();
16 }
```

When writing a functional test for this controller, we can simulate the fired event in the same way that we did with the `File` facade.

```
1 public function testDestroyUser()
2 {
3     Event::shouldReceive('fire')
4         ->once()
5         ->with(['cancellation', Mockery::any()]);
6
7     // Perform any other necessary expectations
8
9     $this->call('DELETE', 'users1');
10 }
```

Even though we referenced the `Event` class explicitly, we're still able to mock it with ease.

Summary

In this chapter, we reviewed Laravel's unique use of facades, which, though confusing at first, allow for the readable static syntax that endeared all of us to Laravel in the first place. Even though it feels as if you're interacting with statics, these are merely illusions (or facades) that instantiate underlying classes, making for complete testability. It's the best of both worlds!

The techniques outlined in this chapter are most useful for small to medium-sized project. However, if you find yourself relying on mocking facades too much, it might be an indication that your code could be better organized.

Chapter 10: Testing Controllers

Testing controllers (a form of functional testing) isn't the easiest thing in the world. Well, let me rephrase that: testing them is a cinch; what's difficult, at least at first, is determining *what* to test.

Should a controller test verify text on the page? Should it touch the database? Should it ensure that variables exist in the view? If this is your first hay-ride, these things can be confusing! Let me help.

Controller tests should verify responses, ensure that the correct database access methods are triggered, and assert that the appropriate instance variables are sent to the view.

Before we continue, it's worth noting that some groups of developers feel that testing controllers is unnecessary. Instead, they prefer integration tests (I'm more inclined to call them functional tests) that will indirectly verify that these controllers work properly. As always, make up your own mind. What allows you to sleep better at night?

What Does a Controller Do?

Before we discuss testing, however, perhaps it's a smart idea to first review which responsibilities a controller should have. As I see things, there are two core areas of focus:

1. Direct traffic, by serving as the application's HTTP interface.
2. Send its own messages to the domain object. As they say, controllers should *tell*, rather than *ask* (Fat Model -> Skinny Controller).

But, as you might expect, in practice, we typically do quite a bit more. Are you performing validation in the model? Is that the controller's responsibility? Are you verifying actions on the model, in order to proceed in different ways based on the result? If you're not careful, before long, your controllers can become massive code smells.

In these situations, it's best to take a step back, and ask yourself, “*What is the responsibility of this class?*” If you find yourself using the word, *and*, too frequently, then a code smell just might be near!



Tip: The less work your controller does, the easier it is to test. This is true for all classes, and is one of the benefits of embracing the single responsibility principle.

3 Steps to Testing Controllers

The process of testing a controller (or any class, really) can be divided into three pieces.

- **Isolate:** Mock all dependencies (perhaps excluding the View).
- **Call:** Trigger the desired controller method.
- **Ensure:** Perform assertions, verifying that the stage has been set properly.

Or, you might recognize this as the more generic pattern, *AAA*:

- **Arrange**
- **Act**
- **Assert**

The Hello World of Controller Testing

The best way to learn these things is through examples. Here's the “*hello world*” of controller testing in Laravel.

```
1 <?php
2
3 class PostsControllerTest extends TestCase {
4
5     public function testIndex()
6     {
7         $this->client->request('GET', 'posts');
8     }
9 }
```

Laravel leverages a handful of Symfony's components to ease the process of

testing routes and views, including `HttpKernel`, `DomCrawler`, and `BrowserKit`. This is why it's paramount that your `PHPUnit` tests inherit from, not `PHPUnit_Framework_TestCase`, but `TestCase`. Don't worry, Laravel still extends the former, but it helps setup the Laravel app for testing, as well as provides a variety of helper assertion methods that you are encouraged to use. More on that shortly.

In the code snippet above, we make a `GET` request to `/posts`, or `localhost:8000/posts`. Assuming that this line is added to a fresh installation of Laravel, Symfony will throw a `NotFoundHttpException`. If working along, try it out by running `phpunit` from the command line.

```
1 $ phpunit
2 1) PostsControllerTest::testIndex
3 Symfony\Component\HttpKernel\Exception\NotFoundException:
```

In *human-speak*, this essentially translates to, “*Hey, I tried to call that route, but you don't have anything registered, fool!*”

As you can imagine, this type of request is common enough to the point that it makes sense to provide a helper method, such as `$this->call()`. In fact, Laravel does that very thing! This means that the previous example can be refactored, like so:

```
1 public function testIndex()
2 {
3     $this->call('GET', 'posts');
4 }
```

Well, that's only partially true. Even though `$this->call()` and `$this->client->request()` will both *call* a particular route, the returned values will differ.

```
1 $crawler = $this->client->request('GET', '/'); // Symfony\Component\DomCraw\ler\Crawler
2
3
4 $response = $this->call('GET', '/'); // Illuminate\Http\Response
```

As a basic rule of thumb, if you intend to *crawl* the DOM - *this bit of text should appear when the page is loaded* - then use `$this->client->request()`, as it will return a `Crawler` instance. Otherwise, the `call` method, which will return a response object, will do just fine.

Overloading is Your Friend

Though we'll stick with the base functionality in this chapter, in my personal projects, I take things a step further by allowing for such methods as `$this->get()`, `$this->post()`, etc. Thanks to PHP overloading, this only requires the addition of a single method, which you could add to `app/tests/TestCase.php`.

```
1 public function __call($method, $args)
2 {
3     if (in_array($method, ['get', 'post', 'put', 'patch',
4         'delete'])) {
5         return $this->call($method, $args[0]);
6     }
7
8     throw new BadMethodCallException;
9 }
```

Now, you're free to write `$this->get('posts')` and achieve the exact same result as the previous two examples. As noted above, however, let's stick with the framework's base functionality for simplicity's sake.

To make the test pass, we only need to prepare the proper route.

```
1 <?php
2
3 Route::get('posts', function()
4 {
5     return 'all posts';
6 });
```

Running `PHPUnit` again will return us to green.

Calling Controller Actions

If you'd prefer to instead call a controller directly, you can instead use the `action` method, like so:

```
1 $response = $this->action('GET', 'PostsController@index');
```

This will trigger the `index` method on `PostsController`. Any applicable parameters may be passed as the third argument.

```
1 $response = $this->action('GET', 'PostsController@show', ['id' => 1]);
```

Laravel's Helper Assertions

A test that you'll find yourself writing repeatedly is one that ensures that a controller passes a particular variable to a view. For example, the `index` method of `PostsController` should pass a `$posts` variable to its associated view, right? That way, the view can filter through all posts, and display them on the page. This is an important test to write!

If it's that common a task, then, once again, wouldn't it make sense for Laravel to provide a helper assertion to accomplish this very thing? Of course it would. And, of course, Laravel does!

`Illuminate\Foundation\Testing\TestCase` includes a number of methods that will drastically reduce the amount of code needed to perform basic assertions. This list includes:

- `assertViewHas`
- `assertResponseOk`
- `assertRedirectedTo`
- `assertRedirectedToRoute`
- `assertRedirectedToAction`
- `assertSessionHas`
- `assertSessionHasErrors`

The following examples calls `GET /posts` and verifies that its views receives the variable, `$posts`.

```
1 public function testIndex()
```

```
2 {
3     $this->call('GET', 'posts');
4
5     $this->assertViewHas('posts');
6 }
```

Tip: When it comes to formatting, I prefer to provide a line break between a test's assertion and the code that prepares the stage.

`assertViewHas` is simply a bit of sugar that inspects the response object - which is returned from `$this->call()` - and verifies that the data associated with the view contains a `posts` variable.

When inspecting the response object, you have two core choices.

- `$response->getOriginalContent()`: Fetch the original content, or the returned `view`. Optionally, you may access the `original` property directly, rather than calling the `getOriginalContent` method.
- `$response->getContent()`: Fetch the evaluated output. If a `View` instance is returned from the route, then `getContent()` will be equal to the HTML output. This can be helpful for DOM verifications, such as “*the view must contain this string.*”

Let's assume that the `posts` route consists of:

```
1 <?php
2
3 Route::get('posts', function()
4 {
5     return View::make('posts.index');
6 });
```

Should we run `phpunit`, it will squawk with a helpful *next step* message:

```
1) PostsControllerTest::testIndex
2 Failed asserting that an array has the key 'posts'.
```

To make it green, we simply fetch the `posts` and pass it to the view.

```
1 Route::get('posts', function()
2 {
3     $posts = Post::all();
4
5     return View::make('posts.index', ['posts', $posts]);
6 });
```

One thing to keep in mind is that, as the code currently stands, it only ensures that the variable, `$posts`, is passed to the view. It doesn't inspect its value. The `assertViewHas` optionally accepts a second argument to verify the value of the variable, as well as its existence.

```
1 public function testIndex()
2 {
3     $this->call('GET', 'posts');
4
5     $this->assertViewHas('posts', 'foo');
6 }
```

With this modified code, unless the view has a variable, `$posts`, that is equal to `foo`, the test will fail. In this situation, though, it's likely that we'd rather not specify a value, but instead declare that the value be an instance of Laravel's `Illuminate\Database\Eloquent\Collection` class. How might we accomplish that? PHPUnit provides a helpful `assertInstanceOf` assertion to fill this very need!

```
1 public function testIndex()
2 {
3     $response = $this->call('GET', 'posts');
4
5     $this->assertViewHas('posts');
6
7     // getData() returns all vars attached to the response.
8     $posts = $response->original->getData()['posts'];
9
10    $this-
>assertInstanceOf('Illuminate\Database\Eloquent\Collection', $pos\ts);
```

```
12 }
```

With this modification, we've declared that the controller **must** pass `$posts` - an instance of `Illuminate\Database\Eloquent\Collection` - to the view. Excellent.

Mocking the Database

There's one glaring problem with our tests so far. Did you catch it?

For each test, a SQL query is being executed on the database. Though this is useful for certain kinds of testing (acceptance, integration), for basic controller testing, it will only serve to decrease performance.

I've drilled this into your skull multiple times at this point. We're not interested in testing Eloquent's ability to fetch records from a database. It has its own tests. Taylor knows it works! Let's not waste time and processing power repeating those same tests.

Instead, it's best to mock the database, and merely verify that the appropriate methods are called with the correct arguments. Or, in other words, we want to ensure that `Post::all()` never fires and hits the database. We know that works, so it doesn't require testing.

This section will depend heavily on the Mockery library. Please review that chapter [from my book](#), if you're not yet familiar with it.

Required Refactoring

Unfortunately, so far, we've structured the code in a way that makes it virtually impossible to test.

```
1 Route::get('posts', function()
2 {
3     // Ouch. We can't test this!!
4     $posts = Post::all();
5
6     return View::make('posts.index')
7         ->with('posts', $posts);
```

```
8 } );
```

This is precisely why it's considered bad practice to nest Eloquent calls into your controllers. Don't confuse Laravel's facades, which are testable and can be swapped out with mocks (`Queue::shouldReceive()`), with your Eloquent models. The solution is to inject the database layer into the controller through the constructor. This requires some refactoring.



Warning: Storing logic within route callbacks is useful for small projects and APIs, but they make testing incredibly difficult. For applications of any considerable size, use controllers.

Let's register a new resource by replacing the posts route with:

```
1 Route::resource('posts', 'PostsController');
```

...and create the necessary resourceful controller with Artisan.

```
1 $ php artisan controller:make PostsController
2 Controller created successfully!
```

Now, rather than referencing the Post model directly, we'll inject it into the controller's constructor. Here's a condensed example that omits all restful methods, except the one that we're currently interested in testing.

```
1 <?php
2
3 class PostsController extends BaseController {
4
5     protected $post;
6
7     public function __construct(Post $post)
8     {
9         $this->post = $post;
10    }
11
12    public function index()
13    {
```

```
14     $posts = $this->post->all();
15
16     return View::make('posts.index')
17         ->with('posts', $posts);
18 }
19
20 }
```



Tip:: It's a better idea to typehint an interface, rather than reference the Eloquent model, itself. But, one thing at a time! Let's work up to that.

Don't worry about manually injecting the Post instance. Like magic, Laravel will do this automatically for you! This is referred to as [automatic resolution](#). More on this shortly.

This is a significantly better way to structure the code. Because the model is now injected, we have the ability to swap it out with a mocked version for testing. Here's an example of doing just that:

```
1 <?php
2
3 class PostsControllerTest extends TestCase {
4
5     public function __construct()
6     {
7         // We have no interest in testing Eloquent
8         $this->mock = Mockery::mock('Eloquent', 'Post');
9     }
10
11    public function tearDown()
12    {
13        Mockery::close();
14    }
15
16    public function testIndex()
17    {
```

```

18     $this->mock
19         ->shouldReceive('all')
20             ->once()
21             ->andReturn('foo');
22
23     $this->app->instance('Post', $this->mock);
24
25     $this->call('GET', 'posts');
26
27     $this->assertViewHas('posts');
28 }
29
30 }
```

The key benefit to this restructuring is that, now, the database will never needlessly be hit. Instead, using Mockery, we merely verify that the `all` method is triggered on the model.

```

1 $this->mock
2     ->shouldReceive('all')
3     ->once();
```

Unfortunately, if you choose to forego coding to an interface, and instead inject the `Post` model into the controller, a bit of trickery has to be used in order to get around Eloquent's use of statics, which can clash with Mockery. This is why we hijack both the `Post` and `Eloquent` classes within the test's constructor, before the official versions have been loaded. This way, we have a clean slate to declare any expectations. The downside, of course, is that we can't default to any existing methods, through the use of Mockery methods, like `makePartial()`.

The IoC Container

Laravel's IoC container drastically eases the process of injecting dependencies into your classes. Each time a controller is requested, it is resolved out of the IoC container. As such, when we need to declare that a mock version of `Post` should be used for testing, we only need to provide Laravel with the instance of `Post` that should be used.

```

1 $this->app->instance('Post', $this->mock);
```

Think of this code as saying, “*Hey Laravel, when you need an instance of Post, I want you to use my mocked version.*” Because the app extends the Container we have access to all IoC methods directly off of it.

Upon instantiation of the controller, Laravel leverages the power of PHP reflection to read the typehint and inject the dependency for you. That’s right; you don’t have to write a single binding to allow for this; it’s automated!

Redirections

Another common expectation that you’ll find yourself writing is one that ensures that the user is redirected to the proper location, perhaps upon adding a new post to the database. How might we accomplish this?

```
1 public function testStore()
2 {
3     $this->mock
4         ->shouldReceive('create')
5         ->once();
6
7     $this->app->instance('Post', $this->mock);
8
9     $this->call('POST', 'posts');
10
11    $this->assertRedirectedToRoute('posts.index');
12 }
```

Assuming that we’re following a restful flavor, to add a new post, we’d POST to the collection, or posts (don’t confuse the POST request method with the resource name, which just happens to have the same name).

```
1 $this->call('POST', 'posts');
```

Then, we only need to leverage another of Laravel’s helper assertions, assertRedirectedToRoute.



Tip: When a resource is registered with Laravel (Route::resource()), the framework will automatically register the necessary named routes. Run `php artisan routes` if you ever

forget what these names are.

You might prefer to also ensure that the `$_POST` superglobal is passed to the `create` method. Even though we aren't physically submitting a form, we can still allow for this, via the `Input::replace()` method, which allows us to "stub" this array. Here's the modified test, which uses Mockery's `with()` method to verify the arguments passed to the method referenced by `shouldReceive`.

```
1 public function testStore()
2 {
3     Input::replace($input = ['title' => 'My Title']);
4
5     $this->mock
6         ->shouldReceive('create')
7         ->once()
8         ->with($input);
9
10    $this->app->instance('Post', $this->mock);
11
12    $this->call('POST', 'posts');
13
14    $this->assertRedirectedToRoute('posts.index');
15 }
```

Paths

One thing that we haven't considered in this test is validation. There should be two separate paths through the `store` method, dependent upon whether the validation passes:

1. Redirect back to the "Create Post" form, and display the form validation errors.
2. Redirect to the collection, or the named route, `posts.index`.

As a best practice, each test should represent but one path through your code.

This first path will be for failed validation.

```

1 public function testStoreFails()
2 {
3     // Set stage for a failed validation
4     Input::replace(['title' => '']);
5
6     $this->app->instance('Post', $this->mock);
7
8     $this->call('POST', 'posts');
9
10    // Failed validation should reload the create form
11    $this->assertRedirectedToRoute('posts.create');
12
13    // The errors should be sent to the view
14    $this->assertSessionHasErrors(['title']);
15 }

```

The code snippet above explicitly declares which errors should exist. Alternatively, you may omit the argument to `assertSessionHasErrors`, in which case it will merely verify that a message bag has been flashed (in translation, your Redirection includes `withErrors($errors)`).

Now for the test that handles successful validation.

```

1 public function testStoreSuccess()
2 {
3     // Set stage for successful validation
4     Input::replace(['title' => 'Foo Title']);
5
6     $this->mock
7         ->shouldReceive('create')
8         ->once();
9
10    $this->app->instance('Post', $this->mock);
11
12    $this->call('POST', 'posts');
13
14    // Should redirect to collection, with a success flash
15    $this->assertRedirectedToRoute('posts.index', ['flash']);

```

```
16 }
```

The production code for these two tests might look like:

```
1 public function store()
2 {
3     $input = Input::all();
4
5     // We'll run validation in the controller for convenience
6     // You should export this to the model, or a service
7     $v = Validator::make($input, ['title' => 'required']);
8
9     if ($v->fails())
10    {
11        return Redirect::route('posts.create')
12            ->withInput()
13            ->withErrors($v->messages());
14    }
15
16    $this->post->create($input);
17
18    return Redirect::route('posts.index')
19        ->with('flash', 'Your post has been created!');
20 }
```

Notice how the Validator is nested directly in the controller? Generally, I'd recommend that you abstract this away to a service. That way, you can test your validation in isolation from any controllers or routes. Nonetheless, let's leave things as they are for simplicity's sake. One thing to keep in mind is that we aren't mocking the Validator, though you certainly could do so. Because this class is a facade, it can easily be swapped out with a mocked version, via the Facade's `shouldReceive` method, without us needing to worry about injecting an instance through the constructor. Win!

```
1 Validator::shouldReceive('make')
2     ->once()
3     ->andReturn(Mockery::mock(['fails' => 'true']));
```

From time to time, you'll find that a method that needs to be mocked should

From time to time, you'll find that a method that needs to be mocked should return an object, itself. Luckily, with Mockery, this is a piece of cake: we only need to create an anonymous mock, and pass an array, which signals the method name and response value, respectively. As such:

```
1 Mockery::mock(['fails' => 'true'])
```

will prepare an object, containing a `fails()` method that returns `true`.

Repositories

To allow for optimal flexibility, rather than creating a direct link between your controller and an ORM, like Eloquent, it's better to code to an interface. The considerable advantage to this approach is that, should you perhaps need to swap out Eloquent for, say, Mongo or Redis, doing so literally requires the modification of a single line. Even better, the controller doesn't ever need to be touched.

Repositories represent the data access layer of your application.

What might an interface for managing the database layer of a Post look like? This should get you started.

```
1 <?php namespace Repositories;  
2  
3 interface PostRepositoryInterface {  
4  
5     public function all();  
6  
7     public function find($id);  
8  
9     public function create($input);  
10  
11 }
```

This can certainly be extended, but we've added the bare minimum methods for the demo: `all`, `find`, and `create`. Notice that the repository interfaces are being stored within `app/repositories`. Because this folder is not autoloaded by default, we need to update the composer `.json` file for the application to

reference it.

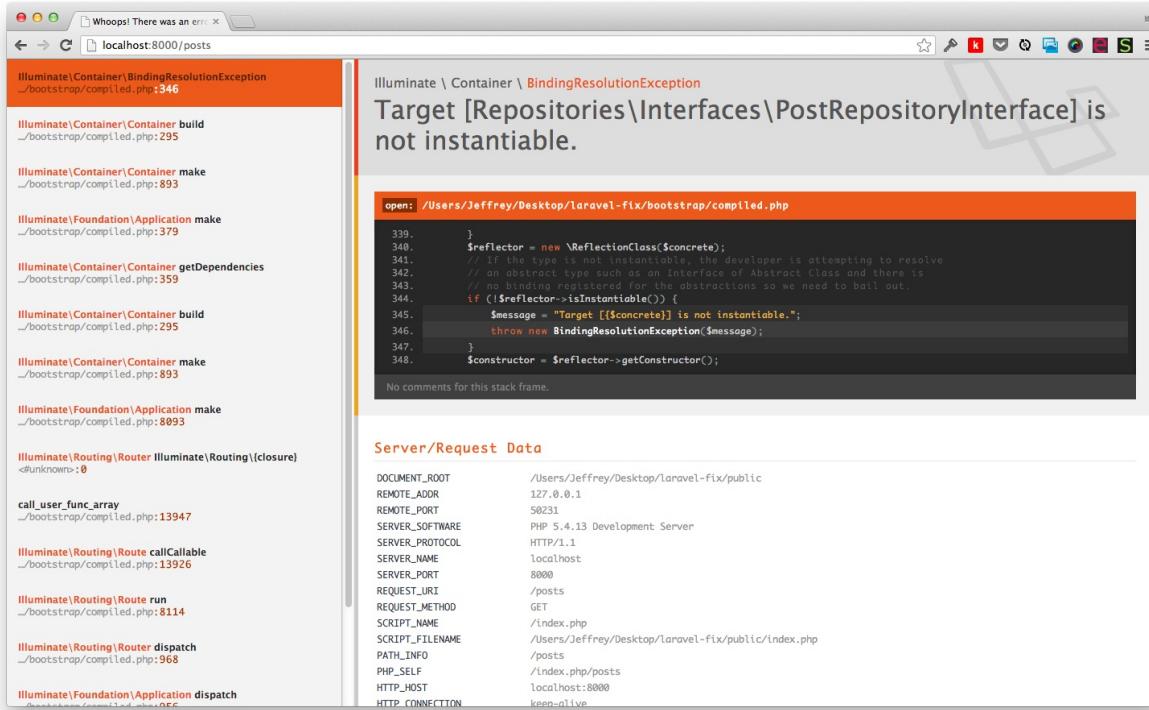
```
1 "autoload": {  
2     "classmap": [  
3         // ....  
4         "app/repositories"  
5     ]  
6 }
```

When a new class is added to this directory, don't forget to composer dump-autoload -o. The -o, (*optimize*) flag is optional, but should always be used, as a best practice.

If you attempt to inject this interface into your controller, Laravel will snap at you. Go ahead; try it out and see. Here's the modified PostController, which has been updated to inject an interface, rather than the Post Eloquent model.

```
1 <?php  
2  
3 use Repositories\PostRepositoryInterface as Post;  
4  
5 class PostsController extends BaseController {  
6  
7     protected $post;  
8  
9     public function __construct(Post $post)  
10    {  
11        $this->post = $post;  
12    }  
13  
14    public function index()  
15    {  
16        $posts = $this->post->all();  
17  
18        return View::make('posts.index', ['posts' =>  
$posts]);  
19    }  
20 }
```

If you run the server and view the output, you'll be met with the dreaded (but beautiful) Whoops error page, declaring that "*PostRepositoryInterface* is not instantiable."



Not Instantiable?

If you think about it, of course the framework is squawking! Laravel is smart, but it's not a mind reader. It needs to be told which implementation of the interface should be used within the controller.

For now, let's add this binding to `app/routes.php`. Later, we'll instead make use of service providers to store this sort of logic.

```
1 App::bind(
2     'Repositories\PostRepositoryInterface',
3     'Repositories\EloquentPostRepository'
4 );
```

Verbalize this function call as, "*Laravel, baby, when you need an instance of PostRepositoryInterface, I want you to use EloquentPostRepository.*"

`app/repositories/EloquentPostRepository` will simply be a wrapper around Eloquent that implements `PostRepositoryInterface`. This way, we're not restricting the API (and every other implementation) to Eloquent's interpretation; we can name the methods however we wish.

```
1 <?php namespace Repositories;
2
3 use Repositories\PostRepositoryInterface;
4 use Post;
5
6 class EloquentPostRepository implements PostRepositoryInterface
{
7
8     public function all()
9     {
10         return Post::all();
11     }
12
13     public function find($id)
14     {
15         return Post::find($id);
16     }
17
18     public function create($input)
19     {
20         return Post::create($input);
21     }
22
23 }
```

Some might argue that the `Post` model should be injected into this implementation for testability purposes. If you agree, simply inject it through the constructor, per usual.

That's all it should take! Refresh the browser, and things should be back to normal. Only, now, your application is far better structured, and the controller is no longer linked to Eloquent.

Let's imagine that, a few months from now, your boss informs you that you need to swap Eloquent out with Redis. Well, because you've structured your application in this future-proof way, you only need to create the new app/repositories/RedisPostRepository implementation:

```
1 <?php namespace Repositories;
2
3 use Repositories\PostRepositoryInterface;
4
5 class RedisPostRepository implements PostRepositoryInterface {
6
7     public function all()
8     {
9         // return all with Redis
10    }
11
12    public function find($id)
13    {
14        // return find one with Redis
15    }
16
17    public function create($input)
18    {
19        // return create with Redis
20    }
21
22 }
```

And update the binding:

```
1 App::bind(
2     'Repositories\PostRepositoryInterface',
3     'Repositories\RedisPostRepository'
4 );
```

Instantly, you're now leveraging Redis in your controller. Notice how app/controllers/PostsController.php was never touched? That's the beauty of it!

Structure

So far in this lesson, our organization has been a bit lacking. IoC bindings in the `routes.php` file? All repositories grouped together in one directory? Sure, that may work in the beginning, but, very quickly, it'll become apparent that this doesn't scale.

In the final section of this chapter, we'll PSR-ify our code, and leverage service providers to register any applicable bindings.



PSR-0 defines the mandatory requirements that must be adhered to for autoloader interoperability.

A PSR-0 loader may be registered with Composer, via the `psr-0` object.

```
1 "autoload": {
2     "psr-0": {
3         "way": "app/lib/"
4     }
5 }
```

The syntax can be confusing at first. It certainly was for me. An easy way to decipher `"way": "app/lib/"` is to think to yourself, *“The base folder for the way namespace is located in app/lib.”* Of course, replace my last name with the name of your project. The directory structure to match this would be:

- app/
 - lib/
 - Way/

Next, rather than grouping all repositories into a `repositories` directory, a more elegant approach might be to categorize them into multiple directories, like so:

- app/
 - lib/

- Way/
 - Storage/
 - Post/
 - PostRepositoryInterface.php
 - EloquentPostRepository.php

It's vital that we adhere to this naming and folder convention, if we want the autoloading to work as expected. The only remaining thing to do is update the namespaces for `PostRepositoryInterface` and `EloquentPostRepository`.

```

1 <?php namespace Way\Storage\Post;
2
3 interface PostRepositoryInterface {
4
5     public function all();
6
7     public function find($id);
8
9     public function create($input);
10
11 }
```

And the implementation:

```

1 <?php namespace Way\Storage\Post;
2
3 use Post;
4
5 class EloquentPostRepository implements PostRepositoryInterface
{
6
7     public function all()
8     {
9         return Post::all();
10    }
11 }
```

```
12     public function find($id)
13     {
14         return Post::find($id);
15     }
16
17     public function create($input)
18     {
19         return Post::create($input);
20     }
21
22 }
```

There we go; that's much cleaner. But what about those pesky bindings? The routes file may be a convenient place to experiment, but it makes little sense to store them there permanently. Instead, we'll use service providers.



Service providers are nothing more than bootstrap classes that can be used to do anything you wish: register a binding, hook into an event, import a routes file, *etc.*

The service provider's `register()` method will be triggered automatically by Laravel.

```
1 <?php namespace Way\Storage;
2
3 use Illuminate\Support\ServiceProvider;
4
5 class StorageServiceProvider extends ServiceProvider {
6
7     // Triggered automatically by Laravel
8     public function register()
9     {
10         $this->app->bind(
11             'Way\Storage\Post\PostRepositoryInterface',
12             'Way\Storage\Post\EloquentPostRepository'
13         );
14     }
15 }
```

```
16 }
```

To make this file known to Laravel, you only need to include it in `app/config/app.php`, within the `providers` array.

```
1 'providers' => array(
2     'Illuminate\Foundation\Providers\ArtisanServiceProvider',
3     'Illuminate\Auth\AuthServiceProvider',
4     // ...
5     'Way\Storage\StorageServiceProvider'
6 )
```

Good; now we have a dedicated file for registering new bindings.

Updating the Tests

With our new structure in place, rather than mocking the Eloquent model, itself, we can instead mock `PostRepositoryInterface`. Here's an example of one such test:

```
1 public function testIndex()
2 {
3     $mock =
Mockery::mock('Way\Storage\Post\PostRepositoryInterface');
4     $mock->shouldReceive('all')->once();
5
6     $this->app-
>instance('Way\Storage\Post\PostRepositoryInterface', $mock)\
7 ;
8
9     $this->call('GET', 'posts');
10
11    $this->assertViewHas('posts');
12 }
```

However, we can improve this. It stands to reason that every method within `PostsControllerTest` will require a mocked version of the repository. As such, it's better to extract some of this prep work into its own method, like so:

```

1 public function setUp()
2 {
3     parent::setUp();
4
5     $this->mock = $this-
>mock('Way\Storage\Post\PostRepositoryInterface');
6 }
7
8 public function mock($class)
9 {
10    $mock = Mockery::mock($class);
11
12    $this->app->instance($class, $mock);
13
14    return $mock;
15 }
16
17 public function testIndex()
18 {
19     $this->mock->shouldReceive('all')->once();
20
21     $this->call('GET', 'posts');
22
23     $this->assertViewHas('posts');
24 }

```

Not bad, ay?

Now, if you want to be super-fly, and are willing to add a touch of test logic to your production code, you could even perform your mocking within the Eloquent model! This would allow for:

```
1 Post::shouldReceive('all')->once();
```

Behind the scenes, this would mock `PostRepositoryInterface`, and update the IoC binding. You can't get much more readable than that!

Allowing for this syntax only requires you to update the `Post` model, or, better, a

`BaseModel` that all of the Eloquent models extend. Here's an example of the former:

```
1 <?php
2
3 class Post extends Eloquent {
4
5     public static function shouldReceive()
6     {
7         $class = get_called_class();
8         $repo = "Way\\Storage\\{$class}\\"
9         "{$class}RepositoryInterface";
10        $mock = Mockery::mock($repo);
11
12        App::instance($repo, $mock);
13
14        return call_user_func_array(
15            [$mock, 'shouldReceive'],
16            func_get_args()
17        );
18    }
19 }
```

If you can manage the inner “*Should I be embedding test logic into production code*” battle, you'll find that this allows for significantly more readable tests.

```
1 <?php
2
3 class PostsControllerTest extends TestCase {
4
5     public function tearDown()
6     {
7         Mockery::close();
8     }
9
10    public function testIndex()
11    {
12        Post::shouldReceive('all')->once();
13    }
14}
```

```

14     $this->call('GET', 'posts');
15
16     $this->assertViewHas('posts');
17 }
18
19 public function testStoreFails()
20 {
21     Input::replace($input = ['title' => '']);
22
23     $this->call('POST', 'posts');
24
25     $this->assertRedirectedToRoute('posts.create');
26     $this->assertSessionHasErrors();
27 }
28
29 public function testStoreSuccess()
30 {
31     Input::replace($input = ['title' => 'Foo Title']);
32
33     Post::shouldReceive('create')->once();
34
35     $this->call('POST', 'posts');
36
37     $this->assertRedirectedToRoute('posts.index',
38     ['flash']);
39 }
40 }
```

It feels good, doesn't it? Hopefully, this chapter hasn't been too overwhelming. The key is to learn how to organize your repositories in such a way to make them as easy as possible to mock and inject into your controllers. As a result of that effort, your tests will be lightning fast!

Crawling the DOM

Because Laravel offers Symfony's [DomCrawler](#) component out of the box, if needed, you also have the ability to inspect the DOM when calling routes.

When requesting a URI with `$this->client->request()`, a Crawler object will be returned, which can then be used to filter the DOM and perform assertions.

Ensure View Contains Text

```
1 public function testLogin()
2 {
3     $crawler = $this->client->request('GET', '/login');
4
5     $h1 = $crawler->filter('h1');
6
7     $this->assertEquals('Please Login', $h1->text());
8 }
```

Above, we're asserting that, when `/login` is requested, we expect to see an `<h1>` tag that contains the text, *Please Login*.



Tip: The `filter` method may be used to filter the DOM, using the simple CSS selectors that we're all familiar with. Alternatively, XPath may be used, but stick with the `cssselector` component, if you can. Its far more readable.

In the example above, the assertion will only pass if the first occurrence of an `<h1>` tag contains *Please Login*. If you need to be more flexible, the assertion could be rewritten, like so:

```
1 $h1 = $crawler->filter('h1:contains("Hello World!")');
2 $this->assertCount(1, $h1);
```

Basic Traversing

The DomCrawler component offers a handful of traversal methods that can be used for further filtering. Many of these are simply helpers that could also be accomplished by modifying the CSS selector directly. Nonetheless, they can be a helpful convenience.

Fetch By Position

```
1 $crawler->filter('h2')->eq(2);
```

Fetch First or Last

```
1 $crawler->filter('ul.tasks li')->first();
2 $crawler->filter('ul.tasks li')->last();
```

Fetch Siblings

```
1 $crawler->filter('.question')->siblings();
```

Fetch Children

```
1 $crawler->filter('ul.tasks')->children();
```

Capture Text Content

Let's say that you need to fetch the value of every list item within a particular unordered list.

```
1 <ul class="tasks">
2   <li>Go to store</li>
3   <li>Finish book</li>
4   <li>Eat dinner</li>
5 </ul>
```

How might you do that? Well, one way would be to use a CSS selector to hunt down the applicable unordered list, map over the list of nodes, and extract the text content for each.

```
1 // Call route
2 $crawler = $this->client->request('GET', '/tasks');
3
4 // Filter down to the desired list
5 $items = $crawler->filter('ul.tasks li');
6
7 // Map over of the list, and return
8 // the text content for each.
9 $tasks = $items->each(function($node, $i)
10 {
11     return $node->text();
```

```
12 } );
```

Now, if we `var_dump($tasks)`, we'll see a list of all tasks. Nifty.

```
1 .array(3) {
2   [0]=>
3   string(11) "Go to store"
4   [1]=>
5   string(11) "Finish book"
6   [2]=>
7   string(10) "Eat dinner"
8 }
```

But, we can do better. The `extract` method can be used to pull out everything from attribute values, to the text content itself. As such, rather than mapping over the list items with `each()`, we can instead clean things up a bit, like so:

```
1 $items = $crawler->filter('ul.tasks li');
2 $tasks = $items->extract('_text');
```

`_text` is a special attribute name that refers to the node's value. This will achieve the exact same result.

If you need to grab a different attribute value from a node, simply substitute `_text` with its name, accordingly. Here's a few examples:

```
1 $className = $node->extract('class');
2 $id = $node->extract('id');
3 $idAndClass = $node->extract(['id', 'class']);
```

Forms

Another helpful functionality that the DomCrawler component provides is the ability to fill out and submit forms.

```
1 public function testRegister()
2 {
3     $crawler = $this->client->request('GET', '/register');
```

```

4
5      // Find the form associated with the submit button
6      // that has a value of 'Submit'
7      $form = $crawler->selectButton('Submit')->form();
8
9      // Fill out the form
10     $form['first'] = 'Jeffrey';
11     $form['last'] = 'Way';
12
13     // Submit the form
14     $r = $this->client->submit($form);
15 }
```

This component can absolutely be helpful in certain situations; however, from my experiences, I prefer to use a more streamlined and readable tool for querying the DOM: Codeception. We'll discuss this tool extensively in one of the following chapters. Until then, have fun with this!

Summary

There are those who would argue that testing controllers in this way is unnecessary. Let your functional and acceptance tests, they'd say, verify the routes and view bindings. As always, these things boil down to preference and testing style. Personally, I like to keep them.

One alternate approach, however, is to remove the mocks and instead treat these as true functional tests, similar to what the Ruby on Rails community advocates. Setting up a test database (in memory, if you can) for your controller tests, though slower, has the benefit of making the tests significantly easier to write and prepare. It's up to you! Make up your own mind.

Chapter 11: The IoC Container

An understanding of the IoC (*Inversion of Control*) pattern is paramount to building flexible and testable applications in Laravel.



IoC: An IoC container provides an easy interface for managing the creation of complex objects, without sacrificing on code readability or terseness. At its core, it allows you to abstract complicated instantiation behind a single line. Dependency injection is an implementation of the IoC pattern. Even better, Laravel's implementation is one of the most powerful in the PHP community. Outside of Laravel, [Pimple](#) is quite popular, as well.

Dependency Injection?

Laravel's IoC container makes the process of leveraging dependency injection to build testable applications as easy as possible. Dependency injection is a pattern for *inserting* a class' dependencies through its constructor (or a *setter* method, if you prefer), rather than hardcoding them - which, as you might have guessed, makes testability a considerable problem.



Definition: Think of dependency injection as a way to allow your future self to inject mocks in place of those dependencies, for testing purposes.

Constructor Injection

```
1 public function __construct(Validator $validator)
2 {
3     $this->validator = $validator;
4 }
```

Setter Injection

```
1 public function setValidator(Validator $validator)
2 {
```

```
3     $this->validator = $validator;
4 }
```

Consider the following class:

```
1 class MyCommand {
2     public function fire()
3     {
4         $generator = new ModelGenerator;
5
6         return $generator->make() ? 'foo' : 'bar';
7     }
8 }
```

This `fire` method is difficult to test - if not impossible (*not without installing [Runkit](#)*). It can be improved by injecting an instance of `ModelGenerator` through the class' constructor.

```
1 class MyCommand {
2     protected $generator;
3
4     public function __construct(ModelGenerator $generator)
5     {
6         $this->generator = $generator;
7     }
8
9     public function fire()
10    {
11        return $this->generator->make() ? 'foo' : 'bar';
12    }
13 }
```

Much better! Think of this as the command asking for the `ModelGenerator`. The test can now be tested quite easily: mock `ModelGenerator`, inject it into the class, and perform an expectation that the `make` method is called, along with your desired return value.

```
1 public function testFire()
```

```

2 {
3     $gen = Mockery::mock('ModelGenerator');
4     $gen->shouldReceive('make')->once()->andReturn(true);
5
6     $command = new MyCommand($gen);
7
8     $this->assertEquals('foo', $command->fire());
9 }

```

See? Not too hard. Dependency injection may initially be a scary term, but it's actually quite descriptive: inject dependencies into a class.

Resolving

Let's review a second example. Consider the following controller.

```

1 <?php
2
3 class UsersController extends BaseController {
4     public function index()
5     {
6         $users = User::all();
7
8         return View::make('users.index', ['users' =>
$users]);
9     }
10 }

```

There's nothing overly wrong with this code. In fact, you've surely written it, in some form or another, many, many times! However, as soon as testability becomes a first-class citizen, this code will no longer suffice. Why? Well, how would you test it without physically hitting the database? There's a couple hacks that you might use, but they involve alias mocks and multiple PHP processes. Like crossing the streams in Ghostbusters, it's just not a good idea. There are a couple ways to manage this.

Solution 1: Defaults

Let's assume that, for a particular project, no IoC container is available. In situations such as this allowing for testable code also has the adverse effect of

situations such as this, awaiting for readable code also has the adverse effect of forcing a laborious, less readable instantiation.

Which would you prefer?

```
1 $thing = new Thing;
```

Or:

```
1 $dep1 = new Dependency;
2 $dep2 = new OtherDependency;
3
4 $thing = new Thing($dep1, $dep2);
```

One solution is to set a sensible default, while still allowing for constructor injection. Here's an example:

```
1 function __construct(Dependency $dep1 = null,
2                         OtherDependency $dep2 = null)
3 {
4     $this->dep1 = $dep1 ?: new Dependency;
5     $this->dep2 = $dep2 ?: new OtherDependency;
6 }
```

With this technique, we have the flexibility to inject mock objects for both `Dependency` and `OtherDependency`. On the flip-side, if we stick with `new Thing()`, those objects will still be instantiated.

For small projects that don't have the luxury of a dedicated container (even though they're freely available on Packagist), this may be a preferable choice.

Solution 2: Resolving

This is a Laravel book, so let's solve it the Laravel way! A first step to improving this code might be to first resolve the `User` model out of the IoC container, rather than referencing the class directly. The following code resolves an instance of `User`, and then triggers its `all` method. Because one isn't currently registered, Laravel will simply fetch the `User` object and return the object.

```
1 public function index()
```

```
2 {
3     $users = App::make('User')->all();
4
5     return View::make('users.index', ['users' => $users]);
6 }
```

We'll discuss binding/resolving more shortly. The most important piece to understand at the moment is that we're asking Laravel to resolve an instance of User out of the IoC container.

When testing the method, Eloquent can be avoided all together (*why would we have any interest in testing that?*) by swapping out the currently registered instance of User with a mock, like so:

```
1 public function testIndex()
2 {
3     App::instance('User', Mockery::mock(['all' => 'foo']));
4     $this->call('GET', 'users');
5 }
```

With this modification, the next time that App::make('User') is called, Laravel will return this new mock object. In this case, as the tested code will require access to an all method, we provide one that simply returns foo.

Don't forget that if an array is passed as the first argument to an anonymous mock, then Mockery will register the keys of that array as methods, and their respective values as the methods' returned values. Here's an example:

```
1 $mock = Mockery::mock(['foo' => 'bar']);
2 $mock->foo(); // bar
```

Should you need to perform expectations on the mock (you likely will), then the test may be modified, like so:

```
1 public function testIndex()
2 {
3     $mock = Mockery::mock();
4     $mock->shouldReceive('all')->once()->andReturn('foo');
```

```
5     App::instance('User', $mock);
6
7     $this->call('GET', 'users');
8 }
```

Repeating this logic for each test can get tedious, though. We can DRY things up by extracting it to its own method that will run before each test, like so:

```
1 <?php
2
3 class UsersControllerTest extends TestCase {
4
5     public function tearDown()
6     {
7         Mockery::close();
8     }
9
10    protected function mock($class)
11    {
12        // Create a mock object, and register the
13        // instance with the IoC container
14        $this->mock = Mockery::mock();
15        App::instance($class, $this->mock);
16    }
17
18    public function testIndex()
19    {
20        // Set an expectation that the all() method
21        // should be called at least once.
22        $this->mock->shouldReceive('all')->once();
23
24        $this->call('GET', 'users');
25    }
26
27 }
```

Now, each test only needs to declare its expectations on the model, and then call the route, per usual.

App Bindings

There will likely be times when you need more control over how classes are resolved out of the container (or so you might think). You can register a binding with the app, and pass a closure as the second argument, which will dictate how this binding will be resolved.

Here's a contrived example that binds the key, `foo`, to a closure that will instantiate the `IKnowKungFoo` class.

```
1 <?php // app/routes.php
2
3 class IKnowKungFoo {}
4
5 // Only for example. Don't do this.
6 App::bind('foo', function()
7 {
8     return new IKnowKungFoo;
9});
```

To trigger that closure and return the new object, use `App::make('foo')`.

```
1 Route::get('/', function()
2 {
3     var_dump(App::make('foo')); // object(IKnowKungFoo)#161
4});
```

Laravel offers an alternate syntax for binding keys to classes. Rather than passing a closure, we could instead reference the name of the class to instantiate, as a string. The following will achieve the exact same result.

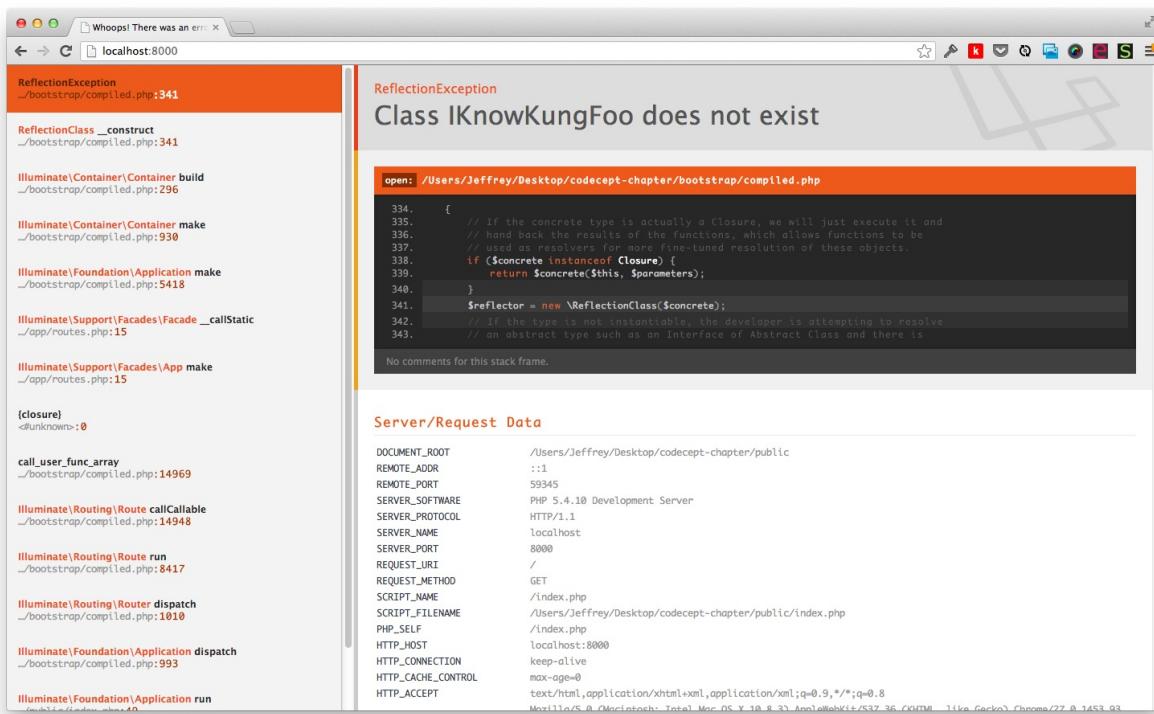
```
1 App::bind('foo', 'IKnowKungFoo');
```

Now, in a real-world situation, there's no need for this. Because we're not performing any real logic within the closure, we could simply do:

```
1 Route::get('/', function()
2 {
3     var_dump(App::make('IKnowKungFoo'));
```

```
4 } );
```

When resolving `IKnowKungFoo`, if Laravel doesn't find an existing registered binding, it'll next look for a class name, called `IKnowKungFoo`. If it finds one, it will simply return a new instance. On the other hand, if no class exists either, then Laravel has no idea what to do, in which case a `ReflectionException` will be thrown:



A `ReflectionException` will be thrown if no binding or class is found.

Interfaces

You might be thinking to yourself, “When would this ever be useful?” Well, as it turns out, the answer is quite frequently - particular when coding to an interface. Imagine a controller that asks for an implementation of `OrderRepositoryInterface`. *Repositories represent the data-access layer of your application.*

```
1 class OrdersController {
2     public function __construct(OrderRepositoryInterface
3 $order)
4     {
```

```
4     $this->order = $order;
5 }
6 }
```

Because controllers, too, are resolved out of the IoC container, Laravel, when presented with this code, will come to a road block, and throw a `BindingResolutionException`. Why? Because it's attempting to instantiate an interface, which obviously isn't allowed. In situations such as this, the framework needs a bit more information from us. Think of this exception as the framework's way of saying:

Okay, your controller is requesting an implementation of `OrderRepositoryInterface` ...but you haven't told me which one to use. I'm adding this to Fail Blog, son.

Perhaps you want to use an Eloquent-specific implementation of the interface: `EloquentOrderRepository`. To use it, create a new binding to make a connection between the interface and its implementation.

```
1 App::bind('OrderRepositoryInterface', 'EloquentOrderRepository');
```

To reiterate, think of this as your way of saying:

“Yo, Laravel - when you need an instance of `OrderRepositoryInterface`, I want you to use `EloquentOrderRepository`. K bro?”

Remember, this line of code is identical to:

```
1 App::bind('OrderRepositoryInterface', function()
2 {
3     return new EloquentOrderRepository;
4});
```

Now, when the `OrdersController` is instantiated, Laravel will correctly pull in the Eloquent implementation of the repository. Why is this so useful? Because, when testing controllers (which you'll soon learn how to do), by allowing for this level of flexibility, injecting a mocked version of the repository will be a

cinch!



Tip: An existing instance may be bound to the container with `App::instance()`.

Example 2: Automatic Resolution

Let's review a second example to see just how far the rabbit hole goes. What if a `UsersController` class requires two dependencies: a repository and user validation service?

```
1 App::bind('UsersController', function()
2 {
3     $repository = new Repositories\User;
4     $validator = new Services\Validators\User;
5
6     return new UsersController($repository, $validator);
7 });
```

For experimentation purposes, you may continue to throw this code in your routes file; however, for real-world projects, you'll likely want to create a service provider to register these types of bindings. Nonetheless, if we run...

```
1 $usersController = App::make('UsersController');
```

...the closure will fire, and return a new instance of `UsersController` with those two dependencies. Fine - we already know this. But, wait: Laravel is a smart little cookie! Through a technique referred to as automatic resolution, Laravel can do the brunt of the work for you, thanks to the power of PHP reflection.

Okay, too much jargon here; let's simplify things. When resolving an object through the IoC container, Laravel will attempt to read the constructor's type-hints and automatically inject the instance for you.



Definition: "PHP 5 comes with a complete reflection API that adds the ability to reverse-engineer classes, interfaces, functions, methods and extensions. Additionally, the reflection API offers ways to retrieve doc comments for functions, classes and methods." - php.net

Consider the following constructor:

```
1 class UsersController {
2     public function __construct(UserRepository $user, Validator
3         $validator)
4     {
5     }
6 }
```

When resolved, Laravel will detect that instances of both UserRepository and Validator are required. It will then instantiate and inject them *automagically*! Folks, this is a very cool thing. I repeat: Laravel will automatically inject instances of those two classes for you; no manual binding necessary.

Extra Credit

If you'd like to learn more about how Laravel allows for this, have a look at Illuminate\Container\Container - specifically, the build method. See below (but don't worry if it's overwhelming).

```
1 <?php
2 //
3 // vendor/laravel/framework/src/Illuminate\Container\Container.php
4 /**
5 * Instantiate a concrete instance of the given type.
6 *
7 * @param string $concrete
8 * @param array $parameters
9 * @return mixed
10 */
11 public function build($concrete, $parameters = array())
12 {
13     // If the concrete type is actually a Closure, we will just
14     // execute it and hand back the results of the functions,
15     // which allows functions to be used as resolvers for more
16     // fine-tuned resolution of these objects.
17     if ($concrete instanceof Closure)
```

```
18     {
19         return $concrete($this, $parameters);
20     }
21
22     $reflector = new \ReflectionClass($concrete);
23
24     // If the type is not instantiable, the developer is
25     // attempting
26     // to resolve an abstract type such as an Interface or
27     // Abstract
28     // Class and there is no binding registered for the
29     // abstractions
30     // so we need to bail out.
31     if ( ! $reflector->isInstantiable())
32     {
33         $message = "Target [$concrete] is not instantiable.";
34
35         throw new BindingResolutionException($message);
36     }
37
38     $constructor = $reflector->getConstructor();
39
40     // If there are no constructors, that means there are no
41     // dependencies then we can just resolve the instances of
42     // the
43     // objects right away, without resolving any other types or
44     // dependencies out of these containers.
45     if (is_null($constructor))
46     {
47         return new $concrete;
48     }
49
50     $parameters = $constructor->getParameters();
51
52     // Once we have all the constructor's parameters we can
53     // create
54     // each of the dependency instances and then use the
55     // reflection
56     // instances to make a new instance of this class, injecting
```

```
the
51     // created dependencies in.
52     $dependencies = $this->getDependencies($parameters);
53
54     return $reflector->newInstanceArgs($dependencies);
55 }
```

The end result is that the previous binding example is unnecessary:

```
1 // Unnecessary
2 App::bind('UsersController', function()
3 {
4     $repository = new Repositories\User;
5     $validator = new Services\Validators\User;
6
7     return new UsersController($repository, $validator);
8});
```

Laravel will automatically do this for you. It's only necessary in situations when the instantiation process is a bit too specific for Laravel to figure out automatically.

Summary

When researching inversion of control on the internet, you'll likely come across dozens of incredibly confusing articles and definitions. Make it easy on yourself: Laravel's IoC container is simply a tool that reverses the direction of responsibility, by assisting with class dependency management. When embraced, your applications will become significantly more flexible and testable. Use it.

Chapter 12: Test-Driving Artisan Commands

Exercise

Though Laravel 3 offered a basic means for executing tasks directly from the command line, the available functionality was somewhat limited. Luckily, version four instead leverages the powerful [Symfony Console component](#), which provides, not only structure, but great flexibility and configuration for your commands. I think you'll love it!

Commands 101

Before we dig into testing commands, let's first review the basic process of creating them.

Scaffolding

Artisan ships with a nifty generator that can scaffold the necessary boilerplate code for new commands. For instance, to create the skeleton for a command that expedites the process of adding new users to a `users` table, from the command-line, run:

```
1 php artisan command:make UserCreatorCommand
```

This will generate `app/commands/UserCreatorCommand.php` with the following boilerplate:

```
1 <?php
2
3 use Illuminate\Console\Command;
4 use Symfony\Component\Console\Input\InputOption;
5 use Symfony\Component\Console\Input\InputArgument;
6
7 class UserCreatorCommand extends Command {
8     protected $name = 'command:name';
9
10    protected $description = 'Command description.';
```

```

11
12     public function __construct()
13     {
14         parent::__construct();
15     }
16
17     public function fire() {}
18
19     protected function getArguments()
20     {
21         return array(
22             array(
23                 'example',
24                 InputArgument::REQUIRED,
25                 'An example argument.'
26             )
27         );
28     }
29
30     protected function getOptions()
31     {
32         return array(
33             array(
34                 'example',
35                 null,
36                 InputOption::VALUE_OPTIONAL,
37                 'An example option.',
38                 null
39             )
40         );
41     }
42 }

```

Take note of the name and description properties:

```

1 protected $name = 'command:name';
2 protected $description = 'Command description.';

```

The values provided here will be referenced when running `php artisan` from the command line.



Tip: The console component also supports pseudo-namespacing. To place multiple commands under a `migrate` heading, as Laravel does by default, you'd name them as `migrate:install`, `migrate:make`, etc. When running `php artisan`, they will be grouped.

For a user creator command, we might go with:

```
1 protected $name = 'user:create';
2 protected $description = 'Create a new record in the users
table.';
```

However, adjusting these values alone won't make the command show up when running `php artisan`; it first has to be registered with Artisan, which can be done from `app/start/artisan.php`.

```
1 <?php // app/start/artisan.php
2
3 Artisan::add(new UserCreatorCommand);
```

That should do it! If we now list the available Artisan commands with `php artisan`, the `user:create` command should now display at the bottom. It's so easy!

```

controller:make      Create a new resourceful controller
db
  db:seed           Seed the database with records
key
  key:generate     Set the application key
migrate
  migrate:install  Create the migration repository
  migrate:make      Create a new migration file
  migrate:refresh   Reset and re-run all migrations
  migrate:reset     Rollback all database migrations
  migrate:rollback  Rollback the last database migration
queue
  queue:listen      Listen to a given queue
  queue:work        Process the next job on a queue
session
  session:table    Create a migration for the session database table
user
  user:create       Create a new record in the users table.
• commands-exercise [develop]

```

List available commands with php artisan.

Arguments

The console component makes the process of passing arguments and options to your command a cinch. Specify any available arguments within the `getArguments` method, like so:

```

1 /**
2  * Get the console command arguments.
3  *
4  * @return array
5 */
6 protected function getArguments()
7 {
8     return array(
9         array(
10            'name',
11            InputArgument::REQUIRED,
12            'Name of the user'
13        )
14    );

```

```
15 }
```

Because this code specifies that the name argument is required, via `InputArgument::REQUIRED` (the inverse being `InputArgument::OPTIONAL`), if we attempt to run the command without arguments, a `RuntimeException` will be thrown.

```
● commands-exercise [develop php artisan user:create]
```

```
[RuntimeException]  
Not enough arguments.
```

```
user:create [--example[="..."]] name
```

```
● commands-exercise [develop ]
```

A `RuntimeException` will be thrown if no argument is passed.

Should you require additional arguments, simply return a comma-separated list of arrays from the `getArguments` method.

To capture the value of the supplied name from your code, use `$this->argument(KEY)`. The `fire` method on the class will automatically be triggered when the command is called from the command-line.

Here's an example of capturing the name argument, and printing the response to the console.

```
1 // app/commands/UserCreatorCommand.php  
2  
3 /**
```

```
4 * Execute the console command.
5 *
6 * @return void
7 */
8 public function fire()
9 {
10     $name = $this->argument('name');
11
12     $this->info("The name that you want to add is {$name}.");
13 }
```

```
● commands-exercise [develop php artisan user:create 'Douglas Quaid'
The name that you want to add is Douglas Quaid.
● commands-exercise [develop ]
```



Tip: The Command class provides a handful of helper methods to print strings to the console, including, but not limited to, `info()`, `error()`, and `line()`. Refer to the `Illuminate\Console\Console` class for a full list.

The users of your command can view a list of the arguments and options, via the help dialog.

```
1 php artisan help user:create
```

```
• commands-exercise [develop] php artisan help user:create
Usage:
  user:create name

Arguments:
  name          Name of the user

Options:
  --help (-h)    Display this help message.
  --quiet (-q)   Do not output any message.
  --verbose (-v) Increase verbosity of messages.
  --version (-V) Display this application version.
  --ansi         Force ANSI output.
  --no-ansi      Disable ANSI output.
  --no-interaction (-n) Do not ask any interactive question.
  --env          The environment the command should run under.
```

Use the help dialog to view accepted arguments and options.

Options

Options are just that: optional. They can be helpful for specifying flags which alter how a particular action is executed.

Perhaps your model generator (which we'll build using TDD in the next section) should accept an optional path to the `models` directory. You can accept this value, via the `getOptions` method, like so:

```
1 /**
2  * Get the console command options.
3  *
4  * @return array
5  */
6 protected function getOptions()
7 {
8     return array(
9         array(
```

```
10         'path',
11         'p',
12         InputOption::VALUE_OPTIONAL,
13         'Path to models directory',
14         'app/models'
15     )
16 );
17 }
```

Each option array can accept five parameters:

- Name of the option
- Optional alias: --path becomes -p
- Option variants, including VALUE_OPTIONAL, VALUE_REQUIRED, VALUE_NONE, VALUE_IS_ARRAY
- Description for help dialog
- Optional default value

Updating this single method now allows the user to override the default path to the `models` directory. Here's an example, assuming a `generate:model` command:

```
1 php artisan generate:model Dog --path="app/foo/models"
```

Alternatively, the user may instead reference the alias that we specified:

```
1 php artisan generate:model Dog -p="app/foo/models"
```

If the `path` option is specified, the value provided should be used as the path to the `models` directory; otherwise, our default of `app/models` will be used.

Single Responsibility Principle

It's important to remember that, while you can throw every ounce of logic into your command class, doing so is considered bad practice. Why? Well, once again, we come back to the single responsibility principle.

A command class is responsible for running a command; it shouldn't be concerned with manipulating the file system, or modifying a database record, or

compiling templates. Yes, you can **make** the class handle that work, but try not to. Instead, make use of dependency injection, so that the class can merely call the necessary methods on its dependencies.

You may inject these dependencies from `app/start/artisan.php`. Below is an example of registering a `ModelGeneratorCommand` command, while injecting an instance of Laravel's `Filesystem` class.

```
1 // app/start/artisan.php
2
3 Artisan::add(
4     new ModelGeneratorCommand($app['files'])
5 );
```

Exercise

Now that the basics of creating commands are instilled, let's move on to the chapter exercise. We'll use test-driven development to build a file generator that is similar to my popular [Laravel Generators tool](#). More specifically, we'll create a package that allows the user to rapidly generate a new model, using a base template as its contents.

Chances are high that, when writing new commands, you'll want to share them. The easiest way to do so is to create a package, using Laravel's helpful `workbench`, push it to GitHub, and make it available on Packagist.

This means that we'll need to follow a slightly different process than was covered earlier in this chapter.

Create the Package

Before calling Laravel's `workbench` command, first, update `app/config/workbench.php` with your name and email address. These values will be used, when Laravel scaffolds a new Composer package for you.

```
1 <?php // app/config/workbench.php
2
3 return array(
4     'name' => 'Jeffrey Way',
5     'email' => 'jeffrey@example.com',
```

```
6 );
```

Next, running `php artisan help workbench`, we can see that the command expects one argument: the name of the vendor and package.

```
1 $ php artisan help workbench
2 package      The name (vendor/name) of the package.
```

Typically, *vendor* will be the name of your company. Alternatively, feel free to use any identifier, or even your last name, as I do.

Let's try it:

```
1 $ php artisan workbench way/generators
2 Package workbench created!
3 Loading composer repositories with package information
4 Installing dependencies (including require-dev)
5   - Installing illuminate/support (dev-master cc16ed1)
6     Cloning cc16ed1445c39846c1c9aaccfc9f1648b28bf3c3
7
8 Writing lock file
9 Generating autoload files
```

This will generate a new `workbench` directory, along with the necessary folder structure (following PSR-0) and files for rapidly preparing your package.

```
1 {  
2     "name": "way/generators",  
3     "description": "",  
4     "authors": [  
5         {  
6             "name": "Jeffrey Way",  
7             "email": "jeffrey@jeffrey-way.com"  
8         }  
9     ],  
10    "require": {  
11        "php": ">=5.3.0",  
12        "illuminate/support": "4.0.x"  
13    },  
14    "autoload": {  
15        "psr-0": {  
16            "Way\\Generators": "src/"  
17        }  
18    },  
19    "minimum-stability": "dev"  
20 }
```

0 Words, INSERT MODE, Line 1, Column 1 Spaces: 4 JSON

Laravel's workbench allows you to rapidly build new packages.

Generating the Command

Next, as we did earlier in the chapter, we'll let Laravel handle the process of generating the boilerplate code for our new model generator command. However, this time, as we don't want the command to be placed within `app/commands`, we need to specify a path to the package directory. Luckily, the command `:make` accepts a `--path` option that allows for this.

```
1 php artisan command:make ModelGeneratorCommand --  
path="workbench/way/genera\  
2 tors/src/Way/Generators/Commands"
```

For now, we'll only add a name and description.

```
1 //
```

```
workbench/.../Way/Generators/Commands/ModelGeneratorCommand.php  
1  
2  
3 protected $name = 'generate:model';  
4 protected $description = 'Generate a new model.';
```

Service Providers

In the previous section, we used `app/start/artisan.php` to register the command with Artisan. This time, though, we'll store the instantiation process within a service provider. When Laravel scaffolded your new package, it included a service provider class, `GeneratorsServiceProvider`. Think of this file as the bootstrap for your package.



Definition: Service providers are simply bootstrap classes for packages. By default, they contain two methods: `boot` and `register`. Within these methods you may do anything you like: include a routes file, register bindings in the IoC container, attach to events, or anything else you wish to do.

In the code below, we register the package's Artisan commands, via the `commands` method on the `ServiceProvider` class.

```
1 <?php //  
workbench/.../Way/Generators/GeneratorsServiceProvider.php  
2  
3 namespace Way\Generators;  
4  
5 use Illuminate\Support\ServiceProvider;  
6  
7 class GeneratorsServiceProvider extends ServiceProvider {  
8     protected $defer = false;  
9  
10    public function register()  
11    {  
12        $this->registerModelGeneratorCommand();  
13  
14        $this->commands(  
15            'generate.model'  
16        );
```

```
17     }
18
19     protected function registerModelGeneratorCommand()
20     {
21         $this->app['generate.model'] = $this->app-
22         >share(function($app)
23         {
24             return new ModelGeneratorCommand;
25         });
26     }
27 }
```

The final step - one that every user of the package must perform manually - is to add the service provider to the providers array in app/config/app.php.

```
1 // app/config/app.php
2
3 'providers' => array(
4     // ....
5     'Way\Generators\GeneratorsServiceProvider'
6 )
```

All service providers listed here will automatically be loaded by Laravel.

It should now work! Run `php artisan` to ensure that it does.

```
db
  db:seed          Seed the database with records
generate
  generate:model  Generate a new model.
key
  key:generate    Set the application key
migrate
  migrate:install Create the migration repository
  migrate:make    Create a new migration file
  migrate:refresh Reset and re-run all migrations
  migrate:reset   Rollback all database migrations
  migrate:rollback Rollback the last database migration
queue
  queue:listen    Listen to a given queue
  queue:work      Process the next job on a queue
session
  session:table  Create a migration for the session database table
• commands-exercise [develop]
```



Tip: If distributing this package through Packagist, you'll want to make a note in your `readme` file that the user needs to add the necessary service provider to `app/config/app.php`.

Testing Artisan Commands

So far in this chapter, we haven't performed an ounce of testing. Now that you're comfortable with the basic workflow for creating commands and packages, let's fix that!

The truth is that a command class shouldn't be responsible for too much. As such, the tests for this class will be fairly simple. Before moving forward, though, we first need to pull in two dependencies: `illuminate/console` and `mockery`.

Open the package's `composer.json` file (*not the master `composer.json` file*), and update the `require` objects, like so:

```
1 "require": {
```

```
2     "php": ">=5.3.0",
3     "illuminate/support": "4.0.x",
4     "illuminate/console": "4.0.x"
5 },
6 "require-dev": {
7     "mockery/mockery": "dev-master"
8 }
```

This specifies that the package expects to have access to Illuminate's console component, as well as Mockery to behave as expected. Run composer update to pull in both of these.

Next, we'll write the first test. When the workbench command generated the package, it also included a folder for any tests. Within that directory, add a new `commands/ModelGeneratorCommandTest.php` file.

The screenshot shows a code editor with a sidebar containing a file tree. The tree includes 'OPEN FILES' and 'FOLDERS' sections, followed by a 'commands-exercise' folder which contains 'app', 'bootstrap', 'public', 'vendor', 'workbench', 'way', 'generators', 'src', and 'Way'. Under 'Way', there are 'Generators' and 'Commands' subfolders, with 'ModelGeneratorCommand.php' selected. Other files in the tree include 'ModelGenerator.php', 'GeneratorsServiceProvider.php', 'tests', 'commands', 'ModelGeneratorCommandTest.php', '.gitkeep', 'vendor', '.gitignore', '.travis.yml', 'composer.json', 'composer.lock', 'phpunit.xml', '.gitattributes', '.gitignore', 'artisan', 'composer.json', 'composer.lock', 'CONTRIBUTING.md', 'phpunit.xml', 'readme.md', and 'server.php'. The main editor area displays the following PHP code:

```
1 <?php namespace Way\Generators\Commands;
2
3 use Way\Generators\Generators\ModelGenerator;
4 use Illuminate\Console\Command;
5 use Symfony\Component\Console\Input\InputOption;
6 use Symfony\Component\Console\Input\InputArgument;
7
8 class ModelGeneratorCommand extends Command {
9
10    /**
11     * The console command name.
12     *
13     * @var string
14     */
15    protected $name = 'generate:model';
16
17    /**
18     * The console command description.
19     *
20     * @var string
21     */
22    protected $description = 'Generate a new model.';
23
24    /**
25
```

Notice how the directory structure of our tests mirror the production files.

Testing output to a console can get a bit tricky with PHPUnit, but, luckily, the Symfony team have already considered this. Their console component provides a test helper that eases the process considerably. For example, to test that a

dummy command simply outputs *The name argument is foo*, we could write:

```
1 <?php //  
workbench/.../tests/commands/ModelGeneratorCommandTest.php  
2  
3 use Way\Generators\Commands\ModelGeneratorCommand;  
4 use Symfony\Component\Console\Tester\CommandTester;  
5  
6 class ModelGeneratorCommandTest extends  
PHPUnit_Framework_TestCase {  
7     public function testOutput()  
8     {  
9         $tester = new CommandTester(new  
ModelGeneratorCommand);  
10  
11         # Pass in any arguments or options  
12         $tester->execute(['name' => 'foo']);  
13  
14         $this->assertEquals(  
15             "The name argument is foo\n",  
16             $tester->getDisplay()  
17         );  
18     }  
19 }
```

The most important bit of this code is that we instantiate Symfony's `CommandTester` class, and pass in an instance of the command that we wish to test - in this case, `ModelGeneratorCommand`. We then trigger its `execute` method, providing any applicable arguments. Eventually, behind the scenes, the `fire` method on your command will be triggered. Lastly, the `getDisplay` method on the `CommandTester` instance is used to fetch any output.

In its current state, running `phpunit` will, of course, fail.

```
1 1) ModelGeneratorCommandTest::testOutput  
2 Failed asserting that two strings are equal.  
3 --- Expected  
4 +++ Actual  
5 @@ @@
```

```
6 - 'The name argument is foo
7 + ''
```

To make it pass, from the `ModelGeneratorCommand` command, we only need to print the necessary string.

```
1 // workbench/.../Generators/Commands/ModelGeneratorCommand.php
2
3 public function fire()
4 {
5     $this->info('The name argument is ' . $this-
6     >argument('name'));
6 }
```

And that returns us to green! Congratulations; you've just tested your first dummy command.

Planning

Before writing a test, it's always a smart idea to determine what you expect to happen. That's one of the core benefits to a TDD cycle. It forces you to think before you write. In this case, when the command fires, I expect it to fetch the path and model name to generate, and then pass that data on to a special model generator class that will take care of the rest. This way, we limit the command class's responsibility. Perhaps the generator will return a boolean, indicating whether or not the model was created successfully.

Expectations

Now that we have a game plan, let's use Mockery to prepare the first test: `testGeneratesModelSuccessfully`.

```
1 <?php //
workbench/.../tests/commands/ModelGeneratorCommandTest.php
2
3 use Way\Generators\Commands\ModelGeneratorCommand;
4 use Symfony\Component\Console\Tester\CommandTester;
5 use Mockery as m;
6
7 class ModelGeneratorCommandTest extends
```

```

PHPUnit_Framework_TestCase {
8     public function tearDown()
9     {
10         $m::close();
11     }
12
13     public function testGeneratesModelSuccessfully()
14     {
15         # We're not interested in testing the model generator
16         $gen =
17         $m::mock('Way\Generators\Generators\ModelGenerator');
18
19         # We only want to ensure that its make() method is
called.
20         # We'll have it return true to mimic a positive
21         outcome.
22         $gen->shouldReceive('make')
23             ->once()
24             ->with('app/models/Foo.php')
25             ->andReturn(true);
26
27         $command = new ModelGeneratorCommand($gen);
28
29         $tester = new CommandTester($command);
30         $tester->execute(['name' => 'foo']);
31
32         # Ensure that the proper response is printed.
33         $this->assertEquals(
34             "Created app/models/Foo.php\n",
35             $tester->getDisplay()
36         );
37     }
38 }

```

Dependency Injection

The first step to making this test pass is to ensure that the generator can be injected into the `ModelGeneratorCommand` class through its constructor. This

way, we can easily swap out the generator with the mocked version, as illustrated above.

```
1 // workbench/.../Generators/Commands/ModelGeneratorCommand.php
2
3 <?php namespace Way\Generators\Commands;
4
5 use Way\Generators\Generators\ModelGenerator;
6 use Illuminate\Console\Command;
7 use Symfony\Component\Console\Input\InputOption;
8 use Symfony\Component\Console\Input\InputArgument;
9
10 class ModelGeneratorCommand extends Command {
11     protected $name = 'generate:model';
12
13     protected $description = 'Generate a new model.';
14
15     protected $generator;
16
17     public function __construct(ModelGenerator $generator)
18     {
19         parent::__construct();
20
21         $this->generator = $generator;
22     }
23     // ...
24 }
```

Generator Class

The next step is to create the generator class that will handle the process of creating the model with the necessary boilerplate code. This will be placed in the Way/Generators/Generators directory.

```
1 <?php // workbench/.../Generators/Generators/ModelGenerator.php
2
3 namespace Way\Generators\Generators;
4
5 class ModelGenerator {
```

```
6     public function make() {}
7 }
```

That will do for now. Remember: we're testing the command class, not the generator itself. That will have its own tests, so don't double-up!

Making the Test Pass

To make the test pass, we need to calculate the proper path to the model that should be created, and send that on to the generator class.

Give the `fire` method below close inspection.

```
1 // workbench/.../Generators/Commands/ModelGeneratorCommand.php
2
3 <?php namespace Way\Generators\Commands;
4
5 use Way\Generators\Generators\ModelGenerator;
6 use Illuminate\Console\Command;
7 use Symfony\Component\Console\Input\InputOption;
8 use Symfony\Component\Console\Input\InputArgument;
9
10 class ModelGeneratorCommand extends Command {
11     protected $name = 'generate:model';
12
13     protected $description = 'Generate a new model.';
14
15     protected $generator;
16
17     public function __construct(ModelGenerator $generator)
18     {
19         parent::__construct();
20
21         $this->generator = $generator;
22     }
23
24     public function fire()
25     {
26         $path = $this->getPath();
27     }
28 }
```

```

28         if ($this->generator->make($path))
29     {
30         $this->info("Created {$path}");
31     }
32 }
33
34     protected function getPath()
35     {
36         return $this->option('path') . '/' . ucwords($this-
37 >argument('name')\
38 ) . '.php';
39     }
40
41     protected function getArguments()
42     {
43         return array(array('name',
44                         InputArgument::REQUIRED,
45                         'Name of the model to generate.'
46                     ),
47                     );
48     }
49 }
50
51     protected function getOptions()
52     {
53         return array(array('path',
54                         null,
55                         InputOption::VALUE_OPTIONAL,
56                         'Path to the models directory.',
57                         'app/models'
58                     )
59                     );
60     }
61 }
62 }
63 }
```

Even though we haven't yet built the generator class beyond its skeleton, phpunit should return green! The reason is because the generator is being mocked with Mockery. The test merely ensures that the `make` method on the generator class is called, and the necessary response is printed to the console, assuming that the file was generated successfully.

But, what if the model generation was not successful? We need a test for that, too!



Tip: Always provide a test for each path through your code.

```
1 // workbench/.../tests/commands/ModelGeneratorCommandTest.php
2
3 public function testAlertsUserIfModelGenerationFails()
4 {
5     $gen = m::mock('Way\Generators\Generators\ModelGenerator');
6
7     # This time, simulate a failed result
8     $gen->shouldReceive('make')
9         ->once()
10        ->with('app/models/Foo.php')
11        ->andReturn(false);
12
13     $command = new ModelGeneratorCommand($gen);
14
15     $tester = new CommandTester($command);
16     $tester->execute(['name' => 'foo']);
17
18     # If generation failed, the output should indicate as much.
19     $this->assertEquals(
20         "Could not create app/models/Foo.php\n",
21         $tester->getDisplay()
22     );
23 }
```

Making this test pass is a cinch.

```

1 // workbench/.../Generators/Commands/ModelGeneratorCommand.php
2
3 public function fire()
4 {
5     $path = $this->getPath();
6
7     if ($this->generator->make($path))
8     {
9         return $this->info("Created {$path}");
10    }
11
12    $this->error("Could not create {$path}");
13 }

```

The final test for this class will assert that, if a custom path option is specified by the user, the getPath method will respond, accordingly. I'm fairly certain that, as the code stands, this will already work, but always feel free to write additional tests - if only for peace of mind.

```

1 // workbench/.../tests/commands/ModelGeneratorCommandTest.php
2
3 public function testCanAcceptCustomPathToModelsDirectory()
4 {
5     $gen = m::mock('Way\Generators\Generators\ModelGenerator');
6
7     # Ensure that the custom path to the directory is correct
8     $gen->shouldReceive('make')
9         ->once()
10        ->with('app/foo/models/Foo.php');
11
12     $command = new ModelGeneratorCommand($gen);
13
14     $tester = new CommandTester($command);
15     $tester->execute(['name' => 'foo', '--path' =>
16 'app/foo/models']);
16 }

```

Without a single change, phpunit still returns green!

The only remaining thing to do is ensure that, when the `ModelGeneratorCommand` class is registered with Artisan in the service provider, we also inject the `ModelGenerator` class. Remember: we got away without doing this, because we manually injected a mocked version of the model generator class.

```
1 <?php //  
workbench/.../Way/Generators/GeneratorsServiceProvider.php  
2 namespace Way\Generators;  
3  
4 use Way\Generators\Commands\ModelGeneratorCommand;  
5 use Way\Generators\Generators\ModelGenerator;  
6 use Illuminate\Support\ServiceProvider;  
7  
8 class GeneratorsServiceProvider extends ServiceProvider {  
9     protected $defer = false;  
10  
11     public function register()  
12     {  
13         $this->registerModelGeneratorCommand();  
14  
15         $this->commands(  
16             'generate.model'  
17         );  
18     }  
19  
20     protected function registerModelGeneratorCommand()  
21     {  
22         $this->app['generate.model'] = $this->app-  
>share(function($app)  
23         {  
24             # Inject the generator into the command  
25             return new ModelGeneratorCommand(new  
ModelGenerator);  
26         } );  
27     }  
28 }
```

And with that, the `ModelGeneratorCommand` class (along with its associated tests)

is finished! But, certainly, we're not done. So far, we've only mocked the model generator for the purposes of testing the command. Next, we need to test the generator, itself.

Testing the Model Generator

To begin testing the model generator, we, of course, need a new test class, which can be stored within the `tests/generators/` directory.

```
1 <?php // tests/generators/ModelGeneratorTest.php
2
3 use Way\Generators\Generators\ModelGenerator;
4 use Mockery as m;
5
6 class ModelGeneratorTest extends PHPUnit_Framework_TestCase {
7     public function tearDown()
8     {
9         m::close();
10    }
11 }
```

The primary purpose of this class is to facilitate the process of compiling a model template with the values provided by the user, and physically creating the file within the proper `models` directory. With that in mind, the first test will be called `testCanGenerateModelUsingTemplate`.

```
1 // tests/generators/ModelGeneratorTest.php
2
3 public function testCanGenerateModelUsingTemplate() {}
```

We must be careful that our tests don't actually create new files. This serves no purpose other than to slow down the tests. This means that we'll need to mock Laravel's `Filesystem` class; however, in order to make use of that class in our package, it needs to be required as a dependency. Don't forget that Laravel is composed of multiple components.

To pull in this package, simply require it within the package's `composer.json` file, and run `composer update`.

```
1 "require": {
2     "php": ">=5.3.0",
3     "illuminate/support": "4.0.x",
4     "illuminate/console": "4.0.x",
5     "illuminate/filesystem": "4.0.x"
6 }
```

Next, we need some means to verify that the compiled template matches our expected output. To allow for this, we'll add a `tests/generators/stubs` directory. This folder will only house a `model.txt` stub, for the purposes of this tutorial.

```
1 <?php // tests/generators/stubs/model.txt
2
3 class Foo extends Eloquent {
4
5 }
```

The contents of this file represent the output that we expect to be written, when `php artisan generate:model foo` is executed.

Now, we can simply mock the `Filesystem`, and ensure that its `put` method is called with the correct path and data, accordingly.

```
1 // tests/generators/ModelGeneratorTest.php
2
3 public function testCanGenerateModelUsingTemplate()
4 {
5     $file = m::mock('Illuminate\Filesystem\Filesystem[put]');
6
7     $file->shouldReceive('put')
8         ->once()
9         ->with('app/models/Foo.php',
10            file_get_contents(__DIR__ . 'stubs/mod\
11 el.txt'));
12
13     $generator = new ModelGenerator($file);
14     $generator->make('app/models/Foo.php');
```

```
14 }
```



Tip: Mockery allows for partial mocks, by specifying a comma separated list of methods to mock within brackets. All unlisted methods will continue to behave as they normally would.

The code above specifies that, when the `make` method on the model generator is called, we expect that the `put` method on Laravel's helpful `Filesystem` class will be called. More importantly, though, it should be called with the correct arguments: the name of the model to generate, and the contents of the model stub that we created earlier.

Running the tests, as you might expect, will return red.

```
1 $ phpunit
2
3 1) ModelGeneratorTest::testCanGenerateModelUsingTemplate
4 Mockery\Exception\InvalidArgumentException: Method
put("app/models/Foo.php", "\"
5 <?php
6
7 class Foo extends Eloquent {
8
9 }") from Illuminate\Filesystem\Filesystem should be called
10 exactly 1 times but called 0 times.
```

It seems that Mockery expected the `put` method to be called with the proper arguments, but that never occurred.

In order to make the test pass, we need a new `templates` directory that will house a stub for each generator (we only have one for this chapter, but you could certainly add more on your own). Here's the template for generating a new model.

```
1 <?php // workbench/.../Generators/Generatortemplatesmodel.txt
2
3 class {{name}} extends Eloquent {
```

```
5 }
```

Notice the {{name}} portion? Think of it as a placeholder that will ultimately be replaced with the model name designated by the user, when running the command. Assuming `php artisan generate:model foo`, {{name}} will be replaced with Foo.

Now, we have a link between the model template and the test stub. If it was compiled correctly, the two should match!

At this point, we only need to write the necessary logic to fetch the compiled template, and pass its contents on to the `Filesystem`'s `put` method. Give the following snippet some study time.

```
1 <?php // workbench/.../Generators/Generators/ModelGenerator.php
2
3 namespace Way\Generators\Generators;
4
5 use Illuminate\Filesystem\Filesystem as File;
6
7 class ModelGenerator {
8     protected $file;
9
10    public function __construct(File $file)
11    {
12        $this->file = $file;
13    }
14
15    public function make($path)
16    {
17        $name = basename($path, '.php');
18        $template = $this->getTemplate($name);
19
20        if (! $this->file->exists($path))
21        {
22            return $this->file->put($path, $template);
23        }
24
25    return false;
```

```

26     }
27
28     protected function getTemplate($name)
29     {
30         $template = $this->file-
>get(__DIR__. 'templatesmodel.txt');
31
32         return str_replace('{{name}}', $name, $template);
33     }
34 }
```

The code is really quite simple. The `getTemplate` method fetches the model template and runs a quick a search and replace on it, using the user's provided model name. The results of this operation will be passed on to the `Filesystem`'s `put` method, which we've mocked for the test. As noted earlier, we don't want to physically create this file; we merely need to ensure that the appropriate method was, in fact, called.

That's all it takes! We're back to green! Don't forget, though, that, because the model generator expects an instance of `Filesystem` upon instantiation, we need to update the service provider to pass it in.

```

1 // workbench/.../Way/Generators/GeneratorsServiceProvider.php
2
3 protected function registerModelGenerator()
4 {
5     $this->app['generate.model'] = $this->app-
>share(function($app)
6     {
7         $generator = new ModelGenerator($app['files']);
8
9         return new ModelGeneratorCommand($generator);
10    });
11 }
```

That should do it! The tests pass; the only remaining thing to do is try it out for real! `cd` to the root of your application, and run:

```
php artisan generate:model foo.
```

As our tests proved would happen, `app/models/Foo.php` will be created with the following boilerplate:

```
1 <?php  
2  
3 class Foo extends Eloquent {  
4  
5 }
```

Go ahead and try out a few inversions of the command to prove that everything is functioning as expected.

```
● commands-exercise [develop] php artisan generate:model foo  
      Created app/models/Foo.php  
● commands-exercise [develop] php artisan generate:model foo  
      Could not create app/models/Foo.php  
● commands-exercise [develop] php artisan generate:model foo --path="app/way/models"  
      Created app/way/models/Foo.php  
● commands-exercise [develop]
```

Great work!

Summary

In this chapter, we leveraged Symfony's Console component to build a model

generator with tests. Testing console output can be a bit tricky, but, luckily, the Symfony team already considered this, and included a `CommandTester` class to ease the process considerably.

Now, even your custom Artisan commands will be test-driven!

Chapter 13: Testing APIs

Testing an API isn't overly different from any other form of functional testing: seed the database, login a user (if applicable), call a route, and verify the response. Once you know what you're doing, it really couldn't be simpler!

APIs in Laravel

One of the wonderful things about Laravel is that it was so clearly created (and updated) to remedy real-world problems and irritations. As a result, often, you'll find that implementing a particular piece of functionality doesn't require more than a few moments. Building APIs is no different. In fact, as I was writing the demo API for this chapter, I found myself laughing at how easy it is. Even five years ago, in PHP, this would have required the better part of a day to implement. Luckily, those days are gone.

Three Components to Writing an API in Laravel

Coding an API from scratch is beyond the scope of this chapter (though I'll cover the process considerably on [Tuts+ Premium](#) in July, 2013). That said, I'd be remiss not to review the essential components.

1. Authentication

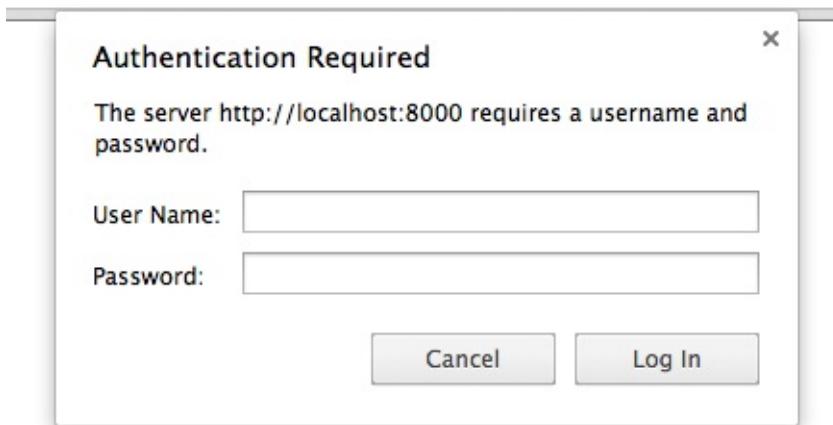
One of Laravel 4's excellent new features is support for basic HTTP authentication. In addition to offering an easy way to create a private section of your web site (when a full login system isn't necessary), HTTP-based authentication can also be a perfect choice for simple APIs.

Triggering basic auth only requires that you reference a native filter: `auth.basic`. Here's an example:

```
1 // app/routes.php
2
3 Route::get('admin', ['before' => 'auth.basic', function()
4 {
5     return 'Secret admin page.';
6 }]);
```

```
6 }]);
```

Believe it or not, that's all you need to password-protect a page or group! Laravel for the win!



HTTP authentication

To instead reference this filter from your controllers, use the `beforeFilter` method within the controller's constructor, like so:

```
1 class MyController {
2     public function __construct()
3     {
4         $this->beforeFilter('auth.basic');
5     }
6 }
```

Now, each route will be protected with basic authentication.

So, how might we leverage this functionality for authenticating API users? Quite simply, in fact: create a resource, and reference the filter before all applicable routes.

The route:

```
1 // app/routes.php
2
3 Route::resource('photos', 'PhotosApiController');
```

And then the controller:

```
1 // app/controllers/PhotosApiController.php
2
3 class PhotosApiController {
4     public function __construct()
5     {
6         $this->beforeFilter('auth.basic');
7     }
8 }
```

Alternatively, to prevent Laravel from setting a user identifier cookie in the session, you could create another filter within `app/filters.php` and reference that.

```
1 Route::filter('auth.basic.once', function()
2 {
3     return Auth::onceBasic();
4});
```

This technique is particularly effective for APIs.



Tip: Basic authentication for an API is the easiest way of verifying a user, however, it's not overly secure. For more significant APIs, you'll likely want to instead opt for an API key and secret. Nonetheless, as this chapter isn't dedicated to API design, let's keep things simple and stick with basic HTTP authentication.

2. Route Prefixing

As this is an API, it's a smart idea to future-proof it by nesting the associated routes, using a prefix, such as `api/v1`.

```
1 Route::group(['prefix' => 'api/v1'], function() {
2     Route::resource('photos', 'PhotosApiController');
3 });
```

Pay close attention to the naming convention for this prefix. By adopting a consistent pattern, such as `api/v1` or `api/v2`, we're able to extend the API's

functionality in the future, while continuing to provide a consistent interface for those who are still dependent upon an older version of the API.

```
• photos-chapter php artisan routes
+-----+-----+-----+
| URI           | Name          | Action        |
+-----+-----+-----+
| GET /api/v1/photos      | api.v1.photos.index | PhotosApiController@index |
| GET /api/v1/photos/create | api.v1.photos.create | PhotosApiController@create |
| POST /api/v1/photos       | api.v1.photos.store | PhotosApiController@store |
| GET /api/v1/photos/{photos} | api.v1.photos.show | PhotosApiController@show |
| GET /api/v1/photos/{photos}/edit | api.v1.photos.edit | PhotosApiController@edit |
| PUT /api/v1/photos/{photos} | api.v1.photos.update | PhotosApiController@update |
| PATCH /api/v1/photos/{photos} |                   | PhotosApiController@update |
| DELETE /api/v1/photos/{photos} | api.v1.photos.destroy | PhotosApiController@destroy |
+-----+-----+-----+
• photos-chapter |
```

Using a prefix for API calls allows for future extensibility.

3. Return JSON

Though you could provide support for returning both XML and JSON from your API, why bother? It's 2013: stick with JSON-only. Besides, what kind of developer prefers XML over JSON when querying a web service? I haven't met these fictional creatures.

By default, when a collection is returned from a controller method, that data will be translated into JSON. This means, if you want to return a JSON-representation for a user with an `id` of 1, you'd only need to write:

```
1 public function show($id)
2 {
3     return User::find($id); // returns JSON
4 }
```

However, convenient as this is, when building APIs, you'll likely want to

provide additional feedback to the API-caller. To manually return a JSON response from a route, use the `json` method on the `Response` class. Here's an example that fetches all photos for the authenticated user, and returns a JSON response.

```
1 class PhotosApiController extends \BaseController {
2     public function index()
3     {
4         return Response::json([
5             'error' => false,
6             'photos' => Auth::user()->photos->toArray()
7         ], 200);
8     }
9 }
```

Don't forget: once a user has been authenticated, you may access their associated `user` object with `Auth::user()`. In effect, to capture the `id` of the authenticated user, use `Auth::user()->id`.

Assuming that all photos are publicly viewable, this will work perfectly; however, if your web app instead limits a photo's visibility to the user associated with the photo, then don't forget to leverage the authentication filter that we reviewed in the previous section.

```
1 class PhotosApiController extends \BaseController {
2     public function __construct()
3     {
4         $this->beforeFilter('auth.basic');
5     }
6
7     public function index()
8     {
9         return Response::json([
10            'error' => false,
11            'photos' => Auth::user()->photos->toArray()
12        ], 200);
13    }
14 }
```

The easiest way to test a web service is with [curl](#). In the image below, notice how, with very little code, we were able to correctly respond to incorrect and valid credentials.

```
❶ photos-chapter curl --user jeffrey@envato.com:wrongpass localhost:8000/api/v1/photos
Invalid credentials.%  
❷ photos-chapter curl --user jeffrey@envato.com:1234 localhost:8000/api/v1/photos
{"error":false,"photos":[{"id":1,"path":"my-image.jpg","caption":"My First Image","user_id":1,"created_at":"2013-05-21 21:13:41","updated_at":"0000-00-00 00:00:00"}]}%  
❸ photos-chapter %
```

Use curl to test your API

Testing APIs

While, as you'll find, testing an API is relatively straight-forward, there are, nonetheless, a few best practices to consider.

Use a Database in Memory

As these are functional tests, we'll of course be hitting a database. If you haven't already, first ensure that you've created a test database, along with its respective configuration file.



Did You Know?: When unit testing, Laravel will automatically set the environment to *testing*, and, subsequently, override all production configuration files with the ones stored in `app/config/testing`.

Assuming that your API is mostly CRUD-based, chances are high that you can opt for a Sqlite database in memory, which should provide a performance boost (*though I've seen mixed benchmarks*). To instruct Laravel to use a DB in memory, update `app/config/testing/database.php` to set a new default driver for testing.

```
1 <?php // app/config/testing/database.php
2
3 return array(
4     'default' => 'sqlite',
5
6     'connections' => array(
7         'sqlite' => array(
8             'driver'    => 'sqlite',
9             'database'  => ':memory:',
10            'prefix'    => '',
11        )
12    )
13 );
```

Pay close attention to the fact that, rather than setting the `database` key to a `.sqlite` file, instead, a database in memory is specified. With this modification, when testing, Laravel will automatically set the environment to `testing` and reference the in-memory DB.

Migrate the Database for Each Test

As we plan to reference a database in memory, it's important to migrate and seed the database before each test, like so:

```
1 class PhotosApiTest extends TestCase {
2     public function setUp()
3     {
4         parent::setUp();
5
6         Artisan::call('migrate');
7         $this->seed();
8     }
9 }
```

The above code leverage the Artisan facade to call the `migrate` command (`php artisan migrate`). This will construct all necessary tables. Secondly, the `seed()` method is used to populate the tables with seed records (if applicable). Alternatively, you could do `Artisan::call('db:seed')`.

Enable Filters

One pitfall that you'll undoubtedly come across is when Laravel seemingly ignores route filters when unit testing. As a result, when testing your API, even though you may expect a test to fail authentication, PHPUnit will still return green (because the filter never fired)! Yikes!

To compensate for this, before each test, always activate the filters.

```
1 public function setUp()
2 {
3     parent::setUp();
4
5     Route::enableFilters();
6 }
```

Set the Authenticated User

Lastly, when testing APIs that require authentication, specify an authenticated user before each test. Let's say that you want to set the currently logged in user to the one with an `id` of 1. There are two ways to do this:

```
1 $this->be(User::find(1));
2 // Or:
3 Auth::loginUsingId(1);
```

Test Examples

With a few guidelines in place, let's review some examples!

User Must Be Authenticated

In our fictional photo storage app, we've decided that access to all photos should be limited to the user in which they're associated. As such, a good first test to write is one that asserts that, if a particular route is requested, and credentials are not provided, *Invalid Credentials* will be returned.

```

1 // app/tests/api/PhotoApiTest.php
2
3 class PhotoApiTest extends TestCase {
4
5     public function setUp()
6     {
7         parent::setUp();
8
9         Route::enableFilters();
10
11         Artisan::call('migrate');
12         $this->seed();
13
14         Auth::loginUsingId(1);
15     }
16
17     public function testMustBeAuthenticated()
18     {
19         // Most tests will assume a logged in user
20         // But not this one.
21         Auth::logout();
22
23         $response = $this->call('GET', 'api/v1/photos');
24
25         $this->assertEquals('Invalid credentials.', $response-
26         >getContent());
27     }
28 }

```

Pretty simple, ay?

Check For Error

Please note that, for all remaining tests, the `setUp()` method will be assumed.

Let's write a test that verifies whether an `error` property exists on the returned object. This should provide some added feedback as to whether the request was successful or not.

```
1 public function testProvidesErrorFeedback()
2 {
3     $response = $this->call('GET', 'api/v1/photos');
4     $data = json_decode($response->getContent());
5
6     $this->assertEquals(false, $data->error);
7 }
```

Fetch All Photos For the Authenticated User

Next, we need a test that fetches all photos for an authenticated users, and verifies whether the correct JSON is returned.

```
1 public function testFetchesAllPhotosForUser()
2 {
3     $response = $this->call('GET', 'api/v1/photos');
4     $content = $response->getContent();
5     $data = json_decode($content);
6
7     // Did we receive valid JSON?
8     $this->assertJson($content);
9
10    // Decoded JSON should offer a photos array
11    $this->assertInternalType('array', $data->photos);
12 }
```

When testing an API, one of the most important assertions you can make is one that verifies whether real JSON was returned. Luckily, as of PHPUnit v3.7, an `assertJson` method is available.

Refactoring

Already, though, I can see that some cleanup is in order. If the tests return green, I'm free to refactor. Let's break this into pieces.

Here's a new test that verifies whether valid JSON was returned from the request.

```
1 public function testReturnsValidJson()
2 {
```

```
3     $response = $this->call('GET', 'api/v1/photos');
4
5     $this->assertJson($response->getContent());
6 }
```

Next, one for verifying that the photos array is available.

```
1 public function testFetchesPhotos()
2 {
3     $response = $this->call('GET', 'api/v1/photos');
4     $data = json_decode($response->getContent());
5
6     $this->assertInternalType('array', $data->photos);
7 }
```

Or, if you're feeling particularly rambunctious, you could write an assertion that decodes a JSON string, and validates whether a particular key exists on it. You crazy kid.

```
1 public function testFetchesPhotos()
2 {
3     $response = $this->call('GET', 'api/v1/photos');
4
5     $this->assertJsonStringHasKey('photos', $response-
6     >getContent());
7 }
8
9 protected function assertJsonStringHasKey($key, $json)
10 {
11     $data = json_decode($json);
12
13     $this->assertInternalType('array', $data->$key);
14 }
```

Updating a Photo

Sure, fetching a list of your photos is great, but what about creating or updating? Making such a request with curl might take the form of:

```
1 curl -X PATCH --user jeffrey@envato.com:1234 -d
"caption='Changed my photo'\
2 " localhost:8000/api/v1/photos/1
```

When working with curl, -x refers to the request type (PUT, DELETE, etc.), and -d is for data that should be sent through.

The following example will set a factory for a Photo and save it to the DB, and then call the necessary API route to update the record.

```
1 public function testUpdatesExistingPhoto()
2 {
3     // Poor man's factory
4     $photo = new Photo;
5     $photo->caption = 'Some Photo';
6     $photo->path = 'foo.jpg';
7     $photo->user_id = 1;
8     $photo->save();
9
10    $updatedPhotoFields = [ 'caption' => 'Updated Photo
Caption' ];
11
12    $response = $this->call('PATCH', 'api/v1/photos/1',
$updatedPhotoFields\
13 );
14
15    $data = json_decode($response->getContent());
16
17    $this->assertEquals('Photo has been updated', $data-
>message);
18    $this->assertEquals('Updated Photo Caption',
Photo::find(1)->caption);
19 }
```

Notice how we're passing the parameters as the third argument to the call method. From the controller, you can capture these values as you normally would when responding to a form's submission.

```
1 $fields = Request::get(); // Or Input::get() if you prefer
2 $caption = Request::get('caption');
```

Once the route has been triggered, we merely inspect the returned JSON and ensure that the message is equal to “*Photo has been updated.*”

Factories

In the previous snippet, we used what I’d refer to as a poor man’s factory. Let’s clean that up using my Factory class that we reviewed in a previous chapter. You can pull it into your project by either installing [Laravel 4 Generators](#) (which includes the package) or [Laravel Test Helpers](#). Personally, I use the custom generators in every project, so that’s what we’ll use here.



Definition: In the simplest possible terms, a factory manages the creation of objects with dummy data.

To replace this big chunk of code:

```
1 // Poor man's factory
2 $photo = new Photo;
3 $photo->caption = 'Some Photo';
4 $photo->path = 'foo.jpg';
5 $photo->user_id = 1;
6 $photo->save();
```

We only need to write:

```
1 Way\_TESTS\Factory::create('Photo');
```

Or, better yet, use the class at the top of the file.

```
1 use Way\Tests\Factory;
2 // ...
3 Factory::create('Photo');
```

This will achieve the exact same result as what we had before, except, this time,

the class will fake the values for each field. To override any fields, pass an array as the second argument to the create method, like so:

```
1 Factory::create('Photo', ['caption' => 'Some caption']);
```

With this modification, the testUpdatesExistingPhoto spec may be updated to:

```
1 public function testUpdatesExistingPhoto()
2 {
3     Factory::create('Photo');
4     $updatedPhotoFields = ['caption' => 'Updated Photo
Caption'];
5
6     $response = $this->call('PUT', 'api/v1/photos/1',
$updatedPhotoFields);
7     $data = json_decode($response->getContent());
8
9     $this->assertEquals('Photo has been updated', $data-
>message);
10    $this->assertEquals('Updated Photo Caption',
Photo::find(1)->caption);
11 }
```

Specifying Options

Basic API design dictates that options should be declared within the query string. Try not to use verbs within your URIs. As such:

```
1 api/v1/photos?color=green
```

...is clearly a smarter choice than:

```
1 api/v1/getGreenPhotos
```

Hopefully, that goes without saying. Let's write a test that specifies a limit for the number of photos that should be returned.

```
1 public function testCanSetALimit()
2 {
```

```

3   // Give a new user two photos
4   Factory::create('User');
5   Factory::create('Photo', ['user_id' => '1']);
6   Factory::create('Photo', ['user_id' => '1']);
7
8   // Declare a limit of 1
9   $response = $this->call('GET', 'api/v1/photos?limit=1');
10  $data = json_decode($response->getContent());
11
12  // Verify that the photos array count is 1, not 2
13  $this->assertCount(1, $data->photos);
14 }

```

To implement this type of functionality, we might write:

```

1 public function index()
2 {
3     $photos = Auth::user()->photos();
4     $photos = $this->applyOptions($photos);
5
6     return Response::json([
7         'error' => false,
8         'photos' => $photos->get()->toArray()
9     ], 200);
10 }
11
12 protected function applyOptions($photos)
13 {
14     if ($limit = Request::get('limit'))
15         $photos->take($limit);
16
17     return $photos;
18 }

```

Summary

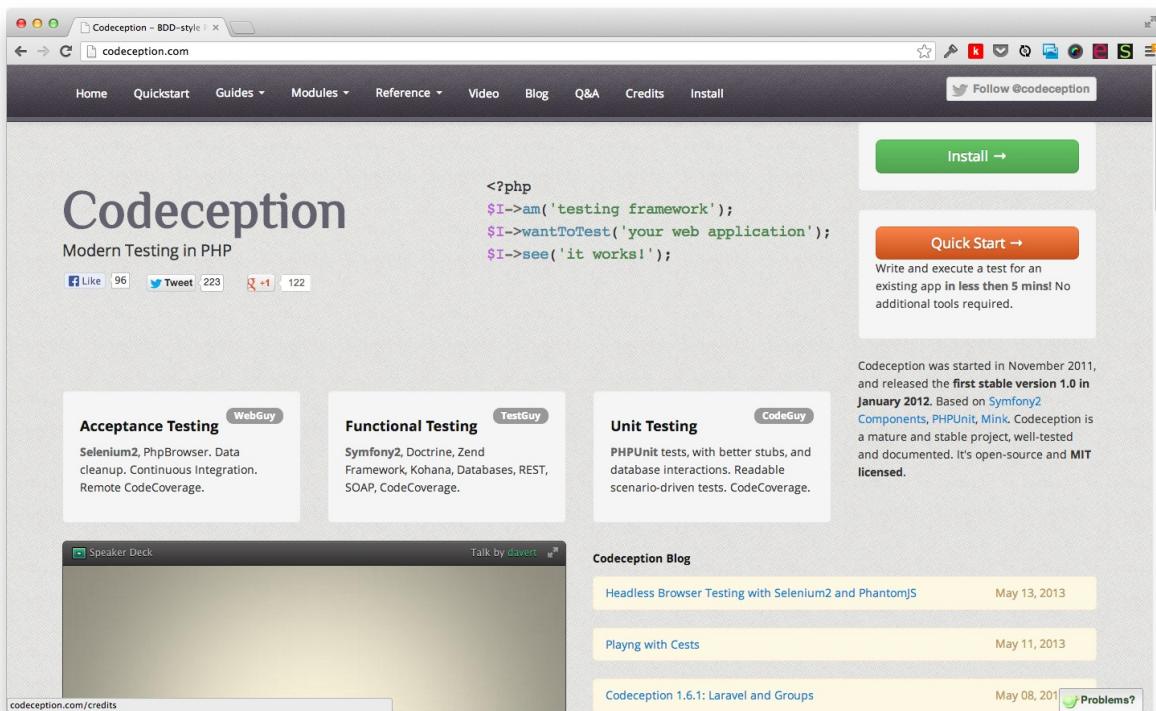
One thing worth noting is that we exclusively relied upon functional (or some might refer to this as integration) testing. In a real-world project, you'd likely want to create unit tests for the PhotosApiController class as well, providing

the best of both worlds.

In this chapter, we used a Sqlite database in memory and Factories to test a fictional photos API. As I noted at the beginning, the basic process should be quite familiar to you at this point. Really, it's no different than any other type of testing: arrange, act, and then assert (AAA).

Chapter 14: Acceptance Testing With Codeception

By itself, PHPUnit does not offer too many helpers for writing functional tests. In a previous chapter, we made use of a couple Symfony components to inspect the DOM (BrowserKit and DomCrawler), however, the process was, in this author's opinion, still too cumbersome. As they say, the more difficult a test is to write, the more likely it is that the test simply won't be written. Wouldn't it be amazing if we could interact with our web apps on a high level using a simple, readable interface that even non-developers could parse? Codeception, which builds upon popular components, like PHPUnit and Mink, is the solution.



codeception.com

“Codeception is kept as simple as possible for any kind of user. PHP developers, QAs, and managers can use Codeception. The only requirements are a basic knowledge of PHP, and theory of automated

testing. Its configuration is kept short; all common issues are already solved.” - codeception.com

An Amuse Bouche

Before we dig further into how Codeception allows us to better test our applications, here’s a quick teaser of the syntax. As a *hello world* example, let’s say we want to ensure that, when /login is requested, the text, “*Login*”, is displayed. Using Codeception, we could write:

```
1 $I = new WebGuy($scenario);
2 $I->amOnPage('/login');
3 $I->see('Login');
```

Notice how readable that is! Codeception places a significant emphasis on readability. Strip away as much clutter as possible, and get right down to what you need to test. It even goes so far as to recommend that you use an `$I` variable when instantiating your scenarios. This is because they’re meant to mimic the feature requests that your client might give you. For example:

```
1 When I am on the login page
2 I expect to see the text Login
```

While tools like Cucumber emphasize plain text scenarios, Codeception tweaks the formula just a bit to remove any potential duplication. Even better, if needed, it can auto-generate plain-text scenarios.

```
1 codecept generate:scenarios
```

But this is just the tip of the iceberg; Codeception can make assertions against what’s stored in the database, perform AJAX requests, submit forms, click links, and much more...all using the same basic syntax demonstrated above.

Testing Refresher

So far in this book, we’ve mostly focused on testing in isolation. There’s a reason for this: roughly eighty percent of your tests should be unit tests. These need to be *greased lightning* fast, so that you can easily run the full suite each time a file is saved.

Surprised that this percentage is so high? Well think about it: if we were building a time machine, we'd definitely want to run some tests for ensuring that the car works correctly for the customer (Marty). For instance:

- Get in the car
- Put key into ignition and start car
- Turn on the time circuits
- Set date to November 5th, 1955
- Ensure that flux capacitor is fluxing
- Step on the gas
- Accelerate to 88mph
- Assert that electric sparks fly

This is an acceptance test! Notice how it flexes the entire system (or Delorean). This is probably a good test to write! But what happens if, for some reason, the car doesn't move when you step on the gas? In the real-world, slamming your head into the steering wheel won't magically fix things. How could you immediately detect what went wrong? It could be a loose wire, could be a faulty pedal, gas might not be getting to the engine thing (*I know nothing about cars - hence, "engine thing"*), who knows? So it seems that testing from a *bird-eye* level, though incredibly important, also has the unavoidable side effect of being a high level test. From way up there, you can't see a thing.

From this, we can deduce that, even though this test is essential, we also require lower level tests, which ensure that each component of the car - the starter, the fuel injection manifold (I know that one from *Back to the Future 3*), the radio - functions as expected.

The key is to find the perfect ratio of unit tests, versus functional and acceptance tests. As a basic rule of thumb (remember that rules may be broken), plan on the number of unit test lines being the same as your production code. 80% unit, with the remainder dedicated to integration, functional, and acceptance tests.

Acceptance Testing

Acceptance tests execute code from the outside in, and will run in an environment that is as close to production as possible. This means that they will hit the database, query real web services, *etc*. In effect, they provide incredibly accurate and real-world results, but, as a side effect, have no choice but to be orders of magnitude slower than your unit tests, which can be executed in a

matter of seconds. If these were the only tests you wrote, they would break the incremental TDD cycle. Waiting even ten seconds for tests to run is out of the question.

To easily remember what acceptance testing refers to, lock on to the word, *accept*. When building new features, the customer (whoever that might be) will typically describe the functionality that they require. We refer to these as *user stories*.

¹ As a user

² To more easily consume Tuts+ Premium content

³ I want to download each course as a zip file

Notice how these stories are quite generalized - almost like a roadmap that charts out where you currently are, and where you're going. They don't delve into the specifics (*when I view this page, I want to see a download button that links to the course's zip file*). There's a time and place to fetch that information from the customer. But, to start, a user story should be condensed. Think of it as a newspaper's two sentence description of some new movie that's out in theaters.

Though it doesn't always play out this way in real life (not your fault), presumably, once this test passes, the feature has been fully implemented, and should be *acceptable* to the customer. If it isn't acceptable, then, at some point, wires became crossed, and the spec wasn't written correctly.

“I've implemented the feature you requested. Do you accept that it works as expected?”

Tools like Cucumber were specifically written to give managers and quality assurance folks the ability to write these high-level tests for you, themselves. It's a cute end-goal, but, I've yet to meet a team that functioned in this way. 95% of the time, you'll find yourself (or someone else on your team) writing the acceptance test, based upon prior discussions with the clients.

The screenshot shows the Cucumber website with a green header bar containing links for WIKI, EXAMPLES, TUTORIALS, TICKETS, MAILING LIST, and IRC. Below the header is the Cucumber logo with the tagline "behaviour driven development with elegance and joy". The main content area is divided into several sections:

- 1: Describe behaviour in plain text**: A code snippet showing a feature file with steps for adding two numbers.
- 2: Write a step definition in Ruby**: A code snippet showing a step definition for the addition feature.
- 3: Run and watch it fail**: A terminal window showing the command \$ cucumber features/addition.feature and its output, which fails because the step definitions are missing.
- 4. Write code to make the step pass**: A code snippet showing the implementation of the step definition for the addition feature.
- 5. Run again and see the step pass**: A terminal window showing the command \$ cucumber features/addition.feature and its output, which now passes.
- 6. Repeat 2-5 until green like a cuke**: A terminal window showing the command \$ cucumber features/addition.feature and its output, which is now green (passed).
- 7. Repeat 1-6 until the money runs out**: A note stating that Cucumber lets software development teams describe how software should behave in plain text, using a business-readable domain-specific language and serving as documentation, automated tests, and more.
- Learn more!**: A link to the Cukeup! conference.
- Download**: Instructions for downloading Cucumber.

Tools like Cucumber were specifically written to give managers and quality assurance folks the ability to write these high-level tests themselves.

Installation

As with any new tool, before we can dig in, we first need to install Codeception. Like Composer, there are a couple different ways to download it.

Global Installation

One way to install Codeception is to manually fetch its archive file.

```
1 wget http://codeception.com/codecept.phar
```

Next, you'll probably want to move it somewhere permanent, so that it can be referenced globally.

```
1 mv codecept.phar /usr/local/bin/codecept
```

Finally, don't forget to ensure that it's executable.

```
1 chmod +x /usr/local/bin/codecept
```

That should do it! Test it out by creating a new tab and running `codecept`. If installed correctly, you'll see a list of all available commands.

```
● Desktop  wget http://codeception.com/codecept.phar
--2013-05-17 16:58:20--  http://codeception.com/codecept.phar
Resolving codeception.com... 207.97.227.245
Connecting to codeception.com[207.97.227.245]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1297773 (1.2M) [application/octet-stream]
Saving to: `codecept.phar.1'

100%[=====] 1,297,773  2.00M/s  in 0.6s

2013-05-17 16:58:21 (2.00 MB/s) - `codecept.phar.1' saved [1297773/1297773]

● Desktop  mv codecept.phar /usr/local/bin/codecept
● Desktop  chmod +x /usr/local/bin/codecept
● Desktop  
```

Installing Codeception globally

Local Installation

A second way that we can pull in Codeception is [through Composer](#).

We first update the composer.json file to reference the Codeception package (I'll be using version 1.6.11 in this chapter).

```
1 "require": {
2     "laravel/framework": "4.0.*"
3 },
4 "require-dev": {
5     "codeception/codeception": "1.6.1.1"
6 },
```

Next, of course, we update Composer to download the new package.

```
1 composer update
```

And that's it! The codecept executable will be located at vendor/bin/codecept. To view the list of commands (as we did before), either add vendor/bin to your system path, or reference its full path, like so:

```
1 vendor/bin/codecept
```

Success!

```
● codecept-chapter vendor/bin/codecept
Codeception version 1.6.1.1

Usage:
[options] command [arguments]

Options:
--help      -h Display this help message.
--quiet     -q Do not output any message.
--verbose   -vvvvv Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug
--version   -V Display this application version.
--ansi      Force ANSI output.
--no-ansi    Disable ANSI output.
--no-interaction -n Do not ask any interactive question.

Available commands:
analyze      Analyzes for non-existent methods and adds them to corresponding helper
bootstrap    Initializes empty test suite and default configuration file
build        Generates base classes for all suites
console      Launches interactive test console
help         Displays help for a command
list         Lists commands
run          Runs the test suites
generate
generate:cept  Generates empty Cept file in suite
generate:cest  Generates empty Cest file in suite
```

Codeception's list of commands

Now that we've successfully installed it on our system, let's do some testing.

Bootstrapping

Codeception is a full-stack testing framework and needs to be *initialized* for each new project.

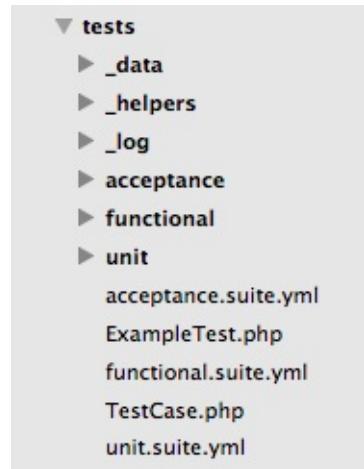
```
1 codecept bootstrap
```

This command will dynamically generate a number of files within a tests/ directory. For language-agnostic projects, this will do just fine; however, for Laravel applications, we want these files to be placed within app/tests. As

such, when bootstrapping, be sure to include the path argument to explicitly set a custom install path.

```
1 codecept bootstrap app
```

Upon running this command, a suite of files will be added to the app/tests directory.



After running codecept bootstrap app

Specifically, make note of how tests are divided into unit, functional, and acceptance. As we reviewed earlier in this book, this is a popular convention that provides an easy way to execute a specific suite of tests. When writing unit tests, you likely don't want to wait for the slow acceptance tests to run as well!

Configuring Acceptance Tests

Because acceptance tests work from the outside in, for Codeception to run properly, you need to be running a server (`php artisan serve`), and must provide the URL for your app within `app/tests/acceptance.suite.yml`.

```
1 class_name: WebGuy
2 modules:
3     enabled:
4         - PhpBrowser
5         - WebHelper
6     config:
7         PhpBrowser:
8             url: 'http://localhost:8000'
```

While you're here, make a mental note that each suite's functionality may be extended with modules. We'll review this more shortly.

Generate a Test

Let's write one dummy test to observe the basic cycle, and then we'll move on to real-world tests. Tests may be created in two ways: manually or generated. The only difference is that, when using the latter approach, Codeception will provide a couple lines of boilerplate code to get you started. Either method will do just fine.

Manual Approach

Create a new file, `app/tests/acceptance/welcomeCept.php`, and append:

```
1 <?php
2
3 $I = new WebGuy($scenario);
4 $I->wantTo('Check the home page for a welcome message');
5 $I->amOnPage('/');
6 $I->see('Welcome');
```

Generator Approach

From the command line, run:

```
1 codecept generate:cept acceptance Welcome
```

If this command is run from the root of your Laravel application, a configuration exception will be thrown, noting that Codeception could not locate the `codeception.yml` file. This is because we set a custom path to the install directory. To compensate, use the `-c` option to set the path to the configuration file as well.

```
1 codecept generate:cept acceptance Welcome -c app
```

This can get cumbersome, though. That's a lot to write for the sole purpose of generating a single file with two lines of boilerplate code. Let's make a couple of changes to remove the need to specify the path to the configuration file.

Move `app/codeception.yml` to the root of your project.

```
1 mv app/codeception.yml codeception.yml
```

Next, update this file, and, within the `paths` object, replace all references to `tests/` with `app/tests`, like so:

```
1 paths:
2   tests: app/tests
3   log: app/tests/_log
4   data: app/tests/_data
5   helpers: app/tests/_helpers
```

That should do it! Now, we can simply write:

```
1 codecept generate:cept acceptance Welcome
```

This will generate `app/tests/acceptance/WelcomeCept.php` with:

```
1 <?php
2
3 $I = new WebGuy($scenario);
4 $I->wantTo('perform actions and see result');
```

Decoding the Command

The `generate` command consists of a few components.

- The name of the command to run (`generate:cept`). Codeception supports two different formats: Cept and Cest. We'll review both of these in the coming chapters.
- The name of the suite to add the file to (*unit, functional, acceptance*)
- The name of the test.

Scenario	Suite name	Test name

codecept generate:cept acceptance Welcome

Decoding the command

Both the manual and generate approach will achieve the same end result. It's up to you to decide which method you most prefer.

Writing the First Test

Depending on which method you chose for creating a new acceptance test, ensure that the acceptance/WelcomeCept.php file contains:

```

1 <?php
2
3 $I = new WebGuy($scenario);
4 $I->wantTo('Check the home page for a welcome message');
5 $I->amOnPage('/');
6 $I->see('Welcome');
```

The wonderful thing about scenario-based tests in Codeception is that the methods above don't require much explanation; they're self explanatory.

Notice how all tests are written in present tense - `$I->see()` rather than `$I->shouldSee()`.

The only method that may require further explanation is `wantTo()`. Though it isn't required, it does help to provide a better description for the test, when viewing the results. For example:

With:

```
1 1) Couldn't check the home page for a welcome message in WelcomeCept.php
```

Without:

```
1 1) WelcomeCept.php
```

As a best practice, opt for readability and always include this method call.

Running All Tests

The codecept run command may be used to trigger all tests in your application.

```
1 codecept run
```

```
● codecept-chapter codecept run
Codeception PHP Testing Framework v1.6.1.1
Powered by PHPUnit 3.7.19 by Sebastian Bergmann.

Suite acceptance started
Trying to check the home page for a welcome message (WelcomeCept.php) - Failed

Suite functional started

Suite unit started

Time: 0 seconds, Memory: 9.25Mb

There was 1 failure:

-----
1) Couldn't check the home page for a welcome message in WelcomeCept.php
Guy couldn't see "Welcome": 'Welcome' in page..
Failed asserting that response contains "welcome". Response was saved to 'log' directory.

Scenario Steps:
2. I see "Welcome"
1. I am on page "/"

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
● codecept-chapter
```

Running tests

In the image above, notice how Codeception will highlight the scenario step that it failed on. This can be incredibly helpful when determining exactly what went wrong. In this case, Codeception failed to find *Welcome* in the response.

Technically, this is a bit misleading. Even if no / route exists, step one will seemingly pass. To ensure that the response code is correct, you may use the `seeResponseCodeIs(200)` method.

To make the test pass, we only need to create a route that returns the string, `Welcome`.

```
1 // app/routes.php
2
3 Route::get('/', function()
4 {
5     return 'Welcome';
6});
```

And we're green!

```
1 Suite acceptance started
2 Trying to check the home page for a welcome message
(WelcomeCept.php) - Ok
3
4 Suite functional started
5 Suite unit started
6
7 Time: 0 seconds, Memory: 9.50Mb
8 OK (1 test, 2 assertions)
```

It's so easy!



Tip: By default, all test suites will be triggered when running `codecept run`. To override this, as the first argument to the `run` command, specify the name of the suite to run: `codecept run acceptance`. This will, in effect, only trigger the tests within `app/tests/acceptance`. For further filtering, a second argument may be passed, which will specify the name of the single test to run. Don't forget that you can view the documentation for any command by preceding its name with `help`: `codecept help run`.

Summary

In this chapter, we reviewed the absolute basics of testing with Codeception, but,

clearly, there is much more to cover.

With the basic syntax under your fingers, in the next *Exercise* chapter, we'll continue using acceptance testing to build a relatively simple login form with authentication.

Chapter 15: Authentication With Codeception Exercise

Because features - and acceptance testing in general - come with a relatively significant performance cost, it's important to limit them to regions of your application that require them. This can include everything from authentication, to billing, to implementing a new feature for your web app. In other words, verifying every possible path is not something that should be done on the acceptance testing level. That's what all the other styles of testing are for.

You see, when coding, you'll find yourself writing tests that span the full spectrum of the application: some acceptance tests to define the end goal for the customer, unit tests to ensure that each component works correctly, some integration tests for verifying the integration of two or more classes, *etc.*

Perhaps you're still wondering why anything but acceptance tests are necessary. I asked this very question at one point in my learning. Why not write a few tests that describe the completed feature, and then code away until those pass? Well, there's a variety of reasons why we don't do this. But, as with anything, the best way to learn why is through experience. In this chapter, we'll exclusively use acceptance tests to build a login feature for a client. As you work along, hopefully, you'll begin to realize where acceptance tests fall short (speed and feedback).

The Feature

Let's imagine that our fictional customer would like to have a new private (password-protected) administration area for their website. The user story might be:

¹ In order to perform administrative tasks

² As the site owner

³ I want to login to a password-protected private area

Translating the Feature for Codeception

The first step is to create a new acceptance test. We'll use the Cest format in this chapter, which allows each *path* to be contained more traditionally within a class method.

```
1 codecept generate:cest acceptance Login
```

Using Codeception, the above user story may be translated to:

```
1 $I->am('Site Owner');
2 $I->wantTo('login to a password-protected area');
3 $I->lookForwardTo('perform administrative tasks');
```

Think of the code above less as performing a specific action, and more defining the story background.

At this point, at a high level, we can write code to interact with the web application in the same way that a human being might. This is why these types of tests are referred to as executing from the outside-in: they exercise the entire application, rather than one or two components (which would be integration testing).

Below is our first test for logging in with proper credentials.

```
1 <?php // app/tests/acceptance/LoginCest.php
2
3 class LoginCest {
4
5     public function logsInUserWithProperCredentials(WebGuy $I)
6     {
7         $I->am('Site Owner');
8         $I->wantTo('login to a password-protected area');
9         $I->lookForwardTo('perform administrative tasks');
10
11         $I->amOnPage('/admin');
12         $I->seeCurrentUrlEquals('/login');
13
14         $I->fillField('email', 'jeffrey@envato.com');
15         $I->fillField('password', '1324');
16         $I->click('Login');
```

```

16
17     $I->seeCurrentUrlEquals('/admin');
18     $I->see('Admin Area', 'h1');
19 }
20
21 }

```

Again, notice how, even if you weren't a developer, you could probably figure out what this code does.

1. Visit the admin page, but expect to be redirected to the login page, as it should be protected.
2. Fill out the username and password, and click the *Login* button
3. Expect to be redirected to `/admin`, and see the text, *Admin Area*

Running the tests at this point will, of course, lead to failure.

```

1 1) Couldn't login to a password-protected area in
LoginCest.loginWithProper\
2 Credentials
3 Guy couldn't see current url equals "/login": Failed
asserting that two str\
4 ings are equal.
5 --- Expected
6 +++ Actual
7 @@ @@
8 - '/login'
9 + '/admin'

10
11 Scenario Steps:
12 4. I see current url equals "/login" <-- RED
13 3. I am on page "/admin"
14 2. So that I perform administrative tasks
15 1. As a Site Owner

```

Looks like we have our first step!

Register Routes

According to the tests, we require two routes:

1. /admin - Should be auth protected
2. /login - For creating a new user session

Per usual, these can be added to app/routes.php.

```
1 <?php // app/routes.php
2
3 Route::get('admin', ['before' => 'auth', function()
4 {
5     // Temporary
6     return '<h1>Admin Area</h1>';
7 }]);
8
9 Route::get('login', function()
10 {
11     return View::make('sessions.create');
12 });
```

If we run the tests again, we'll receive the next step:

```
1 1) Couldn't login to a password-protected area in
LoginCest.loginWithProper\
2 Credentials
3 Guy couldn't fill field "email","jeffrey@envato.com": Field
matching id|nam\
4 e|label|value or css or xpath selector does not
exist
5
6 Scenario Steps:
7 5. I fill field "email","jeffrey@envato.com" <-- RED
8 4. I see current url equals "/login"
9 3. I am on page "/admin"
10 2. So that I perform administrative tasks
11 1. As a Site Owner
```

Clearly, Codeception can't fill in form fields that don't exist.

[Building the Form](#)

BUILDING THE FORM

The next step is to add the login form. As this project doesn't have a master page, let's keep things simple and embed the full HTML here.

```
1 <!-- app/views/sessions/new.blade.php -->
2 <!doctype html>
3 <html>
4     <head>
5         <meta charset="utf-8">
6         <title>Login</title>
7     </head>
8
9     <body>
10        <h1>Login</h1>
11
12        {{ Form::open() }}
13            <div>
14                {{ Form::label('email', 'Email') }}
15                {{ Form::text('email') }}
16            </div>
17
18            <div>
19                {{ Form::label('password',
'Password') }}
20                {{ Form::password('password') }}
21            </div>
22
23            <div>
24                {{ Form::submit('Login') }}
25            </div>
26        {{ Form::close() }}
27    </body>
28 </html>
```

Should we run the tests now, we'll see that, when the form was submitted, Codeception expected to be redirected to /admin, but was not.

¹ Scenario Steps:

² 8. I see current url equals "/admin" <-- RED

```
3 7. I click "Login"
4 6. I fill field "password", "1324"
5 5. I fill field "email", "jeffrey@envato.com"
6 4. I see current url equals "/login"
7 3. I am on page "/admin"
8 2. So that I perform administrative tasks
```

Resources

To make this test pass, we first need a new resourceful controller for managing user sessions.

```
1 php artisan controller:make SessionsController
```

We also need to update the routes file to register these routes.

```
1 // app/routes.php
2
3 Route::resource('sessions', 'SessionsController');
```

Now, we can update the view, and specify that the form should POST to the store method of SessionsController.

```
1 {{ Form::open(['method' => 'post', 'route' => 'sessions.store'])}}
}}
```

Authenticating the User

In order to write authentication code, we first needs a users table within our test database. My [Laravel 4 Generators](#) tool is the quickest way to tackle tasks like this:

```
1 php artisan generate:migration create_users_table --
fields="email:string:un\
2 ique, password:string"
```

This takes care of the migration and schema, but we also need a test record within this table. Laravel offers seed files for this very purpose.

```
1 php artisan generate:seed Users
```

This command will generate the following boilerplate, excluding the test record that I've already inserted.

```
1 <?php // app/database/seeds/UserTableSeeder.php
2
3 class UsersTableSeeder extends Seeder {
4
5     public function run()
6     {
7         $user = new User;
8         $user->email = 'joe@example.com';
9         $user->password = Hash::make('1234');
10        $user->save();
11    }
12
13 }
```

Adding a Test Database

Before moving forward, ensure that you have a test database setup. This can be accomplished by creating `app/config/testing/database.php`. This will have the effect of overriding the default DB settings.

```
1 <?php // app/config/testing/database.php
2
3 return array(
4     'connections' => array(
5         'mysql' => array(
6             'driver'      => 'mysql',
7             'host'        => 'localhost',
8             'database'   => 'my-test-db',
9             'username'   => 'root',
10            'password'   => '1234',
11            'charset'    => 'utf8',
12            'collation'  => 'utf8_unicode_ci',
13            'prefix'     => '',
14        )
15    )
16)
```

```
15      )
16 );
```

Be sure to replace the credentials above with your own. Also, Codeception needs to know these details as well. They can be added to the master `codeception.yml` file that should be in the root of your project.

```
1 modules:
2     config:
3         Db:
4             dsn: 'mysql:host=localhost;dbname=TEST_DB_NAME'
5             user: 'USERNAME'
6             password: 'PASSWORD'
7             dump: app/tests/_data/dump.sql
```

But, that's it; you now have a test database ready to go!

Let's migrate and seed this new users table.

```
1 php artisan migrate --seed --env="testing"
2 Migrated: 2013_05_19_151342_create_users_table
3 Database seeded!
```

At this point, we can authenticate the user within SessionsController.

```
1 // app/controllers/SessionsController
2 public function store()
3 {
4     $creds = [
5         'email' => Input::get('email'),
6         'password' => Input::get('password')
7     ];
8
9     if (Auth::attempt($creds)) return Redirect::to('admin');
10 }
```

Run the tests again, and we get green!

Invalid Credentials

Invalid Credentials

I'd like to write one more test, though. Let's make sure that, if incorrect credentials are specified, then the login page is reloaded, along with an *Invalid Credentials* error message.

```
1 // app/tests/acceptance/LoginCest.php
2
3 public function loginWithInvalidCredentials(WebGuy $I)
4 {
5     $I->amOnPage('/login');
6     $I->click('Login');
7
8     $I->seeCurrentUrlEquals('/login');
9     $I->see('Invalid Credentials', '.flash');
10 }
```

Run the tests:

```
1 1) LoginCest.loginWithInvalidCredentials
2 Guy couldn't see current url equals "/login": Failed asserting
that two str\
3 ings are equal.
4 --- Expected
5 +++ Actual
6 @@ @@
7 - '/login'
8 + '/sessions'

9
10 Scenario Steps:
11 3. I see current url equals "/login" <-- RED
12 2. I click "Login"
13 1. I am on page "/login"
```

This is one of those situations where we don't know exactly what the next step is. That's the downside to writing acceptance tests. We've proven that this piece of functionality doesn't work, but we don't have enough tests to determine precisely what to do next. Now, if we had also written some unit and functional tests, this would never be the case. Nonetheless, we'll continue on.

Let's update the controller's store method to redirect back to the login page if authentication fails.

```
1 // app/controllers/SessionsController.php
2
3 public function store()
4 {
5     $creds = [
6         'email' => Input::get('email'),
7         'password' => Input::get('password')
8     ];
9
10    if (Auth::attempt($creds)) return Redirect::to('admin');
11
12    return Redirect::to('login')->withInput();
13 }
```

The final step is to ensure that the *Invalid Credentials* message displays, as the latest running of the tests show:

```
1 Scenario Steps:
2 4. I see "Invalid Credentials", ".flash" <-- RED
3 3. I see current url equals "/login"
4 2. I click "Login"
5 1. I am on page "/login"
```

In Laravel, we can flash messages when redirecting by passing a session key and value pair to the with method.

```
1 return Redirect::to('login')
2     ->withInput()
3     ->with('message', 'Invalid Credentials');
```

To capture this value from the view, we only need to verify whether the session object has a message key, and, if so, display it within a <div>.

```
1 @if (Session::has('message'))
2     <div class="flash">
```

```
3      {{ Session::get('message') }}
```

```
4  </div>
```

```
5 @endif
```

And that should do it! If working along, manually check your work in the browser to prove that the login form functions as expected.



Tip: Once you embrace a consistent TDD cycle, you'll find that, most of the time, your browser will remain closed.

Summary

Hopefully, if you worked along with this chapter, you found that, though helpful, we require more fine-grained tests to handle the situations when a scenario step fails without providing enough details. In the following chapter, we'll toy around with functional testing in Codeception, which offers mostly the same API, while offering a few bonuses.

Chapter 16: Functional Testing in Codeception

In Codeception, the process of writing functional tests is nearly identical to acceptance testing, which we discussed in the previous chapter. There are only two core differences:

- Functional tests don't require a server to execute. This can provide a performance boost. Instead, requests and responses will be emulated.
- Functional tests provide better output for debugging. Rather than noting that a scenario step failed, functional tests will display the exception message directly in the console.

The *Laravel4* Module

Codeception is extensible in the form of modules. While acceptance tests are framework-agnostic (meaning you could even use Codeception to write tests for a .NET application), the same is not true for your functional tests. Luckily, Codeception provides a *Laravel4* module that can hook into our favorite framework.

The *Laravel4* module may be enabled by editing `functional.suite.yml`. By default, this file will include the `Filesystem` and `TestHelper` modules. Let's extend it to support both `Laravel4` and `Db`.

```
1 class_name: TestGuy
2 modules:
3     enabled: [Filesystem, TestHelper, Laravel4, Db]
```

The DB Module

The *DB* module requires a bit of setup. First, edit the global configuration file - `codeception.yml` - and provide your database credentials within the *DB* configuration block. Don't forget to update the DSN, username, and password with your credentials.

```
1 modules:
```

```
2     config:
3         Db:
4             dsn: 'mysql:host=localhost;dbname=DB_NAME'
5             user: 'USERNAME'
6             password: 'PASSWORD'
7             dump: app/tests/_data/dump.sql
```

The *DB* module's core responsibility is to clean up the database after each test. This way, we can ensure that each test is working with the same data set. To populate your database, Codeception requires a raw SQL dump. You can either tackle this directly from a GUI, like Sequel Pro (*via an “Export” command*), or most systems should have a `mysqldump` executable available from the command line. This SQL dump should be placed within `app/tests/_data/dump.sql`. Using the previously mentioned executable, we can dump our database to this location in a single command:

```
1 mysqldump --opt --user="USERNAME" --password="PASSWORD" DBNAME >
app/tests/\_
2 _data/dump.sql
```

As always, replace values in *all caps* with your own credentials.

Updating TestGuy

The final step is to re-build the `TestGuy` class. Any time that you add or remove a module from a suite, re-run the `build` command. The reason why is because files, like `TestGuy`, `CodeGuy`, and `WebGuy` are dynamically generated, based upon the modules that you've included.

```
1 codecept build
2 /Users/Jeffrey/Desktop/codecept-
chapter/app/tests/acceptance/WebGuy.php gen\
3 erated successfully. 39 methods added
4 /Users/Jeffrey/Desktop/codecept-
chapter/app/tests/functional/TestGuy.php ge\
5 nerated successfully. 52 methods added
6 /Users/Jeffrey/Desktop/codecept-
chapter/app/tests/unit/CodeGuy.php generate\
7 d successfully. 0 methods added
```

Registering a User

In the previous chapter, we wrote an acceptance test for logging a user in. This time, using functional testing, let's write a test for registering a new user. This test should walk through the process of visiting the /register page, filling out the form, and then verifying whether the new user was saved to the database.

As always, we begin by generating a new test - this time, for the functional suite.

```
1 codecept generate:cept functional Registration
```

Next, let's write the spec for registering a user.

```
1 <?php // app/tests/functional/RegistrationCept.php
2
3 $I = new TestGuy($scenario);
4 $I->wantTo('register for a new account');
5 $I->lookForwardTo('be a member');
6
7 $I->amOnPage('/register');
8 $I->see('Register', 'h1');
9 $I->fillField('Email:', 'joe@example.com');
10 $I->fillField('Password:', '1234');
11 $I->click('Register Now');
12
13 $I->seeCurrentUrlEquals('/login');
14 $I->see('You may now sign in!', '.flash');
15 $I->seeInDatabase('users', ['email' => 'joe@example.com']);
```

Did you notice that last method, `seeInDatabase`? This references some of the sugar that the `Db` module provided. With a single line of code, we can peek into the database, and ensure that a new record was, in fact, added to the `users` table.

Don't forget that, for each test, the database will be refreshed.

This time, if we run the tests, rather than Codeception failing on a scenario step without providing any feedback, we'll see that an exception was thrown, along with the stack trace.

```
1 1) Couldn't register for a new account in RegistrationCept.php
2 Symfony\Component\HttpKernel\Exception\NotFoundHttpException:
```

It's squawking because the /register route doesn't exist.

```
1 // app/routes.php
2
3 Route::get('register', function()
4 {
5     return View::make('register');
6 });
```

Next, let's move ahead and create the registration form.

```
1 <!doctype html>
2 <html>
3 <head>
4     <meta charset=utf-8>
5     <title>Register</title>
6 </head>
7
8 <body>
9     <h1>Register</h1>
10
11    {{ Form::open() }}
12        {{ Form::label('email', 'Email:') }}
13        {{ Form::text('email') }}
14
15        {{ Form::label('password', 'Password:') }}
16        {{ Form::text('password') }}
17
18        {{ Form::submit('Register Now') }}
19    {{ Form::close() }}
20 </body>
21 </html>
```

Great! According to the tests, the only remaining thing to do is respond to the POST request, and then create the user. Forgive me for ignoring validation; let's

keep things barebones for the sake of focusing as much as possible on Codeception.

```
1 // app/routes.php
2
3 Route::post('register', function()
4 {
5     $user = new User;
6     $user->email = Input::get('email');
7     $user->password = Input::get('password');
8     $user->save();
9
10    return Redirect::to('login')
11        ->with('message', 'You may now sign in!');
12});
```

Finally, we'll display the flash message to the user, and should then be done!

```
1 <body>
2     <h1>Register</h1>
3
4     @if (Session::has('message'))
5         <div class="flash">
6             {{ Session::get('message') }}
7         </div>
8     @endif
9
10    {{ Form::open() }}
11        {{ Form::label('email', 'Email:') }}
12        {{ Form::text('email') }}
13
14        {{ Form::label('password', 'Password:') }}
15        {{ Form::text('password') }}
16
17        {{ Form::submit('Register Now') }}
18    {{ Form::close() }}
19 </body>
```

Back to green. Did you notice that we implemented this functionality without

ever touching a browser? And the best part is that, for the life of this application, we will have a test that verifies whether or not the registration process works as we expect it to.

```
◆ codecept-chapter codecept run functional
Codeception PHP Testing Framework v1.6.1.1
Powered by PHPUnit 3.7.19 by Sebastian Bergmann.

Suite functional started
Trying to register for a new account (RegistrationCept.php) - Ok
```

Time: 1 second, Memory: 14.50Mb

```
OK (1 test, 4 assertions)
◆ codecept-chapter
```

Back to green

Summary

The fact that this chapter is so short is a testament to how easy Codeception is to use. Tools that are built well should not require hundreds of pages of documentation. It's really quite simple: define how the user wishes to interact with their application using a readable DSL, and then write the necessary code (while still leveraging traditional unit tests) to make it pass.

Chapter 17: Continuous Integration With Travis CI

[Travis](#) is a popular open-source hosted continuous integration service that offers support for a number of languages (originally beginning with Ruby), including PHP. Wait, what, huh? Continuous integration?



Definition: Popularized by Kent Beck and Martin Fowler, and a core *Extreme Programming* development methodology, **continuous integration** is the process of frequently (even multiple times a day) merging (or integrating) your local source revisions in an attempt to avoid what we refer to as *integration hell*. Each integration is then verified by an automated build server, like Travis or [Jenkins](#). This cycle can help to prevent compatibility issues.

Continuous integration assumes that you and your team leverage a version control system, like [Git](#). Even ten years ago, terms like *source control* were scary things. These days, though, it's a basic part of our development process. So chances are, even if you don't realize it, you may already be practicing continuous integration (mostly).

1. Check out a copy of the source.
2. Create a branch and perform the bug fix or addition.
3. Hours (not days or weeks) later, when the tests are green, fetch and merge the latest changes.
4. If no collisions occur (meaning the tests still pass), push your changes to GitHub. To repeat, only integrate when the build is successful.
5. Upon receipt, a CI server will automatically build the project again (in an environment that's as close to production as possible) and trigger all of the tests. If the tests pass, you've successfully integrated.
6. Rinse and repeat.

Item number five above might be the only step (an important one) that you don't follow in your current workflow. Because there is always the possibility that you didn't update your local copy correctly, it's important to have a central continuous integration server that executes automatically. Travis CI is one such example: think of it as the teacher who checks over your work. The ultimate

goal is to have a system for detecting errors within a matter of hours, rather than days or weeks. Git, GitHub, PHPUnit (with tests), and Travis provide the perfect recipe for integrating (excuse the pun) CI into your team's daily workflow.

“Any individual developer’s work is only a few hours away from a shared project state and can be integrated back into that state in minutes. Any integration errors are found rapidly and can be fixed rapidly.” - [Martin Fowler](#)

One advantage to this rapid integration is that it encourages developers to break their work into small, manageable chunks.

Though it may seem that continuous integration is primarily for development teams, this isn’t the case. Even as a solo developer, services like Travis will prove to be quite helpful.

Hello, Travis

Imagine that you have a popular GitHub repository that others frequently contribute to. Next, imagine if, for each pull request, a service, like Travis, would notify you if their committed changes have broken your code. Pretty neat, right? But what if you want to test the package against multiple version of PHP (5.4, 5.3, etc.)? No problem: add a few characters to your configuration file and you’re done.

“Our CI environment provides multiple runtimes (e.g. Node.js or PHP versions), data stores and so on. Because of this, hosting your project on travis-ci.org means you can effortlessly test your library or applications against multiple runtimes and data stores without even having all of them installed locally.” - [travis-ci.org](#)

If you’re worried that I’ve yet again introduced another tool to learn (*when will it end*), you’ll be happy to know that Travis is a cinch to use: connect to GitHub, register a service hook, add a configuration file, and you’re done! Three simple steps.

1. Connect

To get started, the first step is to connect your GitHub account to Travis.

The screenshot shows the Travis CI dashboard for the repository `BitLucid/ninjawars`. The repository has 49 builds, the latest being build 49, which started 1 min 3 sec ago. The build status is green. The commit message is `Merge remote-tracking branch 'origin/news' into news`. The configuration includes environment variables: `INSTALL=full_install`, `INSTALL=integration_install`, and `Php: 5.3.3, 5.3, 5.4`. Below the build details is a **Build Matrix** table:

Job	Duration	Finished	Env	Php
49.1	-	-	INSTALL=full_install	5.3.3
49.2	-	-	INSTALL=full_install	5.3
49.4	1 min 3 sec	-	INSTALL=integration_install	5.3.3

Connect Travis to GitHub

2. Register Hooks

Next, view your profile, and enable one of your GitHub packages that includes tests (you can create a dummy project for testing purposes). In the screenshot below, I'm activating tests for my “Test Helpers” package that we reviewed in the testing models chapter. When checked, Travis will automatically register the necessary service hook with GitHub.



Tip: To manually register service hooks (such as for Packagist) in GitHub, visit the *Settings* page for one of your repositories, click *Service Hooks*, and scroll down to the applicable service.

JeffreyWay/Laravel-Test-Helpers Easier testing in Laravel.



ON

Register the service hooks

3. Configure

As the final step, you have to tell Travis a bit about your project. How can it run your tests, if it doesn't know which language, version, and tools it was built with? Further, some applications may require a test database, setup work, and more. All of this can be declared within a `travis.yml` file, placed in the root of your application.



Did You Know: When creating a package with Laravel's workbench, the framework will automatically generate a `travis.yml` file.

Here's the absolute essentials for a basic PHP project that is dependent upon Composer.

```
1 language: php
2
3 php:
4   - 5.4
5   - 5.3
6
7 before_script:
8   - curl -s http://getcomposer.org/installer | php
9   - php composer.phar install --dev
10
11 script: phpunit
```

This file makes a handful of declarations:

1. Which language are we working with?
2. Which versions of that language must this project be tested against?
3. Anything we need to do before running the tests?
4. Which test script should be used?

Overall, it's fairly simplistic! Remember: this is a Yaml file, so pay close attention to your indentation.

The screenshot shows a code editor window with a dark theme. On the left is a sidebar titled 'OPEN FILES' and 'FOLDERS'. Under 'FOLDERS', there's a single folder named 'laravel-test Helpers'. Inside this folder are several files: 'src', 'tests', 'vendor', '.gitignore', and '.travis.yml'. The file '.travis.yml' is currently selected and open in the main editor area. The code in the file is:

```
language: php
php:
  - 5.4
before_script:
  - curl -s http://getcomposer.org/installer | php
  - php composer.phar install --dev
script: phpunit
```

At the bottom of the editor, it says '13 Words, INSERT MODE, Line 1, Column 1' on the left and 'Spaces: 4' and 'YAML' on the right.

Configuring Travis

Upon adding this new file, commit your changes and push them to GitHub. Once GitHub registers the commit, it will fire off a notification to Travis (your service hooks at work), which, in turn, will read the project's configuration file and execute its tests. Shortly after, you'll receive an email with the results. Here's an example of one I received today, notifying me that the tests all passed.

[Fixed] JeffreyWay/Laravel-Test-Helpers#3 (master - ae6e27e) Inbox x Print Email

 **Travis CI** <notifications@travis-ci.org>
to me, jeffrey ▼

 **Images are not displayed.** Display images below - Always display images from notifications@travis-ci.org

Repository JeffreyWay/Laravel-Test-Helpers
Build #3 <https://travis-ci.org/JeffreyWay/Laravel-Test-Helpers/builds/7473454>
Changeset <https://github.com/JeffreyWay/Laravel-Test-Helpers/compare/4709e1b5ffc...ae6e27ee2564>

Commit ae6e27e (master)
Message Update travis.yml
Author Jeffrey Way
Duration 1 minute and 35 seconds

You can configure recipients for build notifications in your [configuration file](#). Further documentation about Travis CI can be found [here](#). For help please join our IRC channel [irc.freenode.net#travis](#).

Travis will email you the results of the latest integration.

There will be plenty of times, though, when green isn't the color of the day. When the tests invariably fail, Travis will provide a stack trace, detailing the problem.

JeffreyWay/Laravel-Test-Helpers

Easier testing in Laravel.

Current Build History Pull Requests Branch Summary Build #2 Job #2.1

Job	2.1	Commit	4709e1b (master)
State	Failed	Compare	fd7a3476ffca...4709e1b5fffc
Finished	15 minutes ago	Author	Jeffrey Way
Duration	1 min 34 sec	Committer	Jeffrey Way
Message	Remove 5.4 array syntax		
Config	Php: 5.3		

```
1 Using worker: worker-linux-4-1.bb.travis-ci.org:travis-linux-20
2
3 $ git clone --depth=50 --branch=master
4 cd JeffreyWay/Laravel-Test-Helpers
5 git checkout -qf 4709e1b5ffcf22a8abbf1ca691b9a36120e4799
6 phpenv global 5.3
7 php --version
8 PHP 5.3.23 (cli) (built: Mar 31 2013 06:10:11)
9 Copyright (c) 1997-2013 The PHP Group
10 Zend Engine v2.3.0, Copyright (c) 1998-2013 Zend Technologies
11     with Xdebug v2.2.1, Copyright (c) 2002-2012, by Derick Rethans
12 curl -s http://getcomposer.org/installer | php
13 $ php composer.phar install --dev
14 before_script.1
15 before_script.2
16 $ phpunit
17 PHP Parse error: syntax error, unexpected '[' in
18     /home/travis/build/JeffreyWay/Laravel-Test-Helpers/tests/Way/Tests/FactoryTest.php
19     on line 22
20 PHP Stack trace:
21     1. {main}() /home/travis/.phpenv/versions/5.3.23/bin/phpunit:0
22     2. PHPUnit_TextUI_Command::main()
```

Error reports in Travis

Build Configuration

As I noted earlier, Travis is a fairly straight-forward tool. That's what makes it so useful. Having said that, there will certainly be projects that require extended configuration (within your `travis.yml` file). This section will outline various common options.

Absolute Basics

If nothing else, Travis needs to know the language and version of the project.

```
language: php
```

```
2 php:  
3   - 5.4
```

With these three lines alone, Travis will test your project with PHPUnit, against PHP version 5.4. To, say, test against version 5.2 as well, simply add one line:

```
1 language: php  
2 php:  
3   - 5.4  
4   - 5.2
```

Bootstrapping

When running PHPUnit, you will often want to register a bootstrap file, such as Composer's vendor/autoload.php. This sort of configuration should be placed, per usual, within your phpunit.xml file. Remember: Laravel, too, will generate this file for you.

Register Dependencies

Each time that Travis runs your repo's tests, it will build an environment, based upon the configuration file. As such, you must explicitly declare all dependencies or commands that should be executed before running the test script.

The following snippet will download Composer and install all dependencies that are declared within your project's composer.json file.

```
1 before_script:  
2   - curl -s http://getcomposer.org/installer | php  
3   - php composer.phar install --dev
```

Maybe you have a dedicated shell script that will install or prepare various dependencies. That, too, should be added to before_script.

```
1 before_script:  
2   - ./bin/setup-dependencies.sh
```

Notice that these are just simple commands. Anything that you would write in the Terminal may be used here. Perhaps you need to execute an Artisan

~~The terminal may be used here. Perhaps you need to execute an artisan command first:~~

```
1 before_script:  
2     - curl -s http://getcomposer.org/installer | php  
3     - php composer.phar install --dev  
4     - php artisan migrate --seed
```

Notifications

By default, Travis will send email notifications for each commit to the commit author and repository owner. However, this can be modified if necessary.

Turn Off Notifications

```
1 notifications:  
2     email: false
```

Set Recipients

```
1 notifications:  
2     email:  
3         - devmanager@example.com  
4         - logs@example.com  
5         - steve-micromanager@example.com
```

IRC

Travis can even update your team's IRC channel with each commit. Let's say that Taylor wanted Travis to send through the results of each commit to Laravel's IRC channel. To his configuration file, he could add:

```
1 notifications:  
2     irc:  
3         - "chat.freenode.net#laravel"
```

Summary

Continuous integration is not Laravel specific. Nonetheless, it's a pattern that all modern development teams follow in 2013. Are you?

© 2013 Taylor Otwell. All rights reserved. This book is licensed under the Creative Commons Attribution License version 3.0. See the License at <http://creativecommons.org/licenses/by/3.0/>.

Once again, it's an unfortunate truth that scary jargon like this can deter developers. Software development is confusing enough; do we really need to introduce this much confusing terminology? That said, please don't let two words keep you from sleeping better at night. Take a few hours, re-read this chapter, and integrate Travis CI into your development process today.

Frequently Asked Questions (A Living Document)

Wow - you made it this far? Well, I'm sure you have questions. In fact, I guarantee it! Think of this chapter as a living document that will frequently be updated, as I find myself responding to the same questions on Twitter, email, or GitHub. Depending on when you made your purchase, this chapter will either be sparse, or massive!

1. How Do I Test Global Functions?

As a basic rule of thumb, make every effort to avoid recklessly using global functions within your code. If you do, though, you have two options to allow for testability.

Mock It

Let's imagine that a piece of your code uses the native `file_get_contents()` function. Rather than grouping it directly within a particular method, instead, extract it to its own method that you can then mock.

Consider this dummy method that returns the contents of a file.

```
1 public function getContents($file)
2 {
3     return "The content of the given file is: " .
4     file_get_contents($file);
5 }
```

Unfortunately, this function isn't testable without a bit of trickery. Let's fix that by extracting the global function reference.

```
1 public function getContents($file)
2 {
3     return "The contents of the given file is: " . $this-
>fetch($file);
```

```

4 }
5
6 public function fetch($file)
7 {
8     return file_get_contents($file);
9 }
```

Now that we have a wrapper for this function, `getContents` is a cinch to unit test.

```

1 class TheClassTest extends PHPUnit_Framework_TestCase {
2
3     public function testGetContents()
4     {
5         $mock = Mockery::mock('TheClass')->makePartial();
6         $mock->shouldReceive('fetch')
7             ->once()
8             ->with('foo')
9             ->andReturn('bar');
10
11         $sentence = $mock->getContents('foo');
12
13         $this->assertEquals('The contents of the given file
14 is: bar', $sentence);
15     }
16
17 }
```

The Namespacing Trick

A second solution is to use namespacing. To simplify things, this time, we'll assume a collection of functions within a namespace, `App`. Here's an example:

```

1 <?php namespace App\Helpers;
2
3 function showTime()
4 {
5     return time();
```

```
6 }
```

In its current state, triggering the `showTime()` function will, as expected, return a time stamp. As no `time()` function exists in the current namespace, PHP will move up and assume that we're referring to the global version of `time()`.

However, we can add a custom `time()` function that will override the default version. In effect, we're essentially create `App\time()`.

```
1 <?php namespace App;  
2  
3 function time() { return 'foo'; }  
4  
5 function showTime()  
6 {  
7     return time();  
8 }
```

With this modification, when calling `showTime()`, *foo* will be returned! Let's use this in our test.

```
1 <?php namespace App\Helpers;  
2  
3 function time() { return 'foo'; }  
4  
5 class FunctionsTest extends PHPUnit_Framework_TestCase {  
6     public function testShowTime()  
7     {  
8         $this->assertEquals('foo', showTime());  
9     }  
10 }
```

Admittedly, this is a silly example. In fact, we're not really testing anything! Always be careful of that when mocking. But, hopefully, the basic principle has shown through.

If you cannot remove a global function but still need to test a particular method, first attempt to create a wrapper for the function that can be mocked. As a fallback, use the namespacing trick.

2. How Do I Create a SQL Dump for Codeception Tests?

We reviewed this in *Chapter 16*. The easiest way is to knock it out in a single command. For example, to dump a database, called `my-db` into a file, `app/tests/_data/dump.sql`, from the command line, run:

```
1 mysqldump --opt --user="USERNAME" --password="PASSWORD" my-db >
app/tests/_\
2 data/dump.sql
```

If, for some reason, the `mysqldump` executable isn't available to you, there are plenty of MySQL GUIs that will do the job for you. If using the popular [Sequel Pro](#) app, choose *File -> Export*.

3. How Do I Test Protected Methods?

As a basic rule of thumb, you don't test them. Opinions vary a bit on this issue, but, if you'd like my personal advice, only test the public interface. Those protected methods will be tested indirectly in the process. As such, if they don't work properly, your public tests will fail.

4. Should I Test Getters and Setters?

Once again, and as with many things in the testing world, the community is a bit divided on this one. I am of the opinion that they're unnecessary in 90% of the cases. My rule is that, if there's no real logic going on, don't waste time writing a test. However, if your setter manipulates or validates the data in some way, throw a test in there to ensure that it works as expected.

"I get paid for code that works, not for tests, so my philosophy is to test as little as possible to reach a given level of confidence (I suspect this level of confidence is high compared to industry standards, but that could just be hubris). If I don't typically make a kind of mistake (like setting the wrong variables in a constructor), I don't test for it. [...] Ten or twenty years from now, we'll likely have a more universal theory of which tests to write, which tests not to write, and how to tell the difference. In the meantime, experimentation seems in order." - [Kent Beck](#)

5. Why Is PHPUnit Ignoring My Filters?

This is a common pitfall when functional testing. You test a route, expect the filter to be applied, but find that, for some reason, it's ignored. Luckily, the fix is an easy one.

```
1 public function testSomeRoute()
2 {
3     Route::enableFilters();
4
5     // $this->call('GET', 'api/v1/posts');
6 }
```

The `Route` class' `enableFilters` method should activate these filters when testing.

6. Can I Mock a Method in the Class I'm Testing?

This will happen from time to time: you're testing a class, but need to mock one method within that class. Let's review a bit of pseudo code for querying Twitter's Search API.

```
1 <?php
2
3 class TwitterSearch {
4     protected static $url = 'http://search.twitter.com?q=';
5     protected $searchTerm;
6
7     public function __construct($searchTerm)
8     {
9         $this->searchTerm = $searchTerm;
10    }
11
12    public function compile() { /* ... */ }
13
14    public function fetch()
15    {
16        $url = urlencode($this->url . $this->searchTerm);
17
18        return file_get_contents($url);
19    }
20}
```

```
19      }
20 }
```

When testing this class, you may decide to stub or mock that `fetch` method to prevent the code from querying the Twitter API. With [Mockery](#), this is a cinch; what you need is a partial mock.



Definition: A **Partial Mocks** is useful when you only need to mock a couple of the methods on an object, leaving the remainder free to respond to calls as they normally would.

Traditional Partial Mocks

The easiest way to mock that `fetch` method is to use a traditional partial mock, like so:

```
1 $mock = Mockery::mock('TwitterSearch[fetch]', ['laracon']);
2 $mock->shouldReceive('fetch')->once()->andReturn('stub');
```

This code will only mock the `fetch` method on the `TwitterSearch` class, leaving all others untouched and free to behave as they normally would when instantiated. Because of this, it's important to pass in any constructor arguments in the form of an array, as the second argument. Now, when the `fetch()` is triggered at some point in the object, `foo` will be returned, leaving the Twitter API untouched.

Passive Partial Mocks

An alternative approach is to use what Mockery refers to as *passive partial mocking*.

```
1 $mock = Mockery::mock('TwitterSearch')->makePartial();
```

This declares that all methods on the mock object should defer to the original implementation, unless an expectation is declared. So, to reiterate, the difference in this approach is that the presence of an expectation will determine if that method is mocked, or defaults to the original implementation.

```
1 $mock = Mockery::mock('TwitterSearch')->makePartial();
```

```
2 $mock->fetch(); // calls real method
3
4 $mock = Mockery::mock('TwitterSearch')->makePartial();
5 $mock->shouldReceive('fetch')->once()->andReturn('stub');
6 $mock->fetch(); // returns foo
```

Though this won't always be the case, if you find that you're leveraging partial mocks too frequently, this might be an indicator that your code should be refactored to better follow the single responsibility principle. In the case of the above example, this certainly holds true. The `fetch` method, which simply returns the contents of a file as a string, should be extracted to its own class. Then, `TwitterSearch` could merely *ask* for that functionality upon instantiation, allowing for an easy test double.

Goodbye

Well, I suppose this brings us to a close (cue the closing *Saturday Night Live* music). Hopefully, the book lived up to what you expected, and, just maybe, you see testing in your future. You know what? It better dang well be in your future. I worked hard on this book!

The truth, though, is that this is an incredibly complex topic that consists of multiple methodologies, terminologies, frameworks, helpers, and more. No, it's not a skill that you pick up in a few hours. I've been at this for a long time, but still find myself learning new tricks and finding pitfalls every day. I hope the same will be true for you!



Testimonials: If you enjoyed the book, and wouldn't mind providing a short testimonial that will be used in various promotional spots around the web, [do so in this thread](#). Even a sentence or two would be appreciated.

Wait a Second...

But wait - this isn't the end! We're just getting started. *Laravel Testing Decoded* wasn't meant to be released and forgotten. Rest assured that, not only will it continue to be updated to reflect new testing functionality in the Laravel framework, but I also plan to release new chapters throughout 2013.

One of the first supplementary chapters will focus on building the purchase website for this very book. This will provide plenty of real-world scenarios to dig further into acceptance and unit testing.

So, with that in mind - and in closing - I'll leave you with one question: [what do you want to learn next?](#)

Stay in Touch

Let's be friends! I'm incredibly active on [Twitter](#), and, of course, spend much of my days making [Tuts+ Premium](#) the best educational site on the web. Don't be a stranger!