Getting Started with PHPUnit in Laravel

Stephen Rees-Carter · Last update: 6 Jan 2016

Introduction

PHPUnit is one of the oldest and most well known unit testing packages for PHP. It is primarily designed for unit testing, which means testing your code in the smallest components possible, but it is also incredibly flexible and can be used for a lot more than just unit testing.

PHPUnit includes a lot of simple and flexible assertions that allow you to easily test your code, which works really well when you are testing specific components. It does mean, however, that testing more advanced code such as controllers and form submission validation can be a lot more complicated.

To help make things easier for developers, the Laravel PHP framework includes a collection of application test helpers, which allow you to write very simple PHPUnit tests to test complex parts of your application.

The purpose of this tutorial is to introduce you to the basics of PHPUnit testing, using both the default PHPUnit assertions, and the Laravel test helpers. The aim is for you to be confident writing basic tests for your applications by the end of the tutorial.

Prerequisites

This tutorial assumes that you are already familiar with Laravel and know how to run commands within the application directory (such as `php artisan` commands). We will be creating a couple of basic example classes to learn how the different testing tools work, and as such it is recommended that you create a fresh application for this tutorial.

If you have the Laravel installer set up, you can create a new test application by running:

```
laravel new phpunit-tests
```

Alternatively, you can create a new application by using Composer directly:

```
composer create-project laravel/laravel --prefer-dist
```

Other installation options can also be found in the Laravel documentation.

Creating a New Test

The first step when using PHPUnit is to create a new test class. The convention for test classes is that they are stored within `./tests/` in your application directory. Inside this folder, each test class is named as `<name>Test.php`. This format allows PHPUnit to find each test class — it will ignore anything that does not end in `Test.php`.

In a new Laravel application, you will notice two files in the `./tests/` directory: `ExampleTest.php` and `TestCase.php`. The `TestCase.php` file is a bootstrap file for

setting up the Laravel environment within our tests. This allows us to use Laravel facades in tests, and provides the framework for the testing helpers, which we will look at shortly. The `ExampleTest.php` is an example test class that includes a basic test case using the application testing helpers – ignore it for now.

To create a new test class, we can either create a new file manually – or run the helpful Artisan `make:test` command provided by Laravel.

In order to create a test class called `BasicTest`, we just need to run this artisan command:

```
php artisan make:test BasicTest
```

Laravel will create a basic test class that looks like this:

```
<?php
class BasicTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testExample()
    {
        $this->assertTrue(true);
    }
}
```

The most important thing to notice here is the `test` prefix on the method name. Like the `Test` suffix for class names, this `test` prefix tells PHPUnit what methods to run when testing. If you forget the `test` prefix, then PHPUnit will ignore the method.

Before we run our test suite for the first time, it is worth pointing out the default `phpunit.xml` file that Laravel provides. PHPUnit will automatically look for a file

named `phpunit.xml` or `phpunit.xml.dist` in the current directory when it is run. This is where you configure the specific options for your tests.

There is a lot of information within this file, however the most important section for now is the `testsuite` directory definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit ... >

    <testsuites>
        <testsuite name="Application Test Suite">
            <directory>./tests/</directory>
        </testsuite>
    </testsuites>

    ...
</phpunit>
```

This tells PHPUnit to run the tests it finds in the `./tests/` directory, which, as we have previously learned, is the convention for storing tests.

Now that we have created a base test, and are aware of the PHPUnit configuration, it is time to run our tests for the first time.

You can run your PHPUnit tests by running the `phpunit` command:

```
./vendor/bin/phpunit
```

You should see something similar to this as the output:

```
PHPUnit 4.8.19 by Sebastian Bergmann and contributors.

..

Time: 103 ms, Memory: 12.75Mb
```

```
OK (2 tests, 3 assertions)
```

Now that we have a working PHPUnit setup, it is time to move onto writing a basic test.

Note, it counts 2 tests and 3 assertions as the `ExampleTest.php` file includes a test with two assertions. Our new basic test includes a single assertion, which passed.

Writing a Basic Test

To help cover the basic assertions that PHPUnit provides, we will first create a basic class that provides some simple functionality.

Create a new file in your `./app/` directory called `Box.php`, and copy this example class:

```php
<?php
namespace App;

class Box
{
    /**
     * @var array
     */
    protected $items = [];

    /**
     * Construct the box with the given items.
     *
     * @param array $items
     */
    public function __construct($items = [])
    {
        $this->items = $items;
    }

    /**
     * Check if the specified item is in the box.
```

```
     *
     * @param string $item
     * @return bool
     */
    public function has($item)
    {
        return in_array($item, $this->items);
    }

    /**
     * Remove an item from the box, or null if the box is empty.
     *
     * @return string
     */
    public function takeOne()
    {
        return array_shift($this->items);
    }

    /**
     * Retrieve all items from the box that start with the specified letter.
     *
     * @param string $letter
     * @return array
     */
    public function startsWith($letter)
    {
        return array_filter($this->items, function ($item) use ($letter) {
            return stripos($item, $letter) === 0;
        });
    }
}
```

Next, open your `./tests/BasicTest.php` class (that we created earlier), and remove the `testExample` method that was created by default. You should be left with an empty class.

We will now use seven of the basic PHPUnit assertions to write tests for our `Box` class. These assertions are:

- `assertTrue()`

- `assertFalse()`
- `assertEquals()`
- `assertNull()`
- `assertContains()`
- `assertCount()`
- `assertEmpty()`

assertTrue() and assertFalse()

`assertTrue()` and `assertFalse()` allow you to assert that a value is equates to either true or false. This means they are perfect for testing methods that return boolean values. In our `Box` class, we have a method called `has($item)`, which returns true or false when the specified item is in the box or not.

To write a test for this in PHPUnit, we can do the following:

```
assertTrue($box->has('toy'));
        $this->assertFalse($box->has('ball'));
    }
}
```

Note how we only pass a single parameter into the `assertTrue()` and `assertFalse()` methods, and it is the output of the `has($item)` method.

If you run the `./vendor/bin/phpunit` command now, you will notice the output includes:

```
OK (2 tests, 4 assertions)
```

This means our tests have passed.

If you swap the `assertFalse()` for `assertTrue()` and run the `phpunit` command again, the output will look like this:

```
PHPUnit 4.8.19 by Sebastian Bergmann and contributors.

F.

Time: 93 ms, Memory: 13.00Mb

There was 1 failure:

1) BasicTest::testHasItemInBox
Failed asserting that false is true.

./tests/BasicTest.php:12

FAILURES!
Tests: 2, Assertions: 4, Failures: 1.
```

This tells us that the assertion on line 12 failed to assert that a `false` value was `true` - as we switched the `assertFalse()` for `assertTrue()`.

Swap it back, and re-run PHPUnit. The tests should again pass, as we have fixed the broken test.

assertEquals() and assertNull()

Next, we will look at `assertEquals()`, and `assertNull()`.

assertEquals() is used to compare the actual value of the variable to the expected value. We want to use it to check if the value of the `takeOne()` function is an item that is current in the box. As the `takeOne()` method returns a `null` value when the box is empty, we can use `assertNull()` to check for that too.

Unlike `assertTrue()`, `assertFalse()`, and `assertNull()`, `assertEquals()` takes two parameters. The first being the **expected** value, and the second being the **actual** value.

We can implement these assertions in our class as follows:

```
        $this->assertTrue($box->has('toy'));
        $this->assertFalse($box->has('ball'));
    }

    public function testTakeOneFromTheBox()
    {
        $box = new Box(['torch']);

        $this->assertEquals('torch', $box->takeOne());

        // Null, now the box is empty
        $this->assertNull($box->takeOne());
    }
}
```

Run the `phpunit` command, and you should see:

```
OK (3 tests, 6 assertions)
```

assertContains(), assertCount(), and assertEmpty()

Finally, we have three assertions that work with arrays, which we can use to check the `startsWith($item)` method in our `Box` class. `assertContains()` asserts that an expected value exists within the provided array, `assertCount()` asserts the number of items in the array matches the specified amount, and `assertEmpty()` asserts that the provided array is empty.

We can implement tests for these like this:

```
        $this->assertTrue($box->has('toy'));
        $this->assertFalse($box->has('ball'));
    }

    public function testTakeOneFromTheBox()
    {
        $box = new Box(['torch']);
```

```
        $this->assertEquals('torch', $box->takeOne());

        // Null, now the box is empty
        $this->assertNull($box->takeOne());
    }

    public function testStartsWithALetter()
    {
        $box = new Box(['toy', 'torch', 'ball', 'cat', 'tissue']);

        $results = $box->startsWith('t');

        $this->assertCount(3, $results);
        $this->assertContains('toy', $results);
        $this->assertContains('torch', $results);
        $this->assertContains('tissue', $results);

        // Empty array if passed even
        $this->assertEmpty($box->startsWith('s'));
    }
}
```

Save and run your tests again:

```
OK (4 tests, 9 assertions)
```

Congratulations, you have just fully tested the `Box` class using seven of the basic PHPUnit assertions. You can do a lot with these simple assertions, and most of the other, more complex, assertions that are available still follow the same usage pattern.

Testing Your Application

Unit testing each component in your application works in a lot of situations and should definitely be part of your development process, however it isn't all the testing you need to do. When you are building an application that includes complex views, navigation and forms, you will want to test these components

too. This is where Laravel's test helpers make things just as easy as unit testing simple components.

We previously looked at the default files within the `./tests/` directory, and we skipped the `./tests/ExampleTest.php` file. Open it now, and it should look something like this:

```
visit('/')
             ->see('Laravel 5');
    }
}
```

We can see the test in this case is very simple. Without any prior knowledge of how the test helpers work, we can assume it means something like this:

1. when I visit `/` (webroot)
2. I should see 'Laravel 5'

If you open your web browser to our application (you can run `php artisan serve` if you don't have a web server set up), you should see a splash screen with "Laravel 5" on the web root. Given that this test has been passing PHPUnit, it is safe to say that our translation of this example test is correct.

This test is ensuring that the web page rendered at the `/` path returns the text 'Laravel 5'. A simple check like this may not seem like much, but if there is critical information your website needs to display, a simple test like this may prevent you from deploying a broken application if a change somewhere else causes the page to no longer display the right information.

visit(), see(), and dontSee()

Let's write our own test now, and take it one step further.

```
<?php
Route::get('/', function () {
```

```
    return view('welcome');
});

Route::get('/alpha', function () {
    return view('alpha');
});

?>
```

First, edit the `./app/Http/routes.php` file, to add in a new route. For the sake of this tutorial, we will go for a Greek alphabet themed route:

Next, create the view template at `./resources/views/alpha.blade.php`, and save some basic HTML with the Alpha keyword:

```
<h1>Alpha</h1>

<p>This is the Alpha page.</p>
```

Now open it in your browser to ensure it is working as expected: `http://localhost:8000/beta`, and it should display a friendly "This is the Alpha page." message.

Now that we have the template, we will create a new test. Run the `make:test` command:

```
php artisan make:test AlphaTest
```

Then edit the test, using the example test as a guide, but we also want to ensure that our "alpha" page does not mention "beta". To do this, we can use the `dontSee()` assertion, which does the opposite of `see()`.

This means we can do a simple test like this:

```
visit('/alpha')
             ->see('Alpha')
             ->dontSee('Beta');
    }
}
```

Save it and run PHPUnit (`./vendor/bin/phpunit`), and it should all pass, with the status line looking something like this:

```
 OK (5 tests, 12 assertions)
```

Writing Tests First

A great thing about tests is that you can use the **Test Driven Development** (TDD) approach, and write your tests first. After writing your tests, you run them and see that they fail, **then** you write the code that satisfies the tests to make everything pass again. So, let's do that for the next page.

First, make a `BetaTest` class using the `make:test` artisan command:

```
php artisan make:test BetaTest
```

Next, update the test case so it is checking the `/beta` route for "Beta":

```
visit('/beta')
             ->see('Beta')
             ->dontSee('Alpha');
    }
}
```

Now, run the test using `./vendor/bin/phpunit` . The result should be a slightly unfriendly error message, that looks like this:

```
> ./vendor/bin/phpunit
PHPUnit 4.8.19 by Sebastian Bergmann and contributors.

....F.

Time: 144 ms, Memory: 14.25Mb

There was 1 failure:

1) BetaTest::testDisplaysBeta
A request to [http://localhost/beta] failed. Received status code [404].

...

FAILURES!
Tests: 6, Assertions: 13, Failures: 1.
```

We now have an expectation for a missing route. Let's create it.

First, edit the `./app/Http/routes.php` file to create the new `/beta` route:

```
<?php
Route::get('/', function () {
    return view('welcome');
});

Route::get('/alpha', function () {
    return view('alpha');
});

Route::get('/beta', function () {
    return view('beta');
});

?>
```

Next, create the view template at `./resources/views/beta.blade.php`:

```
        Beta

        This is the Beta page.
```

Now, run PHPUnit again, and the result should be back to green.

```
> ./vendor/bin/phpunit
PHPUnit 4.8.19 by Sebastian Bergmann and contributors.

......

Time: 142 ms, Memory: 14.00Mb

OK (6 tests, 15 assertions)
```

We have now implemented our new page using *Test Driven Development* by writing the test first.

click() and seePageIs()

Laravel also provides a helper to allow the test to click a link that exists on the page (`click()`), as well as a way to check what the resulting page is (`seePageIs()`).

Let's use these two helpers to implement links between the Alpha and Beta pages.

First, let's update our tests. Open the `AlphaTest` class, and we will add in a new test method, which will click the 'Next' link found on the "alpha" page to go to the "beta" page.

The new test should look like this:

```php
<?php
class AlphaTest extends TestCase
{
    public function testDisplaysAlpha()
    {
        $this->visit('/alpha')
             ->see('Alpha')
             ->dontSee('Beta');
    }

    public function testClickNextForBeta()
    {
        $this->visit('/alpha')
             ->click('Next')
             ->seePageIs('/beta');
    }
}
```

Notice that we aren't checking the content of either page in our new `testClickNextForBeta()` test method. Other tests are successfully checking the content of both pages, so all we care about is that clicking the "Next" link will send us to `/beta`.

You can run the test suite now, but as expected it will fail as we haven't updated our HTML yet.

Next, we will update the `BetaTest` to do similar:

```
visit('/beta')
             ->see('Beta')
             ->dontSee('Alpha');
    }

    public function testClickNextForAlpha()
    {
        $this->visit('/beta')
             ->click('Previous')
             ->seePageIs('/alpha');
    }
}
```

Next, let's update our HTML templates.

`./resources/views/alpha.blade.php`:

```
        Alpha

        This is the Alpha page.
        Next
```

`./resources/views/beta.blade.php`:

```
        Beta

        This is the Beta page.
        Previous
```

Save the files and run PHPUnit again:

```
> ./vendor/bin/phpunit
PHPUnit 4.8.19 by Sebastian Bergmann and contributors.

F....F..

Time: 175 ms, Memory: 14.00Mb

There were 2 failures:

1) AlphaTest::testDisplaysAlpha
Failed asserting that '<!DOCTYPE html>
```

```
<html>
    <head>
        <title>Alpha</title>
    </head>
    <body>
        <p>This is the Alpha page.</p>
        <p><a href="https://www.semaphoreci.com/beta">Next</a></p>
    </body>
</html>
' does not match PCRE pattern "/Beta/i".

2) BetaTest::testDisplaysBeta
Failed asserting that '<!DOCTYPE html>
<html>
    <head>
        <title>Beta</title>
    </head>
    <body>
        <p>This is the Beta page.</p>
        <p><a href="https://www.semaphoreci.com/alpha">Previous</a></p>
    </body>
</html>
' does not match PCRE pattern "/Alpha/i".

FAILURES!
Tests: 8, Assertions: 23, Failures: 2.
```

We have broken our tests somehow. If you look at our new HTML carefully, you will notice we now have the terms `beta` and `alpha` on the `/alpha` and `/beta` pages, respectively. This means we need to slightly change our tests so they don't match false positives.

Within each of the `AlphaTest` and `BetaTest` classes, update the `testDisplays*` methods to use `dontSee('<page> page')`. This way, it will only match the string, not the term.

The two tests should look like this:

`./tests/AlphaTest.php`:

```php
<?php
class AlphaTest extends TestCase
{
    public function testDisplaysAlpha()
    {
        $this->visit('/alpha')
             ->see('Alpha')
             ->dontSee('Beta page');
    }

    public function testClickNextForBeta()
    {
        $this->visit('/alpha')
             ->click('Next')
             ->seePageIs('/beta');
    }
}
```

`./tests/BetaTest.php`:

```php
<?php
class BetaTest extends TestCase
{
    public function testDisplaysBeta()
    {
        $this->visit('/beta')
             ->see('Beta')
             ->dontSee('Alpha page');
    }

    public function testClickNextForAlpha()
    {
        $this->visit('/beta')
             ->click('Previous')
             ->seePageIs('/alpha');
    }
}
```

Run your tests again, and everything should pass again. We have now tested our new pages, including the Next/Previous links between them.

Continuous Integration for PHPUnit on Semaphore

You can automate running your test by setting up continuous integration on Semaphore.

This way your tests will run on every `git push` and Semaphore comes with all recent versions of PHP preinstalled.

First, sign up for a free Semaphore account if you don't have one already. All there's left to do is to add your repository, and following build steps will be added to run your tests:

```
composer install --prefer-source
phpunit
```

For more information on continuous integration for PHP, refer to Semaphore documentation.

Conclusion

You should notice a common theme across all of the tests that we have written in this tutorial: they are all incredibly simple. This is one of the benefits of learning how to use the basic test assertions and helpers, and trying to use them as much as possible. The simpler you can write your tests, the easier your tests will be to understand and maintain.

Once you have mastered the PHPUnit assertions we have covered in this tutorial, you can find a lot more in the PHPUnit documentation. They all follow the same basic pattern, but you will find that you keep coming back to the basic assertions for most of your tests.

Laravel's test helpers are a fantastic compliment to the PHPUnit assertions, and make testing your application templates easy. That said, it is important to recognize that, as part of our tests, we only checked the critical information – not the entire page. This keeps the tests simple, and allows the page content to