



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

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

This version was published on 2016-01-23



Leanpub

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

© 2015 - 2016 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	i
The Book at a Glance	i
What You Need for This Book	iii
Who should read This Book?	iii
Why PHPUnit?	iv
Is This a CodeIgniter Book?	iv
Is Testing PHP Applications Difficult?	iv
Is Testing CodeIgniter Applications Difficult?	v
Testing is Fun and Easy	v
Conventions Used in This Book	v
Errata	vii
1. What is Automated Testing?	1
1.1 Primitive Example	1
Manual Testing	1
Automated Testing	1
1.2 Why should you write test code?	1
1.3 Finding the Middle Way	1
1.4 What should you test?	1
1.5 TDD or Not TDD	1
2. Setting Up the Testing Environment	2
2.1 Installing CodeIgniter	3
2.2 Installing ci-phpunit-test	3
Enabling Monkey Patching	3
2.3 (Optional) Installing VisualPHPUnit	3
Installing Composer	3
Installing VisualPHPUnit	3
Installing PHPUnit	3
Configuring VisualPHPUnit	3
Configuring PHPUnit XML Configuration File	3
2.4 Installing PHPUnit	3
2.5 (Optional) Installing PsySH	3
2.6 Installing via Composer	3

CONTENTS

Installing Composer	3
Installing CodeIgniter via Composer	3
Installing ci-phpunit-test via Composer	3
Installing PHPUnit via Composer	3
(Optional) Installing PsySH via Composer	3
3. Test Jargon	4
3.1 Testing levels	4
Unit Testing	4
Integration Testing	4
System Testing	4
3.2 Testing Types	4
Functional Testing	4
Database Testing	4
Browser Testing	4
Acceptance Testing	4
3.3 Code Coverage	4
3.4 Fixtures	4
3.5 Test Doubles	4
Mocks and Stubs	4
4. PHPUnit Basics	5
4.1 Running PHPUnit	5
Running All Tests	5
Running a Specific Test Case	7
4.2 Running PHPUnit via Web Browser	7
Running Web Server	7
Running All Tests	8
Running a Specific Test Case	10
4.3 Configuring PHPUnit	12
XML Configuration File	12
Command Line Arguments and Options	14
4.4 Understanding the Basics by Testing Libraries	17
Basic Conventions	18
Data Provider	23
Fixtures	27
Assertions	29
5. Testing a Simple MVC Application	30
5.1 Functional Testing for Controller	31
Controller to Handle Static Pages	31
Manual Testing with a Web Browser	31
Test Case for Page Controller	31

CONTENTS

Checking Code Coverage	31
5.2 Database Testing for Models	31
Preparing the Database	31
News Section	31
Manual Testing with a Web Browser	31
Database Fixtures	31
Test Case for the News Model	31
Checking Code Coverage	31
6. Unit Testing for Models	32
6.1 Why Should You Test Models First?	33
6.2 PHPUnit Mock Objects	33
Playing with Mocks	33
Partial Mocks	33
Verifying Expectations	33
6.3 Testing Models without Database	33
Testing the get_news() Method with Mocks	33
Testing the set_news() Method with Mocks	33
6.4 With the Database or Without the Database?	33
Testing with Little Value	33
When You Write Tests without the Database	33
7. Testing Controllers	34
7.1 Why is Testing Controllers Difficult?	34
7.2 Test Case for the News Controller	34
7.3 Mocking Models	34
7.4 Authentication and Redirection	34
Installing Ion Auth	34
Manual Testing with a Web Browser	34
Testing Redirection	34
Mocking Auth Objects	34
7.5 What if My Controller Needs Something Else?	34
8. Unit Testing CLI Controllers	35
8.1 Dbfixture Controller	35
8.2 Faking is_cli()	35
8.3 Testing exit()	35
8.4 Testing Exceptions	35
8.5 Testing Output	35
8.6 Monkey Patching	35
Patching Functions	35
Patching Class Methods	35
8.7 Checking Code Coverage	35

CONTENTS

- 9. Testing REST Controllers 36**
 - 9.1 Installing CodeIgniter Rest Server 36
 - Fixing the CodeIgniter Rest Server Code 36
 - 9.2 Testing GET Requests 36
 - Getting All of the Data 36
 - Getting One User's Data 36
 - 9.3 Adding Request Headers 36
 - 9.4 Testing POST Requests 36
 - 9.5 Testing JSON Requests 36
 - 9.6 Testing DELETE Requests 36

- 10. Browser Testing with Codeception 37**
 - 10.1 Installing and Configuring Codeception 38
 - What is Codeception? 38
 - Installing Codeception 38
 - What is Selenium Server? 38
 - Installing Selenium Server 38
 - Initializing Codeception 38
 - Configuring Acceptance Tests 38
 - 10.2 Writing Tests 38
 - Conventions for Codeception Acceptance Tests 38
 - Writing Our First Test 38
 - 10.3 Running Tests 38
 - Running Selenium Server 38
 - Running the Web Server 38
 - Running Codeception 38
 - 10.4 Browser Testing: Pros and Cons 38
 - 10.5 Database Fixtures 38
 - 10.6 Test Case for the News Controller 38
 - Database Fixtures 38
 - Testing Page Contents 38
 - Testing Forms 38
 - NewsCept 38
 - 10.7 Testing with Google Chrome 38
 - Installing the ChromeDriver 38
 - Configuring Acceptance Tests 38
 - Running Selenium Server 38
 - Running Tests 38

- 11. CodeIgniter Testing Guide 39**

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 specifically 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?

1.1 Primitive Example

Manual Testing

Automated Testing

1.2 Why should you write test code?

1.3 Finding the Middle Way

1.4 What should you test?

1.5 TDD or Not TDD

2. Setting Up the Testing Environment

2.1 Installing CodeIgniter

2.2 Installing ci-phpunit-test

Enabling Monkey Patching

2.3 (Optional) Installing VisualPHPUnit

Installing Composer

Installing VisualPHPUnit

Installing PHPUnit

Configuring VisualPHPUnit

Configuring PHPUnit XML Configuration File

2.4 Installing PHPUnit

2.5 (Optional) Installing PsySH

2.6 Installing via Composer

Installing Composer

Installing CodeIgniter via Composer

Installing ci-phpunit-test via Composer

Installing PHPUnit via Composer

Creating a phpunit Shortcut

(Optional) Installing PsySH via Composer

3. Test Jargon

3.1 Testing levels

Unit Testing

Integration Testing

System Testing

3.2 Testing Types

Functional Testing

Database Testing

Browser Testing

Acceptance Testing

3.3 Code Coverage

3.4 Fixtures

3.5 Test Doubles

Mocks and Stubs

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.

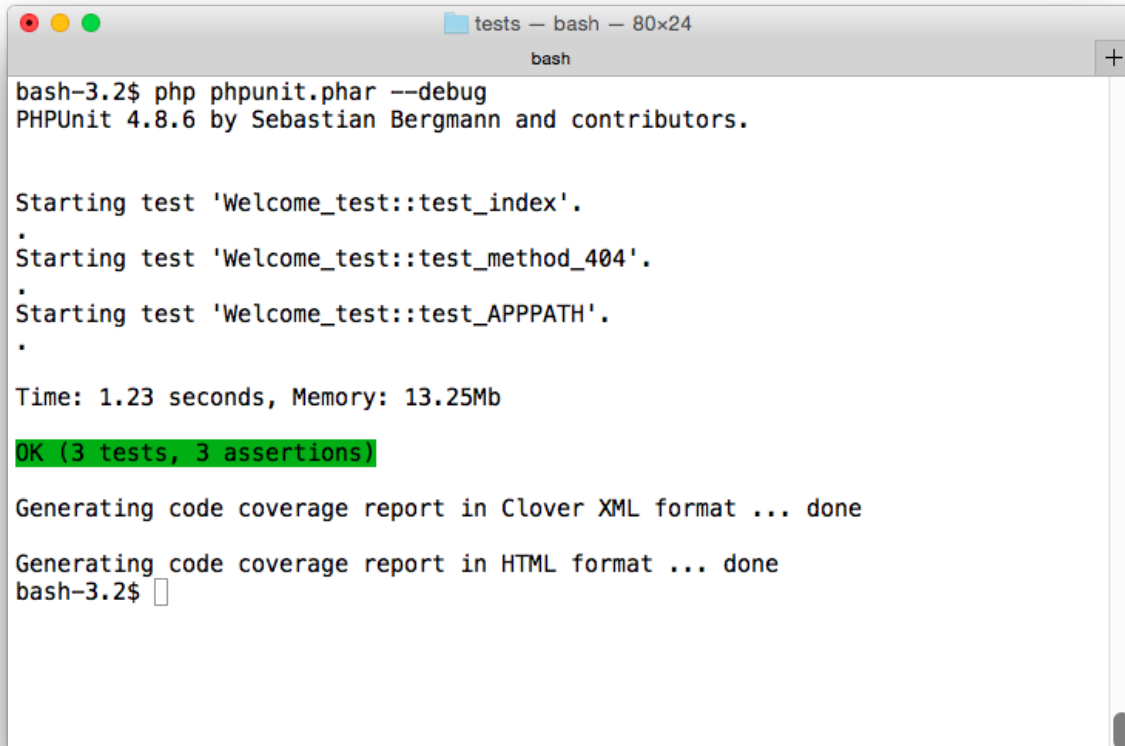
```
$ cd CodeIgniter/application/tests/  
$ php phpunit.phar --debug
```



To Composer users

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

See [Creating phpunit shortcut](#).

A screenshot of a terminal window titled "tests -- bash -- 80x24". The terminal shows the execution of PHPUnit with the --debug flag. The output includes the PHPUnit version (4.8.6), the start of three tests: 'Welcome_test::test_index', 'Welcome_test::test_method_404', and 'Welcome_test::test_APPPATH'. Each test is preceded by a dot (.) indicating it passed. The execution time is 1.23 seconds and memory usage is 13.25Mb. The final result is "OK (3 tests, 3 assertions)", which is highlighted in green in the original image. The terminal also shows messages about generating code coverage reports in Clover XML and HTML formats, both of which are completed. The prompt "bash-3.2\$" is visible at the end of the output.

```
bash-3.2$ php phunit.phar --debug
PHPUnit 4.8.6 by Sebastian Bergmann and contributors.

Starting test 'Welcome_test::test_index'.
.
Starting test 'Welcome_test::test_method_404'.
.
Starting test 'Welcome_test::test_APPPATH'.
.

Time: 1.23 seconds, Memory: 13.25Mb

OK (3 tests, 3 assertions)

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

Generating code coverage report in HTML format ... done
bash-3.2$
```

Run phunit 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](http://xdebug.org/)¹ or [phpdbg](http://phpdbg.com/)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/>

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



To Composer users

```
$ phdbg -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>.

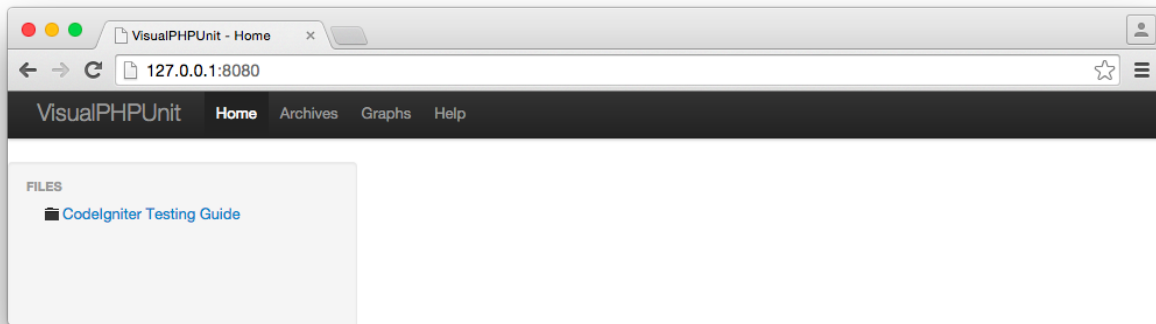
Open your terminal, navigate to the directory in which you've placed VisualPHPUnit, and type the following commands:

```
$ 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/>

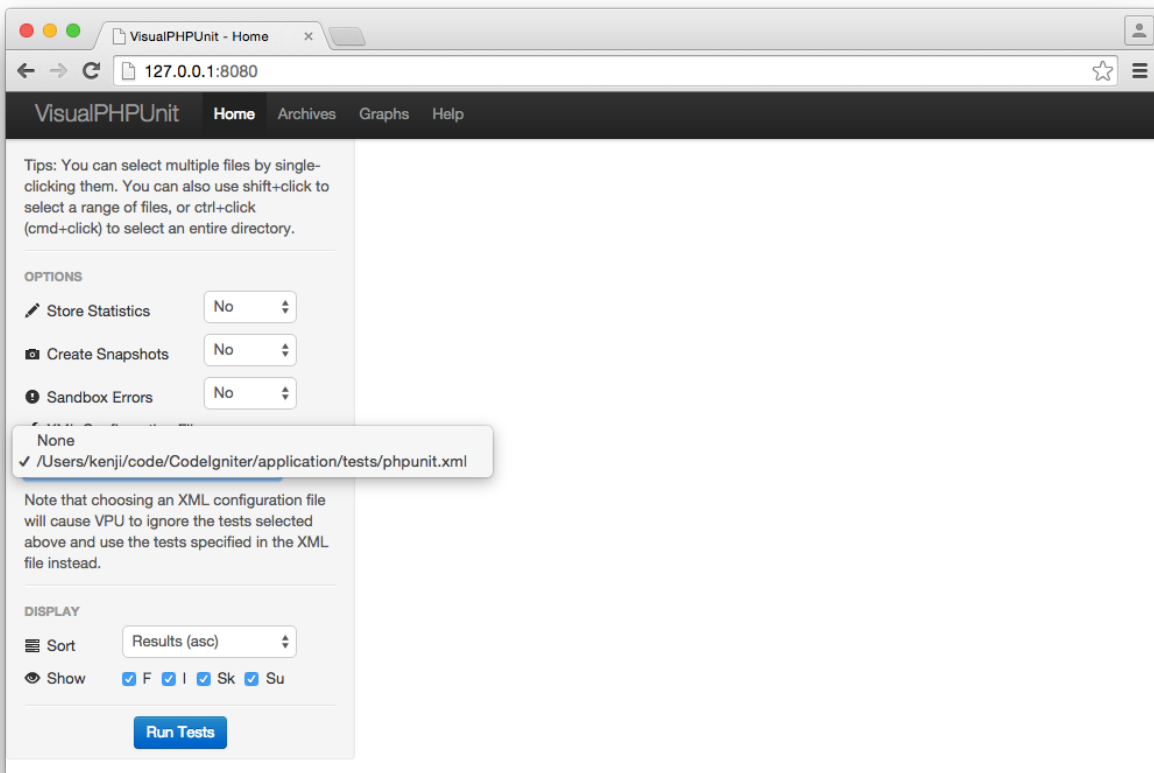
You should see the VisualPHPUnit Home page.



VisualPHPUnit Home

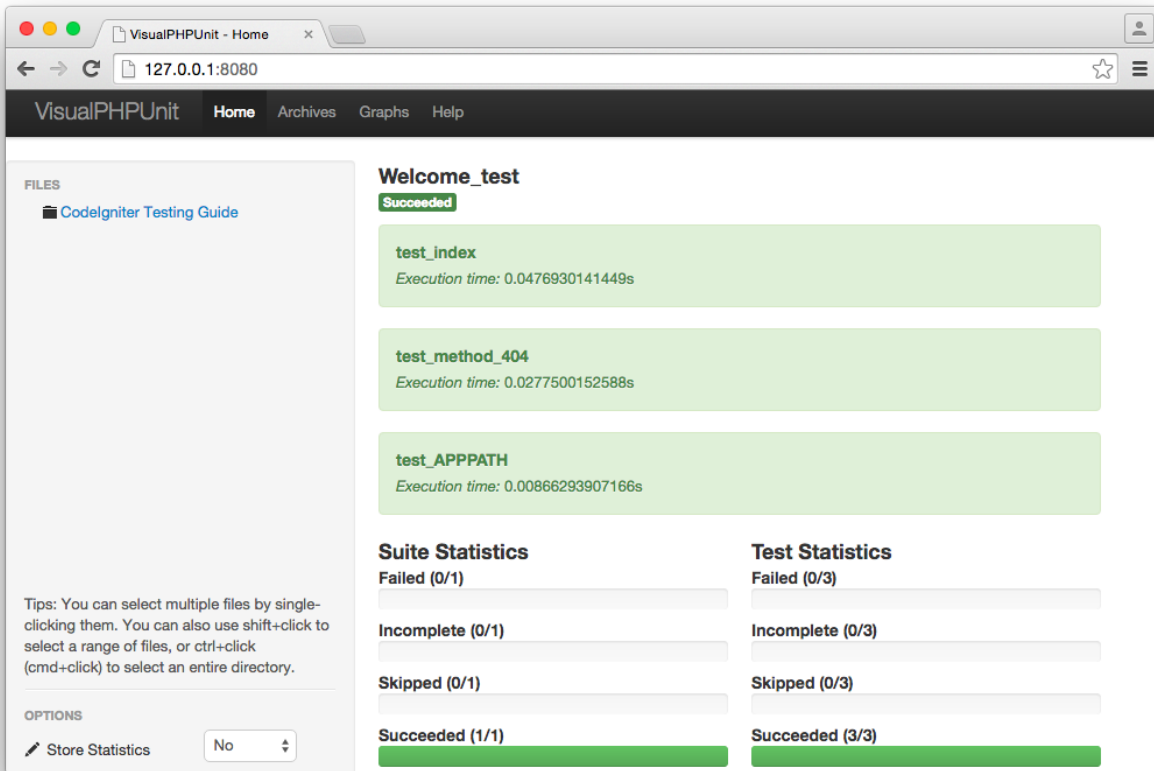
Running All Tests

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

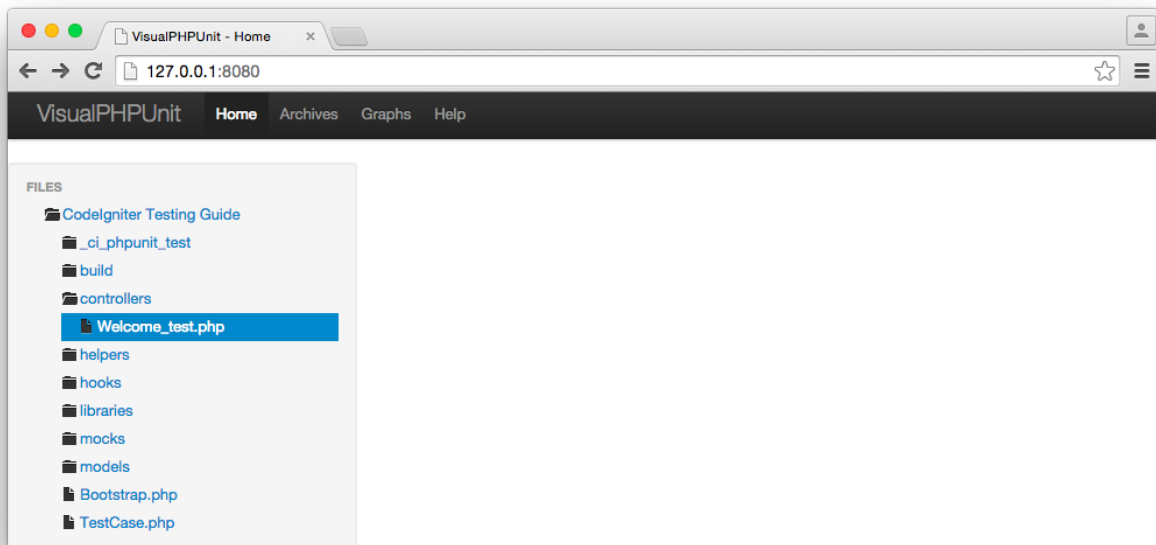
You should see the results like the following.



VisualPHPUnit Run Tests using XML Configuration File

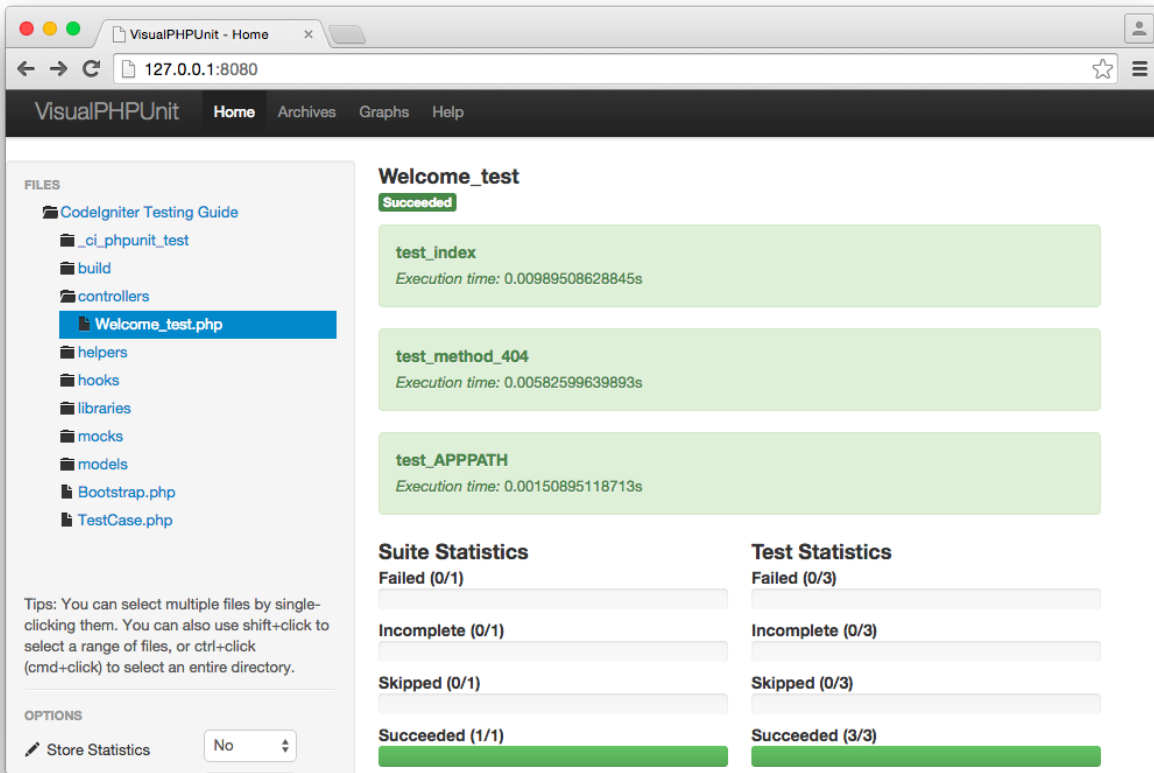
Running a Specific Test Case

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 attributes of the first element, `phpunit`, configure PHPUnit itself.

```
1 <phpunit
2     bootstrap="./Bootstrap.php"
3     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.

```
4 <testsuites>
5     <testsuite name="CodeIgniter Application Test Suite">
6         <directory suffix="test.php">./</directory>
7         <exclude>./_ci_phpunit_test/</exclude>
8     </testsuite>
9 </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.

```
10 <filter>
11     <whitelist>
12         <directory suffix=".php">./controllers</directory>
13         <directory suffix=".php">./models</directory>
14         <directory suffix=".php">./views</directory>
15         <directory suffix=".php">./libraries</directory>
16         <directory suffix=".php">./helpers</directory>
17         <directory suffix=".php">./hooks</directory>
18     </whitelist>
19 </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
```

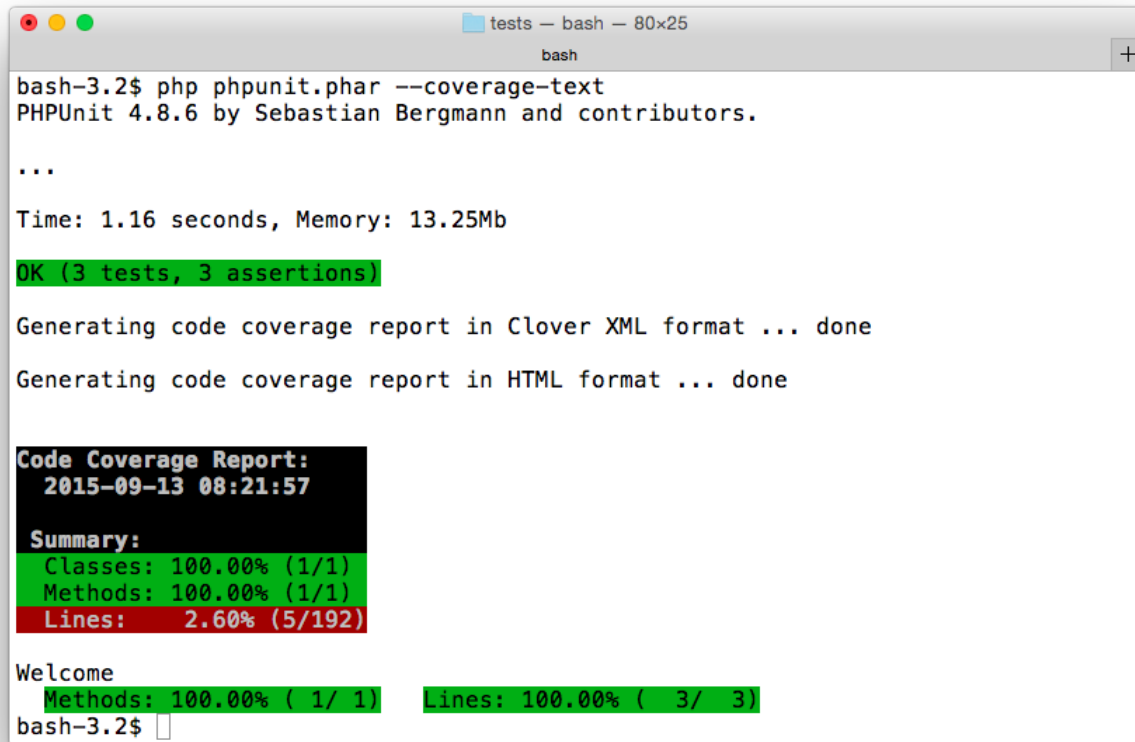
- --debug
- -v/--verbose

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.



```
tests — bash — 80x25
bash
bash-3.2$ php phunit.phar --coverage-text
PHPUnit 4.8.6 by Sebastian Bergmann and contributors.

...

Time: 1.16 seconds, Memory: 13.25Mb

OK (3 tests, 3 assertions)

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

Code Coverage Report:
 2015-09-13 08:21:57

Summary:
Classes: 100.00% (1/1)
Methods: 100.00% (1/1)
Lines: 2.60% (5/192)

Welcome
Methods: 100.00% ( 1/ 1) Lines: 100.00% ( 3/ 3)
bash-3.2$
```

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`

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



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 phunit.phar libraries/Temperature_converter_test.php
```



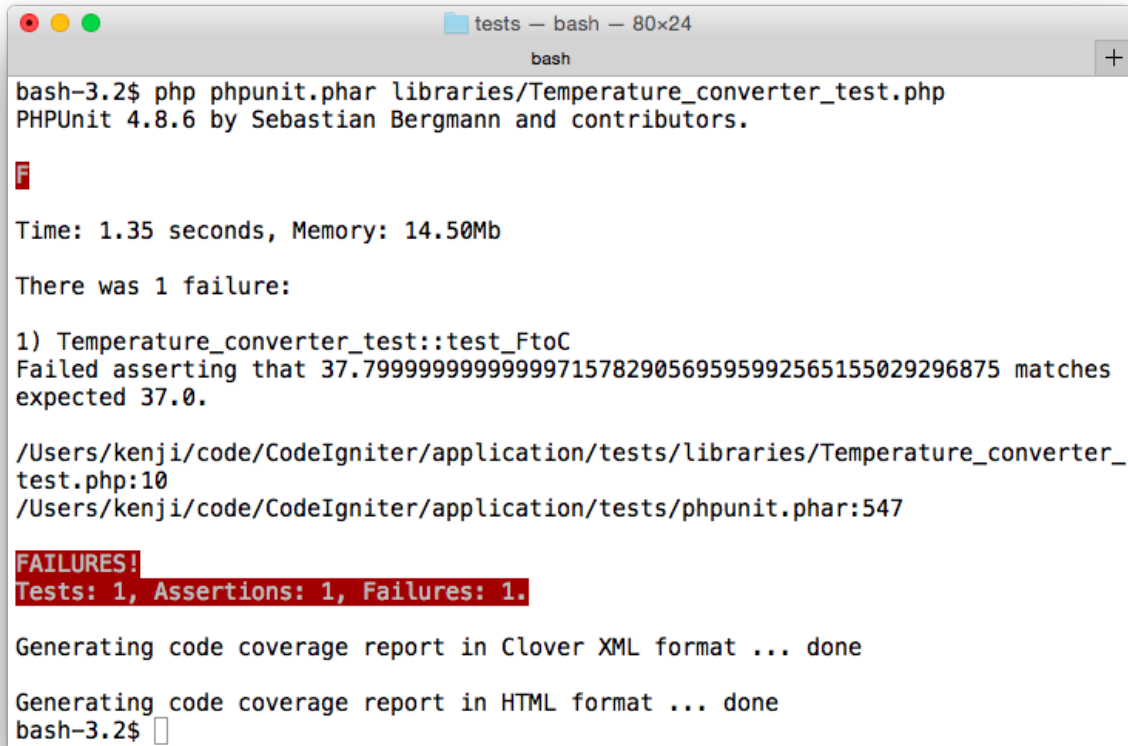
To Composer users

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

See [Creating a phpunit shortcut](#).

³http://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>



```
tests -- bash -- 80x24
bash
bash-3.2$ php phpunit.phar libraries/Temperature_converter_test.php
PHPUnit 4.8.6 by Sebastian Bergmann and contributors.

F

Time: 1.35 seconds, Memory: 14.50Mb

There was 1 failure:

1) Temperature_converter_test::test_FtoC
Failed asserting that 37.79999999999971578290569595992565155029296875 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.

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

Generating code coverage report in HTML format ... done
bash-3.2$
```

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 37.7999999999999971578290569595992565155029296875 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 phunit.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 continue adding tests like this? A common saying in programming is “Don’t repeat yourself”. Continuing to add tests in this manner seems a bit repetitive.



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

```
1 <?php
2
3 class Temperature_converter_test extends TestCase
4 {
5     /**
6      * @dataProvider provide_temperature_data
7      */
8     public function test_FtoC($degree, $expected)
9     {
10         $obj = new Temperature_converter();
11         $actual = $obj->FtoC($degree);
12         $this->assertEquals($expected, $actual, '', 0.01);
13     }
14
15     public function provide_temperature_data()
16     {
17         return [
18             // [Fahrenheit, Celsius]
19             [-40, -40.0],
20             [ 0, -17.8],
21             [ 32,  0.0],
22             [100,  37.8],
23             [212, 100.0],
24         ];
25     }
26 }
```

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 phunit.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 phunit.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\
000000000000710542735760100185871124267578125)'.
.
```

```
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\
99999999999971578290569595992565155029296875)'.
.
```

```
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` matches 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

5.1 Functional Testing for Controller

Controller to Handle Static Pages

Pages Controller

Page Templates

Static Pages

Routing

Manual Testing with a Web Browser

Test Case for Page Controller

Checking Code Coverage

5.2 Database Testing for Models

Preparing the Database

Database Configuration

SQLite

MySQL

Dedicated Test Database

Database Migration

Dbfixture Controller

News Section

News_model Model

News Controller

Views

Routing

Manual Testing with a Web Browser

Database Fixtures

6. Unit Testing for Models

6.1 Why Should You Test Models First?

6.2 PHPUnit Mock Objects

Playing with Mocks

Partial Mocks

Verifying Expectations

6.3 Testing Models without Database

Testing the get_news() Method with Mocks

Creating a Mock Object for CI_Loader

Creating Mocks for get_news()

Writing the Test Method

Creating Another Mocks

Extract the Method to Create Mocks

Writing Another Test Method

Testing the set_news() Method with Mocks

Mocks with Return Map

Writing the Test Method

6.4 With the Database or Without the Database?

Testing with Little Value

When You Write Tests without the Database

7. Testing Controllers

7.1 Why is Testing Controllers Difficult?

7.2 Test Case for the News Controller

7.3 Mocking Models

7.4 Authentication and Redirection

Installing Ion Auth

Database Migrations

Routing

Manual Testing with a Web Browser

Testing Redirection

Mocking Auth Objects

7.5 What if My Controller Needs Something Else?

8. Unit Testing CLI Controllers

8.1 Dbfixture Controller

8.2 Faking is_cli()

8.3 Testing exit()

8.4 Testing Exceptions

8.5 Testing Output

8.6 Monkey Patching

Patching Functions

Patching Class Methods

8.7 Checking Code Coverage

9. Testing REST Controllers

9.1 Installing CodeIgniter Rest Server

Fixing the CodeIgniter Rest Server Code

9.2 Testing GET Requests

Getting All of the Data

Getting One User's Data

9.3 Adding Request Headers

9.4 Testing POST Requests

9.5 Testing JSON Requests

9.6 Testing DELETE Requests

10. Browser Testing with Codeception

10.1 Installing and Configuring Codeception

What is Codeception?

Installing Codeception

What is Selenium Server?

Installing Selenium Server

Initializing Codeception

Configuring Acceptance Tests

Testing with Firefox

10.2 Writing Tests

Conventions for Codeception Acceptance Tests

Writing Our First Test

10.3 Running Tests

Running Selenium Server

Running the Web Server

Running Codeception

10.4 Browser Testing: Pros and Cons

10.5 Database Fixtures

10.6 Test Case for the News Controller

Database Fixtures

Testing Page Contents

Comments

11. CodeIgniter Testing Guide

Thank you for evaluating this sample of *CodeIgniter Testing Guide*.

To purchase this book, please visit <https://leanpub.com/codeigniter-testing-guide>.