symblog

creating a blog in Symfony2

« **[Part 5] - Customising the view: Twig extensions, The sidebar and Assetic** :: Contents

[Part 6] - Testing: Unit and Functional with PHPUnit

Overview

So far we have explored a good amount of ground looking at a number of core concepts with regards to Symfony2 development. Before we continue adding features it is time to introduce testing. We will look at how to test individual functions with unit testing and how to ensure multiple components are working correctly together with functional testing. The PHP testing library **PHPUnit** will be covered as this library is at the centre of the Symfony2 tests. As testing is a large topic it will also be covered in later chapters. By the end of this chapter you will have written a number of tests covering both unit and functional testing. You will have simulated browser requests, populated forms with data, and checked responses to ensure the website pages are outputting correctly. You will also have checked how much coverage your tests have on your applications code base.

Testing in Symfony2

PHPUnit has become the “de facto standard” for writing tests in PHP, so learning it will benefit you in all your PHP projects. Lets also not forget that most of the topics covered in this chapter are language independent and so can be transferred to other languages you.

Tip

If you are planning on writing your own Open Source Symfony2 bundles, you are much more likely to receive interest if your bundle is well tested (and documented). Have a look at the existing Symfony2 bundles available at **Symfony2Bundles**.

Unit Testing

Unit testing is concerned with ensuring individual units of code function correctly when used in isolation. In an Object Oriented code base like Symfony2, a unit would be a class and its methods. For example, we could write tests for the `Blog`and `Comment` Entity classes. When writing unit tests, the test cases should be written independently of other test cases, i.e., the result of test case B should not depend on the result of test case A. It is useful when unit testing to be able to create mock objects that allow you to easily unit test functions that have external dependencies. Mocking allows you to simulate a function call instead of actually executing it. An example of this would be unit testing a class that wraps up an external API. The API class may use a transport layer for communicating with the external API. We could mock the request method of the transport layer to return the results we specify, rather than actually hitting the external API. Unit testing does not test that the components of an application function correctly together, this is covered by the next topic, functional testing.

Functional Testing

Functional testing checks the integration of different components within the application, such as routing, controllers, and views. Functional tests are similar to the manual tests you would run yourself in the browser such as requesting the homepage, clicking a blog link and checking the correct blog is shown. Functional testing provides you with the ability to automate this process. Symfony2 comes complete with a number of useful classes that assist in functional testing including a `Client` that is able to requests pages and submit forms and DOM `Crawler` that we can use to traverse the `Response`from the client.

Tip

There are a number of software development process that are driven by testing. These include processes such as Test Driven Development (TDD) and Behavioral Driven Development (BDD). While these are out side the scope of this tutorial you should be aware of the library written by **everzet** that facilitates BDD called **Behat**. There is also a Symfony2 **BehatBundle** available to easily integrate Behat into your Symfony2 project.

PHPUnit

As stated above, Symfony2 tests are written using PHPUnit. You will need to install PHPUnit in order to run these tests and the tests from this chapter. For detailed **installation instructions** refer to the official documentation on the PHPUnit website. To run the tests in Symfony2 you need to install PHPUnit 3.5.11 or later. PHPUnit is a very large testing library, so references to the official documentation will be made where additional reading can be found.

Assertions

Writing tests is concerened with checking that the actual test result is equal to the expected test result. There are a number of assertion methods available in PHPUnit to assist you with this task. Some of the common assertion methods you will use are listed below.

```
// Check 1 === 1 is true
$this->assertTrue(1 === 1);

// Check 1 === 2 is false
$this->assertFalse(1 === 2);

// Check 'Hello' equals 'Hello'
$this->assertEquals('Hello', 'Hello');

// Check array has key 'language'
$this->assertArrayHasKey('language', array('language' => 'php', 'size' => '1024'));

// Check array contains value 'php'
$this->assertContains('php', array('php', 'ruby', 'c++', 'JavaScript'));
```

A full list of **assertions** is available in the PHPUnit documentation.

Running Symfony2 Tests

Before we begin writing some tests, lets look at how we run tests in Symfony2. PHPUnit can be set to execute using a configuration file. In our Symfony2 project this file is located at `app/phpunit.xml.dist`. As this file is suffixed with `.dist`, you need to copy its contents into a file called `app/phpunit.xml`.

Tip

If you are using a VCS such as Git, you should add the new `app/phpunit.xml` file to the VCS ignore list.

If you have a look at the contents of the PHPUnit configuration file you will see the following.

```
<!-- app/phpunit.xml -->

<testsuites>
    <testsuite name="Project Test Suite">
        <directory>../src/*/*Bundle/Tests</directory>
        <directory>../src/*/Bundle/*Bundle/Tests</directory>
    </testsuite>
</testsuites>
```

The following settings configure some directories that are part of our test suite. When running PHPUnit it will look in the above directories for tests to run. You can also pass additional command line arguments to PHPUnit to run tests in specific directories, instead of the test suite tests. You will see how to achieve this later.

You will also notice the configuration is specifying the bootstrap file located at `app/bootstrap.php.cache`. This file is used by PHPUnit to get the testing environment setup.

```
<!-- app/phpunit.xml -->

<phpunit
    bootstrap                    = "bootstrap.php.cache" >
```

Tip

For more information regarding configuring PHPUnit with an XML file see the **PHPUnit documentation**.

Running the Current Tests

As we used one of the Symfony2 generator tasks to create the `BloggerBlogBundle` back in chapter 1, it also created a controller test for the `DefaultController` class. We can execute this test by running the following command from the root directory of the project. The `-c` option specifies that PHPUnit should load its configuration from the `app` directory.

```
$ phpunit -c app
```

Once the testing has completed you should be notified that the tests failed. If you look at the `DefaultControllerTest` class located at `src/Blogger/BlogBundle/Tests/Controller/DefaultControllerTest.php` you will see the following content.

```
<?php
// src/Blogger/BlogBundle/Tests/Controller/DefaultControllerTest.php

namespace Blogger\BlogBundle\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DefaultControllerTest extends WebTestCase
{
    public function testIndex()
    {
        $client = static::createClient();

        $crawler = $client->request('GET', '/hello/Fabien');

        $this->assertTrue($crawler->filter('html:contains("Hello Fabien")')->count() > 0);
    }
}
```

This is a functional test for the `DefaultController` class that Symfony2 generated. If you remember back to chapter 1, this Controller had an action that handled requests to `/hello/{name}`. The fact that we removed this controller is why the above test is failing. Try going to the URL `http://symblog.dev/app_dev.php/hello/Fabien` in your browser. You should be informed that the route could not be found. As the above test makes a request to the same URL, it will also get the same response, hence why the test fails. Functional testing is a large part of this chapter and will be covered in detail later.

As the `DefaultController` class has been removed, you can also remove this test class. Delete the `DefaultControllerTest` class located at `src/Blogger/BlogBundle/Tests/Controller/DefaultControllerTest.php`.

Unit Testing

As explained previously, unit testing is concerned with testing individual units of your application in isolation. When writing unit tests it is recommend that you replicate the Bundle structure under the Tests folder. For example, if you wanted to test the `Blog` entity class located at `src/Blogger/BlogBundle/Entity/Blog.php` the test file would reside at`src/Blogger/BlogBundle/Tests/Entity/BlogTest.php`. An example folder layout would be as follows.

```
src/Blogger/BlogBundle/
                Entity/
                    Blog.php
                    Comment.php
                Controller/
                    PageController.php
                Twig/
                    Extensions/
                        BloggerBlogExtension.php
                Tests/
                    Entity/
                        BlogTest.php
                        CommentTest.php
                    Controller/
                        PageControllerTest.php
                    Twig/
                        Extensions/
                            BloggerBlogExtensionTest.php
```

Notice that each of the Test files are suffixed with Test.

Testing the Blog Entity - Slugify method

We begin by testing the slugify method in the `Blog` entity. Lets write some tests to ensure this method is working correctly. Create a new file located at `src/Blogger/BlogBundle/Tests/Entity/BlogTest.php` and add the following.

```
<?php
// src/Blogger/BlogBundle/Tests/Entity/BlogTest.php

namespace Blogger\BlogBundle\Tests\Entity;

use Blogger\BlogBundle\Entity\Blog;

class BlogTest extends \PHPUnit_Framework_TestCase
{

}
```

We have created a test class for the `Blog` entity. Notice the location of the file complies with the folder structure mentioned above. The `BlogTest` class extends the base PHPUnit class `PHPUnit_Framework_TestCase`. All tests you write for PHPUnit will be a child of this class. You'll remember from previous chapters that the `\` must be placed in front of the `PHPUnit_Framework_TestCase`class name as the class is declared in the PHP public namespace.

Now we have the skeleton class for our `Blog` entity tests, lets write a test case. Test cases in PHPUnit are methods of the Test class prefixed with `test`, such as `testSlugify()`. Update the `BlogTest` located at `src/Blogger/BlogBundle/Tests/Entity/BlogTest.php` with the following.

```
// src/Blogger/BlogBundle/Tests/Entity/BlogTest.php

// ..

class BlogTest extends \PHPUnit_Framework_TestCase
{
    public function testSlugify()
    {
        $blog = new Blog();

        $this->assertEquals('hello-world', $blog->slugify('Hello World'));
    }
}
```

This is a very simple test case. It instantiates a new `Blog` entity and runs an `assertEquals()` on the result of the `slugify` method. The `assertEquals()` method takes 2 mandatory arguments, the expected result and the actual result. An optional 3rd argument can be passed in to specify a message to display when the test case fails.

Lets run our new unit test. Run the following on the command line.

```
$ phpunit -c app
```

You should see the following output.

```
PHPUnit 3.5.11 by Sebastian Bergmann.

.

Time: 1 second, Memory: 4.25Mb

OK (1 test, 1 assertion)
```

The output from PHPUnit is very simple, Its start by displaying some information about PHPUnit and the outputs a number of . for each test it runs, in our case we are only running 1 test so only 1 . is output. The last statement informs us of the result of the tests. For our `BlogTest` we only ran 1 test with 1 assertion. If you have color output on your command line you will also see the last line displayed in green showing everything executed OK. Lets update the `testSlugify()` method to see what happens when the tests fails.

```
// src/Blogger/BlogBundle/Tests/Entity/BlogTest.php

// ..

public function testSlugify()
{
    $blog = new Blog();

    $this->assertEquals('hello-world', $blog->slugify('Hello World'));
    $this->assertEquals('a day with symfony2', $blog->slugify('A Day With Symfony2'));
}
```

Re run the unit tests as before. The following output will be displayed

```
PHPUnit 3.5.11 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 4.25Mb

There was 1 failure:

1) Blogger\BlogBundle\Tests\Entity\BlogTest::testSlugify
Failed asserting that two strings are equal.
```

```
--- Expected
+++ Actual
@@ @@
-a day with symfony2
+a-day-with-symfony2

/var/www/html/symblog/symblog/src/Blogger/BlogBundle/Tests/Entity/BlogTest.php:15

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

The output is a bit more involved this time. We can see the . for the run tests is replaced by a `F`. This tells us the test failed. You will also see the `E` character output if your test contains Errors. Next PHPUnit notifies us in detail of the failures, in this case, the 1 failure. We can see the `Blogger\BlogBundle\Tests\Entity\BlogTest::testSlugify` method failed because the Expected and the Actual values were different. If you have color output on your command line you will also see the last line displayed in red showing there were failures in your tests. Correct the `testSlugify()` method so the tests execute successfully.

```php
// src/Blogger/BlogBundle/Tests/Entity/BlogTest.php

// ..

public function testSlugify()
{
    $blog = new Blog();

    $this->assertEquals('hello-world', $blog->slugify('Hello World'));
    $this->assertEquals('a-day-with-symfony2', $blog->slugify('A Day With Symfony2'));
}
```

Before moving on add some more test for `slugify()` method.

```php
// src/Blogger/BlogBundle/Tests/Entity/BlogTest.php

// ..

public function testSlugify()
{
    $blog = new Blog();

    $this->assertEquals('hello-world', $blog->slugify('Hello World'));
    $this->assertEquals('a-day-with-symfony2', $blog->slugify('A Day With Symfony2'));
    $this->assertEquals('hello-world', $blog->slugify('Hello    world'));
    $this->assertEquals('symblog', $blog->slugify('symblog '));
    $this->assertEquals('symblog', $blog->slugify(' symblog'));
}
```

Now we have tested the `Blog` entity slugify method, we need to ensure the `Blog` `$slug` member is correctly set when the `$title`member of the `Blog` is updated. Add the following methods to the `BlogTest` file located at`src/Blogger/BlogBundle/Tests/Entity/BlogTest.php`.

```php
// src/Blogger/BlogBundle/Tests/Entity/BlogTest.php

// ..

public function testSetSlug()
{
    $blog = new Blog();

    $blog->setSlug('Symfony2 Blog');
    $this->assertEquals('symfony2-blog', $blog->getSlug());
}

public function testSetTitle()
{
    $blog = new Blog();

    $blog->setTitle('Hello World');
    $this->assertEquals('hello-world', $blog->getSlug());
}
```

We begin by testing the `setSlug` method to ensure the `$slug` member is correctly slugified when updated. Next we check the`$slug` member is correctly updated when the `setTitle` method is called on the `Blog` entity.

Run the tests to verify the `Blog` entity is functioning correctly.

Testing the Twig extension

In the previous chapter we created a Twig extension to convert a `\DateTime` instance into a string detailing the duration since a time period. Create a new test file located at `src/Blogger/BlogBundle/Tests/Twig/Extensions/BloggerBlogExtensionTest.php` and update with the following content.

```php
<?php
// src/Blogger/BlogBundle/Tests/Twig/Extensions/BloggerBlogExtensionTest.php

namespace Blogger\BlogBundle\Tests\Twig\Extensions;

use Blogger\BlogBundle\Twig\Extensions\BloggerBlogExtension;

class BloggerBlogExtensionTest extends \PHPUnit_Framework_TestCase
```

```php
{
    public function testCreatedAgo()
    {
        $blog = new BloggerBlogExtension();

        $this->assertEquals("0 seconds ago", $blog->createdAgo(new \DateTime()));
        $this->assertEquals("34 seconds ago", $blog->createdAgo($this->getDateTime(-34)));
        $this->assertEquals("1 minute ago", $blog->createdAgo($this->getDateTime(-60)));
        $this->assertEquals("2 minutes ago", $blog->createdAgo($this->getDateTime(-120)));
        $this->assertEquals("1 hour ago", $blog->createdAgo($this->getDateTime(-3600)));
        $this->assertEquals("1 hour ago", $blog->createdAgo($this->getDateTime(-3601)));
        $this->assertEquals("2 hours ago", $blog->createdAgo($this->getDateTime(-7200)));

        // Cannot create time in the future
        $this->setExpectedException('\InvalidArgumentException');
        $blog->
```