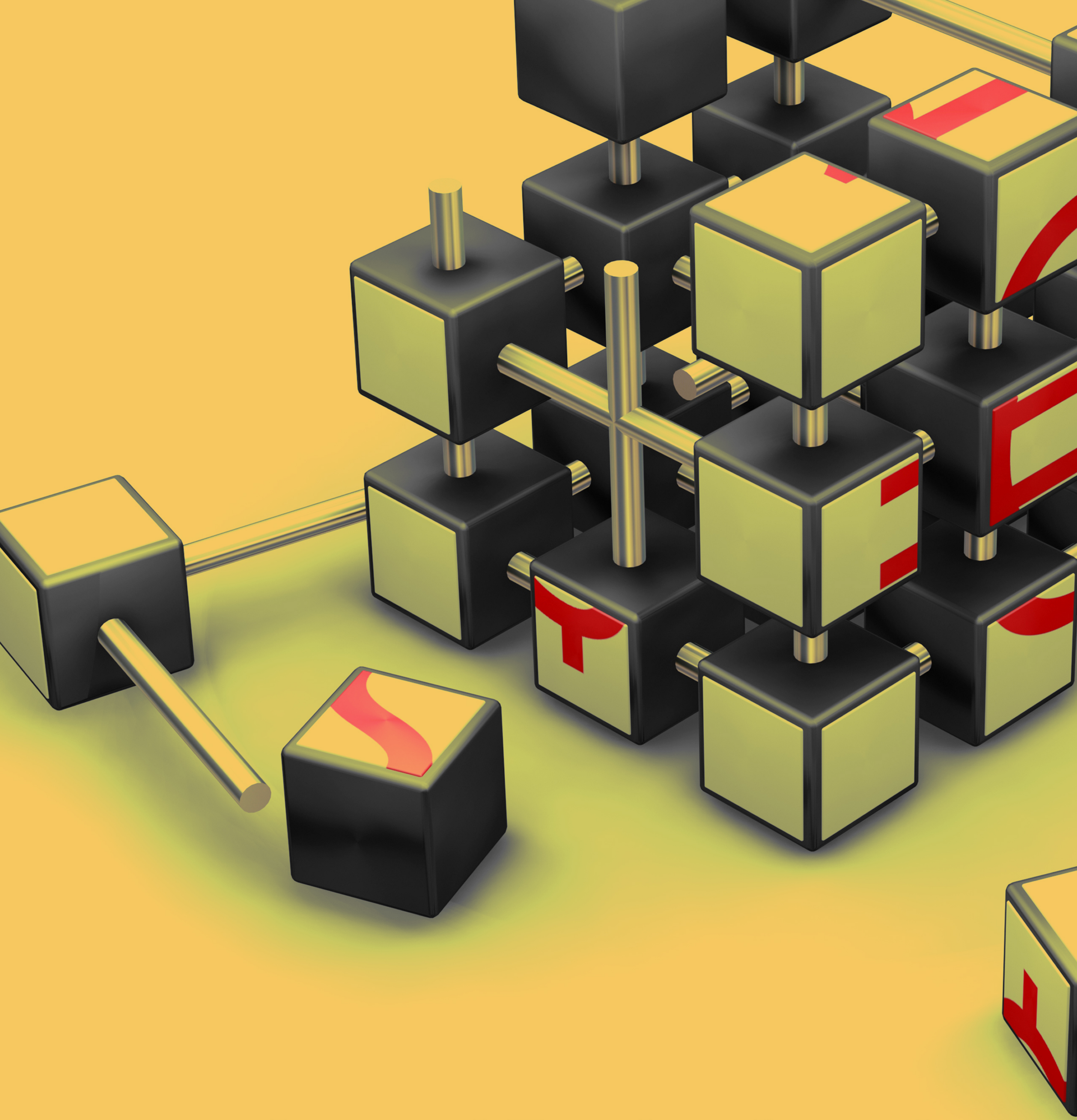


Laravel Testing Decoded

...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 2013-08-06

Tweet This Book!

Please help JeffreyWay by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Just bought @jeffrey_way's new book, Laravel Testing Decoded!

The suggested hashtag for this book is #laravelTesting.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#laravelTesting

Contents

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[1], 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[2]

[1]https://github.com/JeffreyWay/Laravel-Testing-Decoded
[2]http://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[3] might say, all right: let's do this! Here's the testing book you've been waiting for.

[3] http://www.youtube.com/watch?v=LkCNJRfSZBU

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*[4] *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.

[4] http://notes.envato.com/team/jeffrey-wins-a-bet/

> "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

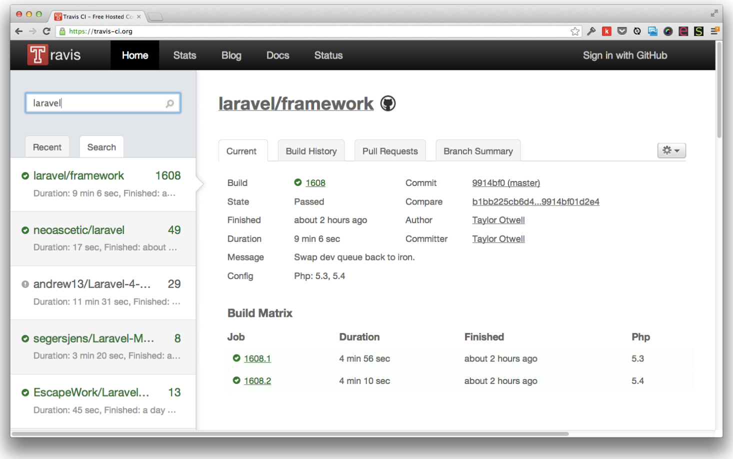
As you begin developing open source software, you'll likely leverage the convenience and power of social coding through GitHub. Eventually (one of the perks), other members of the community will begin contributing to your projects when they encounter bugs or hope to implement new functionality. However, 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[5], 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.

[5] https://travis-ci.org/



laravel/framework 1608
Duration: 9 min 6 sec, Finished: a...

neoascetic/laravel 49
Duration: 17 sec, Finished: about ...

andrew13/Laravel-4-... 29
Duration: 11 min 31 sec, Finished: ...

segersjens/Laravel-M... 8
Duration: 3 min 20 sec, Finished: a...

EscapeWork/Laravel... 13
Duration: 45 sec, Finished: a day ...

laravel/framework

Build	1608	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		

Build Matrix

Job	Duration	Finished	Php
1608.1	4 min 56 sec	about 2 hours ago	5.3
1608.2	4 min 10 sec	about 2 hours ago	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[a]?

[a]http://en.wikipedia.org/wiki/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:

```php
public function fetch($url)
{
    // We can't test this!
    $file = new Filesystem;

    return $this->data = $file->get($url);
}
```

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:

```php
protected $file;

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

public function fetch($url)
{
    return $this->data = $this->file->get($url);
}
```

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.

```php
public function testFetchesData()
{
    $file = Mockery::mock('Filesystem');
    $file->shouldReceive('get')->once()->andReturn('foo');

    $someClass = new SomeClass($file);
    $data = $someClass->fetch('http://example.com');

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

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:

```php
public function __construct(Filesystem $file, Cache $cache)
{
    $this->file = $file;
    $this->cache = $cache;

    $data = $this->file->get('http://example.com');
    $this->write($data);
}
```

Better:

```php
public function __construct(Filesystem $file, Cache $cache)
{
    $this->file = $file;
    $this->cache = $cache;
}
```

The reason why we do this is because, when testing, you'll 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 method should be limited to just a few (one is preferable, even). If, on the other hand, every method is dozens of lines long, this is 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[6]. 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:

```php
function addYearlyInterest($balance)
{
    switch ($this->accountType) {
        case 'checking':
            $rate = $this->getCheckingInterestRate();
            break;

        case 'savings':
            $rate = $this->getSavingsInterestRate();
            break;

        // other types of accounts here
    }

    return $balance + ($balance * $rate);
}
```

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

[6] https://tutsplus.com/2012/04/the-aha-moment/

Define an interface to ensure that you have access to a `getRate` method. You'll often hear interfaces referred to as contracts. This is a nice way to think of them: any implementation, according to the terms of the contract, must implement the given methods.

```
interface BankInterestInterface {
    public function getRate();
}
```

Create a checking implementation of the interface.

```
class CheckingInterest implements BankInterestInterface {
    public function getRate()
    {
        return .01;
    }
}
```

Create a savings implementation of the interface.

```
class SavingsInterest implements BankInterestInterface {
    public function getRate()
    {
        return .03;
    }
}
```

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.

```
function addYearlyInterest($balance, BankInterestInterface $interest)
{
    $rate = $interest->getRate();

    return $balance + ($balance * $rate);
}

$bank = new BankAccount;
$bank->addYearlyInterest(100, new CheckingInterest); // 101
$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.

```
public function testAddYearlyInterest()
{
    $interest = Mockery::mock('BankInterestInterface');
    $interest->shouldReceive('getRate')->once()->andReturn(.03);

    $bank = new BankAccount;
    $newBalance = $bank->addYearlyInterest(100, $interest);

    $this->assertEquals(103, $newBalance);
}
```

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*[7]

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.

[7] https://twitter.com/taylorotwell/status/334789979920285697

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.

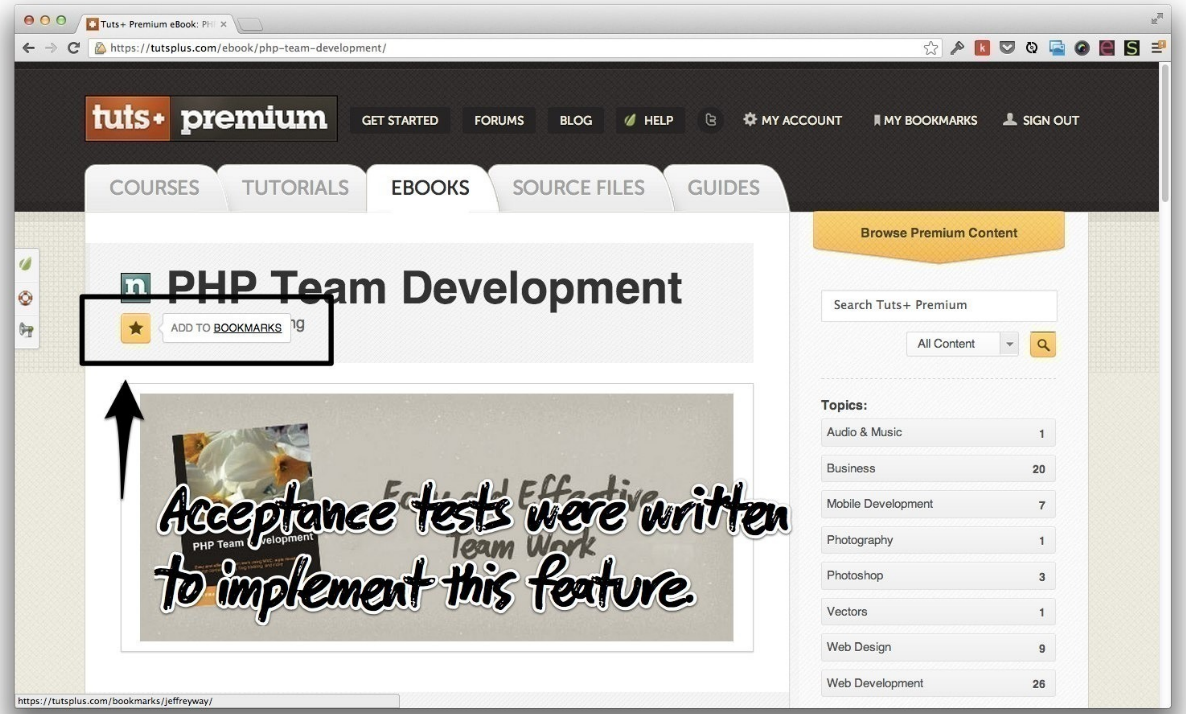
Tip: If functional tests meet a developer's assumptions and requirements, acceptance tests are intended to verify the client's expectations.

Think of Tuts+ Premium[8], 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.

```
In order to keep track of what to learn next
As a member
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.

[8]http://tutsplus.com



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 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 comes 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[9]

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[10]

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.*

[9]http://www.youtube.com/watch?feature=player_detailpage&v=KtHQGs3zFAM#t=77s

[10]http://37signals.com/svn/posts/3159-testing-like-the-tsa