

11

PHPUNIT

OVERVIEW

In this chapter, we will install and configure PHPUnit. We will look at the format of tests and assertions and examine their usage. By the end of this chapter, you will be able to add PHPUnit to a project as well as understand how to write and run test suites.

INTRODUCTION

So far in this book, we have written programs for building various applications, handling exceptions, working with web services, and a number of other purposes. In a business use case, each unit of code needs to be tested for accuracy. This is where unit testing comes into the picture.

In this chapter, we will learn how to test various units of code. We will install PHPUnit and see its usage in detail.

PHPUnit 8.4 requires PHP 7.2 or higher; in this chapter, we will be covering PHPUnit 8.4.

UNIT TEST

A unit test is a small piece of code that your application uses. The idea is to break down your code base into small units. A unit is the smallest component of code. Normally, a unit is made from class methods to test the logic of some code. The purpose of a unit test is to verify that the output works as expected.

You can run multiple tests in one go. The advantage with this approach is that if you run the tests and they all pass, you can then refactor your code and run the tests again to ensure that nothing has been broken. Since each action could be a separate test, you can test against lots of different use cases with the confidence that if you make a change, you know that your code is still stable and runs as expected.

Alternatively, you may want to run the test before the code is finished, in order to see that the test fails, and then run it again once the code is finished to see that the test passes. This way, you would be doing **Test-Driven Development (TDD)**, a practice that we'll explain later on.

To illustrate what a unit test is, the following is an example test. Its purpose is to look at an `$item` array and verify whether it contains an index called `name`. First, the array is defined; then, the variables are set up to state what's expected and where to look for a value. Finally, `assertArrayHasKey` runs this and verifies that the expected value has been found in the array:

```
/** @test */
public function arrayHasKeyName()
{
    //setup
    $item = [
        'name' => 'dave'
    ]
}
```

```

    ];

    //act
    $expected = 'name';
    $actual = $item;
    //assert
    $this->assertArrayHasKey($expected, $actual);
}

```

As long as the **name** index is found in the array, the test will pass and show a green light:

PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

.

1 / 1 (100%)

Time: 30 ms, Memory: 4.00 MB

OK (1 test, 1 assertion)

Figure 11.1: Passed test

Likewise, if **name** is not found in the array, the test will fail:

PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

F

1 / 1 (100%)

Time: 24 ms, Memory: 4.00 MB

There was 1 failure:

1) DcTest::arrayHasKeyNameAttribute
Failed asserting that an array has the key 'name'.

/Users/dave/Documents/Documents - daveismyname/packtphp/chatper11/demo/tests/DcTest.php:20

FAILURES!

Tests: 1, Assertions: 1, Failures: 1.

Figure 11.2: Failed test

WHAT IS PHPUNIT?

PHPUnit is a framework-agnostic testing framework; this means it's not reliant on any third-party framework. In fact, no framework is required apart from PHPUnit. PHPUnit can be used with a group of classes as well as an existing framework.

PHPUnit is a tool for running tests. Unit testing is where code is split into units – small chunks of the minimum code necessary to perform a function – and then tested against expected results.

INSTALLING PHPUNIT

There are two ways to install PHPUnit: using a **PHP Archive (PHAR)** file and with Composer.

EXERCISE 11.1: INSTALLING PHPUNIT WITH PHAR

A PHAR file contains all the files needed for PHPUnit to run.

To install PHPUnit with PHAR, open Terminal and use the **wget** command to download a specific version. At the time of writing, 8.4 is the latest release:

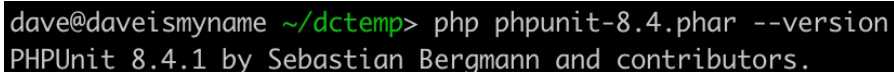
1. Open Terminal and download the latest release using the following code:

```
wget https://phar.phpunit.de/phpunit-8.4.phar
```

2. This will download a file called **phpunit-8.4.phar**. You can verify the file by running the following command in Terminal:

```
php phpunit-8.4.phar --version
```

The output will look as follows:



```
dave@daveismyname ~/dctemp> php phpunit-8.4.phar --version
PHPUnit 8.4.1 by Sebastian Bergmann and contributors.
```

Figure 11.3: Checking the file version

This tells us that we are running PHPUnit 8.4.1.

3. Optionally, you can make the PHAR file executable, meaning you don't need to specify **php** before calling the file. To do this, change the file permissions using a command called **chmod**, which is a command used to change file permissions for both files and folders. In Terminal, type the following command:

```
chmod +x phpunit-8.4.phar
```

This changes the file permissions to allow the file to be executable. Then, you can simply call `./phpunit-8.4.phar` to run it.

In this exercise, we saw how to install PHPUnit on our system via PHAR.

EXERCISE 11.2: INSTALLING PHPUNIT WITH COMPOSER

Since Composer was covered in an earlier chapter, this process will be familiar. In this exercise, we will install PHPUnit with Composer:

1. To install PHPUnit using Composer, add **phpunit/phpunit** to your **composer.json** file (located in the project root directory). Add **phpunit/phpunit** to the **required-dev** section. This tells Composer to only require this in developer mode:

```
{
  "require-dev": {
    "phpunit/phpunit": "^8"
  }
}
```

2. Next, in Terminal, run the following command to install Composer:

```
composer install
```

3. If you are adding this to an existing project update, run the following:

```
composer update
```

4. Another way is to tell Composer to add PHPUnit as a dependency. In Terminal, type the following:

```
composer require --dev phpunit/phpunit ^8
```

This will update your **composer.json** file to the version specified.

5. From this point onward, PHPUnit will be able to be run by calling its executable in Terminal:

```
vendor/bin/phpunit
```

NOTE

You may be tempted to install PHPUnit globally; this is not recommended as each project should manage PHPUnit as a project-specific dependency.

For the rest of this chapter, we will be using Composer to manage PHPUnit.

EXERCISE 11.3: SETTING UP A TESTING ENVIRONMENT

In order to run tests, a place to store the test scripts is needed. As Composer is going to be used, autoloading can be used to load any classes that are needed. Here, we will set up a basic folder structure that can be used as a boilerplate for future projects:

1. Create a project folder named **demo** and store all the folders and files within it. Create the following folders and files:

```
app/  
tests/  
composer.json  
index.php
```

2. Open **composer.json** and add the following JSON code:

```
{  
    "require-dev": {  
        "phpunit/phpunit": "^8"  
    },  
    "autoload": {  
        "psr-4": {  
            "App\\": "app/"  
        }  
    }  
}
```

Composer will install PHPUnit as a dependency to the project and also use **autoload** to autoload files within an **app/** folder under the namespace of **App**. Composer's autoloading feature allows the calling of classes inside **app/** by calling the **App** namespace followed by the class name.

3. Open **index.php** and add the following:

```
<?php require('vendor/autoload.php');
```

This requires the Composer autoloader and adds the ability to autoload files as needed for use later when you want to build an application and run any classes.

4. Open Terminal, navigate to the project, and tell Composer to install PHPUnit and its dependencies as follows:

```
composer install
```

This will install PHPUnit and set up the autoloader. A new folder will be created called **vendor**. This is where all dependencies of the composer are stored. The folder structure will look as follows:

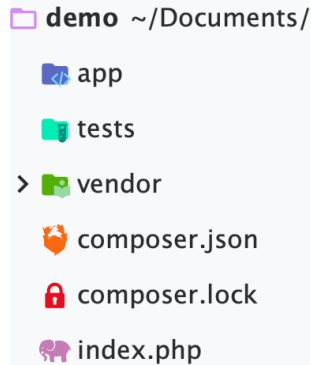


Figure 11.4: The folder structure

The **composer.lock** file is automatically generated when **composer install** is run. This is a cached file of the Composer settings and should not be edited directly.

With this folder structure now in place, it can be used to house all test scripts in the **tests** folder, any classes needed can be autoloaded into the **app** folder, and any third-party code can be installed with Composer. This is all that's needed to start using PHPUnit.

WRITING OUR FIRST TEST

All test classes extend from **PHPUnit\Framework\TestCase** to ensure that you can access all of PHPUnit's test methods and assertions. Tests should be placed within the **tests** folder.

The filename should be saved as Pascal Case. The first letter of each word should be a capital letter and the letters in between should be lowercase. Also, the filename should end with **Test.php**. In this section, we are making a class called **DemoTest**, so the filename should be saved in the **tests** folder as **DemoTest.php**.

Method names should start with **Test** and be followed by what you're testing.

EXERCISE 11.4: CREATING A TEST CLASS

Let's create a very basic test class to introduce what a test looks like:

1. First, import **PHPUnit\Framework\TestCase** as follows:

```
use PHPUnit\Framework\TestCase;
```

2. Create a class that has a suffix of **Test** in the name and extends from **TestCase**:

```
class DemoTest extends TestCase
```

3. Create a method called **public function isTheSame()**. A test can have many methods, and each method can be an individual test. In this example, there is only one test. The methods should start with **test** and be followed by the method name. An alternative approach is to use annotations. Adding **@test** in a DocBlock will inform PHPUnit that the method is a test. This allows you to remove **test** from the method name:

```
<?php
use PHPUnit\Framework\TestCase;
class DemoTest extends TestCase
{
    /**
     * @test
     */
    public function isTheSame()
    {
        $hasValue = false;
        $this->assertSame(true, $hasValue);
    }
}
```

4. Each test can call one of the assertion methods (more on assertions in the next section), such as **assertSame**, which takes two values and compares them to see whether they are the same. In this case, both values need to be **true** for the test to pass:

```
<?php
use PHPUnit\Framework\TestCase;
class DemoTest extends TestCase
{
    public function testIsTheSame()
```



```

    {
        $hasValue = false;
        $this->assertSame(true, $hasValue);
    }
}

```

This test will fail as the first value is **true** but the second is **false**.

5. This test can be run by typing the following into Terminal:

```
vendor/bin/phpunit tests/DemoTest.php.
```

6. To run all tests within the **tests** folder, execute the following:

```
vendor/bin/phpunit tests
```

Upon running, all methods inside **DemoTest** will be executed.

When a test fails, an **F** is printed; if a test passes, a **.** is printed. The time it took to run the test is printed, as well as how much memory was needed to run the tests.

See *Figure 11.5* for the output from Terminal:

```

dave@daveismyname ~/D/p/p/c/demo> vendor/bin/phpunit tests/demotest
PHPUnit 8.3.5 by Sebastian Bergmann and contributors.

F                                                                    1 / 1 (100%)

Time: 61 ms, Memory: 4.00 MB

There was 1 failure:

1) DemoTest::testIsTheSame
Failed asserting that false is identical to true.

/Users/dave/Documents/packt/php/chatper11/demo/tests/demotest.php:10

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

```

Figure 11.5: Result of the unit test

When there is a failure, PHPUnit will inform you. In this case, the failure was **DemoTest::testIsTheSame**. This informs us of the location of the class and method that caused the failure.

THE THREE "A"S FOR UNIT TESTING

There are three fundamental methodologies for writing unit tests.

ARRANGE

Organize your code and set up the primary configuration – for instance, instantiate a class, assign data to a variable, or create an empty array. The idea is to set the stage before acting on it.

ACT

"Act" refers to performing an action, such as populating an array or calling upon code to perform some other action.

ASSERT

Asserting is where you compare the actual results with the expected results. For example, does $X = Y$ indicate equality or is $X = \text{true}$? Take a look at the following:

```
/** @test */
public function arrayPush()
{
    //arrange
    $items = [];

    //act
    $items[] = 'First Item';
    $items[] = 'Second Item';
    $items[] = 'Third Item';

    //assert
    $this->assertCount(3, $items);
}
```

The preceding code is an example of the three "A"s in action. First, an array is arranged (meeting the first "A"), then the array is populated with strings (meeting the second "A"), and, finally, there is an **assert** function to determine whether the array contains three items.

ASSERTIONS

An assertion asserts that a value matches an expected value or does an expected action. PHPUnit comes with assertion methods, which can be used by calling **`$this->`** or **`self::`** followed by the method name.

Another option is to manually include **`src/Framework/Assert/Functions.php`** in your tests. Then, you can call the functions directly. So, instead of **`$this->assertTrue()`**, you would call **`assertTrue()`**.

However, in this chapter, we will only be using the **`$this`** variant. The full list of assertions can be found in the documentation: <https://phpunit.readthedocs.io/en/8.4/assertions.html>.

The following are some of the more popular assertions that are available.

The assertion methods expect the following parameters:

- **`$expected`**: The value that is expected.
- **`$actual`**: The actual value.
- **`$message`**: This is optional but, when used, will be printed with the output. Usually, we use this to add more context.

Here are some of the assertion methods. Remember that these assertions will run only when a test is run:

- **`assertEquals`**: Asserts that the value of the **`$actual`** variable is equal to the **`$expected`** variable. Here's the syntax:

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

- **`assertContains`** (should only be used for arrays): Asserts that the value of the **`$expected`** variable is inside the **`$actual`** array. Consider the following example:

```
$expected = 'php';
$actual = ['html', 'php', 'mysql'];
$this->assertContains($expected, $actual);
```

- **assertStringContainsString:** Asserts that the value of the **\$expected** variable is inside the **\$actual** string. Here's an example:

```
$expected = 'dave';  
$actual = 'dave is in this string';  
$this->assertStringContainsString($expected, $actual);
```

- **assertIsArray:** Asserts that **\$actual** is an array, and so this passes when the type is an array. Here's an example:

```
$actual = [];  
$this->assertIsArray($actual);
```

- **assertIsInt:** Asserts that the value of **\$actual** is an integer, and so this passes when the type is an integer. Consider the following example:

```
$actual = 1;  
$this->assertIsInt($actual);
```

- **assertTrue:** Asserts that the **\$actual** value is **true**. Consider the following example:

```
$actual = true;  
$this->assertTrue($actual);
```

- **assertFalse:** Asserts that the **\$actual** value is **false**:

```
$actual = false;  
$this->assertFalse($actual);
```

EXERCISE 11.5: USING ASSERTIONS WITH TDD

So far, we've looked at assertions, which are used to ensure that our tests pass when the results match what we expect. It turns out that this lends itself really well to a concept called TDD.

The idea behind TDD is that you write your test before writing your code. This way, you can design how you want your code to function before you write it, making it easier to read and plan out.

Another benefit is that you can write minimum-viable tests. This means writing the smallest amount of code necessary to make a test pass; then, once it passes, the code can be improved upon and tested again to ensure that it still works.

For this exercise, let's write tests to examine pages that come from a **Pages** class, which is yet to be written. This will demonstrate both TDD and the usage of assertions:

1. In the **tests** folder, create a new file called **PagesTest.php**. Import **PHPUnit\Framework\TestCase** and define a class called **PagesTest** that extends **TestCase**:

```
<?php
use PHPUnit\Framework\TestCase;
class PagesTest extends TestCase
{

}
```

2. Add a test called **homePageReturnsHello**. The method should create a class called **Pages**. The expected value will be **Hello** and **\$actual** will get the value from **\$pages->getPage(1)**. Finally, run **assertEquals** to check that **\$expected** matches **\$actual**:

```
/** @test */
public function homePageReturnsHello()
{
    $pages = new App\Pages();
    $expected = 'Hello';
    $actual = $pages->getPage('1');
    $this->assertEquals($expected, $actual);
}
```

3. Run the test, which will fail, but that's good because that's what we expected:

```
vendor/bin/phpunit tests -colors=auto
```

The output is as follows:

```
dave@daveismyname ~/D/p/p/c/demo> vendor/bin/phpunit tests --colors=auto
PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

.E                                                                    2 / 2 (100%)

Time: 22 ms, Memory: 4.00 MB

There was 1 error:

1) PagesTest::homePageReturnsHello
Error: Class 'App\Pages' not found

/Users/dave/Documents/packt/php/chatper11/demo/tests/PagesTest.php:10

ERRORS!
Tests: 2, Assertions: 1, Errors: 1.
```

Figure 11.6: Test failed

The test fails as the **Pages** class does not yet exist.

4. Create a new class inside **App** called **Pages.php**:

```
<?php
namespace App;
class Pages
{

}
```

5. Save the file and run the test again:

```
vendor/bin/phpunit tests -colors=auto
```

The output is as follows:

```
dave@daveismyname ~/D/p/p/c/demo> vendor/bin/phpunit tests --colors=auto
PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

.E                                                                    2 / 2 (100%)

Time: 23 ms, Memory: 4.00 MB

There was 1 error:

1) PagesTest::homePageReturnsHello
Error: Call to undefined method App\Pages::getPage()

/Users/dave/Documents/packt/php/chatper11/demo/tests/PagesTest.php:13

ERRORS!
Tests: 2, Assertions: 1, Errors: 1.
```

Figure 11.7: Test failure because of a non-existing method

The test fails again - this time because the **getPage** method does not exist.

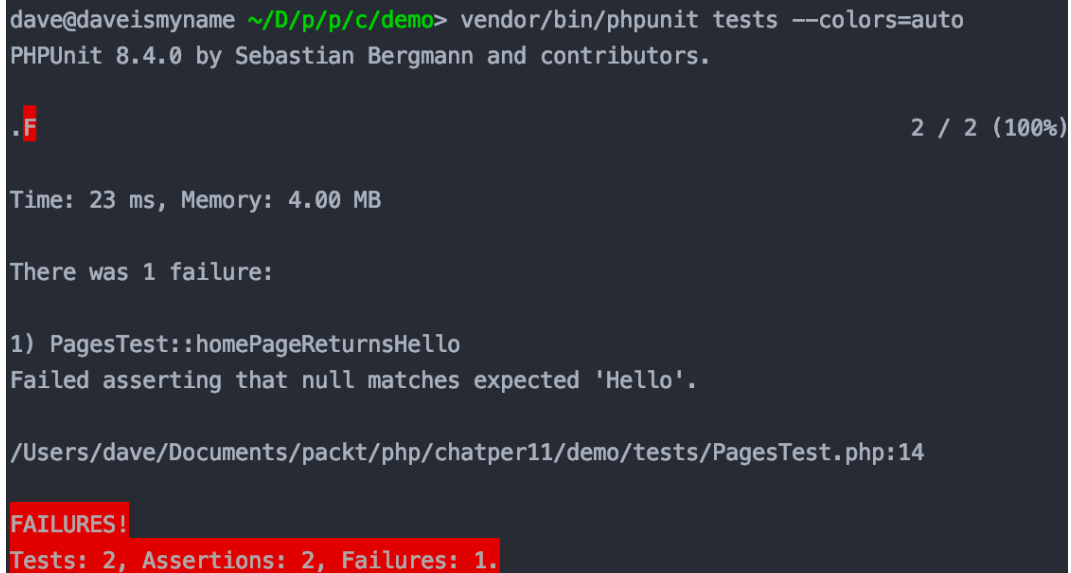
6. In **app\Pages.php**, add a **getPage(\$id)** method:

```
<?php
namespace App;
class Pages
{
    public function getPage($id)
    {
    }
}
```

7. Run the test again:

```
vendor/bin/phpunit tests -colors=auto
```

The output is as follows:



```
dave@daveismyname ~/D/p/p/c/demo> vendor/bin/phpunit tests --colors=auto
PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

.F                                                                    2 / 2 (100%)

Time: 23 ms, Memory: 4.00 MB

There was 1 failure:

1) PagesTest::homePageReturnsHello
Failed asserting that null matches expected 'Hello'.

/Users/dave/Documents/packt/php/chatper11/demo/tests/PagesTest.php:14

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

Figure 11.8: No output returned

Again, the test fails, but this time it knows that the class and method exist. However, the output is wrong. It expects the word **Hello** to be returned but no output exists.

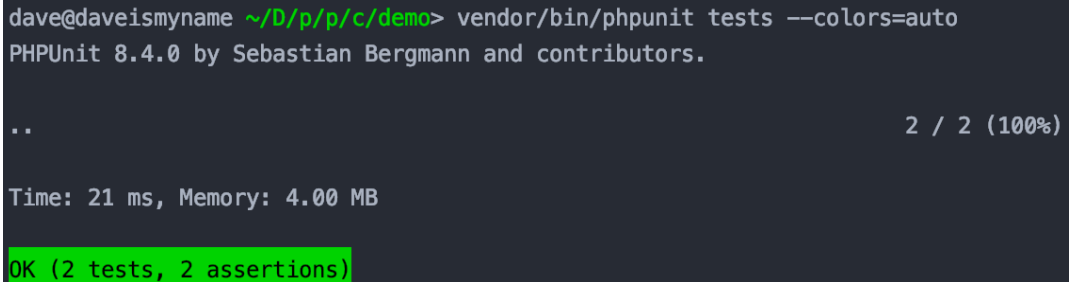
8. Add the following output to the method:

```
<?php
namespace App;
class Pages
{
    public function getPage($id)
    {
        return 'Hello';
    }
}
```


9. Run the test again:

```
vendor/bin/phpunit tests -colors=auto
```

The output is as follows:

A terminal window with a dark background. The prompt is 'dave@daveismyname ~/D/p/p/c/demo>'. The command entered is 'vendor/bin/phpunit tests --colors=auto'. The output shows 'PHPUnit 8.4.0 by Sebastian Bergmann and contributors.', followed by two dots '..' and '2 / 2 (100%)' on the right. Below that is 'Time: 21 ms, Memory: 4.00 MB'. The final line is 'OK (2 tests, 2 assertions)' which is highlighted in green.

```
dave@daveismyname ~/D/p/p/c/demo> vendor/bin/phpunit tests --colors=auto
PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

..                                                                    2 / 2 (100%)

Time: 21 ms, Memory: 4.00 MB

OK (2 tests, 2 assertions)
```

Figure 11.9: Test successful

Success! The test passes as it's returning the expected word, **Hello**. Now that we know the test passes, we can refactor the **Pages** class to be a bit more like a real-world class.

10. Open the **Pages** class and change the method to the following. In this case, a **switch** statement can be used that simulates, say, a database lookup. If the ID passed is **1**, then return **Hello**; if **2** is passed, return **About**:

```
<?php
namespace App;
class Pages
{
    public function getPage($id)
    {
        switch ($id) {
            case '1':
                return 'Hello';
                break;
            case '2':
                return 'About';
                break;
        }
        return 'no page found';
    }
}
```

Running the test again, the test still passes. The refactor was successful:

```
dave@daveismyname ~/D/p/p/c/demo> vendor/bin/phpunit tests --colors=auto
PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

..                                                                  2 / 2 (100%)

Time: 23 ms, Memory: 4.00 MB

OK (2 tests, 2 assertions)
```

Figure 11.10: Refactor successful

11. Open the **PagesTest.php** file and change **getPage(1)** to **getPage(2)**. This will cause the test to fail as the word **Hello** will not be returned. Run the test again:

```
vendor/bin/phpunit tests -colors=auto
```

The output is as follows:

```
dave@daveismyname ~/D/p/p/c/demo> vendor/bin/phpunit tests --colors=auto
PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

.F                                                                    2 / 2 (100%)

Time: 54 ms, Memory: 4.00 MB

There was 1 failure:

1) PagesTest::homePageReturnsHello
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'Hello'
+'About'

/Users/dave/Documents/packt/php/chatper11/demo/tests/PagesTest.php:14

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

Figure 11.11: Output on passing the invalid value

The test fails; this time, it shows that it expected **Hello** but actually got **About**.

This exercise has demonstrated how an assertion can be used and also how TDD helps you to decide on how code should be run and then write the code necessary for success.

Notice how each time the test was run, we needed to specify the `--colors` option. Let's fix that in the next section.

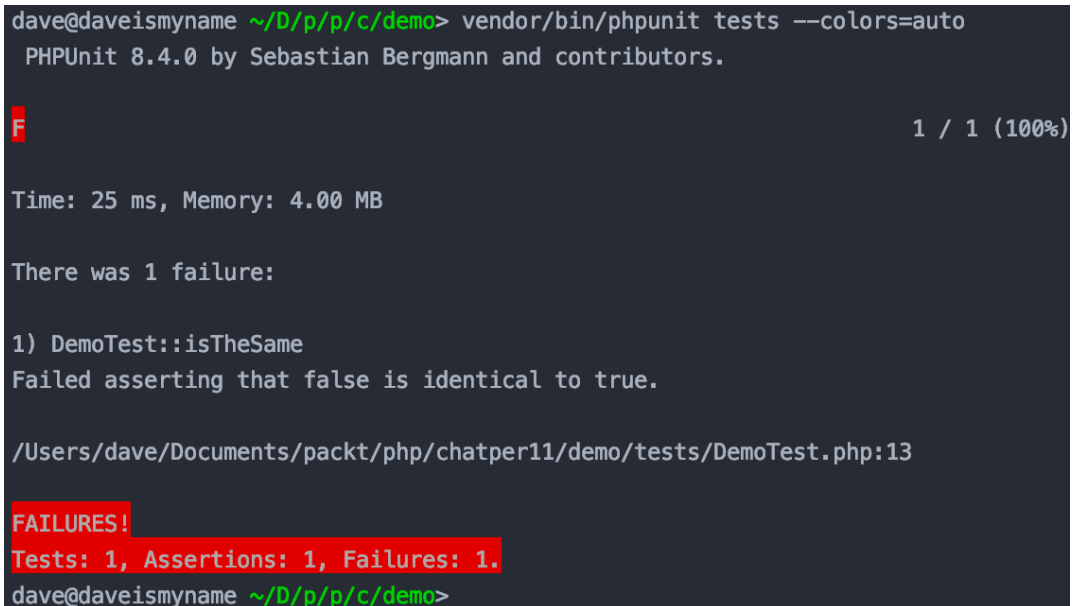
CONFIGURING PHPUNIT WITH PHPUNIT.XML

PHPUnit comes with lots of options that can be used when calling `phpunit`. For instance, so far, any tests that have been executed have returned a black-and-white output. PHPUnit supports adding color to the output by passing a `--colors` flag and specifying an option: **never**, **auto**, or **always**.

Consider the following example:

```
vendor/bin/phpunit tests --colors=auto
```

This would render the following:



```
dave@daveismyname ~/D/p/p/c/demo> vendor/bin/phpunit tests --colors=auto
PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

F                                                                    1 / 1 (100%)

Time: 25 ms, Memory: 4.00 MB

There was 1 failure:

1) DemoTest::isTheSame
Failed asserting that false is identical to true.

/Users/dave/Documents/packt/php/chatper11/demo/tests/DemoTest.php:13

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
dave@daveismyname ~/D/p/p/c/demo>
```

Figure 11.12: Output on passing the `--colors` flag

The preceding failure is highlighted in red; this makes errors much more obvious.

Another common option is to set the path to a script that loads before any tests have run, such as a **bootstrap** file for autoloading. Consider the following example:

```
vendor/bin/phpunit tests --bootstrap vendor/autoload.php
```

Now, if you wanted to use colors and specify the **bootstrap** file, the preceding command would be extended to the following:

```
vendor/bin/phpunit tests --bootstrap vendor/autoload.php --colors=auto
```

Every option added means longer lines of options to add to the commands, which then need typing out each time a test is to be executed. Thankfully, PHPUnit supports a configuration file to house all your options, so they only need to be specified once.

To create this configuration file, create a new file in the project root called **phpunit.xml**. Open the file and write the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit>
  <testsuites>
    <testsuite name="Test suite">
      <directory>tests</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

This is the starter template needed to get started. As the file is an XML file, the first line declares **xml**, its version, and the encoding used.

Next, a **phpunit** node is defined with no options (they will be added later).

Inside the **phpunit** node, a new node called **testsuites** is defined, which stores all the test suites. Only one has been defined here, with a name of **Test suite**. Notice that the directory points to our **tests** folder.

With this in place, instead of typing **vendor/bin/phpunit tests**, you can just type **vendor/bin/phpunit**.

All tests within the **tests** folder will still be tested without you having to specify them.

Back in **phpunit.xml**, options can be added by specifying them inside the **phpunit** node:

```
<phpunit bootstrap="vendor/autoload.php"
    colors="true"
    verbose="true"
    stopOnFailure="false">
```

The complete **phpunit** file now looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit bootstrap="vendor/autoload.php"
    colors="true"
    verbose="true"
    stopOnFailure="false">
    <testsuites>
        <testsuite name="Test suite">
            <directory>tests</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

You can run PHPUnit from Terminal with the following command:

```
vendor/bin/phpunit
```

Here, PHPUnit will apply all settings configured inside **phpunit.xml**. The **vendor/autoload.php** file will run before the tests are executed, the **colors** and **verbose** modes are turned on, and **stopOnFailure** is turned off.

When **verbose** mode is turned on, more detailed output will be printed.

stopOnFailure is set to **false**, so if there is an error, the testing of the remaining tests will not stop. If this was set to **true**, then all testing would stop at the first failure.

A complete list of all configuration options can be found in the documentation at <https://phpunit.readthedocs.io/en/8.4/configuration.html>.

TESTDOX

Another option available to PHPUnit is **testdox**. It looks at all the tests and their methods and converts the camel-case names to sentences. For example, the method name **homePageReturnsHello**, becomes "Home page returns hello." In addition to this, a red **x** or green tick is displayed before the method name output and is grouped under the test name:

```
dave@daveismyname ~/D/p/p/c/demo> vendor/bin/phpunit
PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.3.7
Configuration: /Users/dave/Documents/packt/php/chatper11/demo/phpunit.xml

Demo
✓ Is the same  8 ms

Pages
✓ Home page returns hello  2 ms
✓ About page returns about  1 ms

Time: 72 ms, Memory: 4.00 MB

OK (3 tests, 3 assertions)
dave@daveismyname ~/D/p/p/c/demo>
```

Figure 11.13: Groupings under the test name

TestDox can be turned on by adding the **testdox="true"** option to the **phpunit** node in **phpunit.xml**.

SETUP AND TEARDOWN

PHPUnit has two methods that provide ways to run code before and after each test has run. Each test should be its own unit so that it does not matter what order tests are run in. This means no test should rely on other tests. Each test should start with a clean slate.

In an earlier exercise, we created an instance of each test within the same class. This in itself is not necessarily bad, but it can be avoided. For example, take these two tests:

```
/** @test */
public function homePageReturnsHello()
{
    $pages = new App\Pages();
    $expected = 'Hello';
    $actual = $pages->getPage('1');
    $this->assertEquals($expected, $actual);
}

/** @test */
public function aboutPageReturnsAbout()
{
    $pages = new App\Pages();
    $expected = 'About';
    $actual = $pages->getPage('2');
    $this->assertEquals($expected, $actual);
}
```

They both create an instance of the **App\Pages**. This can be avoided by using a **setUp()** method, where the class is instantiated and then the tests refer to the class property:

```
protected $pages;
protected function setUp(): void
{
    $this->pages = new App\Pages();
}
```

Here are the two steps:

```
public function homePageReturnsHello()
{
    $expected = 'Hello';
    $actual = $this->pages->getPage('1');
    $this->assertEquals($expected, $actual);
}

/** @test */
public function aboutPageReturnsAbout()
{

```

```
$expected = 'About';  
$actual = $this->pages->getPage('2');  
$this->assertEquals($expected, $actual);  
}
```

Now, the **setUp** method sets up the **\$pages** property with an instance of **App\Pages** and the test refers to **\$this->pages** instead. This approach makes the **test** method a little cleaner and still allows each test to be isolated from other tests.

On the flip side, a **tearDown()** method can be used to unset any properties or free up any resources that may have been required during the tests.

EXERCISE 11.6: USING SETUP FOR TESTS

In this exercise, let's run a series of tests that look at the same array of data to determine whether the array has specific keys and to check that it contains only two keys:

1. Create a new class called **ArrayTest.php** inside **tests**. Import **PHPUnit\Framework\TestCase** and define a class called **ArrayTest** that extends **TestCase**:

```
<?php  
use PHPUnit\Framework\TestCase;  
class ArrayTest extends TestCase  
{  
  
}
```

2. Create a protected property called **\$items** and create a protected method called **setUp**. Remember to add **:void** to the end of the method definition, otherwise, PHPUnit will complain that the method is not compatible. Assign an array to **\$this->items** with two keys, **name** and **email**:

```
<?php  
use PHPUnit\Framework\TestCase;  
class ArrayTest extends TestCase  
{  
    protected $items;  
    protected function setUp(): Void  
    {  
        $this->items = [  
            'name' => 'dave',  
            'email' => 'dave@daveismyname.blog'  
        ];  
    }  
}
```



```
    ];
    }
}
```

With this setup, any tests within the **test** class will have access to this array, allowing it to be shared across tests.

3. Create a test called **hasKeyName**. This test should use **assertArrayHasKey**, the expected name will be **name**, and the value will be the array defined in **setUp()**:

```
/** @test */
public function arrayHasKeyName()
{
    $expected = 'name';
    $actual = $this->items;
    $this->assertArrayHasKey($expected, $actual);
}
```

This test will pass as long as the array has a key called **name**; if it does not, then it will fail.

4. Run this test as follows:

```
vendor/bin/phpunit tests/ArrayTest.php
```

The output is as follows:

```
dave@daveismyname ~/D/p/p/c/demo> vendor/bin/phpunit tests/ArrayTest.php
PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.3.7
Configuration: /Users/dave/Documents/packt/php/chatper11/demo/phpunit.xml
Error:        No code coverage driver is available

Array
✓ Array has key name 2 ms

Time: 26 ms, Memory: 4.00 MB

OK (1 test, 1 assertion)
dave@daveismyname ~/D/p/p/c/demo>
```

Figure 11.14: Result of testing

The test passes as the name does exist in the array.

5. Create a test called **hasKeyEmail**. This test should use **assertArrayHasKey**, the expected name will be **email**, and the value will be the array defined in **setUp()**:

```
/** @test */
public function arrayHasKeyEmail()
{
    $expected = 'email';
    $actual = $this->items;
    $this->assertArrayHasKey($expected, $actual);
}
```

This test will also pass as **email** has been defined in the **\$this->items** array in **setUp**. This test ensures that if the **\$this->items** array is changed and the **email** key is removed, our tests will fail:

```
dave@daveismyname ~/D/p/p/c/demo> vendor/bin/phpunit tests/ArrayTest.php
PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.3.7
Configuration: /Users/dave/Documents/packt/php/chatper11/demo/phpunit.xml
Error:         No code coverage driver is available

Array
✓ Array has key name  2 ms
✓ Array has key email 1 ms

Time: 27 ms, Memory: 4.00 MB

OK (2 tests, 2 assertions)
dave@daveismyname ~/D/p/p/c/demo>
```

Figure 11.15: Test case passed

6. Create another test—this time to ensure that there are only two keys in **\$this->items**. Call the test **arrayOnlyHas2Keys**:

```
/** @test */
public function arrayHasOnly2Keys()
{
    $expected = 2;
```

```

    $actual = count($this->items);
    $this->assertEquals($expected, $actual);
}

```

This time, use **assertEquals** to test that **\$this->items** is 2. Using **count(\$this->items)** ensures that the expected number of items will be returned. Run this test to ensure that **\$this->items** contains only two keys:

```

dave@daveismyname ~/D/p/p/c/demo> vendor/bin/phpunit tests/ArrayTest.php
PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.3.7
Configuration: /Users/dave/Documents/packt/php/chatper11/demo/phpunit.xml
Error:        No code coverage driver is available

Array
✓ Array has key name  2 ms
✓ Array has key email 1 ms
✓ Array has only 2 keys 1 ms

Time: 29 ms, Memory: 4.00 MB

OK (3 tests, 3 assertions)

```

Figure 11.16: Result showing that the array has only two keys

With these three tests, we can be sure that the array contains the exact keys needed and that they do not change, and that the total number of keys matches our specified number. Also, this demonstrates the use of **setUp()** to define and share code for each test to ensure that each test is testing with fresh data that does not rely on the order of the tests.

LOGGING RESULTS

The **phpunit.xml** file supports logging results to log files when tests are run. In order to turn on the logs, the options first need to be added to a **logging** node:

```

<logging>
  <log type="coverage-html" target="tests/logs/report"
  lowUpperBound="35"
  highLowerBound="70"/>
  <log type="coverage-clover" target="tests/logs/coverage.xml"/>
  <log type="coverage-php" target="tests/logs/coverage.serialized"/>

```

```
<log type="coverage-text" target="php://stdout"
showUncoveredFiles="false"/>
<log type="junit" target="tests/logs/logfile.xml"/>
<log type="testdox-html" target="tests/logs/testdox.html"/>
<log type="testdox-text" target="tests/logs/testdox.txt"/>
</logging>
```

The target, in this case, has been set to **tests/logs**. This will store all the log files in **tests/logs**. The **logs** folder will be created if it does not already exist.

The logging node can be placed anywhere inside the **phpunit** node; here's a complete file example:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit bootstrap="vendor/autoload.php"
  colors="true"
  verbose="false"
  testdox="false"
  stopOnFailure="false">
  <testsuites>
    <testsuite name="Test suite">
      <directory>tests</directory>
    </testsuite>
  </testsuites>
  <logging>
    <log type="coverage-html" target="tests/logs/report"
lowUpperBound="35"
highLowerBound="70"/>
    <log type="coverage-clover" target="tests/logs/coverage.xml"/>
    <log type="coverage-php" target="tests/logs/coverage.
serialized"/>
    <log type="coverage-text" target="php://stdout"
showUncoveredFiles="false"/>
    <log type="junit" target="tests/logs/logfile.xml"/>
    <log type="testdox-html" target="tests/logs/testdox.html"/>
    <log type="testdox-text" target="tests/logs/testdox.txt"/>
  </logging>
</phpunit>
```

Note that all logs will run automatically. They depend on the options set in the **phpunit** node. For instance, for the **testdoc** logs to run, **testdoc** has to be enabled. The same applies for code coverage: if code coverage is not enabled, the logs will not be generated.

logfile.xml looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
  <testsuite name="Test suite" tests="3" assertions="3" errors="0"
failures="0" skipped="0" time="0.002055">
    <testsuite name="DemoTest"
      file="/Users/dave/Documents/packt/php/chatper11/demo/tests/
DemoTest.php" tests="1" assertions="1" errors="0" failures="0"
skipped="0" time="0.001719">
      <testcase name="testIsTheSame" class="DemoTest"
classname="DemoTest"
        file="/Users/dave/Documents/packt/php/chatper11/demo/tests/
DemoTest.php" line="7" assertions="1" time="0.001719"/>
    </testsuite>
    <testsuite name="PagesTest"
      file="/Users/dave/Documents/packt/php/chatper11/demo/tests/
PagesTest.php" tests="2" assertions="2" errors="0" failures="0"
skipped="0" time="0.000336">
      <testcase name="homePageReturnsHello" class="PagesTest"
classname="PagesTest" file="/Users/dave/Documents/packt/php/
chatper11/demo/tests/
PagesTest.php" line="15" assertions="1" time="0.000277"/>
      <testcase name="aboutPageReturnsAbout" class="PagesTest"
classname="PagesTest"
        file="/Users/dave/Documents/packt/php/chatper11/demo/tests/
PagesTest.php" line="23" assertions="1" time="0.000059"/>
    </testsuite>
  </testsuite>
</testsuites>
```

Likewise, the `testdox.txt` file is a text-based output of the tests run:

```
Demo
[x] Is the same

Pages
[x] Home page returns hello
[x] About page returns about
```

NOTE

The **x** means the test has passed.

`testdox.html` is an HTML version of the tests. It will look as follows:

Demo

✓ Is the same

Pages

✓ Home page returns hello
✓ About page returns about

Figure 11.17: HTML version of the tests

TEST DOUBLES

Unit tests should be tested in isolation for the **System Under Test (SUT)**. These will likely use other parts of the application, commonly known as contributors or dependencies. These fall into the category of test doubles, which refers to any kind of pretend object that represents a real object.

Dummies, stubs, and mocks are often used in PHPUnit for testing purposes, the idea being that instead of using a dependency, we create a fake object that represents it. This allows the input and response to be controlled and therefore tested.

DUMMIES

Dummy objects are objects that are there to satisfy a requirement but not actually be used. A common use case is when a class takes another object as a dependency. In these cases, we don't want to actually use the dependency. A fake object can be used instead to verify that an object has been passed, say, to a constructor. As part of the test suite, some methods may not need the object at all, while others may require the object to be instantiated in the construct.

Dummy values can also be tested by replacing an actual value with a dummy one. Let's create a simple **Order** class inside the **app** folder.

This has one method, called **charge**, which expects **\$amount** and **\$gateway**:

```
<?php
namespace App;
use Exception;
class Order
{
    public function charge($amount, $gateway)
    {
        if ($amount <= 0) {
            return new Exception('amount cannot be 0');
        }
        return $amount;
    }
}
```

The gateway won't be used but is expected. Let's write a test to confirm this.

Create a **OrderTest** test and add a **canCharge** method. Then, create **Order()**, add a charge to **\$expected**, and leave the second parameter as **\$actual**:

```
<?php
use PHPUnit\Framework\TestCase;
use App\Order;
class OrderTest extends TestCase
{
    /** @test */
    public function canCharge()
    {
        $order = new Order();
        $expected = 10;
```

```

        $actual = $order->charge(10);
        $this->assertEquals($expected, $actual);
    }
}

```

This will fail when run as follows:

PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

```
....E..                                     7 / 7 (100%)
```

Time: 90 ms, Memory: 4.00 MB

There was 1 error:

1) OrderTest::CanCharge

ArgumentCountError: Too few arguments to function App\Order::charge(), 1 passed in /Users/dave/ly 2 expected

/Users/dave/Documents/Documents - daveismyname/packt/php/chatper11/demo/app/Order.php:9

/Users/dave/Documents/Documents - daveismyname/packt/php/chatper11/demo/tests/OrderTest.php:15

ERRORS!

Tests: 7, Assertions: 6, Errors: 1.

Figure 11.18: Test case failure

This failed as the second parameter is required, even though it's not actually used. To satisfy this, add a second parameter:

```
$actual = $order->charge(10, 'Stripe');
```

This will now pass with our dummy value added:

PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

```
.....                                     7 / 7 (100%)
```

Time: 56 ms, Memory: 4.00 MB

OK (7 tests, 7 assertions)

dave@daveismyname ~/D/D/p/p/c/demo> █

Figure 11.19: Test case passed with dummy value

STUBS

Stubs are designed to return canned responses to calls made during tests by the SUT.

Imagine that we have contacts stored in a database. For a unit test, we should not connect to the database; that's outside the scope of a unit test. Instead, we can create a stub that represents a contact object:

```
class ContactStub
{
    public function getContact()
    {
        // Do something.
    }
}
```

Then, when the stub is to be used in a test, a **createStub** method is used and the stub class is passed to the method. The **createStub** method will create an instance of the passed class and return an object:

```
$this->createStub(App\ContactStub::class);
```

Once a stub has been created, methods can be called by calling the following:

```
$stub->method('getContact');
```

This will call a method called **getContact**. Any return types specified in the class would be returned to the **method()** call.

To specify what should be returned from the method, a **willReturn()** call can be used:

```
$stub->method('getContact')->willReturn('Joe Bloggs');
```

This would return **Joe Bloggs**.

Using stubs allows the replication of dependencies that would be outside the scope of a unit test; stubbing a class allows the response to be controlled. When testing, you will want fixed input and fixed output for accurate test results.

MOCKS

Replicating an object with a test double where the output is verified against an expected value, ensuring that a particular method has been called, is known as **mocking**. Mocks allow the creation of objects from classes that do not exist; they are fake objects.

The purpose of a mock object is to verify that a method has been called. Optionally, constraints can be added to ensure that parameters are passed in the correct format.

EXERCISE 11.7: MOCK DELETING A BOOK

In this exercise, we're going to mock a class named **Books**. The mock class should call a method name, **delete**, and expect an integer greater than 0. The steps are as follows:

1. Create a test class called **MockTest.php** in the **tests** folder, extend PHPUnit's **TestCase** as normal, and create a class called **MockTest**:

```
<?php
use PHPUnit\Framework\TestCase;
class MockTest extends TestCase
```

2. Create a method with a descriptive name, **willCallDeleteMethodOnBooksClass**:

```
/** @test */
public function willCallDeleteMothodOnBooksClasss()
```

3. Mock a fake class named **Books** by calling **getMockBuilder()**. This will create an instance of the fake class. The **Books** class should have a method called **delete**:

```
$mock = $this->getMockBuilder('Books')
    ->setMethods(['delete'])
    ->getMock();
```

Now, **\$mock** is an object made from the mock class.

4. Next, set the expectations. The method should be called only once. Using **expects(\$this->once)** ensures that it's checked that this runs once. Next, a method called **delete** is expected with an integer greater than 0 using **with()**. The restriction can be added by calling **greaterThan** and a starting integer:

```
$mock->expects($this->once())
    ->method('delete')
    ->with(
        $this->greaterThan(0)
    );
```

5. Finally, call the **delete** method and pass an integer:

```
$mock->delete(2);
```

6. The test now looks like this:

```
public function willCallDeleteMethodOnBooksClasss()  
{  
    $mock = $this->getMockBuilder('Books')  
        ->setMethods(['delete'])  
        ->getMock();  
    $mock->expects($this->once())  
        ->method('delete')  
        ->with(  
            $this->greaterThan(0)  
        );  
    $mock->delete(2);  
}
```

Since **2** has been passed to **\$mock->delete(2);**, the test will pass:

Time: 108 ms, Memory: 4.00 MB

OK (9 tests, 9 assertions)

Figure 11.20: Test case successful

7. If the test method contains the value of `0`, it will fail:

```
$mock->delete(0);
```

The output is as follows:

Time: 104 ms, Memory: 4.00 MB

There was 1 failure:

1) MockTest::willCallDeleteMethodOnBooksClassss

Expectation failed for method name is "delete" when invoked 1 time(s)

Parameter 0 for invocation Books::delete(0) does not match expected value.

Failed asserting that 0 is greater than 0.

/Users/dave/Documents/Documents – daveismyname/packt/php/chatper11/demo/tes

FAILURES!

Tests: 9, Assertions: 8, Failures: 1.

Figure 11.21: Test case failure

The test now fails since 0 is not greater than 0. Mocks ensure that methods are called when they are tested.

ACTIVITY 11.1: CREATING A CONTACT STUB AND TEST

We need to create a stub that represents a contact that would come from an external source. Once it's set, we can use the stub to simulate a fixed contact being returned.

Here are the steps to complete this activity:

1. Create a stub class called **ContactsStub** saved inside **/app**.
2. Define a method called **getContact()** inside the stub and return no output.
3. Create a test called **ContactStubsTest.php** saved within **/tests** that extends PHPUnit.
4. Create a mock from the **ContactStub** class called the **getContact** method and specify its response.
5. Compare a fixed value against the response from the mock and run the test.

Here is the expected output:

We expect a test to pass that will assert that a contact returned from the stub will match a fixed value provided from the test:

```
PHPUnit 8.4.0 by Sebastian Bergmann and contributors.
```

```
.....
```

```
8 / 8 (100%)
```

```
Time: 96 ms, Memory: 4.00 MB
```

```
OK (8 tests, 8 assertions)
```

```
dave@daveismyname ~/D/D/p/p/c/demo>
```

Figure 11.22: Expected output

NOTE



The solution for this activity can be found on page XX.

SUMMARY

In this chapter, we learned about PHPUnit; the reason for writing tests; and how to run tests, whether individually or all at once by running test suites. We also learned about assertions, covering what they are, details of the common ones, and where to find a complete list of assertions. We tried out some of the most used assertions to test that data was in the expected format and generally met our expectations.

We looked at how to write tests, sharing test data between tests within a test case, and setting up and running test suites. We looked at how to configure PHPUnit to turn on options so that they don't have to be manually specified on each test call.

We looked at how to log the results of tests and the different formats that logs can take, as well as where they should be stored.

We started our journey through the land of PHP by writing simple programs to perform basic operations. We then saw how we can control the flow of a program using various control statements. We then worked with functions to build reusable code. We explored the object-oriented programming concepts that enable us to write modular code.

We then saw how the HTTP protocol ensures communication in the same language and structure. We also took a look at the concept of data persistence and connected to a database to fetch and store data. We saw how to handle exceptions and how exceptions can be used to control the flow of a script. We then explored Composer, a PHP tool used to manage third-party libraries in PHP projects, and explored the basic concepts of web services. We connected our application with web services using Guzzle.

We rounded off our learning with a look at unit testing, where we saw how small units of code can be tested.

