CodeIgniter Testing Guide

Beginners Guide to Automated Testing in PHP.
Learn how to write Unit, Functional, and
Acceptance tests for MVC web applications.

Kenji Suzuki and Mat Whitney

CodeIgniter Testing Guide

Beginners' Guide to Automated Testing in PHP.

Kenji Suzuki and Mat Whitney

Tweet This Book!

Please help Kenji Suzuki and Mat Whitney by spreading the word about this book on Twitter!

The suggested hashtag for this book is #CITestGuide.

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?q=#CITestGuide

Contents

Preface

When I learned PHP for the first time, I did not know about writing test code at all. Nobody around me was writing test code. There was no PHPUnit (a testing framework for PHP), yet. In 2004, PHPUnit 1.0.0 was released for PHP4. In the same year, PHPUnit 2.0.0 was released for PHP5. However, I have never used PHPUnit 1 or 2.

When I found CodeIgniter (a PHP web application framework) for the first time, in 2007, it had a Unit testing class, but there was no test code for the framework itself.

Now, in 2015, more than 10 years have passed since PHPUnit 1.0.0. CodeIgniter 3.0 has its own test code with PHPUnit, and the code coverage for those tests is around 60%. We are progressing a bit day by day.

Have you ever written test code for your web application? If you haven't, you may imagine that writing test code will be very difficult or bothersome. Maybe you want to write test code, but don't know how to do so.

It is common to over-estimate the cost of learning something new, and testing is no exception. After reading a tutorial for PHPUnit, I thought, "So how do I test my application?" I had trouble seeing the similarities between the tests in the tutorial and the tests I would need to write for my own application.

This book is a beginners' guide for automated testing of PHP web applications. Of course, you will be able to write test code for any PHP applications after reading this book, but the focus will be on web applications.

I eschew complexity, favoring simple solutions. I use simple and easy to understand solutions first in the book, so you won't get lost. Let's keep going!

The Book at a Glance

If you want to know about this book, this is a great place to start. What follows is a very quick overview of what each chapter covers. This should give you an idea of what's ahead, or serve as a starting point if you want to find a particular portion of the content to review later.

Chapter 1: What is Automated Testing?

Let's begin learning about automated testing. First we will explore the basic concepts of automated testing. We will find out why and what you should test. At the same time, I will explain the ideas and testing policies used by this book.

Chapter 2: Setting Up the Testing Environment

To run tests in your PHP environment, you will need to install some additional software. For this book, this includes *CodeIgniter*, *PHPUnit* and a tool which acts as a bridge between them, *ci-phpunit-test*. If you don't like command line, you can use *VisualPHPUnit* to run tests via your web browser.

Chapter 3: Test Jargon

We define test jargon here. One of the annoying and confusing things in testing is the new vocabulary required to understand it. By the end of this chapter we'll help you understand the difference between Unit, Integration, and System testing; Functional and Acceptance testing; Fixtures and Mocks; and more.

Chapter 4: PHPUnit Basics

In this chapter, we will learn the basics of PHPUnit. We will run PHPUnit and learn how to configure it. After that, we will study PHPUnit conventions and write our first test. We also cover PHPUnit functionality, data providers, fixtures, and assertions.

Chapter 5: Testing a Simple MVC Application

You've already learned how to write test code, so here we will write tests for a CodeIgniter Tutorial application. We will write tests for a controller and a model. In this chapter, we will use the database for model testing.

Chapter 6: Unit Testing for Models

We will learn more about testing models. We will write tests for models without using the database. To do this, we will learn about PHPUnit mock objects.

Chapter 7: Testing Controllers

We will learn more about testing controllers in this and the next two chapters. In this chapter, we will write tests for a controller for reviewing, and write tests with mocking models. We also will write test cases for authentication and redirects.

Chapter 8: Unit Testing CLI Controllers

We will continue learning to write tests for controllers. In this chapter, we will write unit tests for controllers, and learn about monkey patching.

Chapter 9: Testing REST Controllers

In this chapter, we will learn about testing REST controllers. You will learn how to send (emulate) requests with methods other than GET and POST.

Chapter 10: Browser Testing with Codeception

In previous chapters, we have been using PHPUnit. In this chapter, we will learn about another testing tool. We will install *Codeception*, learn to configure it, and write tests which work with the web browser.

What You Need for This Book

I assume you have a general understanding of PHP 5.4 and object-oriented programming (OOP), and you have PHP 5.4, 5.5, or 5.6 installed.

If you know CodeIgniter, you may have an easier time with some parts of this book, but if you don't know it, don't worry. CodeIgniter is an MVC framework that is very easy to learn and understand, and it has great documentation. I will explain CodeIgniter-specific conventions and functionality in this book.

I do not assume that you are using a specific operating system. However, my code examples are written for Mac OS X. Bash commands are provided for Mac OS X and also work on Ubuntu. I have not tested them on Windows, but they will probably work.

We use the following software in this book:

- PHP 5.5 (You can use PHP 5.4 or 5.6)
- CodeIgniter[1] 3.0
- ci-phpunit-test[2] 0.10
- PHPUnit[3] 4.8
- Codeception[4] 2.1
- Selenium Standalone Server[5] 2.48

Who should read This Book?

This book is for PHP developers who don't know *Automated Testing* or *Unit Testing*, or for those looking for help testing CodeIgniter applications.

If one or more of the lines below sounds familiar, this book is perfect for you!

- I have never written test code.
- I want to write test code, but I don't know how.
- I tried to write test code in the past, but I couldn't quite figure it out.

[1] http://www.codeigniter.com/
[2] http://kenjis.github.io/ci-phpunit-test/
[3] https://phpunit.de/
[4] http://codeception.com/
[5] http://www.seleniumhq.org/

Why PHPUnit?

PHPUnit is the de facto standard *Testing Framework* in the PHP world.

These popular PHP frameworks use PHPUnit for their own tests, and they provide support for application testing with PHPUnit:

- CakePHP
- FuelPHP
- Laravel
- Symfony
- Yii
- Zend Framework

CodeIgniter 3.0 uses PHPUnit for testing its system code. Support for application testing with PHPUnit is currently planned for CodeIgniter 4.0.

Is This a CodeIgniter Book?

This book is not specifcally for CodeIgniter, but we use CodeIgniter applications in our examples. Probably 85% of the book's content is not specific to CodeIgniter, and is applicable to testing any PHP application.

So, if you want to learn *Automated Testing* in PHP, this book is still good for you. Most of the techniques outlined in this book can be applied to any other PHP framework, and even to other languages.

In modern web development, you probably use a framework. I don't know what framework you use or like, but if you learn testing with a framework, you can write test code more easily in your real development environment.

CodeIgniter is one of the most easily understood frameworks currently available for PHP. This will allow you to spend more of your time learning about testing, even if you don't know CodeIgniter.

Another reason I chose CodeIgniter as the framework used in this book is that too many CodeIgniter developers don't write test code. So, I'm hoping that by choosing CodeIgniter for my examples, this book will promote better testing practices in the CodeIgniter community.

Is Testing PHP Applications Difficult?

No, but you need the right tools and you need to know how to write tests.

Is Testing CodeIgniter Applications Difficult?

No, at least it is not difficult with CodeIgniter 3.0.

Previously, it was said that testing CodeIgniter applications was difficult, but I will show you why this is no longer the case.

Testing is Fun and Easy

Yes, it is really fun. Do you like to write code? Tests are also code. Good tests will help you write better code.

When people are asked why they don't write test code, it is often because they think it will be difficult or will take too much time.

I will show you how to write tests and try to show you that it can be fun and easy. Writing good tests now will help you catch mistakes earlier, and make it easier to change your code without introducing errors, saving time in the long run.

Conventions Used in This Book

The following typographical conventions are used in this book:

- *Italic*: Indicates new terms or technical terms.
- `Constant width`: Used for program listings, as well as within paragraphs to refer to program elements such as class or function names, variables, statements, and keywords. Also used for Bash commands and their output.

For example, the following is a block of PHP code:

```
1 <?php
2
3 echo 'Hello World!';
```

Sometimes we use diff-style:

```
--- a/CodeIgniter/application/tests/libraries/Temperature_converter_test.php
+++ b/CodeIgniter/application/tests/libraries/Temperature_converter_test.php
@@ -6,7 +6,7 @@ class Temperature_converter_test extends TestCase
     {
         $obj = new Temperature_converter();
         $actual = $obj->FtoC(100);
-        $expected = 37.0;
+        $expected = 37.8;
         $this->assertEquals($expected, $actual, '', 0.01);
     }
 }
```

The listing above is in a format called *unified diff*. It shows the difference between two files. The format starts with two-line header, with the path and name of the original file preceded by `---` on the first line, and the new file preceded by `+++` on the second line. In some situations, each line could include a date/time stamp, as well.

In this case, the original file and the new file are the same file (this might not be the case if the file was moved or renamed, or if the diff was to indicate that the content was moved to a new file). Following this are one or more change hunks which show areas where the files differ.

The line "`@@ -6,7 +6,7 @@`" shows the hunk range. The first set of numbers ("-6,7") indicates the lines in the original file, the second set ("+6,7") indicates the lines in the new file. These numbers are in the format 'start,count', where the ',count' may be omitted if the hunk only includes one line. So, in this example, the hunk starts at line 6 in both files and contains 7 lines in both files.

The contents of the hunk (following the hunk range) contain the lines from the files, with additions preceded with + (highlighted in green here), and deletions preceded with - (highlighted in red here). The remaining lines (not preceded with either a - or +) are provided for context.

In short, remove the line(s) starting with - from the original file and add the line(s) starting with + to get the new file.

This is an example of a Bash command and its output:

```
$ echo 'Hello World!'
Hello World!
```

This is an example of a note or general information.

This is an example of a tip or suggestion.

This is an example of a warning or caution.

This is an example of an exercise.

Errata

Although I have taken care to ensure the accuracy of this content, mistakes do happen. If you notice any mistakes, I would be grateful if you would report them to me. If you find any errata, please file an issue on GitHub https://github.com/kenjis/codeigniter-testing-guide, and I will update the book as soon as possible.

1. What is Automated Testing?

Welcome to the testing world!

In this chapter, we will learn about the concept of automated testing and learn why and what you should test. As we go, we will begin to develop the testing strategy which will be applied by this book.

1.1 Primitive Example

We will begin with a very simple example, a function which returns `'Hello World!'`. For this example, we'll place the function in a file called `hello.php` in a location easily accessible to our server.

hello.php

```
1  <?php
2
3  function hello()
4  {
5      return 'Hello World!';
6  }
```

How do we test this function?

Exercise

At this point, please stop and think about how you would test this function.

Manual Testing

We'll start with a manual test, printing the result of the function in a page viewed by your web browser. We'll place this function in another file, called `hello_test_with_your_eyes.php` in the same location as the `hello.php` file which holds our `hello()` function.

hello_test_with_your_eyes.php

```
<?php

require __DIR__ . '/hello.php';

echo hello();
```

If you navigate to `hello_test_with_your_eyes.php` in your browser and see `Hello World!`, the function and the test both work fine.

Alternatively, you can just run it from PHP's Command Line Interface (CLI):

```
$ php hello_test_with_your_eyes.php
Hello World!
```

This is manual testing.

All PHP developers have tested like this at some point. Manual testing with your web browser is very common. You write some code expecting a certain outcome, then you run the code to verify that you receive the expected outcome. If you look at the web page you built to verify you aren't receiving errors and the outcome looks more or less as you expect, you are already testing your application.

However, if you test your application by yourself, and by manually checking every possible action the user might take, it could take some time to test. If your application is very big, testing every possibility in the entire application could take a very long time, which means you can't possibly test everything very often.

Automated Testing

What if, instead of manually checking the results in a web browser, you write code to test your application? The following is some very primitive test code for our primitive `hello.php` application. We would place this code in a file called `hello_test_automated.php` alongside our application and our manual test.

hello_test_automated.php

```
<?php

error_reporting(-1);

require __DIR__ . '/hello.php';

function assertTrue($condition)
{
    if (! $condition) {
        throw new Exception('Assertion failed.');
    }
}

$actual = hello();
$expected = 'Hello World!';
assertTrue($expected === $actual);
```

If you run this code, you probably won't see anything, unless the `hello()` function returns something unexpected, in which case you will receive a fatal PHP error and a stack trace.

If our `hello()` function does not return `'Hello World!'`, our test will fail. Since the code verifies that the result of executing the function matches the expected result, rather than having to verify the result in a browser (or in the CLI) for ourselves, we call this automated testing. Using automated tests, you could run the test code every time you make a change, or at a certain time every day, because it is PHP code!

When writing an automated test, you think of what you would do to test the code by hand or how you might confirm the code is working properly, then write code which will do the testing for you. In the long run, this means you can spend less time manually testing your code, and it will protect your code base from unexpected changes.

Automated testing has two features:

- it is very fast,
- so you can run tests very often.

Testing Tip: Avoid Slow Tests

Speed is the one of the most important things in testing. If you have slow tests, they prevent you from testing. If you find that one or more of your tests is slow, you should probably try to speed it up.

This example is simple, so you might think the test is nonsense. However, if you expect a certain output given a certain input, the complexity of the function you are testing has very little impact on the complexity of the test, so even very complicated functions might have tests with code which is similar to the test we have created for our primitive `hello` function.

1.2 Why should you write test code?

You don't have to write test code, but even if you choose not to, it is better that you know how to write it. Thinking about how you would write a test for your code makes you consider how your code might fail and what expectations you have for a given piece of code.

According to "Top 12 Reasons to Write Unit Tests[1]," you will get the following benefits when you write test code:

1. Tests Reduce Bugs in New Features
2. Tests Reduce Bugs in Existing Features
3. Tests Are Good Documentation
4. Tests Reduce the Cost of Change
5. Tests Improve Design
6. Tests Allow Refactoring
7. Tests Constrain Features
8. Tests Defend Against Other Programmers
9. Testing Is Fun
10. Testing Forces You to Slow Down and Think
11. Testing Makes Development Faster
12. Tests Reduce Fear

In my opinion, the most important thing is reducing fear. If we have well-tested code, we can change and improve it without fear that we might break existing code.

1.3 Finding the Middle Way

Have you ever heard about the right way to write unit tests?

- Don't access the database.
- Don't use `new` operators (use dependency injection and factories).
- Rewrite your code to be more testable.
- Do not interact with the file system, use a virtual file system instead.

[1] http://www.onjava.com/pub/a/onjava/2003/04/02/javaxpckbk.html

Each of these is true in a sense, but not in all situations. In this book, I don't say such things, but instead recommend following "The Way Of Testivus[2]":

- If you write code, write tests
- Don't get stuck on unit testing dogma
- Embrace unit testing karma
- Think of code and test as one
- The test is more important than the unit
- The best time to test is when the code is fresh
- Tests not run waste away
- An imperfect test today is better than a perfect test someday
- An ugly test is better than no test
- Sometimes, the test justifies the means
- Only fools use no tools
- Good tests fail

Avoid extremes, find the middle way. We prefer simple and easy to understand solutions in this book.

1.4 What should you test?

This is the most difficult question to answer, and in some cases it's nearly impossible to answer. Instead, it is better to ask what we don't have to test.

If we absolutely think we don't have to test *something*, then we don't test it. If we are not sure, we test everything.

However, since we are trying to find the middle way, we'll come back to this idea later, in Chapter 3, Code Coverage. For now, rather than worrying about testing everything, we're going to focus on writing tests wherever we can and acquiring good testing karma.

1.5 TDD or Not TDD

Test Driven Development (TDD) is a software development process in which you write test code first, then write code which passes the tests, and repeat automated testing in very short development cycles.

We prefer TDD, but you can choose whatever development process works for you. If pushed we'd return to *The Way Of Testivus*, which says "*Think of code and test as one*" and "*The best time to test is when the code is fresh*".

According to James Shore[3], the development process in TDD is the following:

[2] http://www.artima.com/weblogs/viewpost.jsp?thread=203994

[3] http://www.jamesshore.com/Blog/Red-Green-Refactor.html

1. **Think**: Figure out what test will best move your code towards completion. (Take as much time as you need. This is the hardest step for beginners.)
2. **Red**: Write a very small amount of test code. Only a few lines… usually no more than five. Run the tests and watch the new test fail: the test bar should turn red. (This should only take about 30 seconds.)
3. **Green**: Write a very small amount of production code. Again, usually no more than five lines of code. Don't worry about design purity or conceptual elegance. Sometimes you can just hardcode the answer. This is okay because you'll be refactoring in a moment. Run the tests and watch them pass: the test bar will turn green. (This should only take about 30 seconds, too.)
4. **Refactor**: Now that your tests are passing, you can make changes without worrying about breaking anything. Pause for a moment. Take a deep breath if you need to. Then look at the code you've written, and ask yourself if you can improve it. Look for duplication and other "code smells." If you see something that doesn't look right, but you're not sure how to fix it, that's okay. Take a look at it again after you've gone through the cycle a few more times. (Take as much time as you need on this step.) After each little refactoring, run the tests and make sure they still pass.
5. **Repeat**: Do it again. You'll repeat this cycle dozens of times in an hour. Typically, you'll run through several cycles (three to five) very quickly, then find yourself slowing down and spending more time on refactoring. Than you'll speed up again. 20-40 cycles in an hour is not unreasonable.

Unfortunately, this book won't serve as a good example of TDD, because we won't write tests first, since we will use existing application code for most of our tests. I recommend you try to write test code first in your real application development.

2. Setting Up the Testing Environment

To run tests in your PHP environment, you will probably need to install some additional software. We will assume you have PHP 5.4, 5.5, or 5.6 installed.

If you are familiar with Composer and prefer it, go to Installing via Composer.

2.1 Installing CodeIgniter

Download the latest version of CodeIgniter 3.0 (in this book we use v3.0.1) from http://www.codeigniter.com/download and unzip it.

For simplicity, we renamed the `CodeIgniter-3.0.1` directory to `CodeIgniter`.

```
CodeIgniter/
├── application/ ... Your application files will be placed here.
├── system/      ... CodeIgniter files, you shouldn't modify these files.
└── user_guide/  ... CodeIgniter User Guide (HTML), this can be omitted.
```

2.2 Installing ci-phpunit-test

ci-phpunit-test is a tool, written by Kenji Suzuki, which makes it easier to work with CodeIgniter 3.0 and PHPUnit together. CodeIgniter has its own Unit testing library, but its capabilities are pretty limited, and PHPUnit is used to test CodeIgniter itself. CodeIgniter's developers plan to provide application testing with PHPUnit in CodeIgniter 4. So ci-phpunit-test is a missing piece for CodeIgniter 3.0, providing the ability to perform application testing with PHPUnit.

Download the latest version of ci-phpunit-test (in this book we use v0.10.0 from https://github.com/kenjis/ci-phpunit-test/releases and unzip it. Copy the `application/tests` directory into the `application` directory in your CodeIgniter installation.

```
CodeIgniter/
└── application/
    └── tests/
```

Enabling Monkey Patching

Next, we enable *Monkey Patching*, because we will need it later.

Monkey Patching is a method used to replace code on the fly without modifying the original code. The Monkey Patching provided by ci-phpunit-test can replace `exit()` or `die()`, as well as some functions and methods of user-defined classes, to prevent unwanted behavior in the code being tested.

To enable monkey patching, open `application/tests/Bootstrap.php` in your editor and remove the comment marks below. Remove the lines 298 and 323.

application/tests/Bootstrap.php

```
290 /*
291  * --------------------------------------------------------------------
292  *  Enabling Monkey Patching
293  * --------------------------------------------------------------------
294  *
295  * If you want to use monkey patching, uncomment below code and configure
296  * for your application.
297  */
298 /* # Remove this line.
299 require __DIR__ . '/_ci_phpunit_test/patcher/bootstrap.php';
300 MonkeyPatchManager::init([
301     'cache_dir' => APPPATH . 'tests/_ci_phpunit_test/tmp/cache',
302     // Directories to patch on source files
303     'include_paths' => [
304         APPPATH,
305         BASEPATH,
306     ],
307     // Excluding directories to patch
308     'exclude_paths' => [
309         APPPATH . 'tests/',
310     ],
311     // All patchers you use.
312     'patcher_list' => [
313         'ExitPatcher',
314         'FunctionPatcher',
315         'MethodPatcher',
316     ],
317     // Additional functions to patch
318     'functions_to_patch' => [
```

```
319         //'random_string',
320     ],
321     'exit_exception_classname' => 'CIPHPUnitTestExitException',
322 ]);
323 */ # Remove this line.
```

2.3 (Optional) Installing VisualPHPUnit

If you don't want to run PHPUnit from the terminal, you might like VisualPHPUnit.

VisualPHPUnit is a visual front-end for PHPUnit. You can run PHPUnit from your web browser.

You don't have to install it. In that case, go to the next section, Installing PHPUnit.

Installing Composer

Unfortunately, when you use VisualPHPUnit, you must install PHPUnit via Composer. So go to Installing Composer, and after installing Composer, go back to here.

Installing VisualPHPUnit

Download the latest version of VisualPHPUnit (in this book we use v2.3.1) from https://github.com/VisualPHPUnit/VisualPHPUnit/releases and unzip it. Put the folder wherever you want.

Installing PHPUnit

To Windows users

I recommend you use *Git Bash* in *Git for Windows* https://git-for-windows.github.io/.

Open your terminal, navigate to the directory in which you've placed VisualPHPUnit and install PHPUnit via Composer:

```
$ cd VisualPHPUnit-2.3.1/
$ composer require "phpunit/phpunit=4.8.*"
```

This command installs PHPUnit and its dependencies.

```
VisualPHPUnit-2.3.1/
├── bin
│    └── phpunit -> ../vendor/phpunit/phpunit/phpunit
└── vendor/
    ├── doctrine/
    ├── phpdocumentor/
    ├── phpspec/
    ├── sebastian/
    ├── phpunit/
    └── symfony/
```

Configuring VisualPHPUnit

Set the full paths for the `application/tests` directory, `application/tests/phpunit.xml` and `application/tests/Bootstrap.php` in the `app/config/bootstrap.php` file:

```
--- a/app/config/bootstrap.php
+++ b/app/config/bootstrap.php
@@ -24,8 +24,8 @@ $config = array(

     // The directories where the tests reside
     'test_directories' => array(
-        'Sample Tests' => "{$root}/app/test",
-        //'My Project' => '/var/www/sites/my.awesome.site.com/laravel/tests',
+        'CodeIgniter Testing Guide'
+            => '/Users/kenji/code/CodeIgniter/application/tests',
     ),


@@ -79,8 +79,7 @@ $config = array(
     // In order for VPU to function correctly, the configuration files must
     // contain a JSON listener (see the README for more information)
     'xml_configuration_files' => array(
-
-
+        '/Users/kenji/code/CodeIgniter/application/tests/phpunit.xml',
     ),
     //'xml_configuration_files' => array(
     //    "{$root}/app/config/phpunit.xml"
@@ -88,8 +87,7 @@ $config = array(

     // Paths to any necessary bootstraps
```

```
     'bootstraps' => array(
-        // '/path/to/bootstrap.php',
-        //'/var/www/sites/my.awesome.site.com/laravel/bootstrap/autoload.php',
+        '/Users/kenji/code/CodeIgniter/application/tests/Bootstrap.php',
     )
 );
```

The listing above is in a format called *unified diff*. It shows the difference between two files. In short, remove the red line starting with - from the original file and add the green line starting with +, to get the new file. See Conventions Used in This Book for details.

Configuring PHPUnit XML Configuration File

To use the PHPUnit XML configuration file to define which tests to run, modify the PHPUnit XML configuration file in your CodeIgniter directory to include the following `listeners` element:

```
--- a/CodeIgniter/application/tests/phpunit.xml
+++ b/CodeIgniter/application/tests/phpunit.xml
@@ -27,4 +27,8 @@
         <log type="coverage-clover" target="build/logs/clover.xml"/>
         <log type="junit" target="build/logs/junit.xml" logIncompleteSkipped="fa\
lse"/>
     </logging>
+    <!-- This is required for VPU to work correctly -->
+    <listeners>
+        <listener class="PHPUnit_Util_Log_JSON"></listener>
+    </listeners>
 </phpunit>
```

Okay, that's all. Go to the next chapter, Test Jargon.

If you use VisualPHPUnit, you can't use PsySH which is introduced in (Optional) Installing PsySH.

2.4 Installing PHPUnit

Download https://phar.phpunit.de/phpunit-4.8.10.phar and rename it to `phpunit.phar`, then put it into your `application/tests` folder. For most environments, that's it, but if you run into issues installing PHPUnit, the installation section of the PHPUnit Manual[1] should help you figure out what's wrong.

[1] https://phpunit.de/manual/4.8/en/installation.html

```
CodeIgniter/
└── application/
    └── tests/
        └── phpunit.phar
```

You can download the latest PHPUnit PHAR from https://phar.phpunit.de/phpunit.phar and download all releases of the PHPUnit PHAR from https://phar.phpunit.de/. PHPUnit 5.0 requires PHP 5.6 or later.

Let's confirm the installed version of PHPUnit. Go to your terminal.

```
$ cd CodeIgniter/application/tests/
$ php phpunit.phar --version
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.
```

To Windows users

I recommend you use *Git Bash* in *Git for Windows* https://git-for-windows.github.io/.

Where to run `phpunit`

You have to go to the `application/tests` directory to run `phpunit`, because this is where the PHPUnit config file `phpunit.xml` is installed.

2.5 (Optional) Installing PsySH

PsySH is an interactive debugger and Read-Eval-Print Loop (REPL) for PHP which enables you to debug your test code via the CLI.

Download the latest version of PsySH (in this book we use v0.6.0-dev) from http://psysh.org/psysh and put it into the `application/tests` directory.

```
CodeIgniter/
└── application/
    └── tests/
        └── psysh
```

Set the permissions for the psysh binary.

```
$ cd CodeIgniter/application/tests/
$ chmod +x psysh
```

Require `psysh` before loading `CIPHPUnitTest` in `application/tests/Bootstrap.php`. Add the line 323.

application/tests/Bootstrap.php

```
323 require __DIR__ . '/psysh'; # Add this line.
324
325 /*
326  * ------------------------------------------------------------------
327  *  Added for CI PHPUnit Test
328  * ------------------------------------------------------------------
329  */
330 require __DIR__ . '/_ci_phpunit_test/CIPHPUnitTest.php';
331 CIPHPUnitTest::init();
```

2.6 Installing via Composer

If you like Composer or the command-line, you can install all of these components via Composer.

Composer is a tool for package (library) management in PHP. It allows you to declare the libraries your project depends on and it will manage (download/install/update) them for you.

Installing Composer

If you use Linux or Mac OS X, you can install Composer from your terminal.

```
$ curl -sS https://getcomposer.org/installer | php
$ sudo mv composer.phar /usr/local/bin/composer
```

If you use Windows, you can download and run Composer-Setup.exe[2].

Installing CodeIgniter via Composer

```
$ composer create-project kenjis/codeigniter-composer-installer CodeIgniter
```

The command listed above installs CodeIgniter 3.0 with a more secure directory structure.

```
CodeIgniter/
├── application/     ... Your application files.
├── composer.json    ... Composer's configuration file.
├── composer.lock    ... Composer's lock file, managed by Composer.
├── public/
|   └── index.php    ... Front controller.
└── vendor/          ... Composer's vendor directory, managed by Composer.
    ├── autoload.php ... Composer's autoloader.
    └── codeigniter/
        └── framework/
            ├── application/ ... You don't use this directory.
            ├── system/      ... CodeIgniter system directory.
            └── user_guide/  ... CodeIgniter User Guide (HTML).
```

The `composer.json` file describes the dependencies of your project. Packages managed by Composer can be found in the `vendor` directory. They are installed, updated, or removed via the `composer` command, so you shouldn't edit any of the files in the `vendor` directory.

Installing ci-phpunit-test via Composer

```
$ cd CodeIgniter/
$ composer require kenjis/ci-phpunit-test --dev
$ php vendor/kenjis/ci-phpunit-test/install.php
```

The commands listed above will install ci-phpunit-test.

[2] https://getcomposer.org/Composer-Setup.exe

```
CodeIgniter/
├── application/
|   └── tests/
└── vendor/
    └── kenjis/
        └── ci-phpunit-test/
```

The last command copies the files inside `vendor/kenjis/ci-phpunit-test` to `application/tests`, so everything can be found in the expected location.

Then we enable *Monkey Patching*. Because we will need it later. See Enabling Monkey Patching and edit `application/tests/Bootstrap.php`.

Installing PHPUnit via Composer

```
$ composer require "phpunit/phpunit=4.8.*" --dev
```

This command installs PHPUnit and its dependencies.

```
CodeIgniter/
└── vendor/
    ├── bin
    |   └── phpunit -> ../phpunit/phpunit/phpunit
    ├── phpdocumentor/
    ├── phpspec/
    └── phpunit/
```

Let's confirm the version of PHPUnit from the terminal.

```
$ cd application/tests/
$ ../../vendor/bin/phpunit --version
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.
```

Where to run `phpunit`

You have to run `phpunit` from the `application/tests` directory, because the PHPUnit config file `phpunit.xml` is there.

Creating a phpunit Shortcut

Now you can use the `phpunit` command, but typing `../../vendor/bin/phpunit` is bothersome. We can create a shortcut using a shell script.

application/tests/phpunit.sh

```
#!/bin/sh

cd `dirname $0`
../../vendor/bin/phpunit $@
```

Then set the permissions for the script.

```
$ chmod +x phpunit.sh
```

Ready, let's try.

```
$ ./phpunit.sh --version
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.
```

Good.

(Optional) Installing PsySH via Composer

PsySH is an interactive debugger and Read-Eval-Print Loop (REPL) for PHP which enables you to debug your test code via the CLI.

```
$ composer require psy/psysh:@stable --dev
```

This command installs PsySH and its dependencies.

```
CodeIgniter/
└── vendor/
    └── bin
        └── psysh -> ../psy/psysh/bin/psysh
```

3. Test Jargon

In this book, we define test jargon as indicated in this chapter. One of the annoying and confusing things in testing is new jargon, and a term's meaning is slightly different depending on the person. This chapter is intended to help you understand the generally-used jargon in testing, as well as the specific meaning we intend when we use that jargon in this book.

If you think something is wrong with the definitions, please let me know.

3.1 Testing levels

Unit Testing

Unit Testing refers to tests that verify the functionality of a specific section of code. In an object-oriented environment, this is usually at the class level. In other words, a unit is usually a class and a unit test would verify the functionality of the class as a whole.

However, when you have, for example, two tightly coupled classes, you and your team may decide that they need to be tested as a single unit. In short, let your team and your code decide what makes up a unit.

If you aren't sure what constitutes a unit in your code base, a class is a good place to start. The smallest unit is a class, or a function if you have some code which uses PHP functions outside of classes. A method (a function defined within a class) is usually not considered a unit, since it can't be tested in isolation from the class which defines it.

Testing Tip: Test Classes, Not Methods

Testing classes rather than individual methods is good practice, because we don't have to concern ourselves with the internal state of a class if it never changes the outcome of a method call. If you write a test to check the internal logic of a class, it may prevent you from refactoring the class later.

Integration Testing

Integration Testing refers to tests that verify the functionality of more than one unit. If all unit tests pass, there might be bugs where two units interact with each other, so unit testing is not enough.

We can say that Functional Testing, Database Testing, and Browser Testing are types of Integration Testing.

System Testing

System Testing, or *End-to-End Testing*, tests a completely integrated system to verify that it meets its requirements. Testing whole web applications using a web browser is a form of system testing, but is often called *Acceptance Testing* in the PHP world.

Naturally, Unit Testing is usually faster than Integration Testing. Integration Testing is usually faster than System Testing. Faster testing is better, because slow tests will be run less often. However, each level of testing is important in its own way in ensuring the proper operation of the application.

3.2 Testing Types

Functional Testing

Functional Testing refers to tests that verify a slice of functionality within the whole system. It does not mean testing PHP functions.

Functional testing is used to verify that specific actions (methods/functions in a controller) generate the expected output when the controller, view(s), model(s) and any necessary libraries are all used together. It could also be used to test functionality provided by a library.

For example a cache library could be tested by verifying that the first call to a specific action populates the cache, while subsequent calls use the cache to fulfill the request. You might also need to call another action to invalidate the cache, then verify that calling the original action repopulates the cache.

We will perform functional testing of our controllers in Chapter 5, Test Case for Page Controller.

Database Testing

Database Testing is testing that uses the database.

We will perform database testing with our model in Chapter 5, Test Case for News Model.

Browser Testing

Browser Testing is testing that uses a web browser or simulates a web browser.

We will perform browser testing of our web application, using Codeception with Firefox and Codeception with Google Chrome, in Chapter 10.

Acceptance Testing

Acceptance Testing is testing whether the product meets the requirements for which it was designed. The name comes from the fact that, in some environments, the customer might define a series of tests which your application will have to pass before they will accept the product.

In most cases, acceptance testing is a specific form of browser testing, which we will cover in Chapter 10.

3.3 Code Coverage

Code coverage tells you which lines of script have been executed during test execution. PHPUnit requires Xdebug[1] or phpdbg[2] to provide code coverage reports.

We often use *Line Coverage*, which measures whether each executable line was executed. If we say code coverage is 60%, then 60% of the lines were executed during test execution.

Code coverage only tells you how many lines were executed. 100% code coverage tells you only that all of your code was executed during the tests. It does not guarantee your code is bug free or that all of your code was actually tested.

If you want to know more about code coverage, I recommend you read the PHPUnit Manual's chapter on Code Coverage Analysis[3].

3.4 Fixtures

Fixture is a known state in which a test will be run. When running a test, if the state differs from one execution of the test to the next, the results may differ as well, so you build one or more fixtures to ensure the tests are repeatable and consistent.

For example, in database testing, we always have to set up the the same database state. If you write a test which runs a query and checks the results without using a fixture to setup the state beforehand, another test which inserts/updates the data may cause your test to fail later. If you're working on a team of developers, the situation may be even worse, as each developer may be unaware of the expectations of the other developer's tests.

Since we are trying to write automated tests, we want the code's environment to be in a known state for each test. We will prepare database fixtures in Chapter 5, Database Fixtures.

[1] http://xdebug.org/

[2] http://phpdbg.com/

[3] https://phpunit.de/manual/4.8/en/code-coverage-analysis.html

3.5 Test Doubles

Test Doubles refer to fake objects utilized for testing purposes. If you unit-test a class, all you have to test is code inside the class. If the class has dependencies, the dependencies are out of scope for the testing, so you don't have to or shouldn't use real dependencies.

For example, if we're writing a test for a class which calls a library (`$this->httpClient` in the code below), we might need to create a test double for that library.

```
$page = $this->httpClient->get('http://www.codeigniter.com/');
```

The class under test needs to call the library's `get()` method with the argument `http://www.codeigniter.com/`, but to retrieve the URL is responsibility of the `$this->httpClient` object. The class under test does not care what happens in the dependency, it only cares about the data which comes back.

So, we can replace it with a fake object like the following in testing.

```
<?php

class FakeHttpClientLib
{
    public function get($url)
    {
        return '<h1>CodeIgniter Rocks</h1>';
    }
}
```

This is a test double. When we use it, we can test our code without an internet connection. Our unit tests will not be affected by the availability of an internet connection, the server status, or changes to the content of the requested web page.

There are many types of Test Doubles, like Mock, Stub, Dummy, Fake, and Spy, but for the purposes of this book, the differences don't really matter.

Mocks and Stubs

Mocks and *Stubs* are types of Test Doubles you see discussed very often. In this book, we use *Mock* as a synonym for *Test Double*.

4. PHPUnit Basics

In this chapter, we will learn the basics of PHPUnit, including how to run PHPUnit, how to configure PHPUnit, and how to write test code. We will also study the basic conventions and functionality of PHPUnit including data providers, fixtures, and assertions.

4.1 Running PHPUnit

There is a sample test case class in ci-phpunit-test, so you can already run some tests.

If you use VisualPHPUnit, go to Running PHPUnit via Web Browser.

Running All Tests

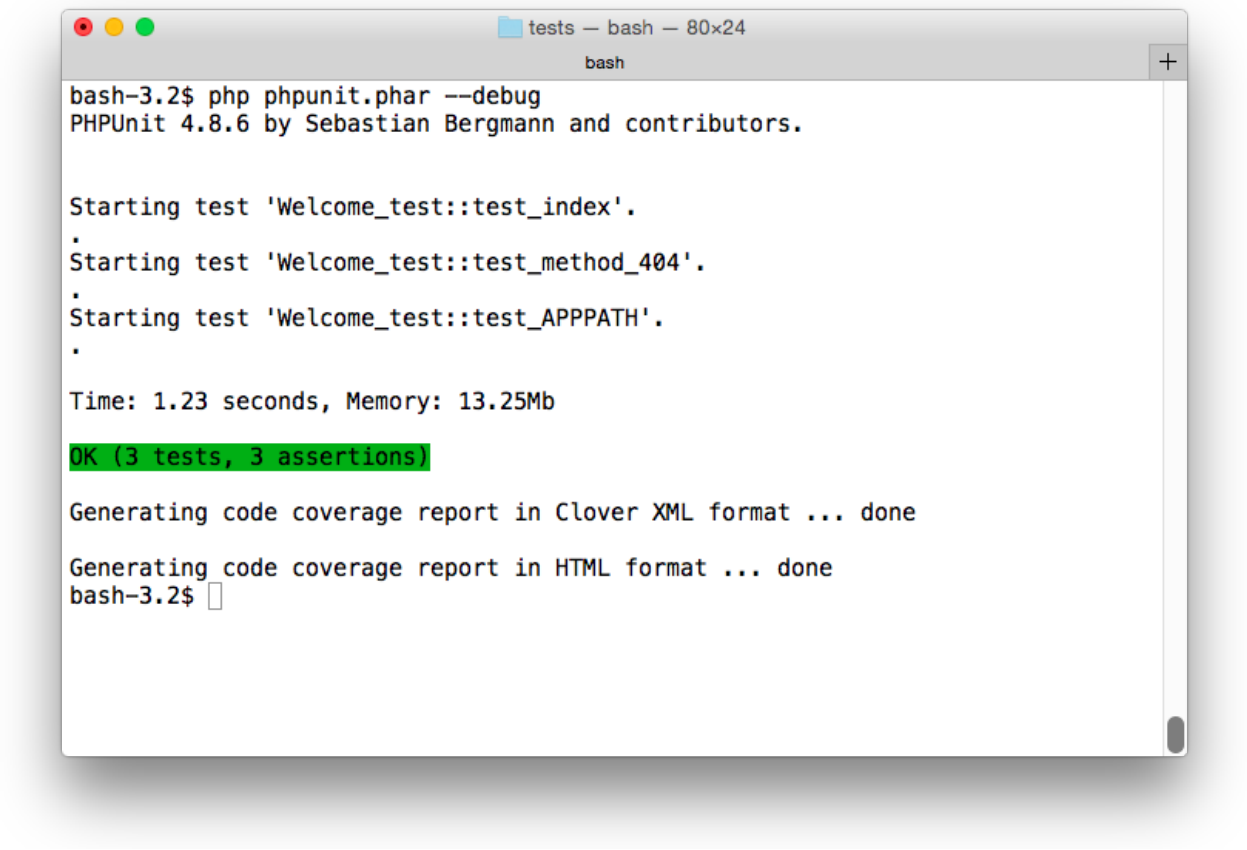
If you run the `phpunit` command without arguments, PHPUnit runs all test case classes. Using the `--debug` option shows additional information which might be of use in debugging.

```
$ cd CodeIgniter/application/tests/
$ php phpunit.phar --debug
```

To Composer users

```
$ ./phpunit.sh --debug
```

See Creating phpunit shortcut.



Run phpunit command

When running the tests, if you see "`OK (3 tests, 3 assertions)`" in the output (highlighted in green in the image above), this means that all tests passed. If one or more of the tests fails, you'll see a `FAILURES!` message (often highlighted in red, depending on your environment).

In the image, "`Starting test 'Welcome_test::test_index'.`" is a debug message. It shows which test case class and method is running. A dot (.) on the next line means the test passed.

"`Generating code coverage report in ...`" is a status message informing us that PHPUnit is generating a code coverage report.

Code Coverage Report

To generate code coverage report, PHPUnit needs Xdebug[1] or phpdbg[2].

If you use Xdebug, you must have Xdebug installed and you will need to enable it before running the tests.

If you use phpdbg, you must have phpdbg installed and you will need to run the following command:

[1] http://xdebug.org/
[2] http://phpdbg.com/

```
$ phpdbg -qrr phpunit.phar --debug
```

To Composer users

```
$ phpdbg -qrr ../../vendor/bin/phpunit --debug
```

To see the HTML code coverage report, open `application/tests/build/coverage/index.html`.

Running a Specific Test Case

You can specify a test file for PHPUnit to run by supplying the filename (and any necessary path information) as an argument when executing PHPUnit.

```
$ php phpunit.phar controllers/Welcome_test.php
```

Okay, go to Configuring PHPUnit.

4.2 Running PHPUnit via Web Browser

Running Web Server

To use VisualPHPUnit, we will use PHP's built-in web server.

If you have a web server like Apache installed, you can use it. For more information, see https://github.com/VisualPHPUnit/VisualPHPUnit#web-server-configuration.

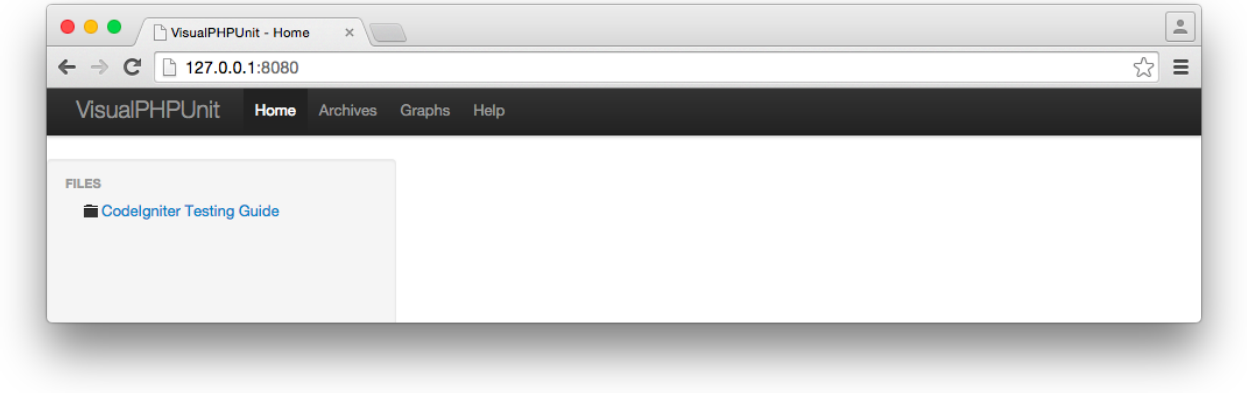
Open your terminal, navigate to the directory in which you've placed VisualPHPUnit, and type the following commands:

```
$ cd app/public/
$ php -S 127.0.0.1:8080
```

Then, access the following URL via your web browser:

- http://127.0.0.1:8080/

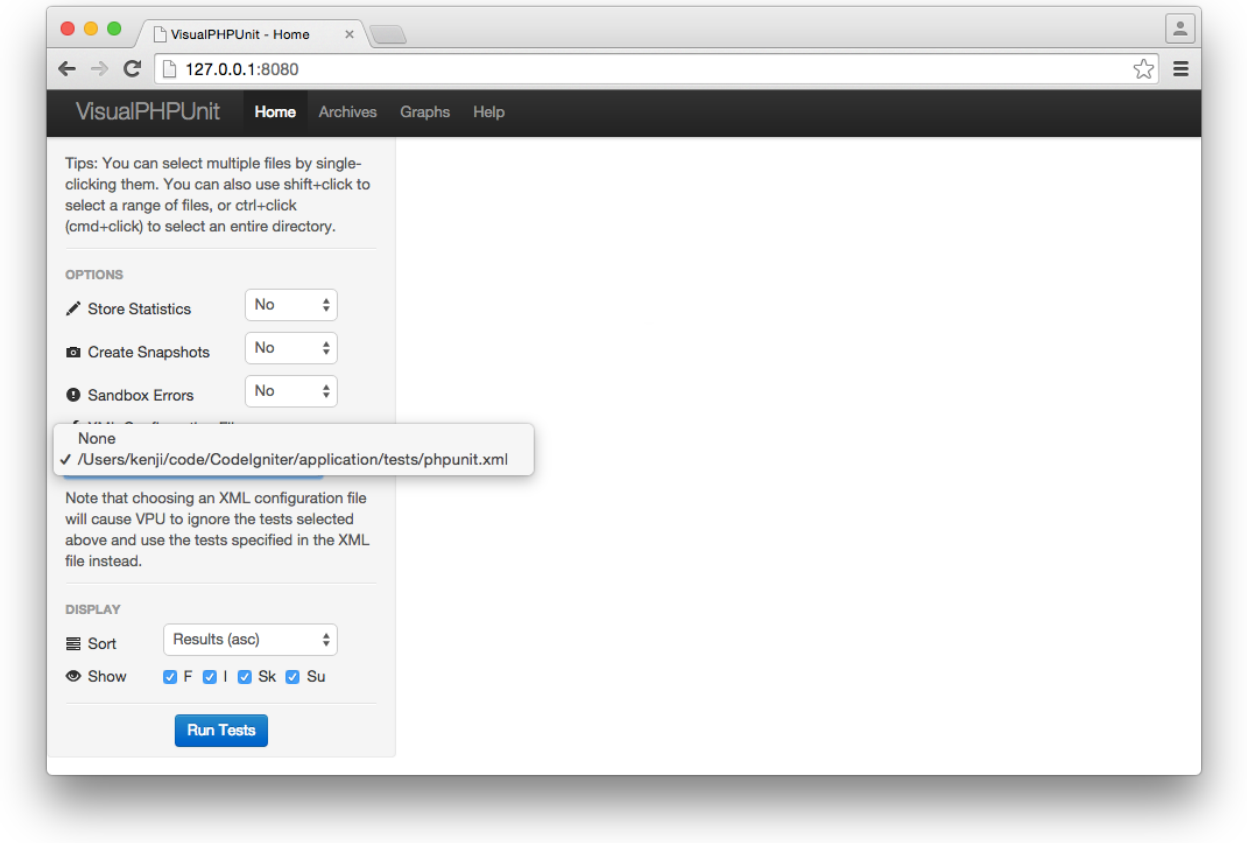
You should see the VisualPHPUnit Home page.



VisualPHPUnit Home

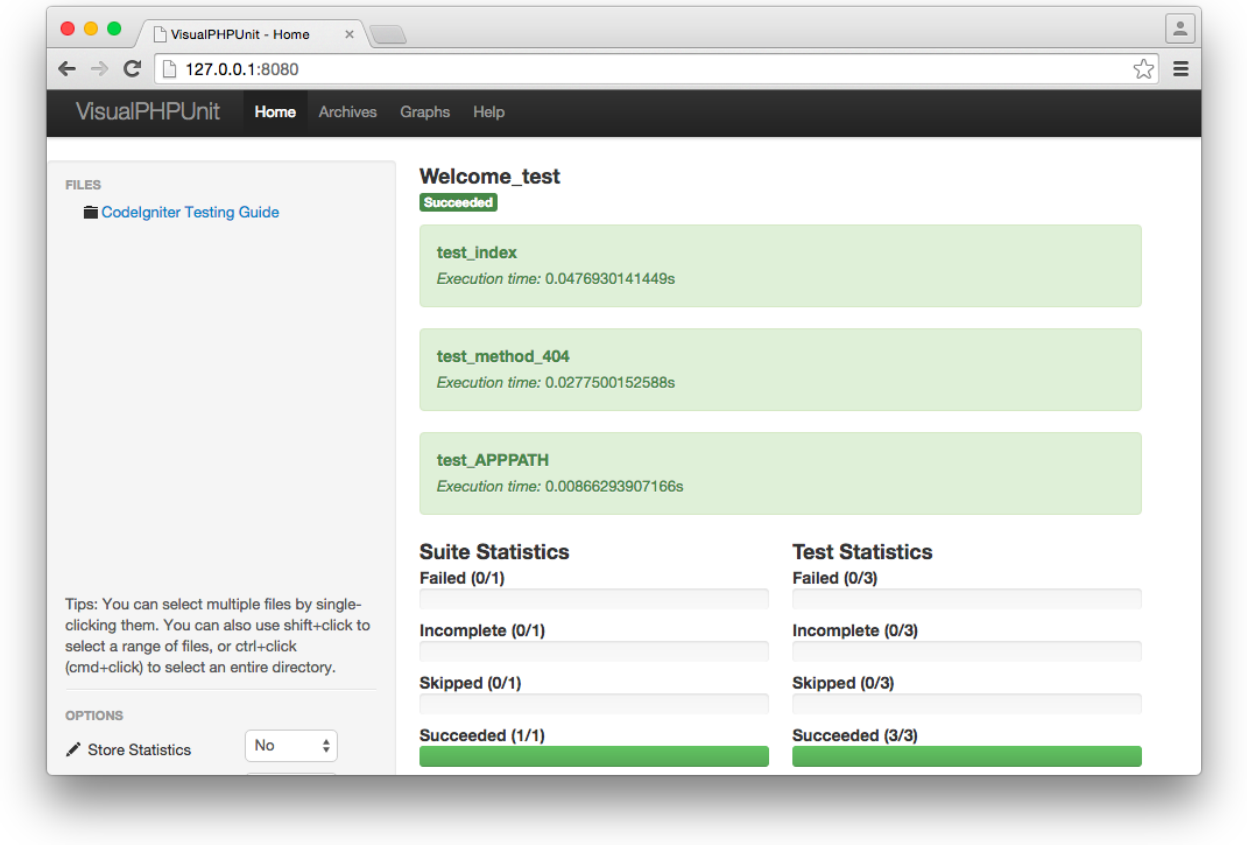
Running All Tests

To run all tests, you use XML Configuration File. Select the XML file you configured, and click the [Run Tests] button or press the [T] key.



VisualPHPUnit XML Configuration File

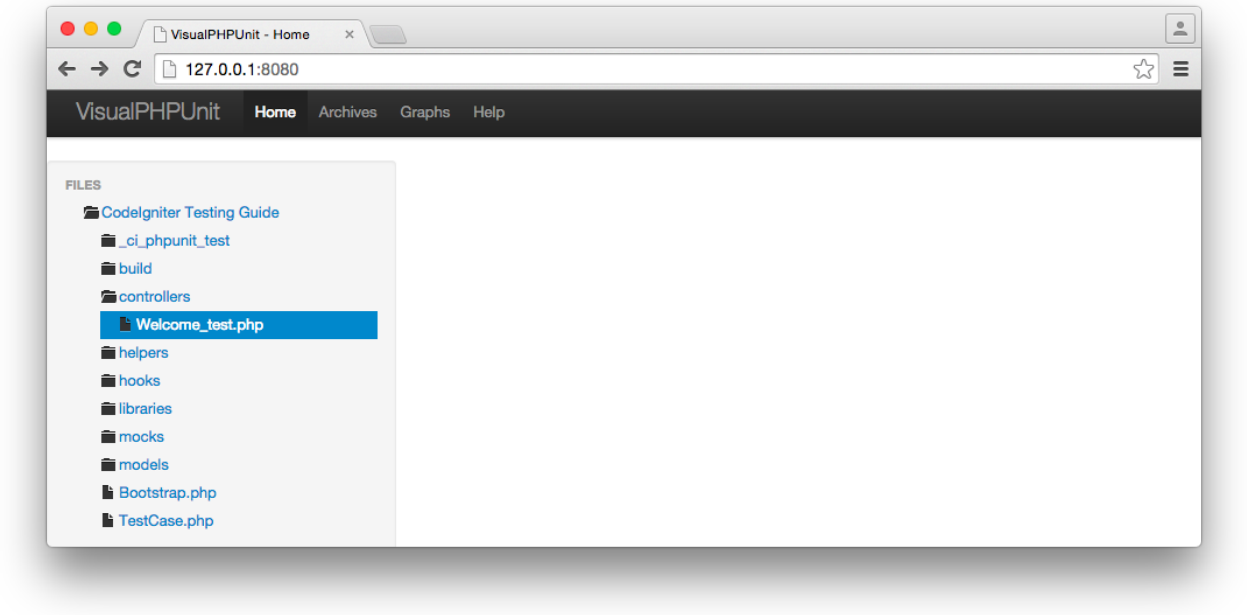
You should see the results like the following.



VisualPHPUnit Run Tests using XML Configuration File

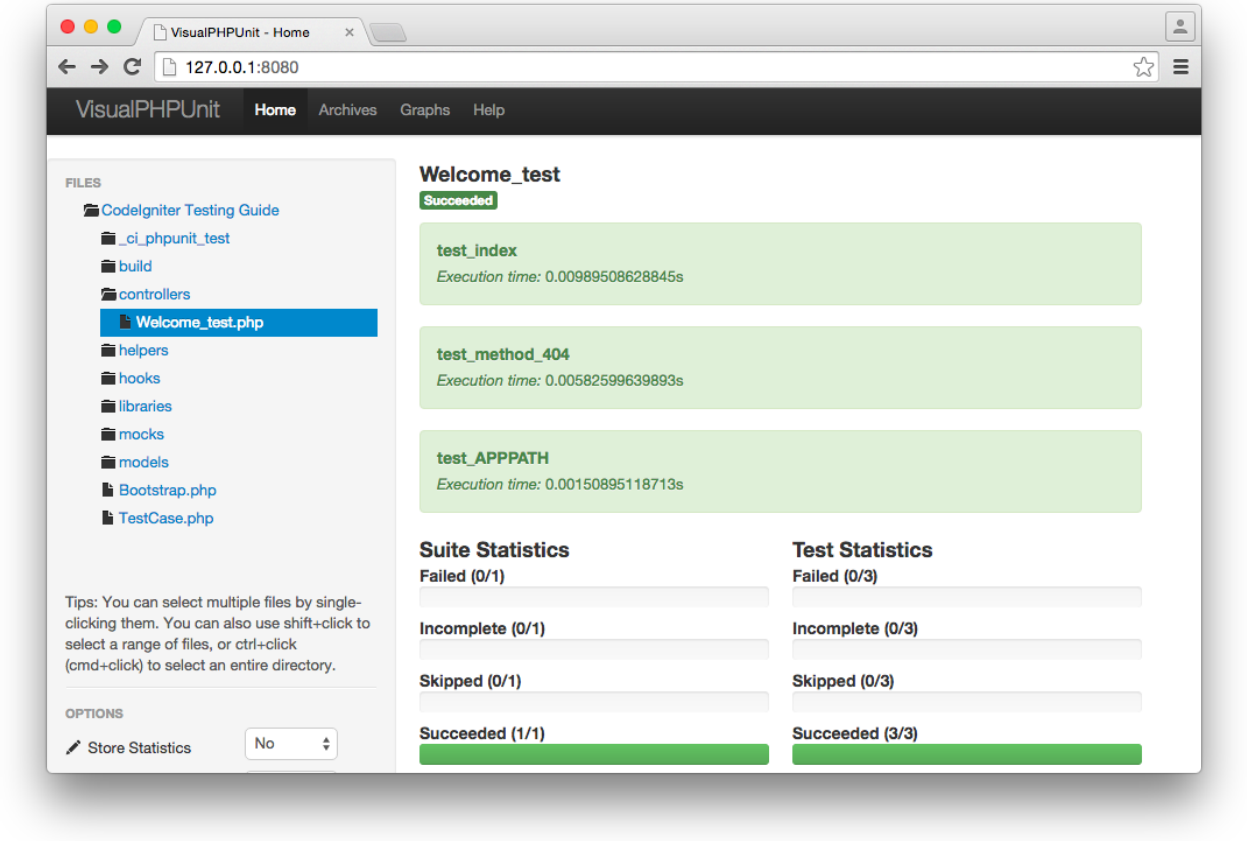
Running a Specific Test Case

To run a specific test case file, select the file, and make sure `XML Configuration File` is `None`. Then click the [Run Tests] button or press the [T] key.



VisualPHPUnit Select a File

You should see the results like the following.



VisualPHPUnit Run Tests

4.3 Configuring PHPUnit

In the previous section, you ran the `phpunit` command and saw the output. Since most of the configuration options were specified by a PHPUnit configuration file, you only specified whether you wanted to include debug output (by using the `--debug` option).

XML Configuration File

`application/tests/phpunit.xml` is the PHPUnit configuration file which specified most of the options used in our previous tests.

phpunit

The attribues of the first element, `phpunit`, configure PHPUnit itself.

```
<phpunit
    bootstrap="./Bootstrap.php"
    colors="true">
```

The `bootstrap` attribute sets the bootstrap file to be used by PHPUnit. The `colors` attribute tells it to use colors in the output. If your terminal doesn't have color capability, change the value of the `colors` attribute from `true` to `false`.

testsuites

The next element, `testsuites`, defines the test suites which will be executed by PHPUnit. A test suite is a set of test case files, usually related in some way. Each `testsuites` element contains one or more `testsuite` elements, each specifying files or directories which will be included (or excluded) in the test and a name for the test suite itself. In this case, our test case files are in `./` (`application/tests`) directory, and we specify a `test.php` suffix, so only files ending in `test.php` within that directory will be included. We are also excluding files in the `./_ci_phpunit_test/` directory.

```
<testsuites>
  <testsuite name="CodeIgniter Application Test Suite">
    <directory suffix="test.php">./</directory>
    <exclude>./_ci_phpunit_test/</exclude>
  </testsuite>
</testsuites>
```

filter

The next element, `filter`, is used to configure code coverage reporting. The `whitelist` element contains elements which define files and directories to be included in the report. The first `directory` element indicates that we include files with suffix `.php` in the `../controllers` (`application/controllers`) directory.

```
<filter>
  <whitelist>
    <directory suffix=".php">../controllers</directory>
    <directory suffix=".php">../models</directory>
    <directory suffix=".php">../views</directory>
    <directory suffix=".php">../libraries</directory>
    <directory suffix=".php">../helpers</directory>
    <directory suffix=".php">../hooks</directory>
  </whitelist>
</filter>
```

The whitelist filter is important in accurately reporting code coverage. If you don't define it, PHPUnit only includes PHP files which run at least one line during test execution. If there are some PHP classes for which you don't write any test code, they probably will not run during test execution, and they are not included in the coverage report. This means you have 0% coverage for those classes, but your coverage report doesn't include the classes in calculating the coverage for your application.

In contrast, some third party libraries which you use may be included in coverage reports if you don't define a whitelist element. You may not want to include the third party libraries, either because you don't want to run the additional tests, or the tests aren't available.

Do We Need to Test Third Party Libraries?

If they are well-tested or they are very stable and you trust them, you probably don't need to test them. Otherwise, it is probably better to write some tests. In most cases, you'll probably want to write some tests to at least cover your own assumptions about the library.

logging

The next element, `logging`, configures the logging of test results. It defines where to put coverage report files.

```
20 <logging>
21   <log type="coverage-html" target="build/coverage"/>
22   <log type="coverage-clover" target="build/logs/clover.xml"/>
23   <log type="junit" target="build/logs/junit.xml" logIncompleteSkipped="false"/>
24 </logging>
```

You can read the details and other configuration options in the PHPUnit Manual https://phpunit.de/manual/4.8/en/appendixes.configuration.html

Command Line Arguments and Options

The `phpunit` command has many options, but we don't use most of them, because it is usually easier to specify them in the configuration file (especially when we want to use the same settings every time we run a test). It might be preferable to specify an option on the command line if you want to change an option for a single execution of the tests or an option isn't available through the configuration file, but you'll usually specify your options in the file.

Here is a list of commonly-used options. If you want to see a full list of options, run `phpunit --help`.

Output

- `--testdox`

This option reports test execution progress in *TestDox* format.

```
$ php phpunit.phar --testdox
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

Welcome_test
 [x] test index
 [x] test method 404
 [x] test APPPATH
```

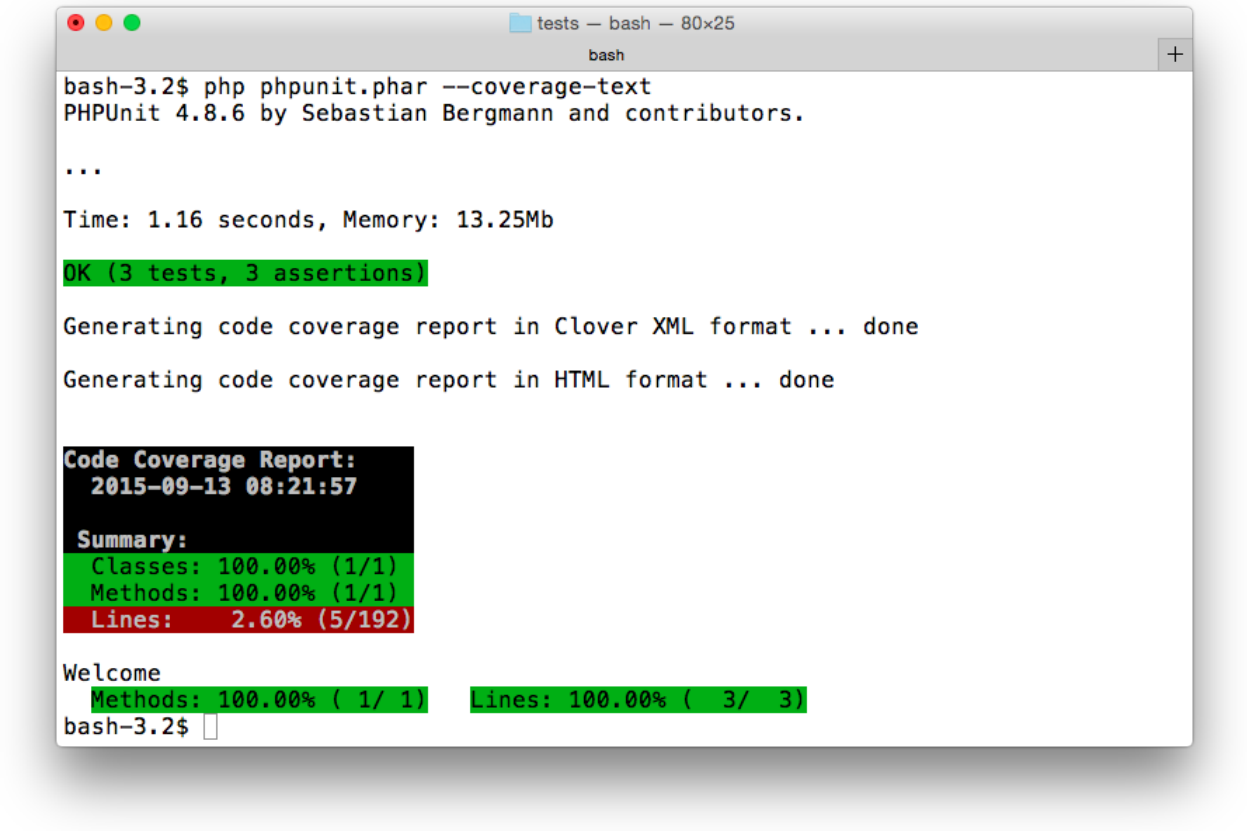
- `--debug`
- `-v`/`--verbose`

We have already used the `--debug` option, which displays debugging information during test execution. The `-v` or `--verbose` option outputs more information. If you get errors during test execution, these options might help you determine the source of the errors.

Coverage Reporting

- `--coverage-text`

This option outputs the code coverage report in a text-only format in the terminal.



phpunit –coverage-text

- `--no-coverage`

This option ignores the code coverage configuration. Code coverage reporting takes time, so this option allows you to skip generating the reports if you just want a quick look at the test results.

Grouping

- `--group`
- `--exclude-group`

These options only run tests from the specified groups, or exclude tests from the specified groups.

We can tag a test case class or method with one or more groups using the `@group` annotation like this:

```
/**
 * @group model
 * @group database
 */
class News_model_test extends TestCase
```

The comment marks should begin with `/**` (a slash followed by two asterisks). If you use `/*`, the annotations will not work. The `@group` tag is an extension of the *PHPDoc* syntax, and may coexist with other PHPDoc tags.

If you want to run tests in the `model` group or `database` group, you can use the `--group` option on the command line as shown:

```
$ php phpunit.phar --group model,database
```

If you want to run tests in the `model` group, but exclude the `database` group, use the `--group` option as before, but add the `--exclude-group` option:

```
$ php phpunit.phar --group model --exclude-group database
```

4.4 Understanding the Basics by Testing Libraries

In CodeIgniter, libraries are the classes located in the `application/libraries` directory. In this section we will write test code for a library class to help us understand the basics of PHPUnit.

Here is the class we will test. It is a calculator which converts Celsius to Fahrenheit, and Fahrenheit to Celsius.

application/libraries/Temperature_converter.php

```
1  <?php
2
3  class Temperature_converter
4  {
5      /**
6       * Converts Celsius to Fahrenheit
7       *
8       * @param float $degree
9       * @return float
10      */
11     public function CtoF($degree)
```

```
12      {
13          return round((9 / 5) * $degree + 32, 1);
14      }
15
16      /**
17        * Converts Fahrenheit to Celsius
18        *
19        * @param float $degree
20        * @return float
21        */
22      public function FtoC($degree)
23      {
24          return round((5 / 9) * ($degree - 32), 1);
25      }
26  }
```

Basic Conventions

Conventions for PHPUnit

Here are basic conventions for PHPUnit.

1. The tests for a class `Class` go into a class `ClassTest`.
2. `ClassTest` extends `PHPUnit_Framework_TestCase` (most of the time).
3. The tests are public methods named `test*`. Alternatively, you can use the `@test` tag in a method's docblock to mark it as a test method.
4. Inside the test methods, assertion methods such as `assertEquals()` are used to assert that an actual value matches an expected value.

Conventions for ci-phpunit-test

We will change some of these conventions for CodeIgniter according to CodeIgniter's coding standards and to provide a convenient way to test.

1. The tests for a class named `Class` go into a class named `Class_test`.
2. `Class_test` extends `TestCase`.
3. The tests are public methods named `test_*`. Alternatively, you can use the `@test` tag in a method's docblock to mark it as a test method.

The `TestCase` class extends `PHPUnit_Framework_TestCase`, so we are, technically, still following the conventions for PHPUnit in terms of extending `PHPUnit_Framework_TestCase`, but the `TestCase` class adds some convenient functionality for running tests in CodeIgniter.

We put the test case files in the `application/tests` directory.

```
CodeIgniter/
└── application/
    └── tests/
        ├── Bootstrap.php ... bootstrap file for PHPUnit
        ├── TestCase.php  ... TestCase class
        ├── controllers/  ... put your controller tests
        ├── libraries/    ... put your library tests
        ├── models/       ... put your model tests
        └── phpunit.xml   ... config file for PHPUnit
```

Our First Test

Now that we know the basic conventions, we can write test code for our `Temperature_converter` class.

application/tests/libraries/Temperature_converter_test.php

```php
<?php

class Temperature_converter_test extends TestCase
{
    public function test_FtoC()
    {
        $obj = new Temperature_converter();
        $actual = $obj->FtoC(100);
        $expected = 37.0;
        $this->assertEquals($expected, $actual, '', 0.01);
    }
}
```

To CodeIgniter users

In this case, ci-phpunit-test autoloads the `Temperature_converter` library, so you don't have to call the `$this->load->library()` method in CodeIgniter.

The `$this->assertEquals()` method is one of PHPUnit's assertion methods. It checks whether two values are equal. The third argument allows us to supply an error message, but we set it to an empty string to use the default message. The fourth argument is the accepted delta, or difference, between the first two values which should be considered equal.

Comparisons of Floating-Point Numbers

Comparisons of floating-point numbers using `$this->assertEquals()` should supply an accepted delta as the fourth argument. For more information, see "What Every Computer Scientist Should Know About Floating-Point Arithmetic[3]" and the PHP Manual[4].

Because we compare floating-point numbers, we have to set the fourth argument. When comparing strings or integers we don't need it, so we often write code like this:

```
$this->assertEquals($expected, $actual);
```

Try running the test case with the `phpunit` command.

```
$ php phpunit.phar libraries/Temperature_converter_test.php
```

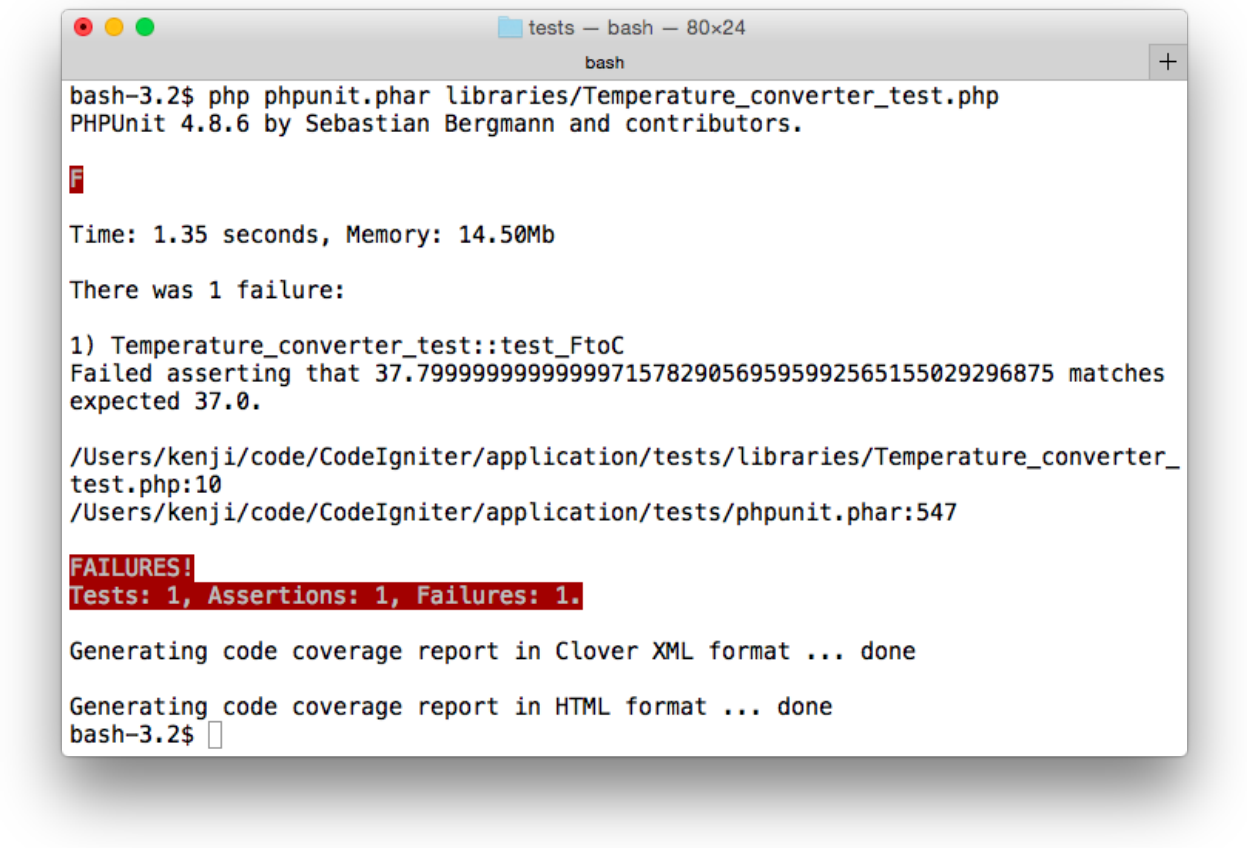
To Composer users

```
$ ./phpunit.sh libraries/Temperature_converter_test.php
```

See Creating a phpunit shortcut.

[3] http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

[4] http://php.net/manual/en/language.types.float.php#language.types.float.comparison



Run phpunit command

This time you should see the red `FAILURES!` message. In fact, there is a bug in our first test. 100 degrees Fahrenheit is not 37.0 Celsius, but 37.8.

You should see a red `F`, because the test failed. The letter marks indicate the following:

- `.` (dot): the test succeeded
- `F`: the test failed
- `I`: the test was incomplete
- `S`: the test was skipped
- `E`: an error occurred while running the test
- `R`: the test is marked as risky

```
There was 1 failure:

1) Temperature_converter_test::test_FtoC
Failed asserting that 37.799999999999971578290569595992565155029296875 matches \
expected 37.0.

/Users/kenji/code/CodeIgniter/application/tests/libraries/Temperature_converter_\
test.php:10
/Users/kenji/code/CodeIgniter/application/tests/phpunit.phar:547

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

You can see which test case class and method failed, as well as the file and line number, which, in this case, is `tests/libraries/Temperature_converter_test.php:10`.

If you fix the test as indicated below, you should see the green `OK` again.

```
--- a/CodeIgniter/application/tests/libraries/Temperature_converter_test.php
+++ b/CodeIgniter/application/tests/libraries/Temperature_converter_test.php
@@ -6,7 +6,7 @@ class Temperature_converter_test extends TestCase
     {
         $obj = new Temperature_converter();
         $actual = $obj->FtoC(100);
-        $expected = 37.0;
+        $expected = 37.8;
         $this->assertEquals($expected, $actual, '', 0.01);
     }
 }
```

The listing above is in a format called *unified diff*. It shows the difference between two files. In short, remove the red line starting with - from the original file and add the green line starting with +, to get the new file. See Conventions Used in This Book for details.

After fixing the test, run `phpunit` again.

```
$ php phpunit.phar libraries/Temperature_converter_test.php
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

.

Time: 1.14 seconds, Memory: 12.50Mb

OK (1 test, 1 assertion)
```

Now we see the green `OK` again.

Data Provider

We have tested one test case. Is it enough? If you think not, you will need to write more tests. How?

application/tests/libraries/Temperature_converter_test.php

```
1  <?php
2
3  class Temperature_converter_test extends TestCase
4  {
5      public function test_FtoC()
6      {
7          $obj = new Temperature_converter();
8
9          $actual = $obj->FtoC(100);
10         $expected = 37.0;
11         $this->assertEquals($expected, $actual, '', 0.01);
12
13         $actual = $obj->FtoC(-40);
14         $expected = -40.0;
15         $this->assertEquals($expected, $actual, '', 0.01);
16     }
17 }
```

Now we have two tests in one test method, but should you contiune adding tests like this? A common saying in programming is "Don't repeat yourself". Continuing to add tests in this manner seems a bit repetetive.

Testing Tip: One Assertion in One Test Method

Using only one (or a few) assertion in one test method is a good practice. It makes it easier to find the cause of failed tests. Do not write tests that test too much.

Fortunately, PHPUnit includes functionality to repeat tests like this. One method of doing so is with a *Data Provider*. Using this, we can write tests like the following:

application/tests/libraries/Temperature_converter_test.php

```
<?php

class Temperature_converter_test extends TestCase
{
    /**
     * @dataProvider provide_temperature_data
     */
    public function test_FtoC($degree, $expected)
    {
        $obj = new Temperature_converter();
        $actual = $obj->FtoC($degree);
        $this->assertEquals($expected, $actual, '', 0.01);
    }

    public function provide_temperature_data()
    {
        return [
            // [Fahrenheit, Celsius]
            [-40, -40.0],
            [  0, -17.8],
            [ 32,   0.0],
            [100,  37.8],
            [212, 100.0],
        ];
    }
}
```

First, the `provide_temperature_data()` method was added. This is the method to provide data for testing. It returns an array of arrays.

Second, the `@dataProvider` tag was added to the docblock of the `test_FtoC()` method. This sets a data provider method name for the test method.

```
    /**
     * @dataProvider provide_temperature_data
     */
```

The `test_FtoC()` method has two parameters (`$degree` and `$expected`). The values in the data provider method are passed to them. The first array `[-40, -40.0]` is passed to the method as `$degree = -40` and `$expected = -40.0`.

PHPUnit repeats the test method for each array in the data provider method.

```
$ php phpunit.phar libraries/Temperature_converter_test.php
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

.....

Time: 1.15 seconds, Memory: 12.50Mb

OK (5 tests, 5 assertions)
```

In the output, you should see five dots and `5 tests, 5 assertions`. If you add the `--debug` option, you can see what's happening more clearly.

```
$ php phpunit.phar --debug libraries/Temperature_converter_test.php
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.


Starting test 'Temperature_converter_test::test_FtoC with data set #0 (-40, -40.\
0)'.
.
Starting test 'Temperature_converter_test::test_FtoC with data set #1 (0, -17.80\
000000000000071054273576010018587112426757812 5)'.
.
Starting test 'Temperature_converter_test::test_FtoC with data set #2 (32, 0.0)'.
.
Starting test 'Temperature_converter_test::test_FtoC with data set #3 (100, 37.7\
999999999999715782905695959925651550292968 75)'.
.
Starting test 'Temperature_converter_test::test_FtoC with data set #4 (212, 100.\
0)'.
.

Time: 1.46 seconds, Memory: 12.50Mb

OK (5 tests, 5 assertions)
```

Testing Tip: Data Provider from Another Class

You can use a data provider from another class by specifying the class in the tag: `@dataProvider Bar::provide_baz_data`. If the class is defined (or autoloadable) and the provider method is public, PHPUnit will use it.

Incomplete Tests

We have written test code for the `FtoC()` method, but there is another method, `CtoF()`, in the class. So we need to test this method, as well.

When we need to write a test method, but we have not finished it, we can use PHPUnit's `$this->markTestIncomplete()` method. This is a marker which can be used to indicate the test is incomplete or not currently implemented.

Add the following method to the bottom of the test case class:

application/tests/libraries/Temperature_converter_test.php

```
    public function test_CtoF()
    {
        $this->markTestIncomplete(
            'This test has not been implemented yet.'
        );
    }
```

An incomplete test is denoted by an `I` in the output of the `phpunit` command, and the `OK` line is yellow, not green, if your terminal has color capability.

```
$ php phpunit.phar libraries/Temperature_converter_test.php --no-coverage
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

.....I

Time: 817 ms, Memory: 7.50Mb

OK, but incomplete, skipped, or risky tests!
Tests: 6, Assertions: 5, Incomplete: 1.
```

If you put a test method without assertions and without `$this->markTestIncomplete()`, you will see the green OK and you cannot determine whether a test is actually successful or just not yet implemented.

Adding Tests

It is easy to add test code for the `CtoF()` method, because `CtoF()` is just the opposite of `FtoC()`. We can reuse the data provider from the `test_FtoC()` method.

Exercise

Please stop here and think about how you would write the test method.

Update the `test_CtoF()` method:

application/tests/libraries/Temperature_converter_test.php

```
    /**
     * @dataProvider provide_temperature_data
     */
    public function test_CtoF($expected, $degree)
    {
        $obj = new Temperature_converter();
        $actual = $obj->CtoF($degree);
        $this->assertEquals($expected, $actual, '', 0.01);
    }
```

The order of the `test_CtoF()` parameters is reversed.

This test case class is okay, but we can still improve it a bit with PHPUnit's `setUp()` method.

Fixtures

A *fixture* is a known state for an application. If you run tests, you must set the world up in a known state before running tests, because any difference in the state may cause changes in the test results.

PHPUnit has some methods for defining fixtures. In this test case, we don't need to do anything, because the class under test has no dependencies except for PHP's internal function, `round()`[5]. The `round()` function also has no dependencies, it just returns a calculated value.

However, we can still improve our test code by setting up our environment.

setUp()

`setUp()` is a method which is called before a test method is run. In other words, PHPUnit calls `setUp()` before running each test method. If you use it, you can share the setup code and the state over multiple test methods.

[5] http://php.net/en/round

```
--- a/CodeIgniter/application/tests/libraries/Temperature_converter_test.php
+++ b/CodeIgniter/application/tests/libraries/Temperature_converter_test.php
@@ -2,13 +2,17 @@

 class Temperature_converter_test extends TestCase
 {
+    public function setUp()
+    {
+        $this->obj = new Temperature_converter();
+    }
+
     /**
      * @dataProvider provide_temperature_data
      */
     public function test_FtoC($degree, $expected)
     {
-        $obj = new Temperature_converter();
-        $actual = $obj->FtoC($degree);
+        $actual = $this->obj->FtoC($degree);
         $this->assertEquals($expected, $actual, '', 0.01);
     }

@@ -29,8 +33,7 @@ class Temperature_converter_test extends TestCase
      */
     public function test_CtoF($expected, $degree)
     {
-        $obj = new Temperature_converter();
-        $actual = $obj->CtoF($degree);
+        $actual = $this->obj->CtoF($degree);
         $this->assertEquals($expected, $actual, '', 0.01);
     }
 }
```

The listing above is in a format called *unified diff*. It shows the difference between two files. In short, remove the red line starting with - from the original file and add the green line starting with + to get the new file. See Conventions Used in This Book for details.

Now, we no longer repeat "`$obj = new Temperature_converter();`" for each test.

tearDown()

PHPUnit also has `tearDown()` method which is called after a test method is run, allowing you to clean up the environment after the test.

Note for ci-phpunit-test

Don't forget to call `parent::tearDown();` if you override the `tearDown()` method.

Other Fixture Methods

- `setUpBeforeClass()` is a static method called before running the first test of the test case class
- `tearDownAfterClass()` is a static method called after running the last test of the test case class

Note for ci-phpunit-test

Don't forget to call `parent::setUpBeforeClass();` if you override the `setUpBeforeClass()` method and `parent::tearDownAfterClass();` if you override the `tearDownAfterClass()` method.

Assertions

PHPUnit has many assertion methods. Here is a list of commonly-used assertions:

- `assertEquals($expected, $actual)` checks whether two values are equal
- `assertSame($expected, $actual)` checks whether two values are equal and the same type
- `assertTrue($condition)` checks whether a *condition* is `true`
- `assertFalse($condition)` checks whether a *condition* is `false`
- `assertNull($variable)` checks whether a *variable* is `null`
- `assertInstanceOf($expected, $actual)` checks the type of an object
- `assertCount($expectedCount, $haystack)` checks whether the number of elements in `$haystack` matches `$expectedCount`
- `assertRegExp($pattern, $string)` checks whether `$string` matchs the regular expression `$pattern`
- `assertContains($needle, $haystack)` checks whether `$needle` is contained in `$haystack`, this method works with strings (`$needle` is a substring of `$haystack`), arrays (`$needle` is an element of `$haystack`), or classes which implement `Iterator` (can be called using `foreach()`)

You can see all of the available methods in the PHPUnit Manual[6].

Testing Tip: Use Specific Assert Methods

Using specific assert methods is a good practice, because it expresses what you want to test and you get more helpful error messages.

[6] https://phpunit.de/manual/4.8/en/appendixes.assertions.html

5. Testing a Simple MVC Application

To begin learning how to write tests, we will start with a simple but essential MVC application, a CodeIgniter Tutorial application included in the CodeIgniter User Guide. It is a basic news application which is thoroughly covered in the Tutorial[1].

The application is licensed under the MIT License. See License Agreement for CodeIgniter and its User Guide.

I changed the coding style of the application source code a bit, and added commentary as comments starting with #.

5.1 Functional Testing for Controller

Since CodeIgniter treats models as optional, we will begin with testing our controller.

Controller to Handle Static Pages

The first thing you're going to see is a controller to handle static pages.

Pages Controller

application/controllers/Pages.php

```
1  <?php
2
3  class Pages extends CI_Controller
4  {
5      public function view($page = 'home')
6      {
7          if (! file_exists(APPPATH.'/views/pages/'.$page.'.php')) {
8              // Whoops, we don't have a page for that!
9              show_404();
10         }
11
12         $data['title'] = ucfirst($page); // Capitalize the first letter
13
```

[1] http://www.codeigniter.com/user_guide/tutorial/

```
14          $this->load->view('templates/header', $data);
15          $this->load->view('pages/'.$page, $data);
16          $this->load->view('templates/footer', $data);
17      }
18  }
```

The `Pages` controller's `view()` method is automatically routed to `pages/view` by CodeIgniter. If you access `http://127.0.0.1:8000/pages/views/home`, CodeIgniter calls the `Pages` controller's `views()` method and passes `home` as the first argument.

The calls to the `$this->load->view()` method load a view file and pass the second argument, `$data`, to the view. In the view file, you can access the value of `$data['title']` by using `$title`.

Page Templates

The views which start with `templates/` are page templates, views containing the header and footer to be used by our site.

application/views/templates/header.php

```
1  <html>
2      <head>
3          <title>CodeIgniter Tutorial</title>
4      </head>
5      <body>
6
7          <h1><?php echo $title; ?></h1>
```

application/views/templates/footer.php

```
1          <em>&copy; 2015</em>
2      </body>
3  </html>
```

Static Pages

The files loaded between the templates are the static content for the pages. For this example, we'll use `home` and `about`.

application/views/pages/home.php

```
<p>
This is `views/pages/home.php`.
</p>
```

application/views/pages/about.php

```
<p>
This is `views/pages/about.php`.
</p>
```

Routing

Next, we'll change the routing definitions in `application/config/routes.php`.

```
--- a/CodeIgniter/application/config/routes.php
+++ b/CodeIgniter/application/config/routes.php
@@ -49,6 +49,7 @@ defined('BASEPATH') OR exit('No direct script access allowed');
 | Examples:    my-controller/index -> my_controller/index
 |      my-controller/my-method -> my_controller/my_method
 */
-$route['default_controller'] = 'welcome';
+$route['default_controller'] = 'pages/view';
 $route['404_override'] = '';
 $route['translate_uri_dashes'] = FALSE;
+$route['(:any)'] = 'pages/view/$1';
```

The listing above is in a format called *unified diff*. It shows the difference between two files. In short, remove the red line starting with - from the original file and add the green line starting with +, to get the new file. See Conventions Used in This Book for further details.

The `default_controller` setting configures the controller/method used when no controller is supplied. Since we've modified the `default_controller` value to `'pages/view'`, if you access `http://127.0.0.1:8000/`, CodeIgniter calls the `Pages` controller's `view()` method without passing any arguments.

The special keyword `(:any)` when used in a route's key will match a URI segment containing any character except for `/`. So, if you access `http://127.0.0.1:8000/about`, CodeIgniter calls the `Pages` controller's `view()` method and passes `'about'` as the first argument.

Manual Testing with a Web Browser

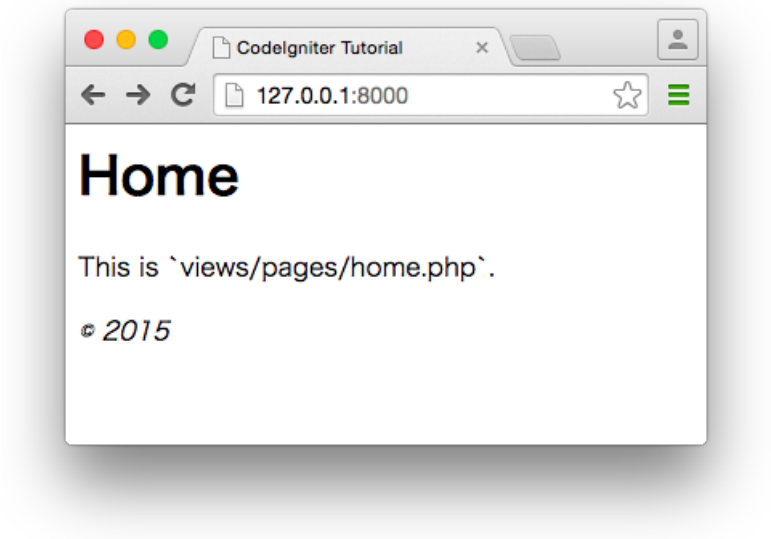
Let's access the web site. In this book, we will use PHP's built-in web server. Open your terminal, navigate to the directory in which you've placed CodeIgniter's `index.php`, and type the following command:

```
$ php -S 127.0.0.1:8000 index.php
```

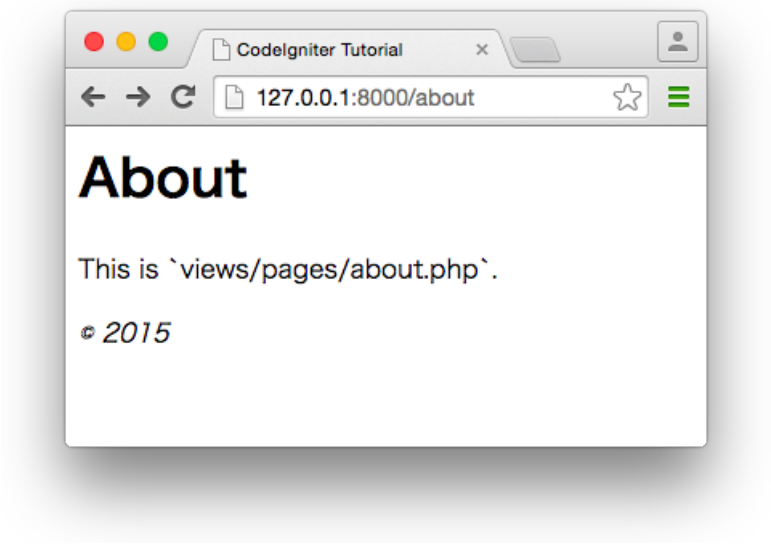
Then, access the following URLs via your web browser:

- http://127.0.0.1:8000/
- http://127.0.0.1:8000/about
- http://127.0.0.1:8000/notfound

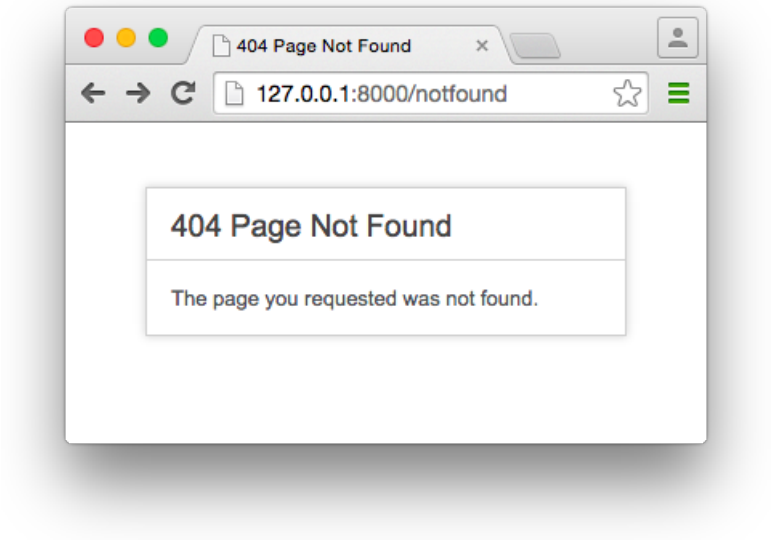
You should see the Home and About pages, followed by a 404 Page Not Found message.



Home



About



404 Page Not Found

Test Case for Page Controller

Some people have said that testing CodeIgniter controllers is difficult, but ci-phpunit-test has a lot of helper methods to simplify this process. Now you can easily write tests for your controllers.

First, we create the test case class in a file located at `application/tests/controllers/Pages_-test.php`.

application/tests/controllers/Pages_test.php

```
<?php

class Pages_test extends TestCase
{

}
```

Then we add our first test method.

application/tests/controllers/Pages_test.php

```
<?php

class Pages_test extends TestCase
{
    public function test_When_you_access_home_Then_you_see_Home()
    {
        $output = $this->request('GET', '/');
        $this->assertContains('<h1>Home</h1>', $output);
    }
}
```

It is simple, isn't it?

The `$this->request()` method in ci-phpunit-test is a method for testing the controller. It sends (emulates) a request to a controller. The first argument is the HTTP method name. The second argument is the URI string. It returns the output (HTML) string from the controller's method.

Many popular frameworks include similar functionality to functional-test controllers.

Run `phpunit` and make sure the test passes.

Next, we'll add two more test methods. Here is the complete listing of our `Pages_test` class.

application/tests/controllers/Pages_test.php

```php
<?php

/**
 * @group controller
 */
class Pages_test extends TestCase
{
    public function test_When_you_access_home_Then_you_see_Home()
    {
        $output = $this->request('GET', '/');
        $this->assertContains('<h1>Home</h1>', $output);
    }

    public function test_When_you_access_about_Then_you_see_About()
    {
        $output = $this->request('GET', 'about');
        $this->assertContains('<h1>About</h1>', $output);
    }

    public function test_When_you_access_notfound_Then_you_get_404()
    {
        $this->request('GET', 'notfound');
        $this->assertResponseCode(404);
    }
}
```

We also added the `@group` tag to tell PHPUnit this test belongs to a group named `controller`.

```
/**
 * @group controller
 */
```

This allows us to run tests in the `controller` group without having to run the rest of our tests.

```
$ php phpunit.phar --group controller
```

When you use `$this->request()`, you can use the `$this->assertResponseCode()` method in ci-phpunit-test. This is an assertion method to check the response code of the request.

So what these test methods do is,

1. Send a request to `/`, and check whether the response output contains `<h1>Home</h1>`
2. Send a request to `/about`, and check whether the response output contains `<h1>About</h1>`
3. Send a request to `/notfound`, and check whether the response code is `404`

These tests are the same thing you might do with your web browser by hand. They are easy to understand and to write, aren't they?

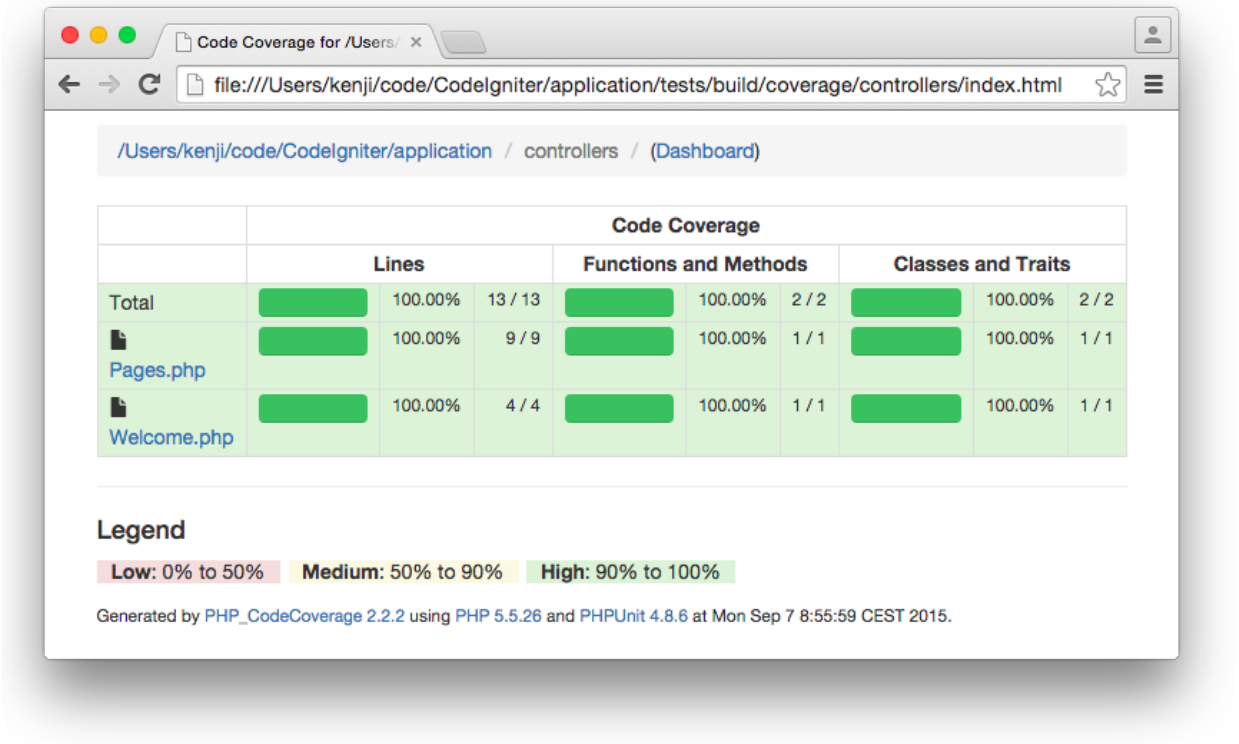
This is called functional testing, because we are testing a controller's functionality.

```
$ php phpunit.phar
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

................

Time: 2.3 seconds, Memory: 13.50Mb

OK (16 tests, 16 assertions)

Generating code coverage report in Clover XML format ... done

Generating code coverage report in HTML format ... done
```

If you run your tests, you should see the green `OK`.

Checking Code Coverage

Let's check code coverage for the controller. Open `application/tests/build/coverage/index.html` in your web browser, then follow the `controllers` link. You'll see coverage for controllers listed like this:



Code Coverage for Controllers

You should see 100% coverage for the Pages controller.

5.2 Database Testing for Models

We have just tested a controller. Next, we will test a model. Before looking at our application code, we should prepare our database.

Preparing the Database

Database Configuration

We use SQLite in this book, but if you like MySQL, of course you can use it.

SQLite

First of all, configure CodeIgniter's database config file. We are using the PDO SQLite3 driver.

```
--- a/CodeIgniter/application/config/database.php
+++ b/CodeIgniter/application/config/database.php
@@ -74,12 +74,12 @@ $active_group = 'default';
 $query_builder = TRUE;

 $db['default'] = array(
-    'dsn'   => '',
+    'dsn'   => 'sqlite:' . APPPATH . 'data/sqlite-database.db',
     'hostname' => 'localhost',
     'username' => '',
     'password' => '',
     'database' => '',
-    'dbdriver' => 'mysqli',
+    'dbdriver' => 'pdo',
     'dbprefix' => '',
     'pconnect' => FALSE,
     'db_debug' => (ENVIRONMENT !== 'production'),
```

The listing above is in a format called *unified diff*. It shows the difference between two files. In short, remove the red line starting with - from the original file and add the green line starting with +, to get the new file. See Conventions Used in This Book for further details.

Create the `application/data` directory to hold the sqlite database file.

If you set "`'dsn' => 'sqlite::memory:'`", you can use an SQLite in-memory database.

If you close the database connection to the SQLite in-memory database, the database ceases to exist. ci-phpunit-test does not close it, but if you close it in your test code or application code, the database will cease to exist.

MySQL

If you use MySQL and the `mysqli` driver, your config file will be something like this:

```
--- a/CodeIgniter/application/config/database.php
+++ b/CodeIgniter/application/config/database.php
@@ -76,9 +76,9 @@ $query_builder = TRUE;
 $db['default'] = array(
    'dsn'   => '',
    'hostname' => 'localhost',
-   'username' => '',
-   'password' => '',
-   'database' => '',
+   'username' => 'dbuser',
+   'password' => 'dbpass',
+   'database' => 'testing',
    'dbdriver' => 'mysqli',
    'dbprefix' => '',
    'pconnect' => FALSE,
```

Dedicated Test Database

It is good practice to use a dedicated database for testing. CodeIgniter includes functionality to handle multiple environments[2]. It reads config files in the `application/config/[environment]` directory, if present. ci-phpunit-test always runs tests in the `testing` environment. So, if you put your database configuration in `application/config/testing/database.php`, this configuration will be used for your tests.

Database Migration

We will create a database table using CodeIgniter's migration library.

Create the `application/database/migrations` directory for migration files, then create a migration file to create the `news` table.

The default location of migration files in CodeIgniter is `application/migrations`, but we have changed it. We will have database seeding files later, which we don't want to have mixed up with our migrations.

[2] http://www.codeigniter.com/user_guide/libraries/config.html#environments

application/database/migrations/20150907090030_Create_news.php

```php
<?php

class Migration_Create_news extends CI_Migration
{
    public function up()
    {
        $this->dbforge->add_field([
            'id' => [
                'type' => 'INT',
                'constraint' => 11,
                'auto_increment' => true
            ],
            'title' => [
                'type' => 'VARCHAR',
                'constraint' => 128,
            ],
            'slug' => [
                'type' => 'VARCHAR',
                'constraint' => 128,
            ],
            'text' => [
                'type' => 'TEXT',
            ],
        ]);
        $this->dbforge->add_key('id', true);
        $this->dbforge->add_key('slug');
        $this->dbforge->create_table('news');
    }

    public function down()
    {
        $this->dbforge->drop_table('news');
    }
}
```

Edit the migration config file to set `migration_enabled` to `true` and set the `migration_version` to the timestamp on our `create_news` migration. We also need to update our `migration_path` to `APPPATH.'database/migrations/'`.

```
--- a/CodeIgniter/application/config/migration.php
+++ b/CodeIgniter/application/config/migration.php
@@ -11,7 +11,7 @@ defined('BASEPATH') OR exit('No direct script access allowed');
 | and disable it back when you're done.
 |
 */
-$config['migration_enabled'] = FALSE;
+$config['migration_enabled'] = TRUE;

 /*
 |--------------------------------------------------------------------------
@@ -69,7 +69,7 @@ $config['migration_auto_latest'] = FALSE;
 | be upgraded / downgraded to.
 |
 */
-$config['migration_version'] = 0;
+$config['migration_version'] = 20150907090030;

 /*
 |--------------------------------------------------------------------------
@@ -81,4 +81,4 @@ $config['migration_version'] = 0;
 | Also, writing permission is required within the migrations path.
 |
 */
-$config['migration_path'] = APPPATH.'migrations/';
+$config['migration_path'] = APPPATH.'database/migrations/';
```

Dbfixture Controller

Create a `Dbfixture` controller to run migrations. We can use the `Migrate` class in CodeIgniter User Guide[3] as a reference.

application/controllers/Dbfixture.php

```
1  <?php
2
3  class Dbfixture extends CI_Controller
4  {
5      public function __construct()
6      {
7          parent::__construct();
8
```

[3] http://www.codeigniter.com/user_guide/libraries/migration.html#usage-example

```
9           // Only accessible via CLI
10          if (is_cli() === false) {
11              exit();
12          }
13      }
14
15      public function migrate()
16      {
17          $this->load->library('migration');
18          if ($this->migration->current() === false) {
19              echo $this->migration->error_string() . PHP_EOL;
20          }
21      }
22  }
```

If the controller is accessible via the web browser, everybody who can access your site can run migrations. This is not good, so we added some logic to check whether it was called via CLI, and exit if not.

Run the `Dbfixture` controller via the CLI to run your migration.

```
$ php index.php dbfixture migrate
```

If there is no output, the migration ran and the `news` table was created successfully. You can confirm this using the `sqlite3` command.

```
$ sqlite3 application/data/sqlite-database.db
SQLite version 3.8.5 2014-08-15 22:37:57
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE "migrations" (
        "version" BIGINT NOT NULL
);
CREATE TABLE "news" (
        "id" INTEGER PRIMARY KEY AUTOINCREMENT,
        "title" VARCHAR NOT NULL,
        "slug" VARCHAR NOT NULL,
        "text" TEXT NOT NULL
);
CREATE INDEX "news_slug" ON "news" ("slug");
sqlite> .exit
```

You can see the `news` table in the database.

News Section

Let's take a look at the source code for the News Section.

News_model Model

The `News_model` class has two methods. One is to get news items from the database, the other is to insert a news item into the database.

application/models/News_model.php

```
<?php

class News_model extends CI_Model
{
    public function __construct()
    {
# $this->load->database() method loads database class. After loading, you can
# use it as "$this->db".
        $this->load->database();
    }

    public function get_news($slug = false)
    {
        if ($slug === false) {
# $this->db->get() method runs the selection query and returns the result.
            $query = $this->db->get('news');
# result_array() method returns the query result as an array of objects.
            return $query->result_array();
        }

# $this->db->get_where() method runs the selection query with a “where” clause
# in the second parameter.
        $query = $this->db->get_where('news', ['slug' => $slug]);
# row_array() method returns a single result row as an array.
        return $query->row_array();
    }

    public function set_news()
    {
# $this->load->helper() method loads CodeIgniter's helper function.
        $this->load->helper('url');

```

```
33  # $this->input->post() method returns $_POST data.
34          $slug = url_title($this->input->post('title'), 'dash', true);
35
36          $data = [
37              'title' => $this->input->post('title'),
38              'slug'  => $slug,
39              'text'  => $this->input->post('text')
40          ];
41
42  # $this->db->insert() method runs insertion query.
43          return $this->db->insert('news', $data);
44      }
45  }
```

News Controller

The `News` controller has three action methods. `index()` shows the news archive. `view()` shows one news item. `create()` shows a form to create a news item.

application/controllers/News.php

```
1   <?php
2
3   class News extends CI_Controller
4   {
5       public function __construct()
6       {
7           parent::__construct();
8   # $this->load->model() method loads "News_model" class. After loading, you can
9   # use it as "$this->news_model".
10          $this->load->model('news_model');
11          $this->load->helper('url_helper');
12      }
13
14      public function index()
15      {
16          $data['news']  = $this->news_model->get_news();
17          $data['title'] = 'News archive';
18
19          $this->load->view('templates/header', $data);
20          $this->load->view('news/index', $data);
21          $this->load->view('templates/footer');
22      }
```

```
23
24      public function view($slug = null)
25      {
26          $data['news_item'] = $this->news_model->get_news($slug);
27
28          if (empty($data['news_item'])) {
29              show_404();
30          }
31
32          $data['title'] = $data['news_item']['title'];
33
34          $this->load->view('templates/header', $data);
35          $this->load->view('news/view', $data);
36          $this->load->view('templates/footer');
37      }
38
39      public function create()
40      {
41          $this->load->helper('form');
42  # $this->load->library() method loads CodeIgniter's Form Validation class. After
43  # loading, you can use it as "$this->form_validation".
44          $this->load->library('form_validation');
45
46          $data['title'] = 'Create a news item';
47
48          $this->form_validation->set_rules('title', 'Title', 'required');
49          $this->form_validation->set_rules('text', 'Text', 'required');
50
51          if ($this->form_validation->run() === false) {
52              $this->load->view('templates/header', $data);
53              $this->load->view('news/create');
54              $this->load->view('templates/footer');
55          } else {
56              $this->news_model->set_news();
57              $this->load->view('news/success');
58          }
59      }
60  }
```

Views

This view file shows the news archive.

application/views/news/index.php

```
<h2><?php echo $title; ?></h2>

<?php foreach ($news as $news_item): ?>

    <h3><?php echo $news_item['title']; ?></h3>
    <div class="main">
        <?php echo $news_item['text']; ?>
    </div>
    <p>
        <a href="<?php echo site_url('news/'.$news_item['slug']); ?>">
            View article
        </a>
    </p>

<?php endforeach; ?>
```

This view file shows one news item.

application/views/news/view.php

```
<?php
echo '<h2>'.$news_item['title'].'</h2>';
echo $news_item['text'];
```

This view includes a form to post a news item.

application/views/news/create.php

```
<h2><?php echo $title; ?></h2>

<?php echo validation_errors(); ?>

<?php echo form_open('news/create'); ?>

    <label for="title">Title</label>
    <input type="input" name="title" /><br />

    <label for="text">Text</label>
    <textarea name="text"></textarea><br />

```

```
13      <input type="submit" name="submit" value="Create news item" />
14
15  </form>
```

The success message is displayed after a news item is created.

application/views/news/success.php

```
1  <h2>Successfully created</h2>
```

Routing

We need to add some routes for the News Section.

```
--- a/CodeIgniter/application/config/routes.php
+++ b/CodeIgniter/application/config/routes.php
@@ -49,7 +49,11 @@ defined('BASEPATH') OR exit('No direct script access allowed'\
);
 | Examples:    my-controller/index -> my_controller/index
 |      my-controller/my-method -> my_controller/my_method
 */
-$route['default_controller'] = 'pages/view';
 $route['404_override'] = '';
 $route['translate_uri_dashes'] = FALSE;
+
+$route['news/create'] = 'news/create';
+$route['news/(:any)'] = 'news/view/$1';
+$route['news'] = 'news';
 $route['(:any)'] = 'pages/view/$1';
+$route['default_controller'] = 'pages/view';
```

This is all we need to get started.

Security Warning

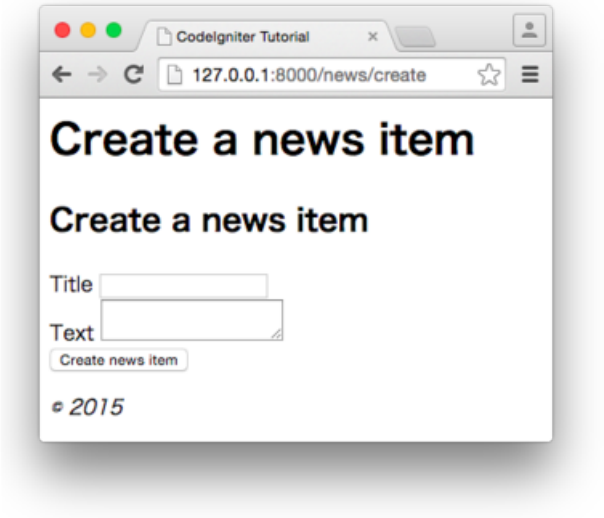
The application code is for tutorial purpose only. It is not secure code. For example, it contains XSS vulnerabilities and has no CSRF protection (unless you enable CSRF protection globally). Do not use this code in a production environment.

Manual Testing with a Web Browser

Let's access the web site. We will use PHP's built-in web server.

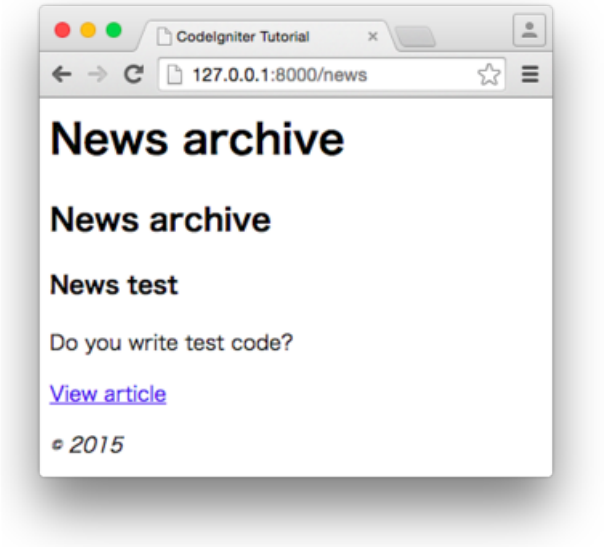
```
$ php -S 127.0.0.1:8000 index.php
```

Then, access http://127.0.0.1:8000/news/create in your web browser. You should see a form like the following:



Create a news item

Fill in and post the form. Then, access http://127.0.0.1:8000/news.



News archive

You can see the news item you just posted. Okay, our News Section works fine.

Database Fixtures

Since the model uses the database to store data, and the data changes if you insert a news item, we have to prepare Database Fixtures before we can write test code for the model.

For example, we start with empty table. When you insert a news item during test execution, the table has one record. You run the test twice, the table has two records. The state of the table changes every time when you run the tests.

If you write a test to check the total count of the news items, the test will fail the next time you run the tests.

Testing Tip: Repeatable

Good tests do not depend on execution order or time. If the tests pass, they should always pass.

So we have to fix the database state before running the tests. In other words, we have to set the database to a known state before we can run the tests.

To do so, we use *Database Migration* and *Database Seeding*. We created a migration file in the Database Migration section.

To run migrations automatically in testing, we modify our `TestCase` class.

application/tests/TestCase.php

```
1  <?php
2
3  class TestCase extends CIPHPUnitTestCase
4  {
5      private static $migrate = false;
6
7      public static function setUpBeforeClass()
8      {
9          parent::setUpBeforeClass();
10
11         // Run migrations once
12         if (! self::$migrate)
13         {
14             $CI =& get_instance();
15             $CI->load->database();
16             $CI->load->library('migration');
17             if ($CI->migration->current() === false) {
18                 throw new RuntimeException($this->migration->error_string());
19             }
```

```
20
21              self::$migrate = true;
22          }
23      }
24  }
```

The `setUpBeforeClass()` method runs migrations once during test execution, before the first test of the first test case class runs.

Database Seeding

Database Seeding is the initial seeding of a database with data. ci-phpunit-test has a database seeder library[4] and a sample seeder class[5].

First, install the seeder library. Copy `ci-phpunit-test/application/libraries/Seeder.php` to your `application/libraries` directory.

Then, create the `application/database/seeds` directory to hold seeder classes, then create a seeder class for the `news` table.

```
CodeIgniter/
└── application/
    ├── database/
    │   └── seeds/
    │       └── NewsSeeder.php
    └── libraries/
        └── Seeder.php
```

application/database/seeds/NewsSeeder.php

```
1   <?php
2
3   class NewsSeeder extends Seeder
4   {
5       private $table = 'news';
6
7       public function run()
8       {
9           $this->db->truncate($this->table);
10          // Reset autoincrement sequence number in SQLite
```

[4] https://github.com/kenjis/ci-phpunit-test/blob/master/application/libraries/Seeder.php
[5] https://github.com/kenjis/ci-phpunit-test/blob/master/application/database/seeds/CategorySeeder.php

```
        if (
            $this->db->dbdriver === 'sqlite3'
            || $this->db->subdriver === 'sqlite'
        ) {
            $this->db->query(
                "DELETE FROM sqlite_sequence WHERE name='$this->table';"
            );
        }

        $data = [
            'id'    => 1,
            'title' => 'News test',
            'slug'  => 'news-test',
            'text'  => 'News text',
        ];
        $this->db->insert($this->table, $data);
    }
}
```

If the seeder is called, it will truncate the `news` table and insert one record.

Testing Tip: Less is More

Insert as little data as possible. Include enough data to satisfy your tests, and no more, as the more data you insert, the longer it takes to run your tests.

Test Case for the News Model

Now we can write tests for the model. Create the test case class.

application/tests/models/News_model_test.php

```
<?php

class News_model_test extends TestCase
{

}
```

Fixtures

First, we need to call the database seeder. We'll add the `setUpBeforeClass()` method to the class.

application/tests/models/News_model_test.php

```
    public static function setUpBeforeClass()
    {
        parent::setUpBeforeClass();

        $CI =& get_instance();
        $CI->load->library('Seeder');
        $CI->seeder->call('NewsSeeder');
    }
```

`get_instance()` is a special function in CodeIgniter. It gets the *CodeIgniter super object* which has many CodeIgniter core objects like the CodeIgniter loader (you use as `$this->load`).

The `setUpBeforeClass()` method is called before the first test of the test case class is run. By seeding our database here, we can start our tests with a known database state.

Next, we add the `setUp()` method to create an instance to be tested.

application/tests/models/News_model_test.php

```
    public function setUp()
    {
        $this->resetInstance();
        $this->CI->load->model('news_model');
        $this->obj = $this->CI->news_model;
    }
```

The `$this->resetInstance()` method in ci-phpunit-test is a method that resets the *CodeIgniter super object*. It creates a new *CodeIgniter super object* and assigns it to `$this->CI`. Then we assign an instance of `News_model` to `$this->obj`.

We use CodeIgniter's loader to instantiate the `News_model`, because it depends on functionality supplied by CodeIgniter. If your model is a pure PHP object which does not extend `CI_Model` (or some child of `CI_Model`), you can setup the test for the model like this:

```
    public function setUp()
    {
        $this->obj = new News_model();
    }
```

Test Methods

Let's start writing test methods.

application/tests/models/News_model_test.php

```
    public function test_When_you_get_all_news_Then_you_get_one_item()
    {
        $result = $this->obj->get_news();
        $this->assertCount(1, $result);
    }
```

Our seeder inserts only one record into the table. So, when we get all news items, we get one item.

Run `phpunit` and make sure the test passes.

The next test is to create a news item. We call the `News_model::set_news()` method. Do you remember that the method depends on CodeIgniter's `$this->input->post()` method?

application/models/News_model.php

```
    public function set_news()
    {
        $this->load->helper('url');

        $slug = url_title($this->input->post('title'), 'dash', true);

        $data = [
            'title' => $this->input->post('title'),
            'slug'  => $slug,
            'text'  => $this->input->post('text')
        ];

        return $this->db->insert('news', $data);
    }
```

The `$this->input->post()` method returns `$_POST` data. We can change the values returned by the method by setting `$_POST` before calling it. So, we can write test code like the following:

application/tests/models/News_model_test.php

```
    public function test_When_you_set_news_item_Then_you_have_two_items()
    {
        $_POST = [
            'title' => 'CodeIgniter is awesome!',
            'text' =>
                'It is easy to understand, easy to write tests, and very fast.',
        ];

        $result = $this->obj->set_news();
        $this->assertTrue($result);
        $this->assertCount(2, $this->obj->get_news());
    }
```

Testing Tip: Avoid Global State

Code that depends on global state (like `$_POST` and other global variables) can be hard to test, because global state can be changed anywhere. Global state is easily forgotten, but the *backupGlobals* feature of PHPUnit restores global variables automatically in order to prevent changes to the global state from causing problems with other tests. See the PHPUnit Manual[6] for details.

Run `phpunit` and make sure the test passes.

Finally, we add a test to get the news item by slug.

application/tests/models/News_model_test.php

```
    public function test_When_you_get_news_by_slug_Then_you_get_the_item()
    {
        $item = $this->obj->get_news('codeigniter-is-awesome');
        $this->assertEquals('CodeIgniter is awesome!', $item['title']);
    }
```

Here is the completed test case class.

[6] https://phpunit.de/manual/4.8/en/fixtures.html#fixtures.global-state

application/tests/models/News_model_test.php

```php
<?php

/**
 * @group model
 * @group database
 */
class News_model_test extends TestCase
{
    public static function setUpBeforeClass()
    {
        parent::setUpBeforeClass();

        $CI =& get_instance();
        $CI->load->library('Seeder');
        $CI->seeder->call('NewsSeeder');
    }

    public function setUp()
    {
        $this->resetInstance();
        $this->CI->load->model('news_model');
        $this->obj = $this->CI->news_model;
    }

    public function test_When_you_get_all_news_Then_you_get_one_item()
    {
        $result = $this->obj->get_news();
        $this->assertCount(1, $result);
    }

    public function test_When_you_set_news_item_Then_you_have_two_items()
    {
        $_POST = [
            'title' => 'CodeIgniter is awesome!',
            'text' =>
                'It is easy to understand, easy to write tests, and very fast.',
        ];

        $result = $this->obj->set_news();
        $this->assertTrue($result);
        $this->assertCount(2, $this->obj->get_news());
```

```
42      }
43
44      public function test_When_you_get_news_by_slug_Then_you_get_the_item()
45      {
46          $item = $this->obj->get_news('codeigniter-is-awesome');
47          $this->assertEquals('CodeIgniter is awesome!', $item['title']);
48      }
49  }
```

Run `phpunit` and make sure the tests pass.

Checking Code Coverage

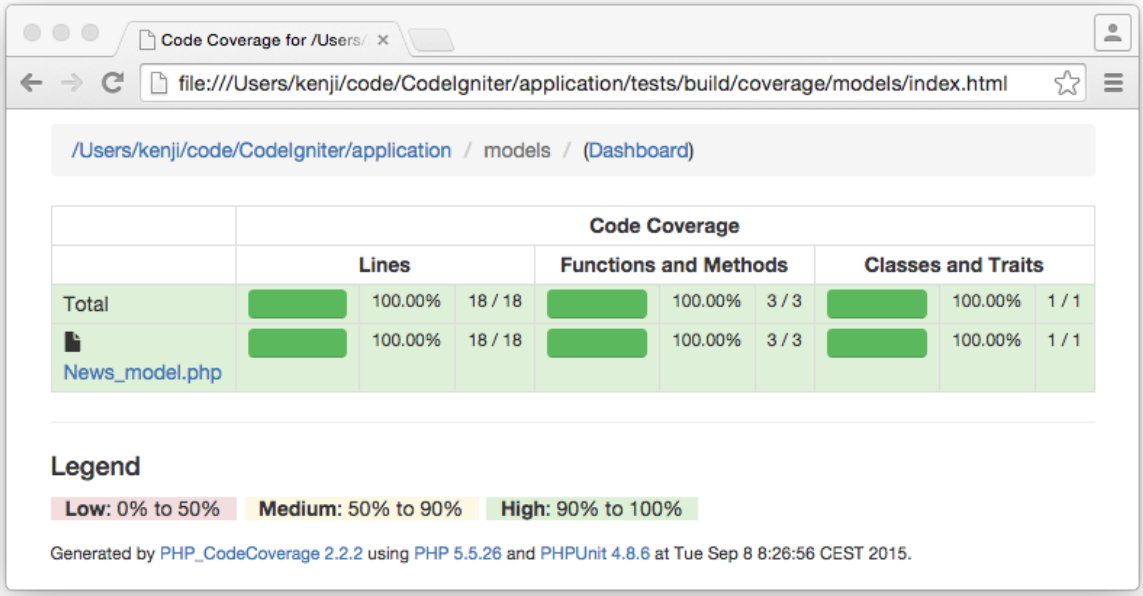
Before checking code coverage, we change our whitelist to exclude files like third party libraries, so we can focus on our application code.

```
--- a/CodeIgniter/application/tests/phpunit.xml
+++ b/CodeIgniter/application/tests/phpunit.xml
@@ -15,6 +15,11 @@
             <directory suffix=".php">../libraries</directory>
             <directory suffix=".php">../helpers</directory>
             <directory suffix=".php">../hooks</directory>
+            <exclude>
+                <directory suffix=".php">../views/errors</directory>
+                <file>../controllers/Dbfixture.php</file>
+                <file>../libraries/Seeder.php</file>
+            </exclude>
         </whitelist>
     </filter>
     <logging>
```

Files in `../views/errors` are CodeIgniter's error page templates. `../controllers/Dbfixture.php` is basically from CodeIgniter's User Guide, and `../libraries/Seeder.php` is from ci-phpunit-test.

```
$ php phpunit.phar
#!/usr/bin/env php
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

...................

Time: 2.98 seconds, Memory: 16.75Mb

OK (19 tests, 20 assertions)

Generating code coverage report in Clover XML format ... done

Generating code coverage report in HTML format ... done
```

Let's check the code coverage for the model. Open `application/tests/build/coverage/index.html` in your web browser, and follow the `models` link. You'll see the coverage for models like this:



Code Coverage for Models

We have 100% coverage.

6. Unit Testing for Models

In this chapter, we will learn how to unit test models using test doubles.

6.1 Why Should You Test Models First?

Models may be the most important part of your application code. The CodeIgniter User Guide says, "*Models are PHP classes that are designed to work with information in your database.*" While this is a good start, we will extend CodeIgniter's definition of *Models* to include classes which may include your *Domain Models* and/or *Business Logic*.

You may also choose to put some of this code in libraries. So, in the context of CodeIgniter, we should test models and libraries first.

In other words, put your most important code, like *Domain Models* or *Business Logic*, in models and libraries, then test them first (and well).

6.2 PHPUnit Mock Objects

Before writing test code for models, let's see what *PHPUnit Mock Objects* are like.

There are some well-known Mocking libraries in PHP. PHPUnit has its own Mocking library (*PHPUnit Mock Objects*), but *Prophecy*. *Phing*, *Mockery*, and *AspectMock* are also well-known. In this book, we only use PHPUnit Mock Objects.

`final`, `private` and `static` methods cannot be mocked by PHPUnit Mock Objects. They are ignored and retain their original behavior. If you really want to mock them, you can do so by using the Monkey Patching functionality provided by ci-phpunit-test.

PHPUnit has a great deal of functionality to create test doubles. If you want to dig into the details, see the PHPUnit Manual https://phpunit.de/manual/4.8/en/test-doubles.html.

Playing with Mocks

We will start by creating mock objects for the `Temperature_converter` class and testing with them.

application/libraries/Temperature_converter.php

```php
<?php

class Temperature_converter
{
    /**
      * Converts Celsius to Fahrenheit
      *
      * @param float $degree
      * @return float
      */
    public function CtoF($degree)
    {
        return round((9 / 5) * $degree + 32, 1);
    }

    /**
      * Converts Fahrenheit to Celsius
      *
      * @param float $degree
      * @return float
      */
    public function FtoC($degree)
    {
        return round((5 / 9) * ($degree - 32), 1);
    }
}
```

First, we create a test case class and write some basic code to create a mock object. Then, if you have installed PsySH, add `eval(\Psy\sh());`. If not, use the `var_dump()` line which has been commented out in the following code.

application/tests/models/Play_with_mocks_test.php

```php
<?php

class Play_with_mocks_test extends TestCase
{
    public function test_mocking_Temperature_converter()
    {
        $mock = $this->getMockBuilder('Temperature_converter')
            ->getMock();
```

```
9           $mock->method('FtoC')->willReturn(99.9);
10
11          eval(\Psy\sh());
12          //var_dump($mock, $mock->FtoC(100), $mock->CtoF(0)); exit;
13      }
14  }
```

`eval(\Psy\sh());` is a breakpoint. If PHP runs the code, *Psy Shell* opens in your terminal and you can debug the code in it.

Run the test case to see it.

```
$ php phpunit.phar models/Play_with_mocks_test.php
#!/usr/bin/env php
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

Psy Shell v0.6.0-dev (PHP 5.5.26 — cli) by Justin Hileman
```

At the breakpoint, *Psy Shell* waits for your input. Type `$mock` and press the [return] (or [enter]) key.

```
Psy Shell v0.6.0-dev (PHP 5.5.26 — cli) by Justin Hileman
$mock
=> Mock_Temperature_converter_7476f2a8 {#38}
```

We can see that `$mock` is an instance of the `Mock_Temperature_converter_7476f2a8` class. `Mock_Temperature_converter_7476f2a8` is a mock class that PHPUnit created on the fly.

We can check the source code by using the `show` command in PsySH.

```
show $mock
class Mock_Temperature_converter_7476f2a8 extends Temperature_converter implemen\
ts PHPUnit_Framework_MockObject_MockObject
Source code unavailable.
```

PsySH's `show` command tells us that `$mock` extends `Temperature_converter` and implements `PHPUnit_Framework_MockObject_MockObject`. Next, run the method `FtoC()`.

```
>>> $mock->FtoC(100)
=> 99.9
>>> $mock->FtoC(50)
=> 99.9
>>> $mock->FtoC(0)
=> 99.9
```

All of the return values are `99.9`, because this is the only value we told the mock to return.

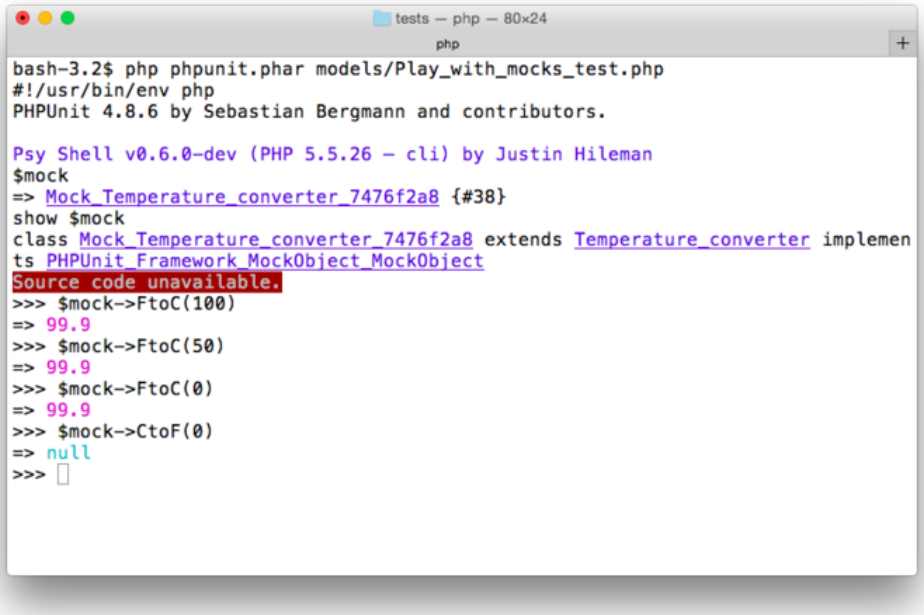
```
$mock = $this->getMockBuilder('Temperature_converter')
    ->getMock();
$mock->method('FtoC')->willReturn(99.9);
```

The last line in this code tells the `$mock` that it has a method named `FtoC` which will return the value `99.9`.

How about running another method?

```
>>> $mock->CtoF(0)
=> null
>>>
```

The methods we did not specify when we created the mock return `null`. This is the default behavior PHPUnit provides when we have not explicitly defined a method.



Psy Shell

Press the [control] key and the [c] key at the same time to terminate the PHPUnit process.

Partial Mocks

Next, how about another method of defining a mock? We will just add `setMethods(['FtoC'])` when we define this mock.

```
--- a/CodeIgniter/application/tests/models/Play_with_mocks_test.php
+++ b/CodeIgniter/application/tests/models/Play_with_mocks_test.php
@@ -8,6 +8,11 @@ class Play_with_mocks_test extends TestCase
             ->getMock();
         $mock->method('FtoC')->willReturn(99.9);

+       $mock2 = $this->getMockBuilder('Temperature_converter')
+           ->setMethods(['FtoC'])
+           ->getMock();
+       $mock2->method('FtoC')->willReturn(99.9);
+
         eval(\Psy\sh());
         //var_dump($mock, $mock->FtoC(100), $mock->CtoF(0)); exit;
     }
```

The listing above is in a format called *unified diff*. It shows the difference between two files. In short, add the green lines starting with +, to get the new file. See Conventions Used in This Book for details.

Run the `phpunit` command and check the results.

```
$ php phpunit.phar models/Play_with_mocks_test.php
#!/usr/bin/env php
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

Psy Shell v0.6.0-dev (PHP 5.5.26 — cli) by Justin Hileman
$mock2->FtoC(100)
=> 99.9
$mock2->FtoC(50)
=> 99.9
$mock2->FtoC(0)
=> 99.9
$mock2->CtoF(0)
=> 32.0
$mock2->CtoF(50)
=> 122.0
```

```
$mock2->CtoF(100)
=> 212.0
```

This time, the method we did not specify when we created the mock returns real values, not `null`. Mocks like this are called *Partial Mocks*. When we use `setMethods()`, we tell PHPUnit to mock the specific methods included in the array, but other methods will call the original class, and can not be configured using our `$mock2` object.

Verifying Expectations

We defined and looked at mock's *output* above. We can also check mock's *input*. In the code below, the mock expects that the `FtoC()` method is called once with the value `0` passed to the parameter.

application/tests/models/Play_with_mocks_test.php

```
    public function test_mock_expectations()
    {
        $mock = $this->getMockBuilder('Temperature_converter')
            ->getMock();
        $mock->expects($this->once())
            ->method('FtoC')
            ->with(0);

        $mock->FtoC(0);
    }
```

There are no assertions in the test method, but it is a real test. If you change or remove "`$mock->FtoC(0);`", the test will fail.

In addition to `$this->once()`, PHPUnit supplies other methods to verify the number of invocations:

- `$this->any()` ... any number of times, including 0
- `$this->once()` ... only once
- `$this->exactly($count)` ... exactly `$count` times
- `$this->at($index)` ... See example below
- `$this->never()` ... never
- `$this->atLeastOnce()` ... at least once

The following is an example of a passing test using the `$this->at()` method.

application/tests/models/Play_with_mocks_test.php

```
    public function test_mock_at_method()
    {
        $mock = $this->getMockBuilder('Temperature_converter')
            ->getMock();
        $mock->expects($this->at(0))
            ->method('FtoC')
            ->with(0);
        $mock->expects($this->at(1))
            ->method('CtoF')
            ->with(100);

        $mock->FtoC(0);
        $mock->CtoF(100);
    }
```

The argument for the `$this->at()` method refers to the zero-based index of all method invocations for a given mock object. So, `$mock->expects($this->at(0))->method('FtoC')->with(0);` configures the mock to expect the first method invocation after the mock is defined to be `FtoC(0)`, and `$mock->expects($this->at(1))->method('CtoF')->with(100);` configures the mock to expect the second method invocation to be `CtoF(100)`. If we changed the index passed to either call to `at()` or we changed the order in which the `FtoC()` and `CtoF()` methods were called, the tests would fail.

6.3 Testing Models without Database

In the previous chapter, we wrote test code for a model that uses the database (see Chapter 5, Test Case for News Model). In functional testing, we need the database to run the tests.

Database testing has some merits in that we can test real database interaction and confirm the stored data. It is intuitive and easy to understand what we are doing, but it takes time. The more fixture data we have to use, the more time it takes to write and run our tests.

We can write test code for models without the database, if we use test doubles.

Testing the get_news() Method with Mocks

Let's look at the `News_model` class again. To unit test it, we have to find all of the dependencies and replace them with mocks.

If class A uses, extends, or otherwise depends on class B, B is a dependency of A.

application/models/News_model.php

```
class News_model extends CI_Model
{
    public function __construct()
    {
        $this->load->database();
    }

    public function get_news($slug = false)
    {
        if ($slug === false) {
            $query = $this->db->get('news');
            return $query->result_array();
        }

        $query = $this->db->get_where('news', ['slug' => $slug]);
        return $query->row_array();
    }

    ...
}
```

The above code has three dependencies:

- `$this->load` (`CI_Loader` object)
- `$this->db` (`CI_DB` object)
- `$query` (`CI_DB_result` object)

`$this->db` and `$query` will be an instance of different classes based on your database configuration. Using our sqlite configuration, `$this->db` will likely be an instance of `CI_DB_pdo_sqlite_driver` and `$query` will likely be an instance of `CI_DB_pdo_result`. CodeIgniter's database library uses the `CI_DB` and `CI_DB_result` base classes to support drivers for multiple databases, so `CI_DB_pdo_sqlite_driver` extends `CI_DB_pdo_driver`, which extends `CI_DB` and `CI_DB_pdo_result` extends `CI_DB_result`. Similarly, the `CI_Loader` object could easily be an instance of `MY_Loader` or some other library which extends the CodeIgniter loader. With this in mind, our mocks target the base classes, rather than the more specific, configuration-dependent classes.

The `__construct()` method depends on `$this->load` (the `CI_Loader` object). As with most core CodeIgniter classes, it is loaded into a property of the core `CI_Controller` object.

The `get_news()` method depends on `$this->db` (the `CI_DB` object, or CodeIgniter's database library) and `$query` (the `CI_DB_result` object, also part of CodeIgniter's database library).

Creating a Mock Object for CI_Loader

First we create a mock object for `CI_Loader`. `$this->load->database()` loads the database classes and opens a connection to the database. We are going to write tests without the database, so we don't want to call the real method. We have to replace it with a mock object.

If we insist on strict unit testing, we have to mock all methods in `CI_Loader`. Since we can't load a lot of classes and other files without `CI_Loader`, we only mock the `database()` method to prevent it from connecting to the database. Yes, this is not strict unit testing, but we are avoiding extremes, so this is not a problem.

The test case class and `setUp()` method will look like this:

application/tests/models/News_model_with_mocks_test.php

```
1  class News_model_with_mocks_test extends TestCase
2  {
3      public function setUp()
4      {
5          // Reset CodeIgniter super object
6          $this->resetInstance();
7
8          // Create mock object for CI_Loader
9          $loader = $this->getMockBuilder('CI_Loader')
10             ->setMethods(['database'])
11             ->getMock();
12         $loader->method('database')->willReturn($loader);
13
14         // Inject mock object into CodeIgniter super object
15         $this->CI->load = $loader;
16
17         if (! class_exists('CI_DB', false)) {
18             // Define CI_DB class
19             eval('class CI_DB extends CI_DB_query_builder { }');
20         }
21
22         $this->obj = new News_model();
23     }
24 }
```

First, we reset the CodeIgniter super object to prevent the previous test state from affecting the next test. Next, we create a partial mock for `CI_Loader` and inject it into the CodeIgniter super object.

Next, we define the `CI_DB` class, if it does not already exist. We replaced the `$this->load->database()` method with a mock method which does nothing, so we have to load the expected

database classes. In fact, the ci-phpunit-test autoloader loads class files (so we don't have to define the `CI_DB_query_builder` class our `CI_DB` class is extending), but the `CI_DB` class is defined by CodeIgniter in a function (`DB()`), which we aren't going to call (since it would try to connect to the database).

We use CodeIgniter's *Query Builder* in this application, so we extended the `CI_DB_query_builder` class. If you don't use Query Builder, extend the `CI_DB_driver` class, instead.

If you find this a little too complicated, you can use the simplified code below for this particular exercise. The real database classes will be initialized and will connect to the database during initialization, but it is just connecting. Since we will not change any data in the database, the database state is the same, and it does not affect our model testing results. However, this will not be the case in future tests.

```
public function setUp()
{
    // Reset CodeIgniter super object
    $this->resetInstance();

    $this->obj = new News_model();
}
```

Creating Mocks for get_news()

Now, we will create mocks for the `get_news()` method.

```
    $result_array = [ ... ];

    // Create mock object for CI_DB_result
    $db_result = $this->getMockBuilder('CI_DB_result')
        ->disableOriginalConstructor()
        ->getMock();
    $db_result->method('result_array')->willReturn($result_array);

    // Create mock object for CI_DB
    $db = $this->getMockBuilder('CI_DB')
        ->disableOriginalConstructor()
        ->getMock();
    $db->expects($this->once())
        ->method('get')
        ->with('news')
        ->willReturn($db_result);
```

We start by defining the `$result_array` which will be returned by our mock for the `CI_DB_result`, then we build the `CI_DB_result` mock, because we will need it to be returned by the `CI_DB` mock.

We added a call to the `disableOriginalConstructor()` method when building the mocks. This disables the call to the original class' constructor, which we don't need to call for this test. If the original constructor is called, a warning occurs, telling us: "Missing argument 1 for CI_DB_result::__construct()".

The method `result_array()` in `$db_result` returns `$result_array`. This allows our mock `CI_DB_result` to be used in the same manner as our code expects to be able to use the object returned by the normal `CI_DB` object.

The mock `$db` expects to be called only once, with the value `'news'` passed to the `get()` method, and returns the mock `$db_result`.

```
$db->get('news');
```

Writing the Test Method

The test method will look like this.

application/tests/models/News_model_with_mocks_test.php

```
public function test_When_call_get_news_without_args_Then_get_all_items()
{
    $result_array = [
        [
          "id"    => "1",
          "title" => "News test",
          "slug"  => "news-test",
          "text"  => "News text",
        ],
        [
          "id"    => "2",
          "title" => "News test 2",
          "slug"  => "news-test-2",
          "text"  => "News text 2",
        ],
    ];

    // Create mock object for CI_DB_result
    $db_result = $this->getMockBuilder('CI_DB_result')
        ->disableOriginalConstructor()
        ->getMock();
    $db_result->method('result_array')->willReturn($result_array);
```

```
        // Create mock object for CI_DB
        $db = $this->getMockBuilder('CI_DB')
            ->disableOriginalConstructor()
            ->getMock();
        $db->expects($this->once())
            ->method('get')
            ->with('news')
            ->willReturn($db_result);

        // Inject mock object into the model
        $this->obj->db = $db;

        $result = $this->obj->get_news();
        $this->assertEquals($result_array, $result);
    }
```

To CodeIgniter users

In this case, ci-phpunit-test's autoloader loads the `News_model` class. So, you don't have to call CodeIgniter's `$this->load->model()` method in the test.

After creating the mocks, we inject the mocked `$db` object into the `db` property of the `News_model` object. This is called *Property Injection*, a form of *Dependency Injection*.

```
        // Inject mock object into the model
        $this->obj->db = $db;
```

Run the `phpunit` command and make sure the tests pass.

Creating Another Mocks

We will add another test method. This time, we will test calling the `get_news()` method with an argument. The code to create the necessary mocks will be like this:

```
        $slug = '...';
        $row_array = [ ... ];

        // Create mock object for CI_DB_result
        $db_result = $this->getMockBuilder('CI_DB_result')
            ->disableOriginalConstructor()
            ->getMock();
        $db_result->method('row_array')->willReturn($row_array);

        // Create mock object for CI_DB
        $db = $this->getMockBuilder('CI_DB')
            ->disableOriginalConstructor()
            ->getMock();
        $db->expects($this->once())
            ->method('get_where')
            ->with('news', ['slug' => $slug])
            ->willReturn($db_result);
```

In this case, the mock `$db` expects to be called only once and returns the mock `$db_result`.

```
    $db->get_where('news', ['slug' => $slug]);
```

Extract the Method to Create Mocks

The code to create mocks is almost the same as the previous code. So we will extract the method to make it easier to re-use.

```
    /**
     * Create Mock Object for CI_DB_result
     *
     * @param string $method method name to mock
     * @param array $return  the return value
     * @return Mock_CI_DB_result_xxxxxxxx
     */
    public function getMock_CI_DB_result($method, $return)
    {
        $db_result = $this->getMockBuilder('CI_DB_result')
            ->disableOriginalConstructor()
            ->getMock();
        $db_result->method($method)->willReturn($return);

        return $db_result;
```

```
    }

    /**
     * Create Mock Object for CI_DB
     *
     * @param string $method method name to mock
     * @param array $args    the arguments
     * @param array $return  the return value
     * @return Mock_CI_DB_xxxxxxxx
     */
    public function getMock_CI_DB($method, $args, $return)
    {
        $db = $this->getMockBuilder('CI_DB')
            ->disableOriginalConstructor()
            ->getMock();
        $mocker = $db->expects($this->once())
            ->method($method);

        switch (count($args)) {
            case 1:
                $mocker->with($args[0]);
                break;
            case 2:
                $mocker->with($args[0], $args[1]);
                break;
            case 3:
                $mocker->with($args[0], $args[1], $args[2]);
                break;
            case 4:
                $mocker->with($args[0], $args[1], $args[2], $args[3]);
                break;
            default:
                break;
        }

        $mocker->willReturn($return);

        return $db;
    }
```

Writing Another Test Method

The whole test method will be like this:

application/tests/models/News_model_with_mocks_test.php

```
    public function test_When_call_get_news_with_slug_Then_get_the_item()
    {
        $slug = 'news-test-2';
        $row_array = [
            "id"    => "2",
            "title" => "News test 2",
            "slug"  => "news-test-2",
            "text"  => "News text 2",
        ];

        // Create mock object for CI_DB_result
        $db_result = $this->getMock_CI_DB_result('row_array', $row_array);

        // Create mock object for CI_DB
        $db = $this->getMock_CI_DB(
            'get_where', ['news', ['slug' => $slug]] , $db_result
        );

        // Inject mock object into the model
        $this->obj->db = $db;

        $item = $this->obj->get_news($slug);
        $this->assertEquals($row_array, $item);
    }
```

Now, we can refactor the previous test method like this:

```
--- a/CodeIgniter/application/tests/models/News_model_with_mocks_test.php
+++ b/CodeIgniter/application/tests/models/News_model_with_mocks_test.php
@@ -28,19 +28,10 @@ class News_model_with_mocks_test extends TestCase
         ];

         // Create mock object for CI_DB_result
-        $db_result = $this->getMockBuilder('CI_DB_result')
-            ->disableOriginalConstructor()
-            ->getMock();
-        $db_result->method('result_array')->willReturn($result_array);
+        $db_result = $this->getMock_CI_DB_result('result_array', $result_array);

         // Create mock object for CI_DB
```

```
-        $db = $this->getMockBuilder('CI_DB')
-            ->disableOriginalConstructor()
-            ->getMock();
-        $db->expects($this->once())
-            ->method('get')
-            ->with('news')
-            ->willReturn($db_result);
+        $db = $this->getMock_CI_DB('get', ['news'], $db_result);

         // Inject mock object into the model
         $this->obj->db = $db;
```

The listing above is in a format called *unified diff*. It shows the difference between two files. In short, remove the red lines starting with - from the original file and add the green lines starting with +, to get the new file. See Conventions Used in This Book for details.

Run the `phpunit` command and make sure the tests pass.

Testing the set_news() Method with Mocks

Let's look at the `News_model::set_news()` method. To unit test it, we have to find all of the dependencies and replace them with mocks.

application/models/News_model.php

```
<?php

class News_model extends CI_Model
{
    ...

    public function set_news()
    {
        $this->load->helper('url');

        $slug = url_title($this->input->post('title'), 'dash', true);

        $data = [
            'title' => $this->input->post('title'),
            'slug'  => $slug,
            'text'  => $this->input->post('text')
        ];
```

```
        return $this->db->insert('news', $data);
    }
}
```

The `set_news()` method has four dependencies:

- `$this->load` (the `CI_Loader` object)
- the `url_title()` function
- `$this->input` (the `CI_Input` object)
- `$this->db` (the `CI_DB` object)

The `url_title()` function is a helper function defined in CodeIgniter's URL helper.

The `$this->load->helper()` method loads the URL helper file, which contains functions like `url_title()` to help manipulate URLs. `url_title()` takes a string as input and creates a human-friendly URL string. It is a simple function and has no side-effects. So we use it as-is and we don't have to mock it.

If you really want strict unit testing, you could mock the function using Monkey Patching, as provided by ci-phpunit-test.

Mocks with Return Map

In the `set_news()` method, `$this->input->post()` is called three times and the return values are assigned to `$data`. In this situation, we can use the `willReturnMap()` method.

```
        // Create mock object for CI_Input
        $input = $this->getMockBuilder('CI_Input')
            ->disableOriginalConstructor()
            ->getMock();
        // Can't use `$input->method()`, because CI_Input has method() method
        $input->expects($this->any())->method('post')
            ->willReturnMap(
                [
                    // post($index = NULL, $xss_clean = NULL)
                    ['title', null, 'News Title'],
                    ['text',  null, 'News Text'],
                ]
            );
```

The signature of the `$this->input->post()` method is "`($index = NULL, $xss_clean = NULL)`". When `$this->input->post('title')` is called, we want it to return `'News Title'`. When `$this->input->post('text')` is called, we want it to return `'News Text'`. So, we set the `willReturnMap()` method using the following array:

```
[
    ['title', null, 'News Title'],
    ['text',  null, 'News Text'],
]
```

Writing the Test Method

We will add the test method to create the mock, inject it and call the `set_news()` method.

application/tests/models/News_model_with_mocks_test.php

```
    public function test_When_post_data_Then_inserted_into_news_table_and_return\
_true()
    {
        // Create mock object for CI_Input
        $input = $this->getMockBuilder('CI_Input')
            ->disableOriginalConstructor()
            ->getMock();
        // Can't use `$input->method()`, because CI_Input has method() method
        $input->expects($this->any())->method('post')
            ->willReturnMap(
                [
                    // post($index = NULL, $xss_clean = NULL)
                    ['title', null, 'News Title'],
                    ['text',  null, 'News Text'],
                ]
            );

        // Inject mock objects into the model
        $this->obj->input = $input;

        $result = $this->obj->set_news();
    }
```

Let's confirm the `$data` values in the `set_news()` method. Add the line `eval(\Psy\sh());` in `News_model.php`. If you have not installed PsySH, use "`var_dump($data); exit;`" instead.

```
--- a/CodeIgniter/application/models/News_model.php
+++ b/CodeIgniter/application/models/News_model.php
@@ -38,6 +38,7 @@ class News_model extends CI_Model
             'slug'  => $slug,
             'text'  => $this->input->post('text')
         ];
+        eval(\Psy\sh());

 # $this->db->insert() method runs insertion query.
         return $this->db->insert('news', $data);
```

Run `phpunit`, then Psy Shell waits for your input. Type `$data` and press the [return] (or [enter]) key.

```
$ php phpunit.phar models/News_model_with_mocks_test.php
#!/usr/bin/env php
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

..Psy Shell v0.6.0-dev (PHP 5.5.26 — cli) by Justin Hileman
$data
=> [
     "title" => "News Title",
     "slug" => "news-title",
     "text" => "News Text",
   ]
```

Okay, after verifying that everything is as expected, remove the `eval(\Psy\sh());` line.

The whole test method will be like this:

application/tests/models/News_model_with_mocks_test.php

```
    public function test_When_post_data_Then_inserted_into_news_table_and_return\
_true()
    {
        // Create mock object for CI_Input
        $input = $this->getMockBuilder('CI_Input')
            ->disableOriginalConstructor()
            ->getMock();
        // Can't use `$input->method()`, because CI_Input has method() method
        $input->expects($this->any())->method('post')
            ->willReturnMap(
                [
                    // post($index = NULL, $xss_clean = NULL)
```

```
                ['title', null, 'News Title'],
                ['text',  null, 'News Text'],
            ]
        );

        // Create mock object for CI_DB
        $db = $this->getMock_CI_DB(
            'insert',
            [
                'news',
                [
                    "title" => "News Title",
                    "slug"  => "news-title",
                    "text"  => "News Text",
                ]
            ],
            true
        );

        // Inject mock objects into the model
        $this->obj->input = $input;
        $this->obj->db = $db;

        $result = $this->obj->set_news();
        $this->assertTrue($result);
    }
```

Run the `phpunit` command and make sure the tests pass. Check the code coverage report, as well.

6.4 With the Database or Without the Database?

We wrote the test case class with a Database Fixture in Chapter 5, Test Case for News Model. We wrote the test case class with Mocks in the previous section.

Which is better? In this case, the test case with the database fixture is simpler and easier to understand. In fact, it seems a bit faster than the test case with mocks on my computer. You will probably prefer testing with the database.

Does it mean you should always write tests with the database? The answer is not so simple.

Testing with Little Value

If we have two classes, `Foo` and `Bar`, like below:

application/libraries/Foo.php

```
<?php

class Foo
{
    private $bar;

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

    public function bar($arg)
    {
        return $this->bar->baz($arg);
    }
}
```

The class `Foo` depends on the class `Bar`.

application/libraries/Bar.php

```
<?php

class Bar
{
    public function baz($arg)
    {
        // Very complex logic
        // ...

        return $return;
    }
}
```

We can write a test case for the `Foo` class using a mock like this:

application/tests/libraries/Foo_test.php

```php
<?php

/**
 * @group library
 */
class Foo_test extends TestCase
{
    public function test_bar()
    {
        $mock = $this->getMockBuilder('Bar')
            ->getMock();
        $mock->method('baz')->willReturn('mocked value');

        // Inject mock object
        $this->obj = new Foo($mock);

        $actual = $this->obj->bar('test');
        $this->assertEquals('mocked value', $actual);
    }
}
```

Is this test useful? What exactly do we test in the test case?

It seems the `$this->assertEquals()` line is almost nonsense, because the return value is from the mock we injected. The only thing we can say from the test is that the method `Foo::bar()` calls the `Bar::baz()` method and returns the result. So our mock is the same as the internal implementation of the `Foo` class.

Why is this test of little value? The `Foo` class has no logic of its own. It is just returning the return value of the `Bar` object's `baz()` method.

Similarly, if our models are simply returning the results straight from the database, unit testing that code using mocked database objects is of little value. In that case, it is better to write database tests.

When You Write Tests without the Database

On the contrary, if our models have a lot of logic to do something like manipulating data from the database, it would be better to write unit testing without the database. It also might be better to think about separating database access from our model classes.

For example, if we add a `$this->news_repository` object to access the database, our model will be like the following, and it will be easier to write tests without the database.

```
    public function get_news($slug = false)
    {
        if ($slug === false) {
            return $this->news_repository->get_all();
        }

        return $this->news_repository->get_one_by_slug($slug);
    }
```

Now, we don't have to create many mocks for database related objects, we just create a mock for the `$this->news_repository` object.

7. Testing Controllers

We will learn more about testing controllers in this and the next two chapters. In this chapter, we will write tests for the News controller for a quick review, then write tests with mock models. We will also write tests for authentication and redirection.

7.1 Why is Testing Controllers Difficult?

In a previous chapter, we wrote test code for a controller (See Chapter 5, Test Case for Page Controller). Those tests are simple, but that is because the controller is very simple.

The controller does not use any models, libraries, or databases. Normally controllers have many dependencies. We can write test code (functional tests), but if one of the dependencies changes, tests may break and fail. The more objects you test at a time, the more fragile your tests will be.

If controllers depend on the database, we have to prepare database fixtures, too. The more objects you test at a time, the more you have to prepare.

What if you try to unit test controllers? In that case, you have to mock a lot of dependencies, and it is not so easy.

Do you remember Our First Test in Chapter 4, which was very easy to write? That is because the class under test had no dependencies at all. In other words, the more dependencies your class has, the more difficult your tests will be to write and maintain.

7.2 Test Case for the News Controller

We did not write tests for the News Controller in Chapter 5. We have a goal to get 100% code coverage, so we will write the tests here. The controller depends on the database, so we need to prepare database fixtures.

Exercise

At this point, please stop and write the test case for the News controller.

application/tests/controllers/News_test.php

```php
<?php

/**
 * @group controller
 * @group database
 */
class News_test extends TestCase
{
    public static function setUpBeforeClass()
    {
        parent::setUpBeforeClass();

        $CI =& get_instance();
        $CI->load->library('Seeder');
        $CI->seeder->call('NewsSeeder');
    }

    public function test_When_access_news_Then_see_news_archive()
    {
        $output = $this->request('GET', '/news');
        $this->assertContains('<h1>News archive</h1>', $output);
        $this->assertContains('<h3>News test</h3>', $output);
    }

    public function test_When_access_news_with_not_existing_slug_Then_get_404()
    {
        $slug = 'not-existing-slug';
        $output = $this->request('GET', "/news/$slug");
        $this->assertResponseCode(404);
    }

    public function test_When_access_news_with_slug_Then_see_the_item()
    {
        $slug = 'news-test';
        $output = $this->request('GET', "/news/$slug");
        $this->assertContains('<h1>News test</h1>', $output);
    }

    public function test_When_post_valid_news_item_Then_see_successful_message()
    {
        $output = $this->request(
```

```
42              'POST',
43              '/news/create',
44              [
45                  'title' => 'CodeIgniter is easy to write tests',
46                  'text'  => 'You can write tests for controllers very easily!',
47              ]
48          );
49          $this->assertContains('<h2>Successfully created</h2>', $output);
50      }
51
52      public function test_When_access_news_Then_see_two_items()
53      {
54          $output = $this->request('GET', '/news');
55          $this->assertContains('<h3>News test</h3>', $output);
56          $this->assertContains(
57              '<h3>CodeIgniter is easy to write tests</h3>', $output
58          );
59      }
60
61      public function test_When_post_invalid_news_item_Then_see_error_messages()
62      {
63          $output = $this->request(
64              'POST',
65              '/news/create',
66              [
67                  'title' => '',
68                  'text'  => '',
69              ]
70          );
71          $this->assertContains('<p>The Title field is required.</p>', $output);
72          $this->assertContains('<p>The Text field is required.</p>', $output);
73      }
74  }
```

We can use the database fixtures (`setUpBeforeClass()` method) from the `News_model_test` class in Chapter 5. We also used the `$this->request()` and `$this->assertResponseCode()` methods in Chapter 5.

The new thing here is that we send (emulate) POST requests. We can set `$_POST` data using the third argument of the `$this->request('POST')` method.

application/tests/controllers/News_test.php

```
    public function test_When_post_valid_news_item_Then_see_successful_message()
    {
        $output = $this->request(
            'POST',
            '/news/create',
            [
                'title' => 'CodeIgniter is easy to write tests',
                'text'  => 'You can write tests for controllers very easily!',
            ]
        );
        $this->assertContains('<h2>Successfully created</h2>', $output);
    }
```

We are done. Run `phpunit` and verify that all of the tests pass. Check the code coverage percentage, as well.

7.3 Mocking Models

In the previous section, we wrote tests with the database. We performed some integrated testing with the controller, model, and views.

If you are not interested in testing the model, you can mock it. So you can write tests for the controller without the model. For example, when you haven't completed the models, you can still write tests for the controllers.

Here is the test case class.

application/tests/controllers/News_with_mocks_test.php

```
1  <?php
2
3  /**
4   * @group controller
5   */
6  class News_with_mocks_test extends TestCase
7  {
8      public function test_When_access_news_Then_see_news_archive()
9      {
10         $result_array = [
11             [
12               "id"    => "1",
```

```
13                "title" => "News test",
14                "slug"  => "news-test",
15                "text"  => "News text",
16            ],
17            [
18                "id"    => "2",
19                "title" => "News test 2",
20                "slug"  => "news-test-2",
21                "text"  => "News text 2",
22            ],
23        ];
24        $this->request->setCallable(
25            function ($CI) use ($result_array) {
26                $news_model = $this->getDouble(
27                    'News_model', ['get_news' => $result_array]
28                );
29                $CI->news_model = $news_model;
30            }
31        );
32
33        $output = $this->request('GET', '/news');
34        $this->assertContains('<h1>News archive</h1>', $output);
35        $this->assertContains('<h3>News test</h3>', $output);
36        $this->assertContains('<h3>News test 2</h3>', $output);
37    }
38 }
```

We use the `$this->request->setCallable()` and `$this->getDouble()` methods in ci-phpunit-test. The `$this->request->setCallable()` method sets a *callable* to run after the controller instantiation. The `$this->getDouble()` method is a helper method to create a PHPUnit partial mock.

```
$news_model = $this->getDouble(
    'News_model', ['get_news' => $result_array]
);
```

The above code is equivalent to the following:

```
$news_model = $this->getMockBuilder('News_model')
    ->disableOriginalConstructor()
    ->setMethods(['get_news'])
    ->getMock();
$news_model->method('get_news')
    ->willReturn($result_array);
```

If you need to call the original constructor, use `true` for the third argument:

```
$form_validation = $this->getDouble(
   'CI_Form_validation', ['run' => true], true
);
```

The following code sets a closure which creates a mock for the model and adds it to the *CodeIgniter super object*. In fact, the CodeIgniter super object is the controller object, and the `$this->request()` method passes it to the parameter of the closure. So, the `$this->news_model` property in the controller will be the mock object when the action method (`News::index()`) is called.

```
$this->request->setCallable(
    function ($CI) use ($result_array) {
        $news_model = $this->getDouble(
            'News_model', ['get_news' => $result_array]
        );
        $CI->news_model = $news_model;
    }
);
```

The `$this->request()` method:

1. Resets the CodeIgniter super object
2. Creates the controller instance (the CodeIgniter super object is a singleton reference in the base `CI_Controller`)
3. Calls the callable set by the `$this->request->setCallable()` method
4. Calls the requested method in the controller

Exercise

At this point, please write the rest of the tests for the News controller using the mock model.

application/tests/controllers/News_with_mocks_test.php

```
    public function test_When_access_news_with_not_existing_slug_Then_get_404()
    {
        $slug = 'not-existing-slug';
        $this->request->setCallable(
            function ($CI) {
                $news_model = $this->getDouble(
                    'News_model', ['get_news' => null]
                );
                $CI->news_model = $news_model;
            }
        );

        $output = $this->request('GET', "/news/$slug");
        $this->assertResponseCode(404);
    }

    public function test_When_access_news_with_slug_Then_see_the_item()
    {
        $slug = 'news-test-2';
        $row_array = [
            "id"    => "2",
            "title" => "News test 2",
            "slug"  => "news-test-2",
            "text"  => "News text 2",
        ];
        $this->request->setCallable(
            function ($CI) use ($row_array) {
                $news_model = $this->getDouble(
                    'News_model', ['get_news' => $row_array]
                );
                $CI->news_model = $news_model;
            }
        );

        $output = $this->request('GET', "/news/$slug");
        $this->assertContains('<h1>News test 2</h1>', $output);
    }

    public function test_When_post_valid_news_item_Then_see_successful_message()
    {
        $this->request->setCallable(
```

```
            function ($CI) {
                $news_model = $this->getDouble(
                    'News_model', ['set_news' => true]
                );
                $CI->news_model = $news_model;
            }
        );

        $output = $this->request(
            'POST',
            '/news/create',
            [
                'title' => 'CodeIgniter is easy to write tests',
                'text'  => 'You can write tests for controllers very easily!',
            ]
        );
        $this->assertContains('<h2>Successfully created</h2>', $output);
    }

    public function test_When_post_invalid_news_item_Then_see_error_messages()
    {
        $output = $this->request(
            'POST',
            '/news/create',
            [
                'title' => '',
                'text'  => '',
            ]
        );
        $this->assertContains('<p>The Title field is required.</p>', $output);
        $this->assertContains('<p>The Text field is required.</p>', $output);
    }
```

Run `phpunit` and verify that all of the tests pass. Check the code coverage percentage, as well.

7.4 Authentication and Redirection

We often create controllers which need to use authentication or redirect the user. We'll show you how to test controllers which use these features. Our examples will use Ion Auth for authentication.

Installing Ion Auth

Ion Auth is a simple and lightweight authentication library for CodeIgniter, licensed under the MIT License. See the License Agreement for Ion Auth.

I changed the coding style of the application source code a bit.

Download the latest version (in this book we use v2.6.0-343-g7d1f6af) from http://github.com/benedmunds/CodeIgniter-Ion-Auth/zipball/2 and unzip it. Copy the following files into your `CodeIgniter/application` directory from the corresponding directory in the Zip package, except for `application/migrations/001_install_ion_auth.php`, which we will place in the `CodeIgniter/application/database/migrations/` directory.

```
CodeIgniter/
└── application/
    ├── config/
    │   └── auth.php
    ├── controller/
    │   └── auth.php
    ├── database/
    │   └── migrations/
    │       └── 001_install_ion_auth.php
    ├── language/
    │   └── english/
    │       ├── auth_lang.php
    │       └── ion_auth_lang.php
    ├── libraries/
    │   ├── Bcrypt.php
    │   └── Ion_auth.php
    ├── models/
    │   └── ion_auth_model.php
    └── views/
        └── auth/
```

Rename the class files to make them work with CodeIgniter 3.0:

```
application/controllers/auth.php       -> Auth.php
application/models/ion_auth_model.php -> Ion_auth_model.php
```

Rename the migration file to use the timestamp format.

```
application/database/migrations/001_install_ion_auth.php
                              -> 20150920182600_install_ion_auth.php
```

Database Migrations

Update the migration config file.

```
--- a/CodeIgniter/application/config/migration.php
+++ b/CodeIgniter/application/config/migration.php
@@ -69,7 +69,7 @@ $config['migration_auto_latest'] = FALSE;
 | be upgraded / downgraded to.
 |
 */
-$config['migration_version'] = 20150907090030;
+$config['migration_version'] = 20150920182600;

 /*
 |--------------------------------------------------------------------------
```

The listing above is in a format called *unified diff*. It shows the difference between two files. In short, remove the red line starting with - from the original file and add the green line starting with +, to get the new file. See Conventions Used in This Book for details.

Run the migrations via CLI.

```
$ php index.php dbfixture migrate
```

If there is no output, the migration ran and the `groups`, `users`, `users_groups`, and `login_attempts` tables were created successfully. You can confirm this using the `sqlite3` command.

```
$ sqlite3 application/data/sqlite-database.db
SQLite version 3.8.5 2014-08-15 22:37:57
Enter ".help" for usage hints.
sqlite> .schema groups
CREATE TABLE "groups" (
        "id" INTEGER PRIMARY KEY AUTOINCREMENT,
        "name" VARCHAR NOT NULL,
        "description" VARCHAR NOT NULL
);
sqlite> .schema users
```

```
CREATE TABLE "users" (
        "id" INTEGER PRIMARY KEY AUTOINCREMENT,
        "ip_address" VARCHAR NOT NULL,
        "username" VARCHAR NOT NULL,
        "password" VARCHAR NOT NULL,
        "salt" VARCHAR NOT NULL,
        "email" VARCHAR NOT NULL,
        "activation_code" VARCHAR NULL,
        "forgotten_password_code" VARCHAR NULL,
        "forgotten_password_time" INT NULL,
        "remember_code" VARCHAR NULL,
        "created_on" INT NOT NULL,
        "last_login" INT NULL,
        "active" TINYINT NULL,
        "first_name" VARCHAR NULL,
        "last_name" VARCHAR NULL,
        "company" VARCHAR NULL,
        "phone" VARCHAR NULL
);
sqlite> .schema users_groups
CREATE TABLE "users_groups" (
        "id" INTEGER PRIMARY KEY AUTOINCREMENT,
        "user_id" MEDIUMINT NOT NULL,
        "group_id" MEDIUMINT NOT NULL
);
sqlite> .schema login_attempts
CREATE TABLE "login_attempts" (
        "id" INTEGER PRIMARY KEY AUTOINCREMENT,
        "ip_address" VARCHAR NOT NULL,
        "login" VARCHAR NOT NULL,
        "time" INT NULL
);
sqlite> .exit
```

You can see all of the tables in the database.

Routing

Add a route for `auth`.

```
--- a/CodeIgniter/application/config/routes.php
+++ b/CodeIgniter/application/config/routes.php
@@ -52,6 +52,8 @@ defined('BASEPATH') OR exit('No direct script access allowed');
 $route['404_override'] = '';
 $route['translate_uri_dashes'] = FALSE;

+$route['auth'] = 'auth';
+
 $route['news/create'] = 'news/create';
 $route['news/(:any)'] = 'news/view/$1';
 $route['news'] = 'news';
```

To CodeIgniter users

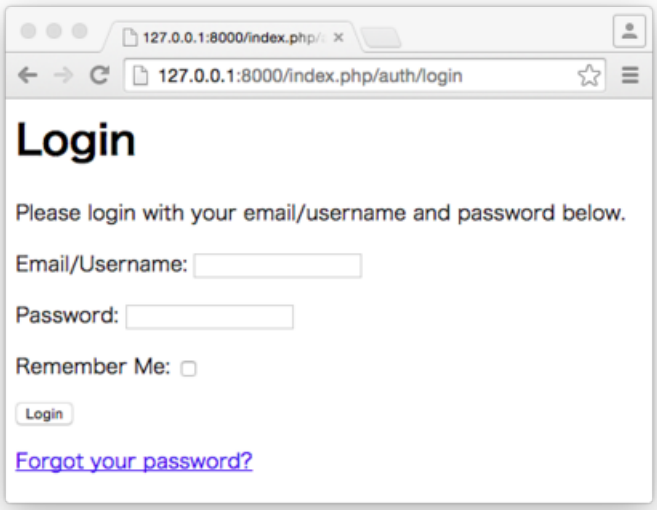
We need to add the route for `auth`, because we have the route "`$route['(:any)'] = 'pages/view/$1';`" after the above code, which prevents the `auth` route from being resolved automatically by CodeIgniter.

Manual Testing with a Web Browser

Let's access the login page using PHP's built-in web server.

```
$ php -S 127.0.0.1:8000 index.php
```

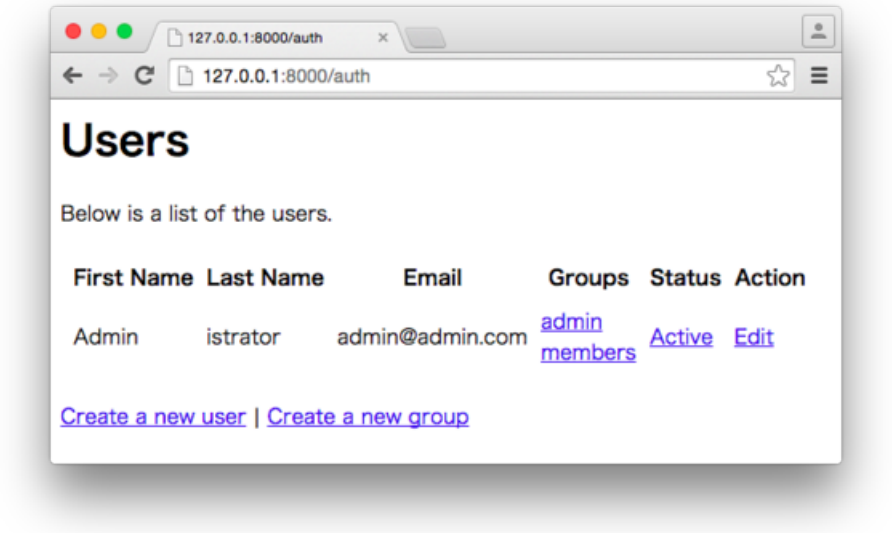
Then, access http://127.0.0.1:8000/auth in your web browser. You should be directed to `auth/login` and see the form below.



Login form

Try to login using the default user. Type `admin@admin.com` in the "Email/Username" field and `password` in the "Password" field, then submit the form using the "Login" button. If you logged in successfully, you should be redirected to the Home page.

Then, access http://127.0.0.1:8000/auth again. You should see the user list:



User list

Testing Redirection

Let's look at an example controller included in Ion Auth.

application/controllers/Auth.php

```php
<?php defined('BASEPATH') or exit('No direct script access allowed');

class Auth extends CI_Controller
{
    public function __construct()
    {
        parent::__construct();
        $this->load->database();
        $this->load->library(['ion_auth', 'form_validation']);
        $this->load->helper(['url', 'language']);

        $this->form_validation->set_error_delimiters(
            $this->config->item('error_start_delimiter', 'ion_auth'),
            $this->config->item('error_end_delimiter', 'ion_auth')
        );

        $this->lang->load('auth');
    }

    // redirect if needed, otherwise display the user list
    public function index()
    {
        if (! $this->ion_auth->logged_in()) {
            // redirect them to the login page
            redirect('auth/login', 'refresh');
        }
        // remove this elseif if you want to enable this for non-admins
        elseif (! $this->ion_auth->is_admin()) {
            // redirect them to the home page because they must be an
            // administrator to view this
            return show_error('You must be an administrator to view this page.');
        } else {
            // set the flash data error message if there is one
            $this->data['message'] =
                (validation_errors()) ? validation_errors()
                : $this->session->flashdata('message');

            //list the users
            $this->data['users'] = $this->ion_auth->users()->result();
            foreach ($this->data['users'] as $k => $user) {
                $this->data['users'][$k]->groups =
```

```
                    $this->ion_auth->get_users_groups($user->id)->result();
            }

            $this->_render_page('auth/index', $this->data);
        }
    }

    ...
}
```

Testing redirection is very simple, because ci-phpunit-test provides a special `redirect()` function by default. You can use the `$this->assertRedirect()` method in ci-phpunit-test with the argument set to a URI string which will be used as the target of the redirect. So, the test case will be like the following:

application/tests/controllers/Auth_test.php

```
<?php

class Auth_test extends TestCase
{
    public function test_When_not_login_user_get_auth_Then_redirected()
    {
        $this->request('GET', 'auth');
        $this->assertRedirect('auth/login');
    }
}
```

Run the `phpunit` command and check that the test passes.

Mocking Auth Objects

If you want to test the output seen by authenticated users, mocking auth objects is easy. You can use the `$this->request->setCallable()` method in ci-phpunit-test to inject mocks.

We add the test method to the bottom of the test case class.

application/tests/controllers/Auth_test.php

```
    public function test_When_admin_user_get_auth_Then_returns_user_list()
    {
        $this->request->setCallable(
            function ($CI) {
                $auth = $this->getDouble(
                    'Ion_auth', ['logged_in' => true, 'is_admin' => true]
                );
                $CI->ion_auth = $auth;
            }
        );

        $output = $this->request('GET', 'auth');
        $this->assertContains('<p>Below is a list of the users.</p>', $output);
    }
```

We used the `$this->getDouble()` helper method in ci-phpunit-test to create a partial mock object. The `logged_in()` and `is_admin()` methods of the object return `true`.

Run the `phpunit` command and check that the tests pass.

Exercise

At this point, please stop and write rest tests for the `Auth` controller.

7.5 What if My Controller Needs Something Else?

We have covered some of the common cases for testing controllers and we will learn more in the next two chapters.

- Chapter 8, Unit Testing CLI Controllers
- Chapter 9, Testing REST Controllers

In my opinion, we will have covered more than 90% of techniques to write test code for controllers by the time we finish Chapter 9. So, you will be able to write test code for almost anything.

There are still some remaining special cases for testing controllers, and you might see legacy code that is very hard to test. If you can't figure out how to write test code for a particular situation, I recommend you look over the documentation of ci-phpunit-test https://github.com/kenjis/ci-phpunit-test/blob/master/docs/HowToWriteTests.md.

Don't worry, if you can't write tests for some controllers, *Browser Testing*, which we will discuss in Chapter 10, could cover your controller testing, so you might be able to avoid writing some more difficult test code with PHPUnit.

As I said before, models and libraries are more important than controllers. So, we should put important code in them, rather than our controllers, and we should test them first and well.

8. Unit Testing CLI Controllers

In this chapter, we will continue learning to write tests for controllers, but we will focus on CLI controllers.

8.1 Dbfixture Controller

All controllers in CodeIgniter can be executed via CLI by default, so CodeIgniter controllers do not have to be written specifically as CLI controllers. You may create controllers, or specific methods in a controller, which you intend to run via CLI. For example, you may add a controller for your cron-jobs.

In Chapter 5, we created a controller intended to be run via CLI: the DBfixture controller.

Before writing tests, we will add two methods to the controller, `seed()` to seed the database, and `all()` to run both migrations and seeding. We will use these methods in a later chapter.

The completed controller looks like this:

application/controllers/Dbfixture.php

```
<?php

class Dbfixture extends CI_Controller
{
    public function __construct()
    {
        parent::__construct();

        // Only accessible via CLI
        if (is_cli() === false) {
            exit();
        }
    }

    public function migrate()
    {
        $this->load->library('migration');
        if ($this->migration->current() === false) {
            echo $this->migration->error_string() . PHP_EOL;
```

```
20          }
21      }
22
23      public function seed()
24      {
25          $this->load->library('seeder');
26          foreach (glob(APPPATH.'database/seeds/*Seeder.php') as $file) {
27              $seeder = basename($file, '.php');
28              $this->seeder->call($seeder);
29          }
30      }
31
32      public function all()
33      {
34          $this->migrate();
35          $this->seed();
36      }
37  }
```

This controller checks to ensure it is running from the CLI in the constructor. The `seed()` method executes all seeder files which have the suffix `Seeder.php` in the `application/database/seeder` directory. The `all()` method runs both migrations and seeders.

The controller is simple but has a few interesting points. Let's write unit tests for it.

8.2 Faking is_cli()

The `is_cli()` function in CodeIgniter returns `true` if the application is running through the command line. We run tests using the `phpunit` command, so the function always returns `true` during test execution.

To test the situation in which `is_cli()` returns `false`, we have to replace the function with a test double. Fortunately, ci-phpunit-test provides a special `is_cli()` function by default. You can set the return value by calling the `set_is_cli()` function in ci-phpunit-test.

Our test case class begins with the following code:

application/tests/controllers/Dbfixture_test.php

```php
<?php

/**
 * @group controller
 */
class Dbfixture_test extends TestCase
{
    public function setUp()
    {
        $this->resetInstance();
        $this->obj = new Dbfixture();
    }

    public function tearDown()
    {
        parent::tearDown();
        set_is_cli(true);
    }

    public function test_When_not_running_under_CLI_Then_exit()
    {
        set_is_cli(false);
        $obj = new Dbfixture();
    }
}
```

We added the `tearDown()` method to make sure we don't forget to reset the return value of the `is_cli()` function to `true` after the test.

8.3 Testing exit()

By the way, there is a call to `exit()` in the controller code. Can we test the code like that? Let's try it.

```
$ php phpunit.phar controllers/Dbfixture_test.php
#!/usr/bin/env php
PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

E

Time: 1.35 seconds, Memory: 15.50Mb

There was 1 error:

1) Dbfixture_test::test_When_not_running_under_CLI_Then_exit
CIPHPUnitTestExitException: exit() called in Dbfixture::__construct()

...
```

We have an error, because the *Monkey Patching* feature in ci-phpunit-test converts `exit()` or `die()` to throw an exception (`CIPHPUnitTestExitException`).

Monkey Patching allows us to replace code on the fly without modifying the original code. The Monkey Patching provided by ci-phpunit-test can patch `exit()` or `die()`, as well as some functions and methods of user-defined classes.

We will learn more about monkey patching later in this chapter.

8.4 Testing Exceptions

We know that monkey patching will replace `exit()` with `CIPHPUnitTestExitException`, so we can catch the exception to test it.

PHPUnit allows us to use the annotations `@expectedException`, `@expectedExceptionMessage`, `@expectedExceptionMessageRegExp`, and `@expectedExceptionCode` to test exceptions. We can write a test method using `@expectedException`:

application/tests/controllers/Dbfixture_test.php

```
    /**
     * @expectedException CIPHPUnitTestExitException
     */
    public function test_When_not_running_under_CLI_Then_exit()
    {
        set_is_cli(false);
        new Dbfixture();
    }
```

Run the `phpunit` command and check that the test passes.

The following code demonstrates how we can use `@expectedExceptionMessage`:

```
    /**
     * @expectedException CIPHPUnitTestExitException
     * @expectedExceptionMessage exit() called in Dbfixture::__construct()
     */
```

Or we can write tests using `try` and `catch`. In this case, we could assert something after catching the exception.

```
    public function test_When_not_running_under_CLI_Then_exit()
    {
        set_is_cli(false);

        try {
            new Dbfixture();
        } catch (CIPHPUnitTestExitException $e) {
            $this->assertNull($e->exit_status);
        }
    }
```

The `exit_status` property of the `CIPHPUnitTestExitException` object contains the argument of the call to `exit()`. In this case, it is `null`.

8.5 Testing Output

The `migrate()` method in the controller does not return (to be accurate, it returns `null`). It prints an error message via `echo`, if the `$this->migration->current()` method returns `false`. To test this behavior, we can use the `$this->expectOutputString()` method in PHPUnit.

application/tests/controllers/Dbfixture_test.php

```
    public function test_When_migration_error_Then_you_see_the_error()
    {
        // Create mock object for CI_Loader
        $loader = $this->getDouble(
            'CI_Loader',
            [
                'library' => null,
            ]
        );

        // Create mock object for CI_Migration
        $error_msg = 'Migration error';
        $migration = $this->getDouble(
            'CI_Migration',
            [
                'current' => false,
                'error_string' => $error_msg,
            ]
        );

        // Inject mock objects
        $this->obj->load      = $loader;
        $this->obj->migration = $migration;

        $this->expectOutputString($error_msg . PHP_EOL);
        $this->obj->migrate();
    }
```

We used the `$this->getDouble()` helper method in ci-phpunit-test to create a partial mock object. The `library()` method of the `CI_Loader` object will return `null`. The `current()` method of the `CI_Migration` object will return `false`, and the `error_string()` method will return `'Migration error'`.

Since the `migrate()` method in the controller outputs `'Migration error'`, we can check the string using the `$this->expectOutputString()` method.

PHPUnit also has a `$this->expectOutputRegex()` method, which allows you to set the expectation that the output matches a regular expression.

Run the `phpunit` command and ensure that the tests pass.

8.6 Monkey Patching

Monkey Patching allows us to replace code on the fly without modifying the original code. The monkey patching provided by ci-phpunit-test can patch `exit()` or `die()`, as well as some global functions and methods of user-defined classes.

We have already used it to patch `exit()` in the previous section. Without monkey patching, we can't usually test code which contains `exit()`.

Monkey patching is a very powerful tool for testing, so use it with caution.

Testing Tip: Untestable without Monkey Patching

If you can't write tests without monkey patching, it may be a sign that your application code should be improved.

It would probably be better not to use monkey patching if you don't have to do so. It has a negative impact on the speed of testing. You will probably need it when you come across otherwise untestable code which you can't or don't want to modify, like third party libraries.

Patching Functions

Let's look at the `seed()` method of the controller. To write unit tests, we have to find all of the dependencies and replace them with mocks.

application/controllers/Dbfixture.php

```
    public function seed()
    {
        $this->load->library('seeder');
        foreach (glob(APPPATH.'database/seeds/*Seeder.php') as $file) {
            $seeder = basename($file, '.php');
            $this->seeder->call($seeder);
        }
    }
```

The above code has five dependencies:

- `$this->load` (`CI_Loader` object)
- `glob()` function
- `APPPATH` constant
- `basename()` function

- `$this->seeder` (`Seeder` object)

The `glob()` function uses the hard-coded argument `APPPATH.'database/seeds/*Seeder.php'`.

`basename()`[1] is a simple function and has no side-effects, so we use it as is. We don't have to mock it.

`glob()`[2] depends on the file system and has a hard-coded argument. We should mock it.

We can create mock objects, but how do we mock a global function? In this case, we can use Monkey Patching in ci-phpunit-test, but it has a few limitations. Some functions may cause errors if you attempt to replace them, see the documentation[3] for details.

By default, only some global functions, like `mt_rand()`, `uniqid()`, `md5()`, `time()`, and `date()`, can be patched. To patch the `glob()` function, we add it to the array in the monkey patching configuration in `application/tests/Bootstrap.php`.

```
--- a/CodeIgniter/application/tests/Bootstrap.php
+++ b/CodeIgniter/application/tests/Bootstrap.php
@@ -316,6 +316,7 @@ MonkeyPatchManager::init([
    // Additional functions to patch
    'functions_to_patch' => [
        //'random_string',
+       'glob',
    ],
    'exit_exception_classname' => 'CIPHPUnitTestExitException',
 ]);
```

The listing above is in a format called *unified diff*. It shows the difference between two files. In short, add the green line starting with + to the original file, to get the new file. See Conventions Used in This Book for details.

Now we can patch the `glob()` function with the `MonkeyPatch::patchFunction()` method. The first argument is the name of the function to patch. The second argument is the return value. The third argument is the class and method name in which to apply this patch.

[1] http://php.net/en/basename
[2] http://php.net/en/glob
[3] https://github.com/kenjis/ci-phpunit-test/blob/master/docs/HowToWriteTests.md#patching-functions

```
MonkeyPatch::patchFunction('glob', [
    '/application/database/seeds/NewsSeeder.php',
    '/application/database/seeds/BookSeeder.php',
    '/application/database/seeds/ShopSeeder.php',
], 'Dbfixture::seed');
```

The `glob` function will be patched only in the `Dbfixture::seed()` method, and it will return the array passed to the `MonkeyPatch::patchFunction()` method's second argument.

If you call `MonkeyPatch::patchFunction()` without the third argument, all of the functions (located in `include_paths` and not in `exclude_paths` in the monkey patching configuration in the `application/tests/Bootstrap.php` file) called during the test method execution will be replaced.

Using `MonkeyPatch::patchFunction()`, we can write a test method like the following:

application/tests/controllers/Dbfixture_test.php

```
1   public function test_When_run_seed_Then_seeders_are_called()
2   {
3       // Patch on glob() function
4       MonkeyPatch::patchFunction('glob', [
5           '/application/database/seeds/NewsSeeder.php',
6           '/application/database/seeds/BookSeeder.php',
7           '/application/database/seeds/ShopSeeder.php',
8       ], 'Dbfixture::seed');
9
10      // Create mock object for CI_Loader
11      $loader = $this->getDouble(
12          'CI_Loader',
13          [
14              'library' => null,
15          ]
16      );
17
18      // Create mock object for Seeder
19      $seeder = $this->getDouble('Seeder', ['call' => null]);
20      $this->verifyInvokedMultipleTimes($seeder, 'call', 3, [
21          ['NewsSeeder'],
22          ['BookSeeder'],
23          ['ShopSeeder'],
24      ]);
```

```
25
26          // Inject mock objects
27          $this->obj->load   = $loader;
28          $this->obj->seeder = $seeder;
29
30          $this->obj->seed();
31      }
```

The `$this->verifyInvokedMultipleTimes()` method is a helper method for PHPUnit Mock Objects in ci-phpunit-test.

```
$this->verifyInvokedMultipleTimes($seeder, 'call', 3, [
    ['NewsSeeder'],
    ['BookSeeder'],
    ['ShopSeeder'],
]);
```

The above code is equivalent to the following:

```
$seeder->expects($this->exactly(3))
    ->method('call')
    ->withConsecutive(
        ['NewsSeeder'],
        ['BookSeeder'],
        ['ShopSeeder'],
    );
```

The `withConsecutive()` method can take any number of arrays as arguments, depending on the calls you want to test against. Each array is a list of constraints corresponding to the arguments of the method being mocked, as when calling `with()`.

Run the `phpunit` command and ensure the tests pass.

Patching Class Methods

You can patch user-defined class methods with `MonkeyPatch::patchMethod()`. The first argument is the class name. The second argument is an array containing the method name and its return value:

```
        MonkeyPatch::patchMethod('Dbfixture', [
            'migrate' => null,
            'seed'    => null,
        ]);
```

With the code above, the `migrate()` and `seed()` methods of the `Dbfixture` class will return `null`.

We can write test code for the `all()` method of the `Dbfixture` class like this:

application/tests/controllers/Dbfixture_test.php

```
1     public function test_When_run_all_Then_migrate_and_seed_are_called()
2     {
3         MonkeyPatch::patchMethod('Dbfixture', [
4             'migrate' => null,
5             'seed'    => null,
6         ]);
7         MonkeyPatch::verifyInvokedOnce('Dbfixture::migrate');
8         MonkeyPatch::verifyInvokedOnce('Dbfixture::seed');
9
10        $this->obj->all();
11    }
```

The `MonkeyPatch::verifyInvokedOnce()` method verifies that a patched class method or a patched function was invoked only once.

In addition to `verifyInvokedOnce()`, there are other methods to verify the number of times a monkey-patched method or function has been invoked:

- `MonkeyPatch::verifyInvoked($class_method, $params)` ... at least once
- `MonkeyPatch::verifyInvokedOnce($class_method, $params)` ... only once
- `MonkeyPatch::verifyInvokedMultipleTimes($class_method, $count, $params)` ... exactly *count* times
- `MonkeyPatch::verifyNeverInvoked($class_method, $params)` ... never

Run the `phpunit` command and verify that the tests pass.

8.7 Checking Code Coverage

Before checking code coverage, remove the `controllers/Dbfixture.php` line in `phpunit.xml`:

```
--- a/CodeIgniter/application/tests/phpunit.xml
+++ b/CodeIgniter/application/tests/phpunit.xml
@@ -17,7 +17,6 @@
            <directory suffix=".php">../hooks</directory>
            <exclude>
                <directory suffix=".php">../views/errors</directory>
-               <file>../controllers/Dbfixture.php</file>
                <file>../libraries/Seeder.php</file>
            </exclude>
        </whitelist>
```

The listing above is in a format called *unified diff*. It shows the difference between two files. In short, remove the red line starting with - from the original file, to get the new file. See Conventions Used in This Book for details.

Run `phpunit` and verify that all tests pass. Check the code coverage percentage, as well.

9. Testing REST Controllers

You can also test your REST controllers easily. The examples in this chapter will use CodeIgniter Rest Server.

9.1 Installing CodeIgniter Rest Server

CodeIgniter Rest Server is a fully RESTful server implementation for CodeIgniter.

Download the latest version (in this book we use v2.7.2) from https://github.com/chriskacerguis/codeigniter-restserver/releases and unzip it. Copy the following files into your `CodeIgniter/application` directory from the corresponding directory in the Zip package.

```
CodeIgniter/
└── application/
    ├── config/
    │   └── rest.php
    ├── controller/
    │   └── api/
    │       └── Example.php
    └── libraries/
        ├── Format.php
        └── REST_Controller.php
```

CodeIgniter Rest Server is licensed under the MIT License. See the License Agreement for CodeIgniter Rest Server.

Fixing the CodeIgniter Rest Server Code

Up to the present we have never modified application code to write tests, because we don't normally have to do so. However, we have to modify CodeIgniter Rest Server in order to be able to write tests.

Testing Tip: Untestable Code

If you can't write tests or it is hard to write tests, it is a sign that your application code design should be improved.

First, replace `require` with `require_once` in the `Example` controller. If there is more than one REST controller, we load them at the same time during test execution, which will result in "Fatal error: Cannot redeclare class REST_Controller" if `require` is used.

```
--- a/CodeIgniter/application/controllers/api/Example.php
+++ b/CodeIgniter/application/controllers/api/Example.php
@@ -3,7 +3,7 @@
 defined('BASEPATH') OR exit('No direct script access allowed');

 // This can be removed if you use __autoload() in config.php OR use Modular Ext\
ensions
-require APPPATH . '/libraries/REST_Controller.php';
+require_once APPPATH . '/libraries/REST_Controller.php';

 /**
  * This is an example of a few basic user interaction methods you could use
```

The listing above is in a format called *unified diff*. It shows the difference between two files. In short, remove the red line starting with - from the original file and add the green line starting with +, to get the new file. See Conventions Used in This Book for details.

Second, add the catch block below to `REST_Controller.php`. CodeIgniter Rest Server calls `exit()` and monkey patching provided by ci-phpunit-test converts it to an exception, but the Rest Server catches it by default. We want to catch the exception thrown by the monkey patch and re-throw it.

```
--- a/CodeIgniter/application/libraries/REST_Controller.php
+++ b/CodeIgniter/application/libraries/REST_Controller.php
@@ -653,6 +653,11 @@ abstract class REST_Controller extends CI_Controller {
         {
             call_user_func_array([$this, $controller_method], $arguments);
         }
+        catch (CIPHPUnitTestExitException $ex)
+        {
+            // This block is for ci-phpunit-test
+            throw $ex;
+        }
         catch (Exception $ex)
         {
             // If the method doesn't exist, then the error will be caught and a\
n error response shown
```

9.2 Testing GET Requests

Let's access the web API using the `curl` command.

curl is an open source command line tool and library for transferring data. It supports many protocols, including HTTP, HTTPS, FTP, IMAP, POP3, and SMTP.

Open your terminal, navigate to the directory in which CodeIgniter's `index.php` is located, and type the following command.

```
$ php -S 127.0.0.1:8000 index.php
```

Getting All of the Data

When we send a GET request to `api/example/users`, it returns all of the users' data in JSON format. Open another terminal, and type the following command. The `-X` or `--request` option specifies the HTTP method.

```
$ curl -X GET 'http://127.0.0.1:8000/api/example/users'
```

You should see the following JSON output:

```
[{"id":1,"name":"John","email":"john@example.com","fact":"Loves coding"},{"id":2\
,"name":"Jim","email":"jim@example.com","fact":"Developed on CodeIgniter"},{"id"\
:3,"name":"Jane","email":"jane@example.com","fact":"Lives in the USA","0":{"hobb\
ies":["guitar","cycling"]}}]
```

You can add the `-i` or `--include` option to see the HTTP response headers.

```
$ curl -X GET -i 'http://127.0.0.1:8000/api/example/users'
HTTP/1.1 200 OK
Host: 127.0.0.1:8000
Connection: close
X-Powered-By: PHP/5.5.26
Content-Type: application/json; charset=utf-8

[{"id":1,"name":"John","email":"john@example.com","fact":"Loves coding"},{"id":2\
,"name":"Jim","email":"jim@example.com","fact":"Developed on CodeIgniter"},{"id"\
:3,"name":"Jane","email":"jane@example.com","fact":"Lives in the USA","0":{"hobb\
ies":["guitar","cycling"]}}]
```

If you want to see verbose output, add the `-v` or `--verbose` option.

```
$ curl -X GET -v 'http://127.0.0.1:8000/api/example/users'
*   Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 8000 (#0)
> GET /api/example/users HTTP/1.1
> Host: 127.0.0.1:8000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Host: 127.0.0.1:8000
< Connection: close
< X-Powered-By: PHP/5.5.26
< Content-Type: application/json; charset=utf-8
<
* Closing connection 0
[{"id":1,"name":"John","email":"john@example.com","fact":"Loves coding"},{"id":2\
,"name":"Jim","email":"jim@example.com","fact":"Developed on CodeIgniter"},{"id"\
:3,"name":"Jane","email":"jane@example.com","fact":"Lives in the USA","0":{"hobb\
ies":["guitar","cycling"]}}]
```

Next, we write a test case class for it. The controller calls `exit()`, so we have to catch the `CIPHPUnitTestExitException` exception.

application/tests/controllers/api/Example_test.php

```php
<?php

/**
 * @group controller
 */
class Example_test extends TestCase
{
    public function test_When_get_users_Then_returns_all_user_data()
    {
        try {
            $this->request('GET', 'api/example/users');
        } catch (CIPHPUnitTestExitException $e) {
            $output = ob_get_clean();
        }

        $data = json_decode($output, true);
        $this->assertCount(3, $data);
        $this->assertResponseCode(200);
```

```
19      }
20  }
```

`$this->request('GET', 'api/example/users')` throws an exception, so we catch it. The `$this->request()` method has turned on output buffering to capture the output of the controller, so we get the buffer contents using PHP's `ob_get_clean()`[1] function in the `catch` block.

The value of `$output` is a string in JSON format. We can convert it to an array using the `json_decode()`[2] function in PHP. Then we assert that the output data has three elements, and the HTTP response code is 200 (OK).

Run the `phpunit` command, and verify that the test passes.

Getting One User's Data

If we specify a user ID, it returns the user's data in JSON format.

```
$ curl -X GET 'http://127.0.0.1:8000/api/example/users/id/1'
{"id":1,"name":"John","email":"john@example.com","fact":"Loves coding"}
```

We can write a test method like the following:

application/tests/controllers/api/Example_test.php

```
    public function test_When_get_valid_user_id_Then_returns_the_user_data()
    {
        $json =
            '{"id":1,"name":"John","email":"john@example.com",'
            . '"fact":"Loves coding"}';

        $output = $this->request('GET', 'api/example/users/id/1');
        $this->assertJsonStringEqualsJsonString($json, $output);
        $this->assertResponseCode(200);
    }
```

In this case, the controller does not call `exit()`, so we don't need the `try` and `catch` blocks. The `$this->assertJsonStringEqualsJsonString()` method in PHPUnit checks whether the value of the JSON string received from the request (the second argument) matches the expected JSON string (the first argument).

Run the `phpunit` command, and verify that the tests pass.

If there is no user with the provided ID, it returns an error message.

[1] http://php.net/en/ob_get_clean
[2] http://php.net/en/json_decode

```
$ curl -X GET -i 'http://127.0.0.1:8000/api/example/users/id/5'
HTTP/1.1 404 Not Found
Host: 127.0.0.1:8000
Connection: close
X-Powered-By: PHP/5.5.26
Content-Type: application/json; charset=utf-8

{"status":false,"error":"User could not be found"}
```

It returns the response code 404 (Not Found), instead of 200 (OK). We can write a test method to look for this condition:

application/tests/controllers/api/Example_test.php

```
    public function test_When_get_not_found_user_id_Then_returns_error()
    {
        $output = $this->request('GET', 'api/example/users/id/5');
        $data = json_decode($output, true);
        $this->assertFalse($data['status']);
        $this->assertResponseCode(404);
    }
```

Run the `phpunit` command, and verify that the tests pass.

Next, if you specify an invalid ID, like a negative number, it returns the response code 400 (Bad Request).

```
$ curl -X GET -i 'http://127.0.0.1:8000/api/example/users/id/-5'
HTTP/1.1 400 Bad Request
Host: 127.0.0.1:8000
Connection: close
X-Powered-By: PHP/5.5.26
Content-type: text/html; charset=UTF-8
```

We can write the following test method:

application/tests/controllers/api/Example_test.php

```
    public function test_When_get_invalid_user_id_Then_returns_bad_request()
    {
        try {
            $this->request('GET', 'api/example/users/id/-5');
        } catch (CIPHPUnitTestExitException $e) {
            $output = ob_get_clean();
        }
        $this->assertResponseCode(REST_Controller::HTTP_BAD_REQUEST);
    }
```

We can use the class constant `REST_Controller::HTTP_BAD_REQUEST` to specify the expected HTTP response code.

Run the `phpunit` command, and verify that the tests pass.

9.3 Adding Request Headers

If you add an Accept header with the `-H` or `--header` option, you can get the response in another format.

```
$ curl -X GET 'http://127.0.0.1:8000/api/example/users/id/1' \
  -H 'Accept: application/csv'
```

```
id,name,email,fact
1,John,john@example.com,"Loves coding"
```

Because of technical difficulties with the publishing system of this book, I changed the style of the long Bash command to use more than one line.

In this case, we can use the `$this->request->setHeader()` method in ci-phpunit-test to add a request header.

application/tests/controllers/api/Example_test.php

```
    public function test_When_send_accept_header_csv_Then_get_csv_response()
    {
        $this->request->setHeader('Accept', 'application/csv');
        $output = $this->request('GET', 'api/example/users/id/1');

        $this->assertEquals(
            'id,name,email,fact
1,John,john@example.com,"Loves coding"
',
            $output
        );
        $this->assertResponseCode(200);
        $this->assertResponseHeader(
            'Content-Type', 'application/csv; charset=utf-8'
        );
    }
```

You can assert a response header using ci-phpunit-test's `$this->assertResponseHeader()` method.

Run the `phpunit` command, and verify that the tests pass.

9.4 Testing POST Requests

You can send a POST request with curl, specifying the POST data with the `-d` or `--data` option. It returns the response code 201 (Created).

```
$ curl -X POST -i 'http://127.0.0.1:8000/api/example/users' \
  -d 'name=John&email=john@example.com'


HTTP/1.1 201 Created
Host: 127.0.0.1:8000
Connection: close
X-Powered-By: PHP/5.5.26
Content-Type: application/json; charset=utf-8

{"id":100,"name":"John","email":"john@example.com","message":"Added a resource"}
```

If you want to send data which needs to be urlencoded, use the `--data-urlencode` option instead of the `-d` option.

```
$ curl -X POST 'http://127.0.0.1:8000/api/example/users' \
  --data-urlencode "name=M&M's" \
  --data-urlencode 'email=mms@example.com'


{"id":100,"name":"M&M's","email":"mms@example.com","message":"Added a resource"}
```

We can write the following test method:

application/tests/controllers/api/Example_test.php

```
    public function test_When_post_user_data_Then_returns_created()
    {
        $post = [
            'name'  => 'John',
            'email' => 'john@example.com',
        ];

        $output = $this->request('POST', 'api/example/users', $post);
        $data = json_decode($output, true);

        $this->assertEquals($post['name'], $data['name']);
        $this->assertEquals($post['email'], $data['email']);
        $this->assertResponseCode(REST_Controller::HTTP_CREATED);
    }
```

You can pass the `$_POST` data as the third argument of the `$this->request('POST')` method.

Run the `phpunit` command, and verify that the tests pass.

9.5 Testing JSON Requests

You can send a JSON request, too. In this case, we use the `--data-binary` option and specify the "`Content-Type: application/json`" header.

```
$ curl -X POST 'http://127.0.0.1:8000/api/example/users' \
  -H 'Content-Type: application/json' \
  --data-binary '{"name":"John","email":"john@example.com"}'
```

```
{"id":100,"name":"John","email":"john@example.com","message":"Added a resource"}
```

The above request does not work properly with most of the web servers built-in with PHP, because of a bug[3]. The bug has been fixed only on PHP 5.6.13 or later. If you try to use this request, please use PHP 5.6.13 or later, or another web server, like Apache.

We can test this request with the following method:

application/tests/controllers/api/Example_test.php

```
    public function test_When_post_json_user_data_Then_returns_created()
    {
        $json = '{"name":"John","email":"john@example.com"}';
        $json_array = json_decode($json, true);

        $this->request->setHeader('Content-Type', 'application/json');
        $output = $this->request('POST', 'api/example/users', $json);
        $data = json_decode($output, true);

        $this->assertEquals($json_array['name'], $data['name']);
        $this->assertEquals($json_array['email'], $data['email']);
        $this->assertResponseCode(REST_Controller::HTTP_CREATED);
    }
```

You can set the request body (a JSON string) with the third argument of the `$this->request()` method.

Run the `phpunit` command, and verify that the tests pass.

9.6 Testing DELETE Requests

You can send a DELETE request:

[3] https://bugs.php.net/bug.php?id=66606

```
$ curl -X DELETE -i 'http://127.0.0.1:8000/api/example/users/id/1'
HTTP/1.1 204 No Content
Host: 127.0.0.1:8000
Connection: close
X-Powered-By: PHP/5.5.26
Content-Type: application/json; charset=utf-8
```

It returns the response code 204 (No Content). We can write the following method to test this:

application/tests/controllers/api/Example_test.php

```
    public function test_When_delete_user_Then_returns_no_content()
    {
        $output = $this->request('DELETE', 'api/example/users/id/1');
        $this->assertResponseCode(REST_Controller::HTTP_NO_CONTENT);
    }
```

Run the `phpunit` command, and verify that the tests pass.

If you specify an invalid user ID, then it returns the response code 400 (Bad Request):

```
$ curl -X DELETE -i 'http://127.0.0.1:8000/api/example/users/id/-1'
HTTP/1.1 400 Bad Request
Host: 127.0.0.1:8000
Connection: close
X-Powered-By: PHP/5.5.26
Content-type: text/html; charset=UTF-8
```

We can write a test method like the following:

application/tests/controllers/api/Example_test.php

```
    public function test_When_delete_invalid_user_id_Then_returns_bad_request()
    {
        try {
            $this->request('DELETE', 'api/example/users/id/-1');
        } catch (CIPHPUnitTestExitException $e) {
            $output = ob_get_clean();
        }
        $this->assertResponseCode(REST_Controller::HTTP_BAD_REQUEST);
    }
```

Run the `phpunit` command, and verify that the tests pass.

At the moment, the line coverage of the controller is 93.10%. We can see untested lines in the coverage report. The green lines are executed, the pink lines are not executed.

```
33      public function users_get()
34      {
35          // Users from a data store e.g. database
36          $users = [
37              ['id' => 1, 'name' => 'John', 'email' => 'john@example.com', 'fact' => 'Loves coding'],
38              ['id' => 2, 'name' => 'Jim', 'email' => 'jim@example.com', 'fact' => 'Developed on CodeIgniter'],
39              ['id' => 3, 'name' => 'Jane', 'email' => 'jane@example.com', 'fact' => 'Lives in the USA', ['hobb
40          ];
41
42          $id = $this->get('id');
43
44          // If the id parameter doesn't exist return all the users
45
46          if ($id === NULL)
47          {
48              // Check if the users data store contains users (in case the database result returns NULL)
49              if ($users)
50              {
51                  // Set the response and exit
52                  $this->response($users, REST_Controller::HTTP_OK); // OK (200) being the HTTP response code
53              }
54              else
55              {
56                  // Set the response and exit
57                  $this->response([
58                      'status' => FALSE,
59                      'error' => 'No users were found'
60                  ], REST_Controller::HTTP_NOT_FOUND); // NOT_FOUND (404) being the HTTP response code
61              }
62          }
```

Code Coverage for Example.php

The untested lines are sample code to be used when no users are found, but the lines are never executed, because the users' data `$users` is hard coded in the method.

If this were a real application, we would probably get the users' data from the database. In that case, we can write test code using database fixtures or a mocked model.

There is no way to inject variables like `$user`, we can't write test code to get 100% coverage in this case. Since the controller is an example, 93% is good enough.

10. Browser Testing with Codeception

In previous chapters, we have been using PHPUnit, but in this chapter we will explore some other tools. We will install Codeception and Selenium Server, learn how to configure them, and how to write browser tests.

10.1 Installing and Configuring Codeception

What is Codeception?

Codeception is a PHP testing framework which provides three levels of testing: Acceptance Tests, Functional Tests, and Unit Tests. We use *Acceptance Tests* as a synonym for *Browser Tests* in this chapter.

Any web site can be covered with acceptance tests provided by Codeception. To use functional tests, we need a module for our framework, but we don't have a module for CodeIgniter. For this reason, we will only use Codeception for acceptance tests in this chapter.

In previous chapters, we have been using PHPUnit with ci-phpunit-test providing framework integration with CodeIgniter. Since we don't have framework integration for Codeception, we will be using Codeception on its own, which should simplify the process of learning to use it for acceptance tests.

Installing Codeception

Download the latest version (in this book we use v2.1.4) from http://codeception.com/install and place the downloaded `codecept.phar` in the root directory of your PHP project.

```
CodeIgniter/
└── codecept.phar
```

To Composer users

Codeception may also be installed via Composer as follows:

```
$ cd CodeIgniter/
$ composer require codeception/codeception --dev
```

Once installed via Composer, use `vendor/bin/codecept` in place of `php codecept.phar` in the remaining instructions in this chapter.

What is Selenium Server?

Selenium is a project which automates browsers made up of three parts: the WebDriver, the Server, and the client. *Selenium WebDriver* is a driver to manipulate a browser. *Selenium Server* is a Java server application which communicates with Selenium client (in our case, facebook/webdriver[1] in Codeception) and Selenium WebDriver.

In summary, Codeception acts as our Selenium client, and uses Selenium Server to control the browser(s) used in our tests. Selenium Server uses the WebDriver(s) to control the browser(s) specified when we run Codeception.

Installing Selenium Server

Download the latest version (in this book we use v2.48.2) from http://docs.seleniumhq.org/download/ and place the downloaded `selenium-server-standalone-x.x.x.jar` in the root directory of your PHP project.

```
CodeIgniter/
└── selenium-server-standalone-2.48.2.jar
```

Initializing Codeception

To begin, initialize Codeception by running the following command:

```
$ php codecept.phar bootstrap
```

This command creates some files and a `tests` directory. You should see output similar to this:

```
 Initializing Codeception in .../CodeIgniter

File codeception.yml created       <- global configuration
tests/unit created                 <- unit tests
tests/unit.suite.yml written       <- unit tests suite configuration
tests/functional created           <- functional tests
tests/functional.suite.yml written <- functional tests suite configuration
tests/acceptance created           <- acceptance tests
tests/acceptance.suite.yml written <- acceptance tests suite configuration
tests/_output was added to .gitignore
 ---
tests/_bootstrap.php written <- global bootstrap file
```

[1] https://github.com/facebook/php-webdriver

```
Building initial Tester classes
Building Actor classes for suites: acceptance, functional, unit
 -> AcceptanceTesterActions.php generated successfully. 0 methods added
\AcceptanceTester includes modules: PhpBrowser, \Helper\Acceptance
AcceptanceTester.php created.
 -> FunctionalTesterActions.php generated successfully. 0 methods added
\FunctionalTester includes modules: \Helper\Functional
FunctionalTester.php created.
 -> UnitTesterActions.php generated successfully. 0 methods added
\UnitTester includes modules: Asserts, \Helper\Unit
UnitTester.php created.

Bootstrap is done. Check out .../CodeIgniter/tests directory
```

In this book, we only use Codeception for acceptance testing, so all of the test case files are placed in the `tests/acceptance` directory.

```
CodeIgniter/
├── codeception.yml  ... Codeception's global configuration file.
└── tests/
    ├── acceptance/          ... Your test files will be placed here.
    ├── acceptance.suite.yml ... Configuration file for acceptance tests.
    └── ...
```

Configuring Acceptance Tests

Testing with Firefox

We run acceptance tests with our web browser. First, we will use Firefox, because Selenium Standalone Server has a built-in Firefox driver.

Set the browser and the URL of our site in `acceptance.suite.yml`.

```
--- a/CodeIgniter/tests/acceptance.suite.yml
+++ b/CodeIgniter/tests/acceptance.suite.yml
@@ -7,6 +7,7 @@
 class_name: AcceptanceTester
 modules:
     enabled:
-        - PhpBrowser:
-            url: http://localhost/myapp
+        - WebDriver:
```

```
+            url: http://127.0.0.1:8000/
+            browser: 'firefox'
         - \Helper\Acceptance
```

The listing above is in a format called *unified diff*. It shows the difference between two files. In short, remove the red lines starting with - from the original file, and add the green lines starting with + to get the new file. See Conventions Used in This Book for additional details.

10.2 Writing Tests

Conventions for Codeception Acceptance Tests

Here are some basic conventions for the Codeception acceptance tests.

1. Test files have the suffix `Cept.php`.
2. The test code begins with "`$I = new AcceptanceTester($scenario);`".
3. The `$I->wantTo()` method defines your scenario (test name).

We put the files in `tests/acceptance` directory.

`Cept` is a scenario-based format for test case file in Codeception. While we don't use it in this book, Codeception also has a `Cest` format, which combines scenario-driven test approach with OOP design. If you want to group multiple testing scenarios into one, you should consider using `Cest` format. See the Cest Classes documentation[2] for details.

Writing Our First Test

Now that we know the basic conventions, we can start to write our first acceptance test.

[2] http://codeception.com/docs/07-AdvancedUsage#Cest-Classes

tests/acceptance/WelcomeCept.php

```
<?php
$I = new AcceptanceTester($scenario);
$I->wantTo('ensure that frontpage works');
$I->amOnPage('/');
$I->see('Home');
```

By reading the code above, you probably have a good idea of what this test does.

We must always write the line "`$I = new AcceptanceTester($scenario);`" at the top of the test code. The `$I->wantTo()` method defines our scenario (test name). The rest of the test accesses `/`, and checks whether the page received in the response contains the string `Home`.

10.3 Running Tests

Running Selenium Server

To run tests, we need to start Selenium server to control the web browser(s). Run the server with the following command:

```
$ java -jar selenium-server-standalone-2.48.2.jar
```

Running the Web Server

We also need a web server. Open another terminal, and run PHP's built-in web server:

```
$ CI_ENV=testing php -S 127.0.0.1:8000 index.php
```

We added `CI_ENV=testing` to run it under the `testing` environment. Because we want to set the environment of the CodeIgniter running on the server to a value of `testing`. It is the same thing you set `CI_ENV` in your .htaccess or Apache configuration file.

Our PHPUnit tests run in the `testing` environment, because ci-phpunit-test changes the environment automatically. But in this case, the PHP process of the web server is different from the process running Codeception. So we have no way to change the environment of the CodeIgniter on the server from our test code.

Running Codeception

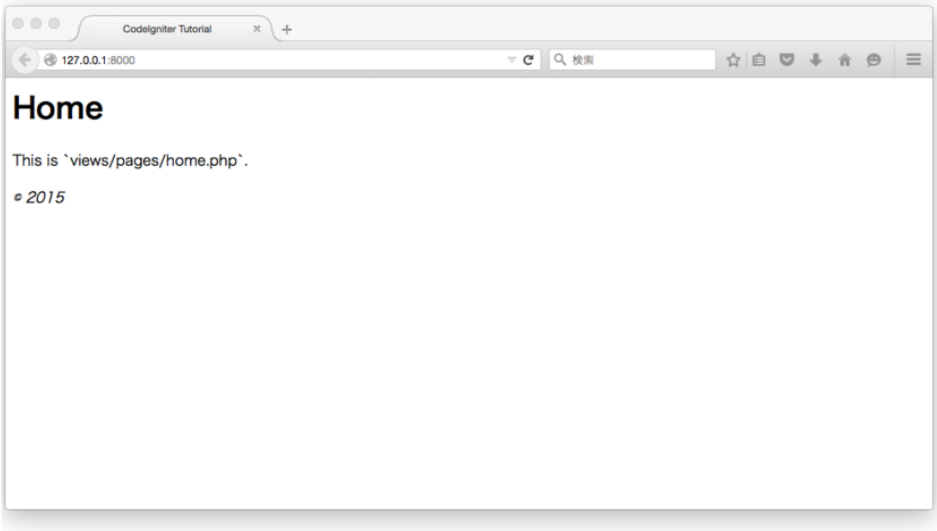
Since we will be using Firefox[3], it needs to be installed, as well.

Open another terminal. If you use the `codecept run` command, Codeception runs all tests:

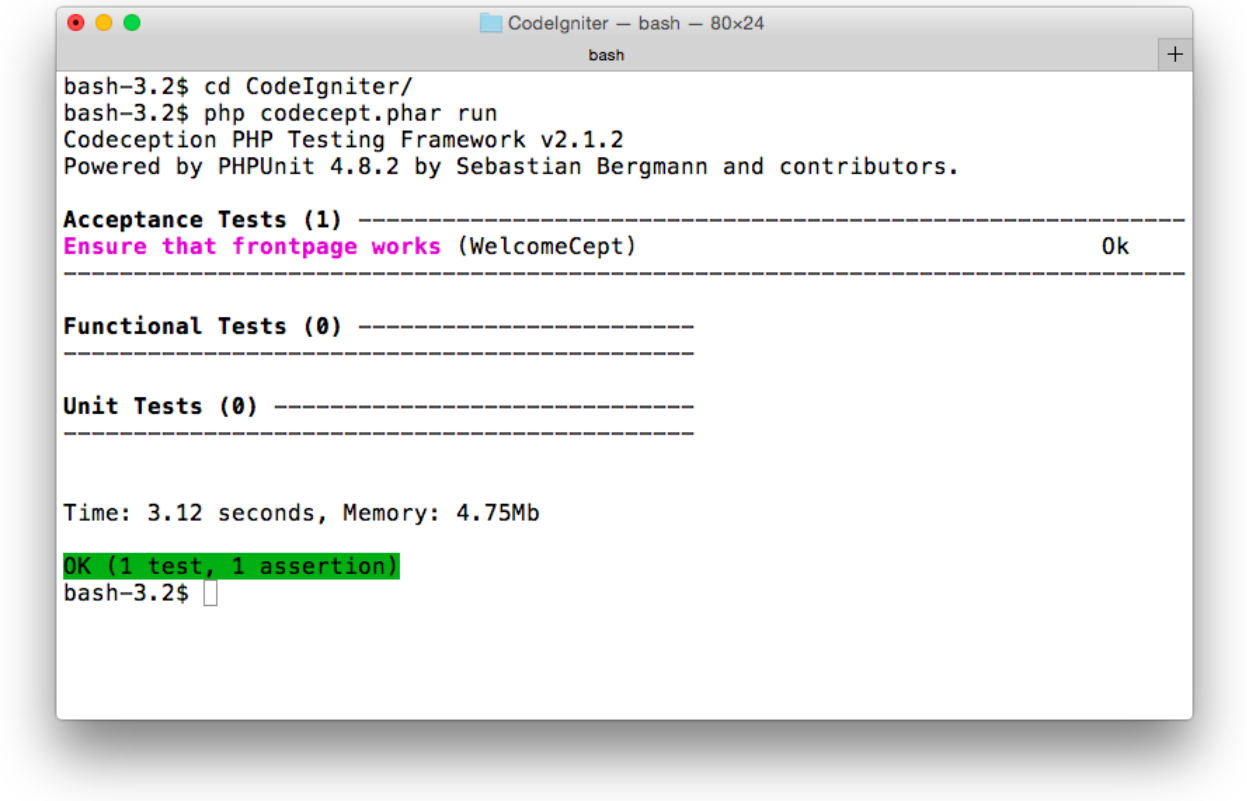
[3] https://www.firefox.com/

```
$ php codecept.phar run
```

You should see Firefox start up and access `http://127.0.0.1:8000`.



Firefox



Execute the codecept run command

You should see a green `OK` in your terminal. Our first test passed, but we also see the lines below:

```
Functional Tests (0) ----------
--------------------------------------------

Unit Tests (0) ----------
--------------------------------------------
```

We don't have any functional tests or unit tests with Codeception. Since we're using PHPUnit for those tests, we don't need them in Codeception. We can run only the acceptance tests using the following command:

```
$ php codecept.phar run acceptance
Codeception PHP Testing Framework v2.1.4
Powered by PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

Acceptance Tests (1) ------------------------------------------------------------
Ensure that frontpage works (WelcomeCept)                                  Ok
----------------------------------------------------------------------------------


Time: 3.75 seconds, Memory: 4.75Mb

OK (1 test, 1 assertion)
```

10.4 Browser Testing: Pros and Cons

There are pros and cons in browser testing.

Pros:

- can test whole system (it is system testing)
- can test JavaScript and Ajax requests
- less affected by changes in source code
- can be run on any web site
- can be shown to your clients and managers

Cons:

- they are really slow (it requires running web server, web browser and database)
- affected by changes in views
- fewer checks can lead to false-positive results
- rendering and JavaScript issues can lead to unpredictable results

10.5 Database Fixtures

The browser test tests the whole system, so we need the database. That means we need database fixtures, too. Codeception includes functionality to restore a SQL dump file to the database when running acceptance tests. However, since we have been using database migrations and seeders for our tests, we don't have a SQL dump file.

We could generate a SQL dump file, but the migrations and seeders have many benefits of their own, as discussed in Chapter 5. Since we already have the migrations, seeders, and the capability to run them, we should make use of them here, as well.

We use the `Dbfixture` controller, which we tested in Chapter 8, and run the controller before executing the acceptance tests. If we do it this way, we don't have to maintain a SQL dump file, and we make good use of our existing testing infrastructure.

application/controllers/Dbfixture.php

```
<?php

class Dbfixture extends CI_Controller
{
    public function __construct()
    {
        parent::__construct();

        // Only accessible via CLI
        if (is_cli() === false) {
            exit();
        }
    }

    public function migrate()
    {
        $this->load->library('migration');
        if ($this->migration->current() === false) {
            echo $this->migration->error_string() . PHP_EOL;
        }
    }

    public function seed()
    {
        $this->load->library('seeder');
        foreach (glob(APPPATH.'database/seeds/*Seeder.php') as $file) {
            $seeder = basename($file, '.php');
            $this->seeder->call($seeder);
        }
    }

    public function all()
    {
        $this->migrate();
```

```
35          $this->seed();
36      }
37  }
```

This controller checks to ensure it is being run from the CLI, and the `all()` method runs migrations, and executes all seeder files which have the suffix `Seeder.php` in the `application/database/seeder` directory.

10.6 Test Case for the News Controller

We wrote a test case for the News controller in Chapter 7, Test Case for the News Controller. Now, we'll write the same tests with Codeception.

tests/acceptance/NewsCept.php

```
1  <?php
2  $I = new AcceptanceTester($scenario);
3  $I->wantTo('ensure that News works');
```

Database Fixtures

Before running tests, we have to prepare our database. We have already made the `Dbfixture` controller to help us do this. How do we run the controller from our Codeception tests?

We can run it with the `system()`[4] function in PHP.

tests/acceptance/NewsCept.php

```
1  <?php
2
3  system("CI_ENV=testing php index.php dbfixture all");
4
5  $I = new AcceptanceTester($scenario);
6  $I->wantTo('ensure that News works');
```

To Composer users

If you installed CodeIgniter via Composer, the path of `index.php` is different.

```
system("CI_ENV=testing php public/index.php dbfixture all");
```

[4] http://php.net/en/system

Testing Page Contents

We can write tests the same way we did in `WelcomeCept`. Set the URI with the `$I->amOnPage()` method and check the result with the `$I->see()` method.

tests/acceptance/NewsCept.php

```
$I->amGoingTo('access the archive page');
$I->amOnPage('/news');
$I->see('News archive');
$I->see('News test');
$I->see('News text', '.main');
$I->seeInTitle('CodeIgniter Tutorial');
```

Comments

The `$I->amGoingTo()` method allows you to describe the actions you are going to perform. It is just a method of adding comments to your test. You can also use the `$I->expect()` and `$I->expectTo()` methods to add comments describing the expected results.

```
$I->amGoingTo('access the archive page');
```

Assertions

The `$I->see()` method is an assertion method. It allows us to check whether the argument string is on the page.

```
$I->see('News archive');
$I->see('News test');
```

We can use the second argument to specify a CSS class which we expect to be applied to the string.

```
$I->see('News text', '.main');
```

We can also check the page title.

```
$I->seeInTitle('CodeIgniter Tutorial');
```

To learn more about assertion methods, see the documentation's assertions section[5].

Testing Forms

Codeception also allows us to fill out and submit forms. We use the `$I->fillField()` method to fill out the fields in the form, then use the `$I->click()` method to click the submit button.

[5] http://codeception.com/docs/03-AcceptanceTests#Assertions

tests/acceptance/NewsCept.php

```
$I->amGoingTo('post a valid item');
$I->fillField('title', 'CodeIgniter is easy to write tests');
$I->fillField('text', 'You can write tests for controllers very easily!');
$I->click('Create news item');
$I->see('Successfully created');
```

If you want to know more about form manipulation, see the documentation's Forms section[6].

NewsCept

The following is the complete code listing for `NewsCept.php`.

tests/acceptance/NewsCept.php

```
1  <?php
2
3  system("CI_ENV=testing php index.php dbfixture all");
4
5  $I = new AcceptanceTester($scenario);
6  $I->wantTo('ensure that News works');
7
8  $I->amGoingTo('access the archive page');
9  $I->amOnPage('/news');
10 $I->see('News archive');
11 $I->see('News test');
12 $I->see('News text', '.main');
13 $I->seeInTitle('CodeIgniter Tutorial');
14
15 $I->amGoingTo('access an item which does not exist');
16 $I->amOnPage('/news/not-existing-slug');
17 $I->see('404 Page Not Found');
18 $I->dontSeeInTitle('CodeIgniter Tutorial');
19
20 $I->amGoingTo('access an existing item');
21 $I->amOnPage('/news/news-test');
22 $I->see('News test');
23 $I->seeInTitle('CodeIgniter Tutorial');
24
25 $I->amGoingTo('access the create page');
```

[6] http://codeception.com/docs/03-AcceptanceTests#Forms

```
26 $I->amOnPage('/news/create');
27 $I->see('Create a news item');
28
29 $I->amGoingTo('post a valid item');
30 $I->fillField('title', 'CodeIgniter is easy to write tests');
31 $I->fillField('text', 'You can write tests for controllers very easily!');
32 $I->click('Create news item');
33 $I->see('Successfully created');
34
35 $I->amGoingTo('access the archive page');
36 $I->amOnPage('/news');
37 $I->see('News test');
38 $I->see('CodeIgniter is easy to write tests');
39
40 $I->amGoingTo('post an invalid item');
41 $I->amOnPage('/news/create');
42 $I->click('Create news item');
43 $I->see('The Title field is required.');
44 $I->see('The Text field is required.');
```

Run it and confirm that the tests pass. Don't forget to run Selenium server and the web server before running the tests.

```
$ php codecept.phar run acceptance tests/acceptance/NewsCept.php
```

You should see a green `OK` like the following:

```
Codeception PHP Testing Framework v2.1.4
Powered by PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

Acceptance Tests (1) ------------------------------------------------------------
Ensure that news works (NewsCept)                                          Ok
---------------------------------------------------------------------------------


Time: 5.86 seconds, Memory: 5.25Mb

OK (1 test, 14 assertions)
```

If you want to know more about the functionality of Codeception's acceptance tests, see the Acceptance Testing documentation[7].

[7] http://codeception.com/docs/03-AcceptanceTests

10.7 Testing with Google Chrome

To run tests with Google Chrome, we need to install Google Chrome[8] and the ChromeDriver for Selenium.

Installing the ChromeDriver

You can find the driver at http://www.seleniumhq.org/download/ or https://github.com/SeleniumHQ/selenium/wiki/ChromeDriver.

You may have to follow a few links to get to the latest driver. In this book, we use v2.20[9]. Download the driver for your platform and unzip it. Move the driver to the root directory of your PHP project.

```
CodeIgniter/
└── chromedriver
```

To Windows users

The name of the driver file for Windows is `chromedriver.exe`.

Configuring Acceptance Tests

To use the Chrome driver in your tests, define different environments inside the `env` root in `acceptance.suite.yml`. Name the environments (firefox, chrome, etc.), then redefine any configuration parameters that were previously set.

```
--- a/code/CodeIgniter/tests/acceptance.suite.yml
+++ b/code/CodeIgniter/tests/acceptance.suite.yml
@@ -11,3 +11,11 @@ modules:
             url: http://127.0.0.1:8000/
             browser: 'firefox'
         - \Helper\Acceptance
+env:
+    chrome:
+        modules:
+            config:
+                WebDriver:
+                    browser: 'chrome'
+    firefox:
+        # nothing changed
```

[8] http://www.google.com/chrome/

[9] http://chromedriver.storage.googleapis.com/index.html?path=2.20/

The listing above is in a format called *unified diff*. It shows the difference between two files. In short, add the green lines starting with + to the original file. See Conventions Used in This Book for details.

Running Selenium Server

When you run Selenium server, add the driver file path with the following option:

```
$ java -jar selenium-server-standalone-2.48.2.jar \
  -Dwebdriver.chrome.driver=./chromedriver
```

Because of technical difficulties in the publishing system of this book, I changed the style of the long Bash command by breaking it into more than one line.

To Windows users

The name of the driver file for Windows is `chromedriver.exe`.

Now, we are ready to start running tests with Chrome.

Running Tests

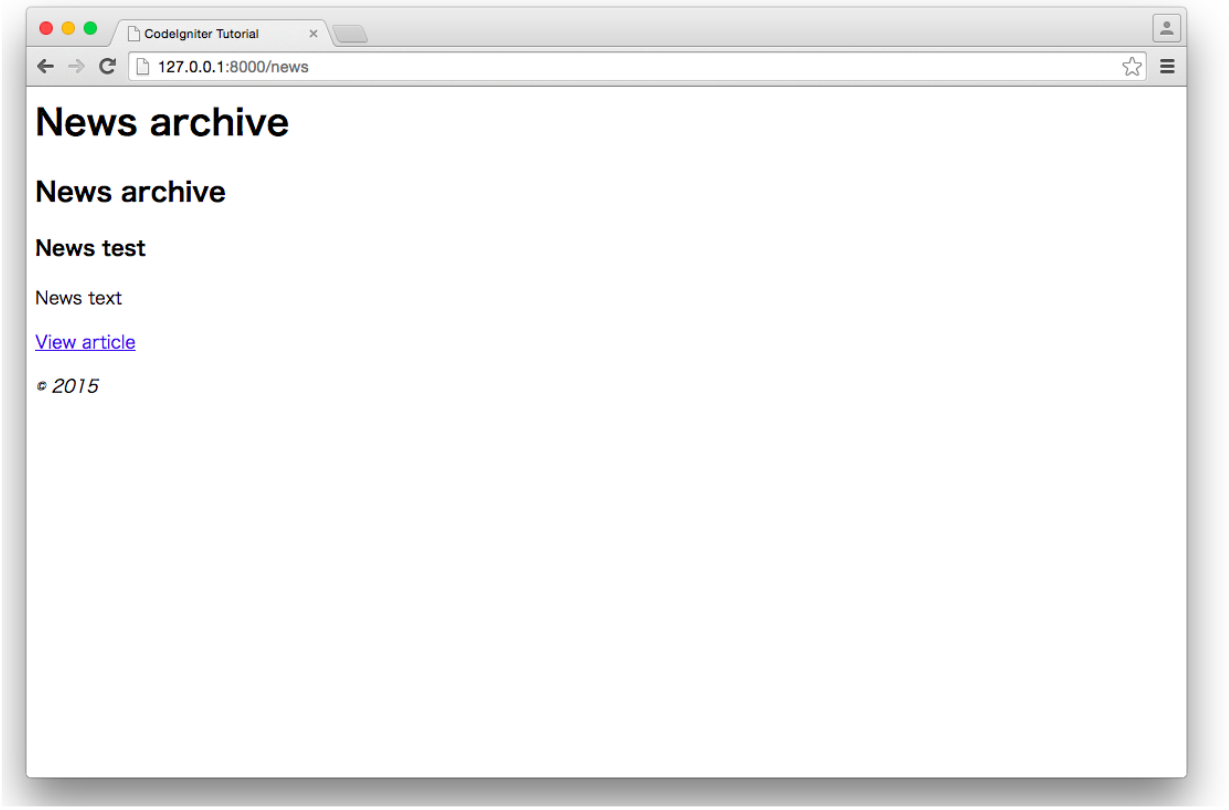
Run PHP's built-in web server before running Codeception:

```
$ CI_ENV=testing php -S 127.0.0.1:8000 index.php
```

Run the `codecept` command with the `--env` option:

```
$ php codecept.phar run acceptance --env chrome
```

You should see Google Chrome start up and run the tests.



Chrome

```
Codeception PHP Testing Framework v2.1.4
Powered by PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

Acceptance-chrome Tests (2) ----------------------------------------------------
Ensure that news works (NewsCept)                                          Ok
Ensure that frontpage works (WelcomeCept)                                  Ok
-------------------------------------------------------------------------------


Time: 6.4 seconds, Memory: 5.00Mb

OK (2 tests, 15 assertions)
```

You can also run the tests in multiple browsers:

```
$ php codecept.phar run acceptance --env firefox --env chrome
Codeception PHP Testing Framework v2.1.4
Powered by PHPUnit 4.8.10 by Sebastian Bergmann and contributors.

Acceptance-firefox Tests (2) ----------------------------------------------------
Ensure that news works (NewsCept)                                          Ok
Ensure that frontpage works (WelcomeCept)                                  Ok
---------------------------------------------------------------------------------

Acceptance-chrome Tests (2) -----------------------------------------------------
Ensure that news works (NewsCept)                                          Ok
Ensure that frontpage works (WelcomeCept)                                  Ok
---------------------------------------------------------------------------------


Time: 11.74 seconds, Memory: 5.25Mb

OK (4 tests, 30 assertions)
```

The output above shows that we have run all tests with two browsers, and the result is green.

11. Congratulations

You've learned how to write tests for your applications. I hope you have enjoyed the book.

Writing good tests now will help you catch mistakes earlier, and make it easier to change your code without introducing errors, saving time in the long run. It could also improve the design of your code.

Last Message

We dedicated more pages to controller testing than any other type of testing in this book, but that is just because testing controllers is a complex topic. It is more important to test models (or libraries) in most CodeIgniter applications. Of course, it is better to test both controllers and models, but testing controllers is less important than testing models.

Generally speaking, fat controllers are a bad practice. The most important code, usually *Domain Models* or *Business Logic*, should be in your models (or libraries). This is the code for which tests should be written first.

Feedback

If you find any errors or have suggestions, please share them by opening an issue on GitHub[1] and I'll address them promptly.

Testimonials: If you have enjoyed this book, and you don't mind providing a short testimonial to be used in various places (e.g. the book's web page), please leave a comment on this page[2]. Even a short sentence or two would be appreciated.

Thank you for your support. I hope you like this book, and I hope to hear from you soon on GitHub, Twitter, or via email.

[1] https://github.com/kenjis/codeigniter-testing-guide/issues
[2] https://github.com/kenjis/codeigniter-testing-guide/issues/1

Appendix A

How to Speed Up Testing

If you have slow tests, they may prevent you from testing, so your tests should be fast. This appendix will give you some hints to speed up your tests.

Whenever possible, you should make your tests faster. However, if you keep writing tests, you can end up with so many tests that running your whole test suite will take a long time, no matter how fast the individual tests may be.

Still, there is usually something you can do to speed up your tests.

Speed Up without Code Modifications

Without modifying your application/test code, there are still some things which might make your tests run faster:

- change the version of PHPUnit you are using
- install PHPUnit via Composer
- disable code coverage reporting
- disable Xdebug
- use phpdbg instead of Xdebug
- upgrade to PHP 7
- use a SQLite in-memory database

Here are the benchmarks of a certain CodeIgniter project:

PHP	Coverage	Xdebug	option	phpdbg	PHPUnit	Test Time (sec)	Total Time (sec)
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	4.6.10 (Phar)	32.35	35.634
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	4.8.21 (Phar)	26.53	29.920
5.6.18-dev	No	2.4.0RC3	no-coverage	n/a	4.8.21 (Phar)	3.29	3.426
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	4.8.21 (Composer)	21.60	24.922

PHP	Coverage	Xdebug	option	phpdbg	PHPUnit	Test Time (sec)	Total Time (sec)
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	5.0.10 (Phar)	25.99	29.350
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	5.1.3 (Phar)	26.14	29.512
5.6.18-dev	No	n/a	n/a	n/a	4.6.10 (Phar)	1.51	1.638
5.6.18-dev	No	n/a	n/a	n/a	4.8.21 (Phar)	1.51	1.633
5.6.18-dev	No	n/a	n/a	n/a	4.8.21 (Composer)	1.30	1.416
7.0.3-dev	Yes	n/a	n/a	0.5.0	4.8.21 (Composer)	2.00	2.728
7.0.3-dev	No	n/a	n/a	n/a	4.6.10 (Phar)	0.799	0.834
7.0.3-dev	No	n/a	n/a	n/a	4.8.21 (Phar)	0.792	0.825
7.0.3-dev	No	n/a	n/a	n/a	4.8.21 (Composer)	0.732	0.769

Test Time is the time output by PHPUnit. *Total Time* is the whole time measured by the `time` command. If *Coverage* is *Yes*, it includes the time to generate the coverage report. Both Time columns are displayed in seconds.

The actual commands are like the following:

```
1 $ time php phpunit-4.8.21.phar
2 $ time ../../vendor/bin/phpunit
3 $ time phpdbg -qrr ../../vendor/bin/phpunit
```

Unfortunatley the following combinations do not work properly:

- PHP 7 + phpdbg + PHPUnit (Phar) ... 2 errors
- PHP 7 + Xdebug ... Segmentation fault

You need PHP 7 to use phpdbg for coverage reporting.

Changing PHPUnit

PHP	Coverage	Xdebug	option	phpdbg	PHPUnit	Test Time (sec)	Total Time (sec)
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	4.6.10 (Phar)	32.35	35.634
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	4.8.21 (Phar)	26.53	29.920
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	4.8.21 (Composer)	21.60	24.922
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	5.0.10 (Phar)	25.99	29.350
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	5.1.3 (Phar)	26.14	29.512

PHPUnit 4.8.21 is faster than 4.6.10.

PHPUnit 4.8.21 installed via Composer is a little faster than the same version of PHPUnit Phar.

Updating PHPUnit does not guarantee that the tests will speed up, but there may be other benefits to keeping it up to date.

Disable Coverage Reporting

If you want to make your tests faster, disable generating the code coverage report.

PHP	Coverage	Xdebug	option	phpdbg	PHPUnit	Test Time (sec)	Total Time (sec)
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	4.8.21 (Phar)	26.53	29.920
5.6.18-dev	No	2.4.0RC3	no-coverage	n/a	4.8.21 (Phar)	3.29	3.426

The `--no-coverage` option for PHPUnit greatly improves the test speed (in this example, from 26.53 sec to 3.29 sec).

To use the `--no-coverage` option, you need PHPUnit 4.8 or later.

Disable Xdebug

If you disable Xdebug, your test will be a bit faster than with Xdebug and the `--no-coverage` option.

PHP	Coverage	Xdebug	option	phpdbg	PHPUnit	Test Time (sec)	Total Time (sec)
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	4.6.10 (Phar)	32.35	35.634
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	4.8.21 (Phar)	26.53	29.920
5.6.18-dev	No	2.4.0RC3	no-coverage	n/a	4.8.21 (Phar)	3.29	3.426
5.6.18-dev	No	n/a	n/a	n/a	4.6.10 (Phar)	1.51	1.638
5.6.18-dev	No	n/a	n/a	n/a	4.8.21 (Phar)	1.51	1.633

Use phpdbg instead of Xdebug

If you only need the code coverage report, you could generate it using phpdbg using the following options:

```
$ phpdbg -qrr phpunit.phar
```

This is not a fair comparison, but phpdbg appears to be faster than Xdebug for code coverage reporting.

PHP	Coverage	Xdebug	option	phpdbg	PHPUnit	Test Time (sec)	Total Time (sec)
5.6.18-dev	Yes	2.4.0RC3	n/a	n/a	4.8.21 (Composer)	21.60	24.922
7.0.3-dev	Yes	n/a	n/a	0.5.0	4.8.21 (Composer)	2.00	2.728

You need PHP 7 and PHPUnit 4.8 or later to use phpdbg for coverage reporting. If you use PHP 5, you must use Xdebug for coverage reporting.

Upgrading to PHP 7

PHP	Coverage	Xdebug	option	phpdbg	PHPUnit	Test Time (sec)	Total Time (sec)
5.6.18-dev	No	n/a	n/a	n/a	4.6.10 (Phar)	1.51	1.638
5.6.18-dev	No	n/a	n/a	n/a	4.8.21 (Phar)	1.51	1.633
5.6.18-dev	No	n/a	n/a	n/a	4.8.21 (Composer)	1.30	1.416
7.0.3-dev	No	n/a	n/a	n/a	4.6.10 (Phar)	0.799	0.834
7.0.3-dev	No	n/a	n/a	n/a	4.8.21 (Phar)	0.792	0.825
7.0.3-dev	No	n/a	n/a	n/a	4.8.21 (Composer)	0.732	0.769

As you may already know, PHP 7 is much faster than 5.6. So, if you can use PHP 7, you can make your tests faster, too.

SQLite in-memory database

If you use SQLite, you can improve the performance of your tests by switching to the SQLite in-memory database. If you use *Query Builder*, and do not use database-specific functionality, it may be possible to switch your application's database to SQLite for testing.

To use a SQLite in-memory database, update or create `application/config/testing/database.php` like the following:

application/config/testing/database.php

```
$db['default'] = array(
    'dsn'      => 'sqlite::memory:',
    'hostname' => 'localhost',
    'username' => '',
    'password' => '',
    'database' => '',
    'dbdriver' => 'pdo',
    'dbprefix' => '',
    'pconnect' => FALSE,
    'db_debug' => (ENVIRONMENT !== 'production'),
    'cache_on' => FALSE,
    'cachedir' => '',
    'char_set' => 'utf8',
```

```
    'dbcollat' => 'utf8_general_ci',
    'swap_pre' => '',
    'encrypt' => FALSE,
    'compress' => FALSE,
    'stricton' => FALSE,
    'failover' => array(),
    'save_queries' => TRUE
);
```

Appendix B

How to Read ci-phpunit-test

If you want to customize or improve ci-phpunit-test, you need to read its source code. This is a guide to help you do so.

Structure of ci-phpunit-test

ci-phpunit-test consists of the following major parts:

- Bootstrap ... Bootstrap for PHPUnit.
- Autoloader ... Autoloader for application and ci-phpunit-test class files.
- Replaced Classes and Functions ... Replacements for CodeIgniter code.
- New Functions ... Functions added for testing.
- TestCase classes ... TestCase classes provided by ci-phpunit-test.
- Request-related classes ... `$this->request()` and related functionality.
- Helper classes ... Helper or short-cut methods for test code.
- Monkey Patch library ... Library to provide monkey patching capability.

```
tests/
├── Bootstrap.php  ... Bootstrap
├── TestCase.php   ... TestCase class for users
├── _ci_phpunit_test/
|   ├── CIPHPUnitTest.php            ... Bootstrap
|   ├── CIPHPUnitTestAutoloader.php  ... Autoloader class
|   ├── CIPHPUnitTestCase.php        ... Base TestCase class
|   ├── CIPHPUnitTestDouble.php      ... Helper class to create Mock objects
|   ├── CIPHPUnitTestFileCache.php   ... Cache class for Autoloader
|   ├── CIPHPUnitTestReflection.php  ... Helper class to access private methods
|   ├── CIPHPUnitTestRequest.php     ... Request related class (main)
|   ├── CIPHPUnitTestRouter.php      ... Request related class for routing
|   ├── CIPHPUnitTestSuperGlobal.php ... Request related class for super globals
|   ├── alias/                       ... Class aliases
|   ├── autoloader.php               ... Autoloader
|   ├── exceptions/                  ... Exception classes
```

```
|   ├── functions.php               ... New Functions
|   ├── patcher/                    ... Monkey Patch library
|   ├── replacing/                  ... Replaced Classes and Functions
|   └── tmp/
|       └── cache/                    ... Cache folder
└── phpunit.xml    ... PHPUnit configuration file
```

Bootstrap

The bootstrap file for PHPUnit is `tests/Bootstrap.php`. It is executed once before running test cases. It has three parts:

1. A nearly complete copy of CodeIgniter's `index.php` file.
2. Initialization of the Monkey Patch library.
3. Calls the `CIPHPUnitTest::init()` method.

`CIPHPUnitTest::init()` loads and registers the autoloader, which loads new functions as well as the replaced classes and functions. It also executes CodeIgniter's `core/CodeIgniter.php`, but not all of it. You can see the replaced version of it in `_ci_phpunit_test/replacing/core/CodeIgniter.php`.

Autoloader

The file `tests/_ci_phpunit_test/autoloader.php` registers the autoloader. The `CIPHPUnitTestAutoloader::load()` method provides the autoloader's functionality. It autoloads ci-phpunit-test classes, and application classes like controllers, models, and libraries.

Replaced Classes and Functions

Some CodeIgniter classes and functions are replaced by ci-phpunit-test to make it easier to write tests. You can see all of the replaced function and class names in How to Write Tests[3]. It is recommended that you read that document before continuing.

Most of the functions and classes replaced by ci-phpunit-test were replaced for one of two reasons:

1. To reset the CodeIgniter super object
2. To provide behavior needed only for testing

[3] https://github.com/kenjis/ci-phpunit-test/blob/master/docs/HowToWriteTests.md#can-and-cant

The first is required primarily because the CodeIgniter super object is a form of singleton.

Singleton is a design pattern that restricts the instantiation of a class to one object. In fact, the CodeIgniter super object is not restricted to one instance, but under normal circumstances it is expected that only one instance will exist.

Singletons can be very difficult to test, unless you can reset them to a known state for each test (or for various parts of a test). So, ci-phpunit-test has the capability to reset CodeIgniter's super object. Some functions which were replaced to make this possible include `load_class()` and `is_loaded()`.

The second is necessary in part because portions of the CodeIgniter code do things which would prevent the tests from running. For example, `redirect()`, `show_404()` and `show_error()` call `exit()`, so they stop phpunit execution. ci-phpunit-test replaces them with versions which do not call `exit()`.

Another example is the `CI_Load` class. When we run tests, we call more than one controller, and may load a model or library multiple times. The `CI_Load` class has code to prevent loading something more than once, so we need to disable that code.

If you install CodeIgniter and ci-phpunit-test via Composer, you can check the differences in the `CI_Load` class using the following commands:

```
$ cd vendor/kenjis/ci-phpunit-test
$ bin/check-diff.sh
```

In another example of replacing classes to provide behavior only required for testing, `set_status_header()` is replaced to allow it to function when run from the CLI, and to save the status code in the `CI_Output` object, giving ci-phpunit-test better access to the information.

New Functions

New functions are needed for the same two reasons:

1. To reset the CodeIgniter super object
2. To provide behavior needed only for testing

The `reset_instance()` function resets the CodeIgniter super object.

The `load_class_instance()` function provides behavior which is only needed for testing. By injecting a CodeIgniter instance directly into the `load_class()` function, it can manipulate the state of the function. The `set_is_cli()` function allows ci-phpunit-test to change the return value of CodeIgniter's `is_cli()` function.

TestCase classes

The `CIPHPUnitTestCase` class is the base class of the `TestCase` class. The `TestCase` class is intended to allow user customization, so it has no logic by default.

The `CIPHPUnitTestCase` class provides many methods for testing, including assertions, setup, and tear down. It also transfers some methods to other objects (e.g. `CIPHPUnitTestRequest` and `CIPHPUnitTestDouble`).

Request related classes

The `CIPHPUnitTestRequest` class provides the `$this->request()` method. It emulates an HTTP request to a controller or a URI string. If you pass a URI string to `$this->request()`, the `CIPHPUnitTestRequest` object resolves the route using the `CIPHPUnitTestRouter` object.

The `CIPHPUnitTestSuperGlobal` object sets the super global variables for the request. After ci-phpunit-test v0.10.1, it will also be used to set global variables for the CodeIgniter core objects.

Helper classes

There are two helper classes. One is `CIPHPUnitTestDouble`, which helps to create the PHPUnit Mock object and verify invocations. The other is `CIPHPUnitTestReflection`, which helps to access private methods/properties using PHP Reflection.

Helper classes provide only helper methods or short-cuts. The `$this->getDouble()` and the `$this->verifyInvoked()` methods are transferred to the `CIPHPUnitTestDouble` object.

The `CIPHPUnitTestReflection` class can be used as a `ReflectionHelper` class. This is setup by a class alias which can be found in the `_ci_phpunit_test/alias` directory.

Monkey Patch library

The Monkey Patch library provides the monkey patching functionality used by ci-phpunit-test. It can be used as a stand alone library, but I recommend you use it only for testing.

It provides three patchers:

- `ExitPatcher` ... converts `exit()` to an exception
- `FunctionPatcher` ... patches functions
- `MethodPatcher` ... patches methods in user-defined classes

Appendix C

References

1. Bergmann, Sebastian. "PHPUnit Manual," 2015, accessed 2015-09-10, https://phpunit.de/manual/4.8/en/.
2. British Columbia Institute of Technology. "CodeIgniter User Guide," 2015-08-07, accessed 2015-09-10, http://www.codeigniter.com/user_guide/.
3. Bergmann, Sebastian, Stefan Priebsch. *Real-World Solutions for Developing High-Quality PHP Frameworks and Applications*. Wiley Publishing, Inc., 2011. ISBN: 978-0-470-87249-9.
4. Jones, Paul M. *Modernizing Legacy Applications In PHP*. Leanpub, 2015-07-19.
5. Ezell, Lonnie. *Practical CodeIgniter 3*. Leanpub, 2015-08-07.
6. Hartjes, Chris. *The Grumpy Programmer's PHPUnit Cookbook*. Leanpub, 2015-06-17.
7. Lockhart, Josh. *Modern PHP*. O'Reilly Media, Inc., 2015-02-09. ISBN: 978-1-491-90501-2.
8. Burke, Eric M. and Brian M. Coyner. "Top 12 Reasons to Write Unit Tests," 2003-04-02, accessed 2015-09-10, http://www.onjava.com/pub/a/onjava/2003/04/02/javaxpckbk.html.
9. Savoia, Alberto. "The Way of Testivus - Unit Testing Wisdom From An Ancient Software Start-up," 2007-04-26, accessed 2015-09-10, http://www.artima.com/weblogs/viewpost.jsp?thread=203994.
10. Shore, James. "Red-Green-Refactor," 2005-11-30, accessed 2015-09-11, http://www.jamesshore.com/Blog/Red-Green-Refactor.html
11. Wikipedia, "Software testing," 2015-09-09, accessed 2015-09-11, https://en.wikipedia.org/wiki/Software_testing
12. Way, Jeffrey. *Laravel Testing Decoded*. Leanpub, 2013-08-06.

License Agreement for CodeIgniter and its User Guide

License Agreement for CodeIgniter Rest Server

License Agreement for Ion Auth