Unit Testing

You might have noticed some subtle but nonetheless important bugs in the framework we built in the previous chapter. When creating a framework, you must be sure that it behaves as advertised. If not, all the applications based on it will exhibit the same bugs. The good news is that whenever you fix a bug, you are fixing a bunch of applications too.

Today's mission is to write unit tests for the framework we have created by using PHPUnit_. Create a PHPUnit configuration file in example.com/phpunit.xml.dist:

.. code-block:: xml

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

<phpunit backupGlobals="false"
         backupStaticAttributes="false"
         colors="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="false"
         stopOnFailure="false"
         syntaxCheck="false"
         bootstrap="vendor/autoload.php"
>
    <testsuites>
        <testsuite name="Test Suite">
            <directory>./tests</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

This configuration defines sensible defaults for most PHPUnit settings; more interesting, the autoloader is used to bootstrap the tests, and tests will be stored under the example.com/tests/ directory.

Now, let's write a test for "not found" resources. To avoid the creation of all dependencies when writing tests and to really just unit-test what we want, we are going to use test doubles_. Test doubles are easier to create when we rely on interfaces instead of concrete classes. Fortunately, Symfony2 provides such interfaces for core objects like the URL matcher and the controller resolver. Modify the framework to make use of them::

```
<?php

// example.com/src/Simplex/Framework.php

namespace Simplex;

// ...

use Symfony\Component\Routing\Matcher\UrlMatcherInterface;
use Symfony\Component\HttpKernel\Controller\ControllerResolverInterface;

class Framework
{
```

```
    protected $matcher;
    protected $resolver;

    public function __construct(UrlMatcherInterface $matcher, ControllerResolverInterface $resolver)
    {
        $this->matcher = $matcher;
        $this->resolver = $resolver;
    }

    // ...
}
```

We are now ready to write our first test::

```
<?php

// example.com/tests/Simplex/Tests/FrameworkTest.php

namespace Simplex\Tests;

use Simplex\Framework;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Exception\ResourceNotFoundException;

class FrameworkTest extends \PHPUnit_Framework_TestCase
{
    public function testNotFoundHandling()
    {
        $framework = $this->getFrameworkForException(new ResourceNotFoundException());

        $response = $framework->handle(new Request());

        $this->assertEquals(404, $response->getStatusCode());
    }

    protected function getFrameworkForException($exception)
    {
        $matcher = $this->getMock('Symfony\Component\Routing\Matcher\UrlMatcherInterface');
        $matcher
            ->expects($this->once())
            ->method('match')
            ->will($this->throwException($exception))
        ;
        $resolver = $this->getMock('Symfony\Component\HttpKernel\Controller\ControllerResolverInterface');

        return new Framework($matcher, $resolver);
    }
}
```

This test simulates a request that does not match any route. As such, the `match()` method returns a `ResourceNotFoundException` exception and we are testing that our framework converts this exception to a 404 response.

Executing this test is as simple as running `phpunit` from the `example.com` directory:

.. code-block:: bash

```
$ phpunit
```

.. note::

I do not explain how the code works in details as this is not the goal of
this book, but if you don't understand what the hell is going on, I highly
recommend you to read PHPUnit documentation on `test doubles`_.

After the test ran, you should see a green bar. If not, you have a bug either in the test or in the framework code!

Adding a unit test for any exception thrown in a controller is just as easy::

```
public function testErrorHandling()
{
    $framework = $this->getFrameworkForException(new \RuntimeException());

    $response = $framework->handle(new Request());

    $this->assertEquals(500, $response->getStatusCode());
}
```

Last, but not the least, let's write a test for when we actually have a proper Response::

```
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Controller\ControllerResolver;

public function testControllerResponse()
{
    $matcher = $this->getMock('Symfony\Component\Routing\Matcher\UrlMatcherInterface');
    $matcher
        ->expects($this->once())
        ->method('match')
        ->will($this->returnValue(array(
            '_route' => 'foo',
            'name' => 'Fabien',
            '_controller' => function ($name) {
                return new Response('Hello '.$name);
            }
        )))
    ;
    $resolver = new ControllerResolver();

    $framework = new Framework($matcher, $resolver);

    $response = $framework->handle(new Request());

    $this->assertEquals(200, $response->getStatusCode());
    $this->assertContains('Hello Fabien', $response->getContent());
}
```

In this test, we simulate a route that matches and returns a simple controller. We check that the response status is 200 and that its content is the one we have set in the controller.

To check that we have covered all possible use cases, run the PHPUnit test coverage feature (you need to enable XDebug_ first):

.. code-block:: bash

```
$ phpunit --coverage-html=cov/
```

Open example.com/cov/src_Simplex_Framework.php.html in a browser and check that all the lines for the Framework class are green (it means that they have been visited when the tests were executed).

Thanks to the simple object-oriented code that we have written so far, we have been able to write unit-tests to cover all possible use cases of our framework; test doubles ensured that we were actually testing our code and not Symfony2 code.

Now that we are confident (again) about the code we have written, we can safely think about the next batch of features we want to add to our framework.

.. _PHPUnit: https://phpunit.de/manual/current/en/ .. _test doubles: https://phpunit.de/manual/current/en/test-doubles.html .. _XDebug: http://xdebug.org/