AN INTRODUCTION TO SYMFONY 6
(for people that already know OO-PHP and some MVC stuff)

by
Matt Smith, Ph.D.
https://github.com/dr-matt-smith

Acknowledgements

Thanks to Ryan Weaver, for suggesting I update things to Symfony 6 in 2022 :-)

Thanks to the PHP and Symfony open-source international communities.

Table of Contents

Part I

Symfony Testing

1

Unit testing in Symfony

1.1 Testing in Symfony

Symfony is built by an open source community. There is a lot of information about how to test Symfony in the official documentation pages:

- Symfony testing
- Testing with user authentication tokens
- How to Simulate HTTP Authentication in a Functional Test

1.2 Installing Simple-PHPUnit (project `test01`)

Symfony has as special 'testpack' that works with PHPUnit. Add this to your project as follows:

```
$ composer require --dev symfony/test-pack
```

You should now see a `/tests` directory created.

Run our tests - we have none, so should get a message telling us no tests were executed:

```
$ bin/phpunit

PHPUnit 9.5.19
```

```
No tests executed!
```

1.3 Creating a test class

Let's create a simple test (1 + 1 = 2!) to check everything is working okay.

Create a new class `/tests/SimpleTest.php` containing the following:

```php
<?php
namespace App\Tests;

use PHPUnit\Framework\TestCase;

class SimpleTest extends TestCase
{
    public function testOnePlusOneEqualsTwo()
    {
        // Arrange
        $num1 = 1;
        $num2 = 1;
        $expectedResult = 2;

        // Act
        $result = $num1 + $num2;

        // Assert
        $this->assertEquals($expectedResult, $result);
    }
}
```

Note the following:

- test classes are located in directory `/tests`
  - or a suitably named sub-directory, matching the `/src` namespaced folder they are testing
- test classes end with the suffix `Test`, e.g. `SimpleTest`
- simple test classes extend the superclass `\PHPUnit\Framework\TestCase`
  - if we add a `uses` statement `use PHPUnit\Framework\TestCase` then we can simple extend `TestCase`
- simple test classes are in namespace `App\Tests`

– the names and namespaces of test classes testing a class in `/src` will reflect the namespace of the class being tested

– i.e. If we write a class to test `/src/Controller/DefaultController.php` it will be `/tests/Controller/DefaultControllerTest.php`, and it will be in namespace `App\Tests\Controller`

– so our testing class architecture directly matches our source code architecture

1.4 Running our tests

Run our tests - we have 1 now, so should get a message telling us our test ran and passed:

```
$ bin/phpunit

Testing
.                                                                   1 / 1 (100%)

Time: 00:00.022, Memory: 10.00 MB

OK (1 test, 1 assertion)
```

Dots are good. For each passed test you'll see a full stop. Then after all tests have run, you'll see a summary:

```
1 / 1 (100%)
```

This tells us how many passed, out of how many, and what the pass percentage was. In our case, 1 out of 1 passed = 100%.

1.5 Testing other classes (project `test02`)

our testing structure mirrors the code we are testing

Let's create a very simple class `Calculator.php` in `/src/Util`[1], and then write a class to test our class. Our simple class will be a very simple calculator:

- method `add(...)` accepts 2 numbers and returns the result of adding them
- method `subtract()` accepts 2 numbers and returns the result of subtracting the second from the first

so our `Calculator` class is as follows:

[1] Short for 'Utility' - i.e. useful stuff!

```php
<?php
namespace App\Util;

class Calculator
{
    public function add($n1, $n2)
    {
        return $n1 + $n2;
    }

    public function subtract($n1, $n2)
    {
        return $n1 - $n2;
    }
}
```

1.6 The class to test our calculator

We now need to write a test class to test our calculator class. Since our source code class is `/src/Util/Calculator.php` then our testing class will be `/tests/Util/CalculatorTest.php`. And since the namespace of our source code class was `App\Util` then the namespace of our testing class will be `App\Tests\Util`. Let's test making an instance-object of our class `Calculator`, and we will make 2 assertions:

- the reference to the new object is not NULL
- invoking the `add(...)` method with arguments of (1,1) and returns the correct answer (2!)

Here's the listing for our new test class `/tests/Util/CalculatorTest.php`:

```php
namespace App\Tests\Util;

use App\Util\Calculator;
use PHPUnit\Framework\TestCase;

class CalculatorTest extends TestCase
{
    public function testCanCreateObject()
    {
        // Arrange
        $calculator = new Calculator();
```

```
        // Act

        // Assert
        $this->assertNotNull($calculator);
    }

    public function testAddOneAndOne()
    {
        // Arrange
        $calculator = new Calculator();
        $num1 = 1;
        $num2 = 1;
        $expectedResult = 2;

        // Act
        $result = $calculator->add($num1, $num2);

        // Assert
        $this->assertEquals($expectedResult, $result);
    }
}
```

Note:

- we had to add `use` statements for the class we are testing (`App\Util\Calculator`) and the PHP Unit TestCase class we are extending (`use PHPUnit\Framework\TestCase`)

Run the tests - if all goes well we should see 3 out of 3 tests passing:

```
$ bin/phpunit

Testing
...                                                                 3 / 3 (100%)

Time: 00:00.014, Memory: 10.00 MB

OK (3 tests, 3 assertions)
```

1.7 Using a data provider to test with multiple datasets

Rather than writing lots of methods to test different additions, let's use a **data provider** (via an annotation comment), to provide a single method with many sets of input and expected output

values:

Here is our testing method:

```
/**
 * @dataProvider additionProvider
 */
public function testAdditionsWithProvider($num1, $num2, $expectedResult)
{
    // Arrange
    $calculator = new Calculator();

    // Act
    $result = $calculator->add($num1, $num2);

    // Assert
    $this->assertEquals($expectedResult, $result);
}
```

and here is the data provider (an array of arrays, with the right number of values for the parameters of `testAdditionsWithProvider(...)`:

```
public function additionProvider()
{
    return [
        [1, 1, 2],
        [2, 2, 4],
        [0, 1, 1],
    ];
}
```

Take special note of the annotation comment immediately before method `testAdditionsWithProvider(...)`:

```
/**
 * @dataProvider additionProvider
 */
```

The special comment starts with `/**`, and declares an annotation `@dataProvider`, followed by the name (identifier) of the method. Note especially that there are no parentheses `()` after the method name.

When we run Simple-PHPUnit now we see lots of tests being executed, repeatedly invoking `testAdditionsWithProvider(...)` with different arguments from the provider:

```
$ vendor/bin/simple-phpunit
PHPUnit 5.7.27 by Sebastian Bergmann and contributors.
```

```
Testing Project Test Suite
......                                                          6 / 6 (100%)

Time: 65 ms, Memory: 4.00MB

OK (6 tests, 6 assertions)
```

1.8 Configuring testing reports

In additional to instant reporting at the command line, PHPUnit offers several different methods of recording test output text-based files.

PHPUnit (when run with Symfony's Simple-PHPUnit) reads configuration settings from file `phpunit.dist.xml`. Most of the contents of this file (created as part of the installation of the Simple-PHPUnit package) can be left as their defaults. But we can add a range of logs by adding the following 'logging' element in this file.

Many projects follow a convention where testing output files are stored in a directory named `build`. We'll follow that convention below - but of course change the name and location of the test logs to anywhere you want.

Add the following into file `phpunit.dist.xml`:

```
<logging>
    <junit outputFile="junit.xml"/>
    <teamcity outputFile="./build/teamcity.txt"/>
    <testdoxHtml outputFile="./build/testdox.html"/>
    <testdoxText outputFile="./build/testdox.txt"/>
    <testdoxXml outputFile="./build/testdox.xml"/>
    <text outputFile="./build/logfile.txt"/>
</logging>
```

Figure 1.1 shows a screenshot of the contents of the created `/build` directory after Simple-PHPUnit has been run.

The `.txt` file version of **test dox** (**testdoxText**) is perhaps the simplest output - showing `[x]` next to a passed method and `[ ]` for a test that didn't pass. The text output turns the test method names into more English-like sentences:

```
Simple (App\Tests\Simple)
 [x] One plus one equals two

Calculator (App\Tests\Util\Calculator)
```

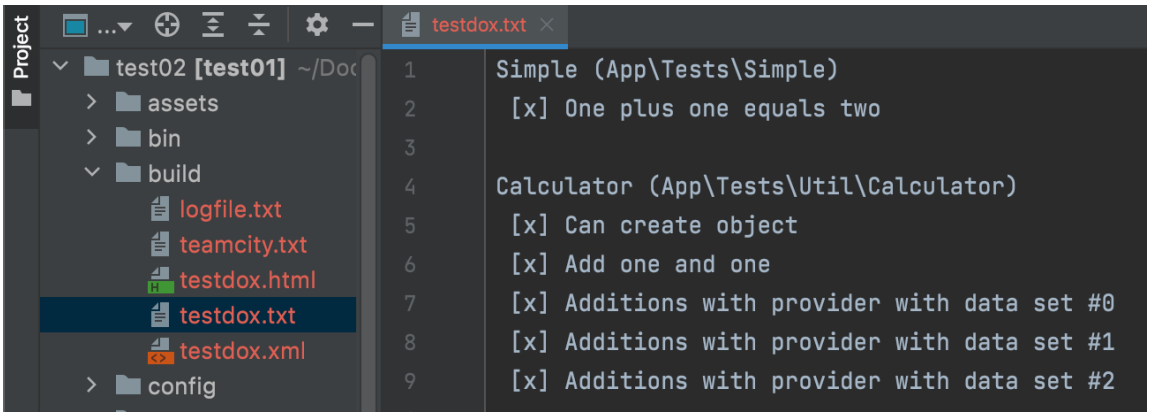

Figure 1.1: Contents of directory /build.

```
[x] Can create object
[x] Add one and one
[x] Additions with provider with data set #0
[x] Additions with provider with data set #1
[x] Additions with provider with data set #2
```

1.9 Testing for exceptions (project test03)

If our code throws an **Exception** while a test is being executed, and it was not caught, then we'll get an **Error** when we run our test.

For example, let's add a `divide(...)` method to our utility `Calculator` class:

```
public function divide($n, $divisor)
{
    if(empty($divisor)){
        throw new \InvalidArgumentException("Divisor must be a number");
    }

    return $n / $divisor;
}
```

In the code above we are throwing an `\InvalidArgumentException` when our `$divisor` argument is empty (0, null etc.).

Let's write a valid test ($1/1 = 1$) in class `CalculatorTest`:

```
public function testDivideOneAndOne()
{
    // Arrange
```

```
        $calculator = new Calculator();
        $num1 = 1;
        $num2 = 1;
        $expectedResult = 1;

        // Act
        $result = $calculator->divide($num1, $num2);

        // Assert
        $this->assertEquals($expectedResult, $result);
    }
```

This should pass.

Now let's try to write a test for 1 divided by zero. Not knowing how to deal with exceptions we might write something with a `fail(...)` instead of an `assert...`:

```
    public function testDivideOneAndZero()
    {
        // Arrange
        $calculator = new Calculator();
        $num1 = 1;
        $num2 = 0;
        $expectedResult = 1;

        // Act
        $result = $calculator->divide($num1, $num2);

        // Assert - FAIL - should not get here!
        $this->fail('should not have got here - divide by zero not permitted');
    }
```

But when we run simple-phpunit we'll get an error since the (uncaught) Exceptions is thrown before our `fail(...)` statement is reached:

```
$ vendor/bin/simple-phpunit
PHPUnit 9.5.19

Testing
.......E                                                          8 / 8 (100%)

Time: 00:00.028, Memory: 10.00 MB

There was 1 error:
```

```
1) App\Tests\Util\CalculatorTest::testDivideOneAndZero
InvalidArgumentException: Divisor must be a number

/Users/matt/test03/src/Util/Calculator.php:18
/Users/matt/test03/tests/Util/CalculatorTest.php:82

ERRORS!
Tests: 8, Assertions: 7, Errors: 1.
```

And our logs will confirm the failure:

```
Simple (App\Tests\Simple)
 [x] One plus one equals two

Calculator (App\Tests\Util\Calculator)
 [x] Can create object
 [x] Add one and one
 [x] Additions with provider with data set #0
 [x] Additions with provider with data set #1
 [x] Additions with provider with data set #2
 [x] Divide one and one
 [ ] Divide one and zero
```

1.10 PHPUnit `expectException(...)`

PHPUnit allows us to declare that we expect an exception - but we must declare this **before** we invoke the method that will throw the exception.

Here is our improved method, with `expectException(...)` and a better `fail(...)` statement, that tells us which exception was expected and not thrown:

```php
public function testDivideOneAndZero()
{
    // Arrange
    $calculator = new Calculator();
    $num1 = 1;
    $num2 = 0;
    $expectedResult = 1;

    // Expect exception - BEFORE you Act!
    $this->expectException(\InvalidArgumentException::class);
```

```
        // Act
        $result = $calculator->divide($num1, $num2);

        // Assert - FAIL - should not get here!
        $this->fail("Expected exception {\InvalidArgumentException::class} not thrown");
    }
```

Now all our tests pass:

```
$ vendor/bin/simple-phpunit
PHPUnit 9.5.19

Testing
........                                                          8 / 8 (100%)

Time: 00:00.026, Memory: 10.00 MB

OK (8 tests, 8 assertions)
```

1.11 Testing for custom Exception classes

While the built-in PHP Exceptions are find for simple projects, it is very useful to create custom exception classes for each project you create. Working with, and testing for, objects of custom Exception classes is very simple in Symfony:

1. Create your custom Exception class in `/src/Exception`, in the namespace `App\Exception`. For example you might create a custom Exception class for an invalid Currency in a money exchange system as follows:

```
// file: /src/Exception/UnknownCurrencyException.php
<?php

namespace App\Exception;

class UnknownCurrencyException extends \Exception
{
    public function __construct($message = null)
    {
        if(empty($message)) {
            $message = 'Unknown currency';
        }
```

```
        parent::__construct($message);
    }
}
```

2. Ensure your `/src/Util/Calculator.php` source code throws an instance of your custom Exception. For example:

```
use App\Exception\UnknownCurrencyException;

...

public function euroOnlyExchange(string $currency)
{
    $currency = strtolower($currency);
    if('euro' != $currency){
        throw new UnknownCurrencyException();
    }

    // other logic here ...
}
```

3. In your tests your must check for the expected custom Exception class. E.g. using the annotation approach:

```
use App\Exception\UnknownCurrencyException;

...

public function testInvalidCurrencyException()
{
    // Arrange
    $calculator = new Calculator();
    $currency = 'I am not euro';

    // Expect exception - BEFORE you Act!
    $this->expectException(UnknownCurrencyException::class);

    // Act
    // ... code here to trigger exception to be thrown ...
    $calculator->euroOnlyExchange($currency);

    // Assert - FAIL - should not get here!
    $this->fail("Expected exception {\Exception} not thrown");
```

```
}
```

1.12 Checking Types with assertions

Sometimes we need to check the **type** of a variable. We can do this using the `assertInternalType(...)` method.

For example:

```
$result = 1 + 2;

// check result is an integer
$this->assertInternalType('int', $result);
```

Learn more in the PHPUnit documentation:

- https://phpunit.de/manual/6.5/en/appendixes.assertions.html#appendixes.assertions.assertInternalType

1.13 Same vs. Equals

There are 2 similar assertions in PHPUnit:

- `assertSame(...)`: works like the `===` identity operator in PHP
- `assertEquals(...)`: works like the `==` comparison

When we want to know if the values inside (or referred to) by two variables or expressions are equivalent, we use the weaker `==` or `assertEquals(...)`. For example, do two variables refer to object-instances that contain the same property values, but may be different objects in memory.

When we want to know if the values inside (or referred to) by two variables are exactly the same, we use the stronger `===` or `assertSame(...)`. For example, do two variables both refer to the same object in memory.

The use of `assertSame(...)` is useful in unit testing to check the types of values - since the value returned by a function must refer to the same numeric or string (or whatever) literal. So we could write another way to test that a function returns an integer result as follows:

```
$expectedResult = 3;
$result = 1 + 2;

// check result is an integer
$this->assertSame($expectedResult, $result);
```

2

Web testing

2.1 Testing controllers with `WebTestCase`

Symfony provides a package for simulating web clients so we can (functionally) test the contents of HTTP Responses output by our controllers.

2.2 Creating a new project for web testing (project `test04`)

Do the following to setup a new project for web testing:

```
// create and 'cd' into project
$ symfony new --webapp test4_webTesting
$ cd test4_webTesting

// add testing package
$ composer req --dev symfony/test-pack
```

2.3 Create home page with a Default controller

Make a new `DefaultController` class:

```
symfony console make:controller Default
```

Let's edit the generated template to include the message `Hello World`. Edit `/templates/default/index.html.twig`

```
{% extends 'base.html.twig' %}

{% block body %}
<h1>Hello World</h1>

Hello World from the default controller
{% endblock %}
```

Let's also set the URL to simply `/`, and the route name to `defalt` for this route in `/src/Controller/DefaultController.php`:

```
class DefaultController extends AbstractController
{
    #[Route('/', name: 'default')]
    public function index(): Response
    {
        $template = 'default/index.html.twig';
        $args = [];
        return $this->render($template, $args);
    }
}
```

If we run a web server and visit the home page we should see our 'hello world' message in a browser - see Figure 2.1.

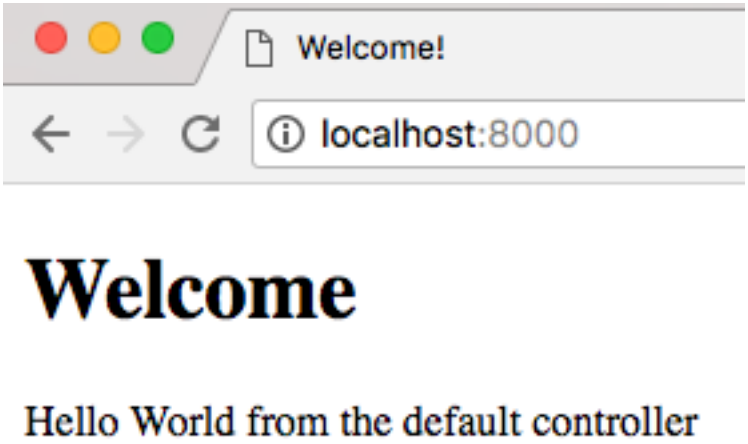

Figure 2.1: Home page.

2.4 Using the 'make' tool to create a web test

The Symfony console 'make' tool can be used to create web test classes:

```
$ symfony console make:test

// choose 'WebTestCase' when asked:

 Which test type would you like?:
  [TestCase       ] basic PHPUnit tests
  [KernelTestCase ] basic tests that have access to Symfony services
  [WebTestCase    ] to run browser-like scenarios, but that don't execute JavaScript code
  [ApiTestCase    ] to run API-oriented scenarios
  [PantherTestCase] to run e2e scenarios, using a real-browser or HTTP client and a real web server
 > WebTestCase

// name the new class 'HomePageTest' when asked:

Choose a class name for your test, like:
 * UtilTest (to create tests/UtilTest.php)
 * Service\UtilTest (to create tests/Service/UtilTest.php)
 * \App\Tests\Service\UtilTest (to create tests/Service/UtilTest.php)

 The name of the test class (e.g. BlogPostTest):
 > HomePageTest

 created: tests/HomePageTest.php

  Success!

 Next: Open your new test class and start customizing it.
 Find the documentation at https://symfony.com/doc/current/testing.html#functional-tests
```

2.5 Automating a test for the home page contents

Let's write a test class for our `DefaultController` class. So we create a new test class `/tests/Controller/DefaultControllerTest.php`. We'll write 2 tests, one to check that we get a 200 OK HTTP success code when we try to request `/`, and secondly that the content received in the HTTP Response contains the text `Hello World`:

```
namespace App\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;
use Symfony\Component\HttpFoundation\Response;

class DefaultControllerTest extends WebTestCase
{
    // methods go here
}
```

We see our class must extend `WebTestCase` from package `Symfony\Bundle\FrameworkBundle\Test\`, and also makes use of the Symfony Foundation `Response` class.

Our method to test for a 200 OK Response code is as follows:

```
public function testHomepageResponseCodeOkay()
{
    // Arrange
    $url = '/';
    $httpMethod = 'GET';
    $client = static::createClient();

    // Assert
    $client->request($httpMethod, $url);
    $statusCode = $client->getResponse()->getStatusCode();

    // Assert
    $this->assertSame(Response::HTTP_OK, $statusCode);
}
```

NOTE: Another way to check the Response status code is to use the `$this->assertResponseStatusCodeSame(<code` method. For example:

```
$this->assertResponseStatusCodeSame(Response::HTTP_OK);
```

We see how a web client object `$client` is created and makes a GET request to `/`. We see how we can interrogate the contents of the HTTP Response received using the `getResponse()` method, and within that we can extract the status code, and compare with the class constant `HTTP_OK` (200).

Here is our method to test for a Level 1 heading containing **exactly** `Hello World` (case sensitive):

```
public function testHomepageContentContainsHelloWorld(): void
{
    // Arrange
    $url = '/';
```

```
        $httpMethod = 'GET';
        $client = static::createClient();
        $searchText = 'Hello World';
        $cssSelector = 'h1';

        // Act
        $crawler = $client->request($httpMethod, $url);
        $content = $client->getResponse()->getContent();

        // Assert
        $this->assertResponseIsSuccessful();
        $this->assertSelectorTextContains($cssSelector, $searchText);
    }
```

We see how we can use the `assertSelectorTextContains` string method to search for the string `Hello World` in the content of the HTTP Response.

When we run PHPUnit we can see success both from the full-stops at the CLI, and in our log files. For example we case see the human-friendly `teestdox` report in `/build/testdox.txt` as follows:

```
Default Controller (App\Tests\Controller\DefaultController)
 [x] Homepage response code okay
 [x] Homepage content contains hello world
```

2.6 Testing text case-insensitively

I lost 30 minutes thinking my web app wasn't working! This was due to the difference between `Hello world` and `Hello World`: [w]orld vs [W]orld.

Luckily there is a specific assertion for testing text that ignores case:

- `assertStringContainsStringIgnoringCase(<needle>, <haystack>)`

Here's a full test method for case insensitively:

```
    public function testHomepageContentContainsHelloWorldIgnoreCase()
    {
        // Arrange
        $url = '/';
        $httpMethod = 'GET';
        $client = static::createClient();
        $searchText = 'heLLo worLD';
        $cssSelector = 'body';
```

```
        // Act
        $crawler = $client->request($httpMethod, $url);
        $content = $client->getResponse()->getContent();

        // Assert
        $this->assertStringContainsStringIgnoringCase($searchText, $content);
    }
```

2.7 Test multiple pages with a data provider

Avoid duplicating code when only the URL and search text changes, by writing a testing method fed by arrays of test input / expected values from a data provider method.

Here is a method with a provider, testing for `hello world` in the home page (route `/`), and `about` for the `/about` route:

```
    /**
     * @dataProvider basicPagesTextProvider
     */
    public function testPublicPagesContainBasicText(string $url, string $searchText)
    {
        // Arrange
        $httpMethod = 'GET';
        $client = static::createClient();

        // Act
        $crawler = $client->request($httpMethod, $url);
        $content = $client->getResponse()->getContent();

        // Assert
        $this->assertStringContainsStringIgnoringCase($searchText, $content);
    }

    public function basicPagesTextProvider(): array
    {
        return [
            ['/', 'hello WORLD'],
            ['/about', 'about'],
        ];
    }
```

2.8 Count the number of elements

We can use the `filter(...)` method of the crawler object to retrieve an array of elements matching a CSS selector.

For example, this statement asserts that there should be exactly 4 elements with CSS class `comment`:

```
$this->assertCount(4, $crawler->filter('.comment'));
```

2.9 Testing different content returned by the Response

Our test classes are subclasses of the Symfony `WebTestCase` class, which provides a number of methods for testing the contents of the HTTP Response.

So there are several useful assertions we can make based on CSS element selectors, including:

- `$this->assertSelectorExists($cssSelector)`
  - this assets that a given element is present in the Response content (such as `#formHeading`, `h1`, `title`, `body` etc.)
- `$this->assertSelectorNOTExists($cssSelector)`
  - as above but that the selector is NOT present in the Response
- `$this->assertSelectorTextContains($cssSelector, $searchText)`
  - this assets that a given element (such as `h1`, `title`, `body` etc.) contains some given text
- `$this->assertSelectorTextNotContains($cssSelector, $searchText)`
  - as above but that the search text is NOT present in selected element of the Response
- `$this->assertSelectorTextSame($cssSelector, $searchText)`
  - this assets that a given element (such as `h1`, `title`, `body` etc.) contains some given text

Here we see a test with many of these assertions demonstrated:

```
public function testHomepageBodyContentContainsHelloAndNotDinasaur(): void
{
    // Arrange
    $url = '/';
    $httpMethod = 'GET';
    $client = static::createClient();
    $searchText = 'Hello World';
    $textNotInPage = 'Dinosaur';
    $cssSelector = 'body';
```

```
        // Act
        $crawler = $client->request($httpMethod, $url);
        $content = $client->getResponse()->getContent();

        // Assert
        $this->assertResponseIsSuccessful();

        // has a 'body' element
        $this->assertSelectorExists($cssSelector);

        // does NOT have a 'footer' element
        $this->assertSelectorNotExists('footer');

        // 'body' contains 'Hello'
        $this->assertSelectorTextContains($cssSelector, $searchText);

        // 'body' dopes NOT contains 'Dinosaur'
        $this->assertSelectorTextNotContains($cssSelector, $textNotInPage);

        // 'h1' exact text of 'Hello World'
        $this->assertSelectorTextSame('h1', $searchText);
    }
```

2.10 Testing route followed after link clicked or form submission

Another useful assertion is `$this->assertRouteSame(<routeName>)` which asserts that the Response now received is from a link or redirect followed, corresponding to the given route name.

For example, the following test checks that after clicking the `/about` link, the Response is from the `/about` route (more about links in the next section):

```
    public function testHomepageResponseProperties(): void
    {
        // Arrange
        $url = '/';
        $httpMethod = 'GET';
        $client = static::createClient();
        $client->followRedirects();
        $homeRoute = 'home';
        $aboutRoute = 'about';
```

```
        // Act
        $crawler = $client->request($httpMethod, $url);

        // click ABOUT link
        $linkText = 'about';
        $link = $crawler->selectLink($linkText)->link();
        $client->click($link);

        // now on page for about route
        $this->assertRouteSame($aboutRoute);
    }
```

2.11 Testing links (project `test05`)

We can test links with our web crawler as follows:

- get reference to crawler object when you make the initial request

  ```
  $httpMethod = 'GET';
  $url = '/about';
  $crawler = $client->request($httpMethod, $url);
  ```

- select a link with:

  ```
  $linkText = 'login';
  $link = $crawler->selectLink($linkText)->link();
  ```

- click the link with:

  ```
  $client->click($link);
  ```

- then check the content of the new request

  ```
  // set $expectedText to what should in page when link has been followed ...
  // Assert
  $content = $client->getResponse()->getContent();
  $this->assertStringContainsString($expectedText, $content);

  $this->assertSelectorTextContains($cssSelector, $expectedText);
  ```

For example, if we create a new 'about' page Twig template '/templates/default/about.html.twig':

```
{% extends 'base.html.twig' %}
```

```
{% block body %}
<h1>About page</h1>

    <p>
        About this great website!
    </p>

{% endblock %}
```

and a `DefaultController` method to display this page when the route matches `/about`:

```
#[Route('/about', name: 'about')]
public function about(): Response
{
    $template = 'default/about.html.twig';
    $args = [];
    return $this->render($template, $args);
}
```

If we add to our base Twig template links to the homepage and the about, in template `/templates/base.html.twig`:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
    </head>
    <body>

    <nav>
        <ul>
            <li>
                <a href="{{ url('homepage') }}">home</a>
            </li>
            <li>
                <a href="{{ url('about') }}">about</a>
            </li>
        </ul>
    </nav>

        {% block body %}{% endblock %}
```

```
    </body>
</html>
```

We can now write a test method to:

- request the homepage `/`
- select and click the `about` link
- test that the content of the new response is the 'about' page if it contains 'about page'

Here is our test method:

```
public function testHomePageLinkToAboutWorks()
{
    // Arrange
    $url = '/';
    $httpMethod = 'GET';
    $client = static::createClient();
    $client->followRedirects();
    $searchText = 'About';
    $linkText = 'about';
    $cssSelector = 'body';

    // Act
    $crawler = $client->request($httpMethod, $url);
    $link = $crawler->selectLink($linkText)->link();
    $client->click($link);
    $content = $client->getResponse()->getContent();

    // Assert
    $this->assertStringContainsString($searchText, $content);

    $this->assertSelectorTextContains($cssSelector, $searchText);
}
```

2.11.1 Instruct client to 'follow redirects'

In most cases we want our testing web-crawler client to follow redirects. So we need to add the `$client->followRedirects(true)` statement immediately after creating the client object.

```
public function testExchangePage()
{
    $httpMethod = 'GET';
    $url = '/calc';
```

```
    $client = static::createClient();
    $client->followRedirects(); // <<<<<<<<<<<<<<<<< default is 'true'
    $client->request($httpMethod, $url);
    $this->assertSame(Response::HTTP_OK, $client->getResponse()->getStatusCode());
}
```

3

Testing web forms

3.1 Testing forms (project `test06`)

Testing forms is similar to testing links, in that we need to get a reference to the form (via its submit button), then insert out data, then submit the form, and examine the content of the new response received after the form submission.

Assume we have a Calculator class as follows in `/src/Util/Calculator.php`:

```
namespace App\Util;

class Calculator
{
    public function add($n1, $n2)
    {
        return $n1 + $n2;
    }

    public function subtract($n1, $n2)
    {
        return $n1 - $n2;
    }

    public function divide($n, $divisor)
```

```
    {
        if(empty($divisor)){
            throw new \InvalidArgumentException("Divisor must be a number");
        }

        return $n / $divisor;
    }

    public function process($n1, $n2, $process)
    {
        switch($process){
            case 'subtract':
                return $this->subtract($n1, $n2);
                break;
            case 'divide':
                return $this->divide($n1, $n2);
                break;
            case 'add':
            default:
                return $this->add($n1, $n2);
        }
    }
}
```

Assume we also have a `CalculatorController` class in `/src/Controller/`:

```
namespace App\Controller;

use App\Util\Calculator;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class CalcController extends Controller
{
    ... methods go here ...
}
```

There is a calculator home page that displays the form Twig template at `/templates/calc/index.html.twig`:

```
#[Route('/calculator', name: 'app_calculator_index')]
public function index(): Response
{
```

```
        $template = 'calculator/index.html.twig';
        $args = [];
        return $this->render($template, $args);
    }
```

and a 'process' controller method to received the form data (n1, n2, operator) and process it: There is a calculator home page that displays the form Twig template at `/templates/calculator/index.html.twig`:

```
use Symfony\Component\HttpFoundation\Request;
use App\Util\Calculator;

...

#[Route('/calculator/process', name: 'app_calculator_process')]
public function processAction(Request $request): Response
{
    // extract name values from POST data
    $n1 = $request->request->get('num1');
    $n2 = $request->request->get('num2');
    $operator = $request->request->get('operator');

    $calc = new Calculator();
    $answer = $calc->process($n1, $n2, $operator);

    $template = 'calculator/result.html.twig';
    $args =  [
        'n1' => $n1,
        'n2' => $n2,
        'operator' => $operator,
        'answer' => $answer
    ];

    return $this->render($template, $args);
}
```

The Twig template to display our form looks as follows `/templates/calculator/index.html.twig`:

```
{% extends 'base.html.twig' %}

{% block body %}
<h1>Calculator home</h1>
```

```
    <form method="post" action="{{ url('app_calculator_process') }}">
        <p>
            Num 1:
            <input type="text" name="num1" value="1">
        </p>
        <p>
            Num 2:
            <input type="text" name="num2" value="1">
        </p>
        <p>
            Operation:
            <br>
            ADD
            <input type="radio" name="operator" value="add" checked>
            <br>
            SUBTRACT
            <input type="radio" name="operator" value="subtract">
            <br>
            DIVIDE
            <input type="radio" name="operator" value="divide">
        </p>

        <p>
            <input type="submit" name="calc_submit">
        </p>
    </form>

{% endblock %}
```

and the Twig template to confirm received values, and display the answer `result.html.twig` contains:

```
<h1>Calc RESULT</h1>
<p>
    Your inputs were:
    <br>
    n1 = {{ n1 }}
    <br>
    n2 = {{ n2 }}
    <br>
    operator = {{ operator }}
<p>
```

```
    answer = {{ answer }}
```

3.2 Create a `CalculatorTest` class

Using the Symfony make tool, let's create a `CalculatorTest` class in `/src/Tests/`:

```
$symfony console make:test

 Which test type would you like?:
  [TestCase       ] basic PHPUnit tests
  [KernelTestCase ] basic tests that have access to Symfony services
  [WebTestCase    ] to run browser-like scenarios, but that don't execute JavaScript code
  [ApiTestCase    ] to run API-oriented scenarios
  [PantherTestCase] to run e2e scenarios, using a real-browser or HTTP client and a real web server
 > WebTestCase


Choose a class name for your test, like:
 * UtilTest (to create tests/UtilTest.php)
 * Service\UtilTest (to create tests/Service/UtilTest.php)
 * \App\Tests\Service\UtilTest (to create tests/Service/UtilTest.php)

 The name of the test class (e.g. BlogPostTest):
 > CalculatorTest

 created: tests/CalculatorTest.php

  Success!
```

3.3 Test we can get a reference to the form

Let's test that can see the form page

```
public function testHomepageResponseCodeOkay()
{
    // Arrange
    $url = '/calculator';
    $httpMethod = 'GET';
    $client = static::createClient();
    $expectedResult = Response::HTTP_OK;
```

```
        // Assert
        $client->request($httpMethod, $url);
        $statusCode = $client->getResponse()->getStatusCode();

        // Assert
        $this->assertSame($expectedResult, $statusCode);
    }
```

Let's test that we can get a reference to the form on this page, via its 'submit' button:

```
    public function testFormReferenceNotNull()
    {
        // Arrange
        $url = '/calculator';
        $httpMethod = 'GET';
        $client = static::createClient();
        $crawler = $client->request($httpMethod, $url);
        $buttonName = 'calc_submit';

        // Act
        $buttonCrawlerNode = $crawler->selectButton($buttonName);
        $form = $buttonCrawlerNode->form();

        // Assert
        $this->assertNotNull($form);
    }
```

NOTE: We have to give each form button we wish to test either a `name` or `id` attribute. In our example we gave our calculator form the `name` attribute with value `calc_submit`:

```
    <input type="submit" name="calc_submit">
```

3.4 Submitting the form

Assuming our form has some default values, we can test submitting the form by then checking if the content of the response after clicking the submit button contains test 'Calc RESULT':

```
    public function testCanSubmitAndSeeResultText()
    {
        // Arrange
        $url = '/calculator';
        $httpMethod = 'GET';
```

```
        $client = static::createClient();
        $crawler = $client->request($httpMethod, $url);
        $expectedContentAfterSubmission = 'Calc RESULT';
        $expectedContentLowerCase = strtolower($expectedContentAfterSubmission);
        $buttonName = 'calc_submit';

        // Act
        $buttonCrawlerNode = $crawler->selectButton($buttonName);
        $form = $buttonCrawlerNode->form();

        // submit the form
        $client->submit($form);

        // get content from next Response
        $content = $client->getResponse()->getContent();
        $contentLowerCase = strtolower($content);

        // Assert
        $this->assertContains($expectedContentLowerCase, $contentLowerCase);
    }
```

3.5 Entering form values then submitting

Once we have a reference to a form (`$form`) entering values is completed as array entry:

```
$form['num1'] = 1;
$form['num2'] = 2;
$form['operator'] = 'add';
```

So we can now test that we can enter some values, submit the form, and check the values in the response generated.

Let's submit 1, 2 and `add`:

```
public function testSubmitOneAndTwoAndValuesConfirmed()
{
    // Arrange
    $url = '/calculator';
    $httpMethod = 'GET';
    $client = static::createClient();
    $crawler = $client->request($httpMethod, $url);
    $buttonName = 'calc_submit';
```

```
        // Act
        $buttonCrawlerNode = $crawler->selectButton($buttonName);
        $form = $buttonCrawlerNode->form();

        $form['num1'] = 1;
        $form['num2'] = 2;
        $form['operator'] = 'add';

        // submit the form & get content
        $crawler = $client->submit($form);
        $content = $client->getResponse()->getContent();

        // Assert
        $this->assertStringContainsString(
            '1',
            $content
        );
        $this->assertStringContainsString(
            '2',
            $content
        );
        $this->assertStringContainsString(
            'add',
            $content
        );
    }
```

The test above tests that after submitting the form we see the values submitted confirmed back to us.

3.6 Testing we get the correct result via form submission

Assuming all our `Calculator`, methods have been inidividudally **unit tested**, we can now test that after submitting some values via our web form, we get the correct result returned to the user in the final response.

Let's submit 1, 2 and `add`, and look for `3` in the final response:

```
    public function testSubmitOneAndTwoAndResultCorrect()
```

```
{
    // Arrange
    $url = '/calculator';
    $httpMethod = 'GET';
    $client = static::createClient();
    $num1 = 1;
    $num2 = 2;
    $operator = 'add';
    $expectedResult = 3;
    // must be string for string search
    $expectedResultString = $expectedResult . '';
    $buttonName = 'calc_submit';

    // Act

    // (1) get form page
    $crawler = $client->request($httpMethod, $url);

    // (2) get reference to the form
    $buttonCrawlerNode = $crawler->selectButton($buttonName);
    $form = $buttonCrawlerNode->form();

    // (3) insert form data
    $form['num1'] = $num1;
    $form['num2'] = $num2;
    $form['operator'] = $operator;

    // (4) submit the form
    $crawler = $client->submit($form);
    $content = $client->getResponse()->getContent();

    // Assert
    $this->assertStringContainsString($expectedResultString, $content);
```

That's it - we can now select forms, enter values, submit the form and interrogate the response after the submitted form has been processed.

3.7 Selecting form, entering values and submitting in one step

Using the **fluent** interface,, Symfony allows us to combine the steps of selecting the form, setting form values and submitting the form. E.g.:

```
$client->submit($client->request($httpMethod, $url)->selectButton($buttonName)->form([
    'num1'  => $num1,
    'num2'  => $num2,
    'operator'  => $operator,
]));
```

So we can write a test with fewer steps if we wish:

```
public function testSelectSetValuesSubmitInOneGo()
{
    // Arrange
    $url = '/calc';
    $httpMethod = 'GET';
    $client = static::createClient();
    $num1 = 1;
    $num2 = 2;
    $operator = 'add';
    $expectedResult = 3;
    // must be string for string search
    $expectedResultString = $expectedResult . '';
    $buttonName = 'calc_submit';

    // Act
    $client->submit($client->request($httpMethod, $url)->selectButton($buttonName)->form([
            'num1'  => $num1,
            'num2'  => $num2,
            'operator'  => $operator,
    ]));
    $content = $client->getResponse()->getContent();

    // Assert
    $this->assertStringContainsString($expectedResultString, $content);
}
```

3.8 Using a Data Provider to test forms

Let's list some operations and numbers and expected answers as a Data Provider array, and have a single test method automatically loop through testing all those data sets.

```
public function equationsProvider(): array
{
    return [
        [1, 1, 'add', 2],
        [1, 2, 'add', 3],
        [5, 2, 'subtract', 3],
        [5, 4, 'subtract', 1],
    ]
}
```

We can now write a parameterized test method, with the required speical annotation comment naming the Data Provider method:

```
/**
 * @dataProvider equationsProvider
 */
public function testSelectSetValuesSubmitInOneGoWithProvider(int $num1, int $num2, string $operator
{
    // Arrange
    $url = '/calculator';
    $httpMethod = 'GET';
    $client = static::createClient();

    // must be string for string search
    $buttonName = 'calc_submit';

    // Act
    $client->submit($client->request($httpMethod, $url)->selectButton($buttonName)->form([
        'num1'  => $num1,
        'num2'  => $num2,
        'operator'  => $operator,
    ]));
    $content = $client->getResponse()->getContent();

    // Assert
    $this->assertStringContainsString($expectedAnswer, $content);
}
```

We can see our 4 data sets used in teh TextDox output file (`/build/textdox.txt`):

```
Calculator (App\Tests\Controller\Calculator)
 [x] Can visit calculator page okay
 [x] Form reference not null
 [x] Can submit and see result text
 [x] Submit one and two and values confirmed
 [x] Submit one and two and result correct
 [x] Select set values submit in one go
 [x] Select set values submit in one go with provider with data set #0
 [x] Select set values submit in one go with provider with data set #1
 [x] Select set values submit in one go with provider with data set #2
 [x] Select set values submit in one go with provider with data set #3
```

4

Code coverage and xDebug

4.1 xDebug

for code coverege in PHP almost all projects use the open source xDebug tool. You'll need to have this installed to be able to generate code coverage reports for your project.

- https://xdebug.org/

4.2 Code Coverage

It's good to know how **much** of our code we have tested, e.g. how many methods or logic paths (e.g. if-else- branches) we have and have not tested.

Code coverage reports can be text, XML or nice-looking HTML. See Figure **??** for a screenshot of an HTML coverage report for a `Util` class with 4 methods. We can see that while `add` and `divide` have been fully (100%) covered by tests, methods `subtract` and `process` are insufficiently covered.

This is known as code coverage, and easily achieved by:

1. Adding a line to the PHPUnit configuration file (`php.ini`)
2. Ensuring the **xDebug** PHP debugger is installed and activated

See Appendix **??** for these stesp.

Code Coverage								
	Functions and Methods				Lines			
Total		50.00%	2 / 4	CRAP		72.73%	8 / 11	
Calculator		50.00%	2 / 4	9.30		72.73%	8 / 11	
add		100.00%	1 / 1	1		100.00%	1 / 1	
subtract		0.00%	0 / 1	2		0.00%	0 / 1	
divide		100.00%	1 / 1	2		100.00%	3 / 3	
process		0.00%	0 / 1	4.59		66.67%	4 / 6	

Figure 4.1: Screenshot of HTML coverage report.

4.3 Generating Code Coverage HTML report

Add the following element as a child to the `<logging>` element in file `phpuninit.xml.dist`:

```
<log type="coverage-html" target="./build/report"/>
```

So the full content of the `<logging>` element is now:

```
<logging>
    <log type="coverage-html" target="./build/report"/>
    <log type="junit" target="./build/logfile.xml"/>
    <log type="testdox-html" target="./build/testdox.html"/>
    <log type="testdox-text" target="./build/testdox.txt"/>
    <log type="tap" target="./build/logfile.tap"/>
</logging>
```

Now when you run `vendor/bin/simple-phpunit` you'll see a new directory `report` inside `/build`. Open the `index.html` file in `/build/report` and you'll see the main page of your coverage report. See Figure 4.2.

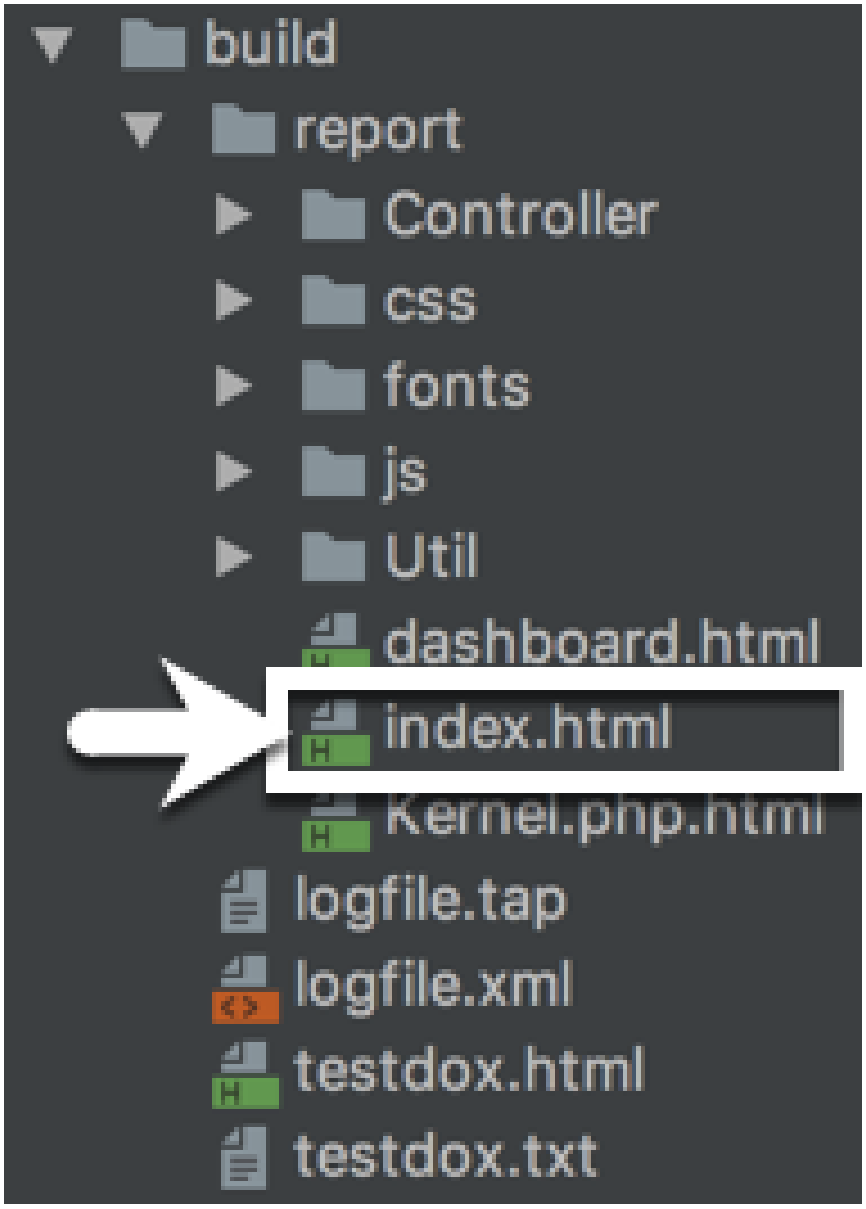

Figure 4.2: Build files showing `index.html` in `/build/report`.

4.4 Tailoring the 'whitelist'

PHPUnit decides which soruces file to analyse and build coverage reports for by using a 'whitelist' - i.e. a list of just those files and/or directories that we are interested in at this point in time. The whitelist is inside the `<filter>` element in PHPUnity configuration file 'phpunit.xml.dist'.

the default whitelist is `./src` - i.e **all** files in our source directory. But, for example, this will include Kernel, which we generally don't touch. So if you want to go **GREEN** for everything in your coverage report, then you can list only those directories inside `/src` that you are interested in.

For our example above we were working with classes in `/src/Util` and `src/Controller`, so that's what we can list in our 'whitelist'. You can always 'disable' lines in XML by wrapping an XML command around them `<-- ... -->`, which we've done below to the default `./src/` white list element:

```
<filter>
    <whitelist>
        <!--
            // ignore this element for now ...
            <directory>./src/</directory>
        -->
        <directory>./src/Controller</directory>
        <directory>./src/Util</directory>
    </whitelist>
</filter>
```

Part II

Publishing Symfony websites

5

Publishing your Symfony website

5.1 Requirements for Symfony publishing

You need the following to run Symfony on an internet host:

- an up to date version of PHP (8.1+ at present)
- a MySQL database (or another DB type supported by Symfony)
- a way to setup your project code (e.g. Composer)
- a way to setup your database (e.g. SSH terminal to run fixtures or MySQL dumps or client connection)

Traditional hosting companies, that don't offer SSH terminals, may require you to use FTP and online MySQL clients (such as PHPMyAdmin) to setup your project. Once setup, they'll run fine, but there can be a bunch of fiddly steps with online Control Panels and so on.

Having setup several PHP and Symfony projects for companies and organisations with traditional hosting companies in the past, I can say from experience that unless you are doing it every day, it's a fiddly business, especially when you want to be spending your time adding and testing features to the website project rather than administering the site.

5.2 Simplest ways to host Symfony projects

There are 2 easy ways to host Symfony websites, both supporting **CD (Continuous Deployment)** whereby commits pushed to the **master** branch of a Github (or similar) cloud repository are pulled down and the app restarted automatically, triggered by web "hooks" - event messages to the hosting servers each time a new commit is pushed:

- Symfony Cloud, from Sensio Labs, the creators of Symfony. See Figure 5.1.

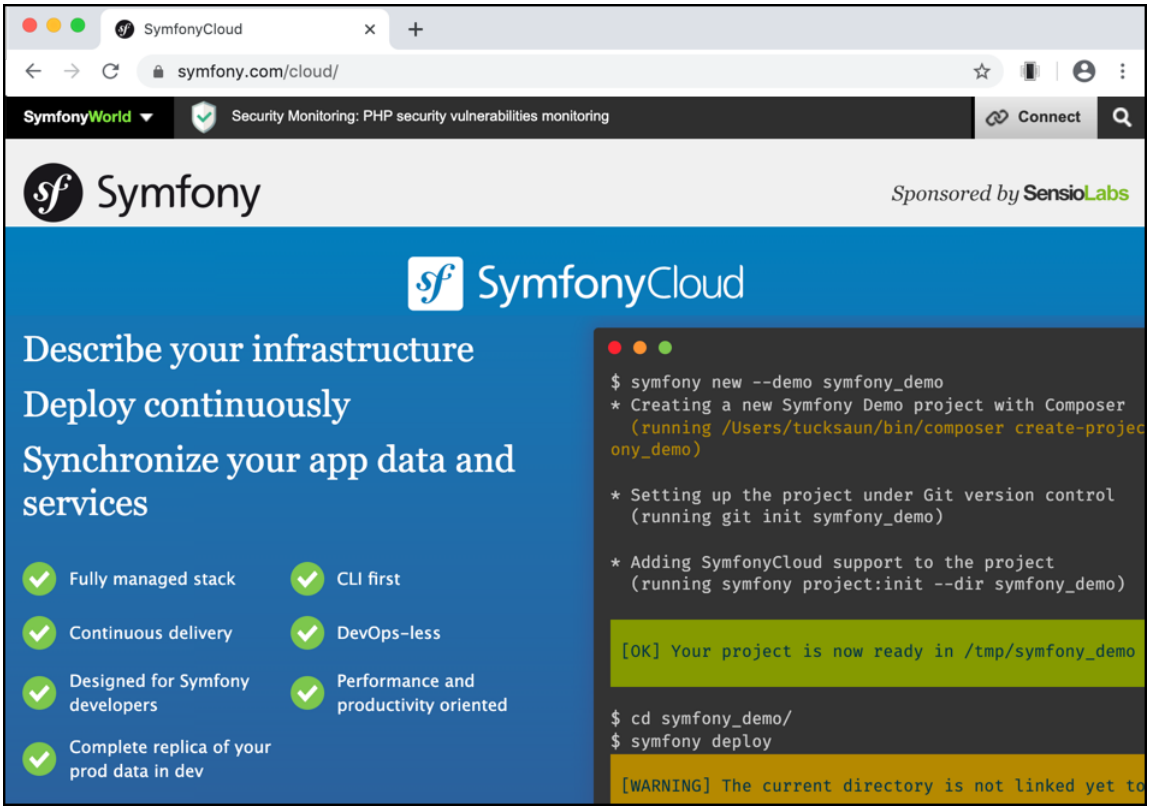

Figure 5.1: Symfony Cloud - from the creators of Symfony.

- PAAS - PHP-As-A-Service hosting companies
  - these companies specialise in PHP projects, and provide PHP environment variables, MySQL integrations, Github hooks and so on. Fore example see Figure 5.2 to see the site for Fortrabbit.com.

Since it's cheaper, and still very straightforward, we'll go through the steps for publishing with **Fortrabbit**.

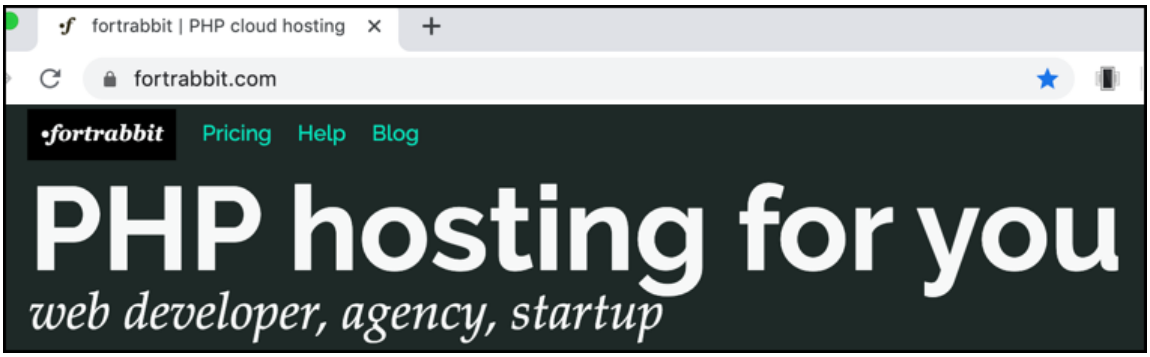

Figure 5.2: Fortrabbit.com - PHP-As-A-Service hosting.

6

Setting up project ready for Fortrabbit

6.1 Updating the names of our MySQL variables to match Fortrabbit ones

By simply changing the names (identifiers) of the variables in our `.env` file, it means our application will work locally with these settings and also work with no further changes when published to Fortrabbit.

Choose a **NEW** database name, e.g. I've chosen `week10demo` here. Now edit your project's `.env` file to now use variables `MYSQL_USER`, `MYSQL_PASSWORD`, `MYSQL_HOST` and `MYSQL_DATABASE` as follows, with **your** local root password and new database name:

```
MYSQL_USER=root
MYSQL_PASSWORD=passpass
MYSQL_HOST=127.0.0.1:3306
MYSQL_DATABASE=week10demo
DATABASE_URL=mysql://${MYSQL_USER}:${MYSQL_PASSWORD}@${MYSQL_HOST}/${MYSQL_DATABASE}
```

See Figure 6.1 to see these Fortrabbit MySQL variables for Symfony project.

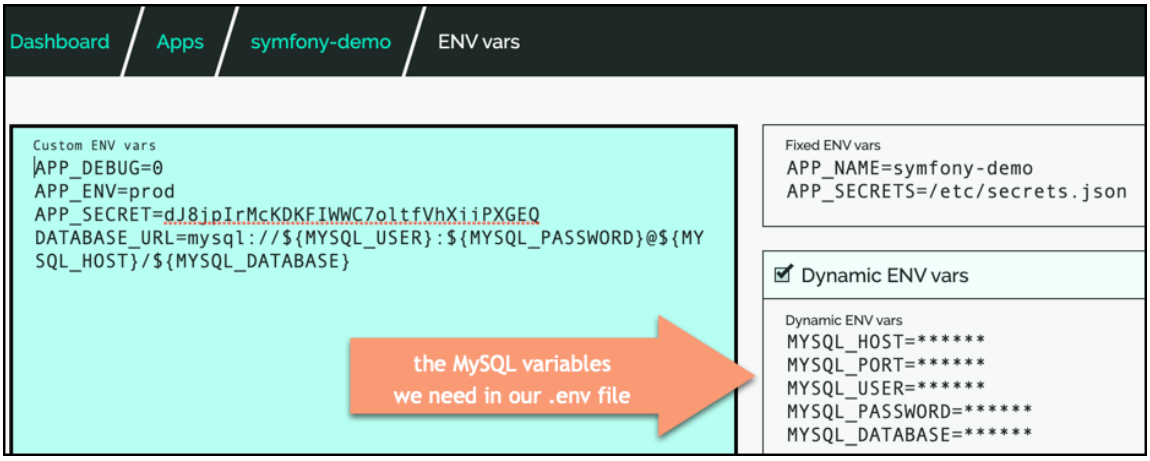

Figure 6.1: The Fortrabbit MySQL environment variables.

6.2 Create new DB locally, and make fresh migrations

Since both locally, and remotely we'll have a new DB, do the following to keep things in step:

1. Delete any contents in folder `/migrations` folder (but not the folder itself)
2. Create the new local database schema with `symfony console doctrine:database:create`
3. Create a migration for this new database with `symfony console make:migration`

We now have a new clean migration ready to use with our remote database.

6.3 Creating the `/public/.htaccess` Apache server routing file

This is a solved problem, since there is a Symfony community Composer Flex "recipe" to copy into our project the file we need.

Type the following a the command line:

```
composer require symfony/apache-pack
```

You'll be asked to say "yes" since this is a community contribution and not officially part of the Symfony project:

```
$ composer require symfony/apache-pack

Using version ^1.0 for symfony/apache-pack
./composer.json has been updated
Loading composer repositories with package information
```

```
Updating dependencies (including require-dev)
Restricting packages listed in "symfony/symfony" to "5.0.*"
Package operations: 1 install, 0 updates, 0 removals
  - Installing symfony/apache-pack (v1.0.1): Loading from cache
Writing lock file
Generating autoload files
ocramius/package-versions: Generating version class...
ocramius/package-versions: ...done generating version class
Symfony operations: 1 recipe (b11f4293313a650b4596551c0c2bb403)
  -  WARNING  symfony/apache-pack (>=1.0): From github.com/symfony/recipes-contrib:master
    The recipe for this package comes from the "contrib" repository, which is open to community con
    Review the recipe at https://github.com/symfony/recipes-contrib/tree/master/symfony/apache-pack

    Do you want to execute this recipe?
    [y] Yes
    [n] No
    [a] Yes for all packages, only for the current installation session
    [p] Yes permanently, never ask again for this project
    (defaults to n): y
```

Say "y" here!

```
  - Configuring symfony/apache-pack (>=1.0): From github.com/symfony/recipes-contrib:master
Executing script cache:clear [OK]
Executing script assets:install public [OK]

Some files may have been created or updated to configure your new packages.
Please review, edit and commit them: these files are yours.
```

You should then see a new file `.htaccess` in the `/public` folder. See Figure 6.2.

That's it - we've now prepared our local Symfony project for publishing at Fortrabbit!

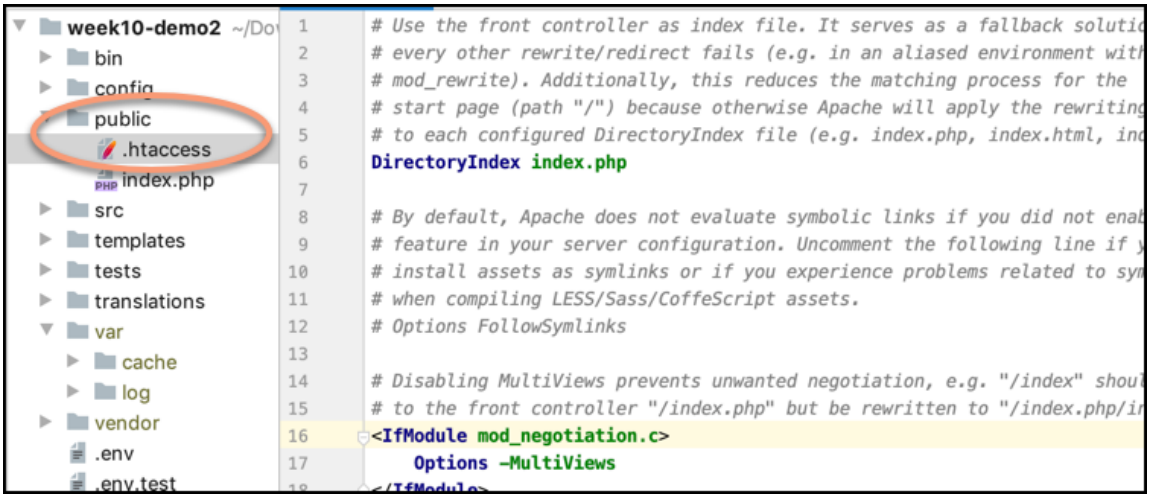

Figure 6.2: Screenshot of recipe-created `/public/.htaccess` file.

7

Publishing with Fortrabbit.com

7.1 Main steps for PAAS publishing with Fortrabbit

Having setup our Symfony project (with `.htaccess` and renamed `.env` MySQL variables and a fresh DB migration), we need to be able to the following to get our Symfony project published with Fortrabbit:

- create a Fortrabbit account, and setup SSH security keys so our local computer can securely communicate with the Fortrabbit servers
  - follow the steps in the Fortrabbit SHH keys documentation
  - see Figure 7.1
- link a local project to a Fortrabbit Github repository (seee steps in next section)
  - so we can **push** our code to the Fortrabbit repo to trigger a rebuild of the project
- work in an SSH terminal to run migrations & fixtures etc. (see later this chapter)

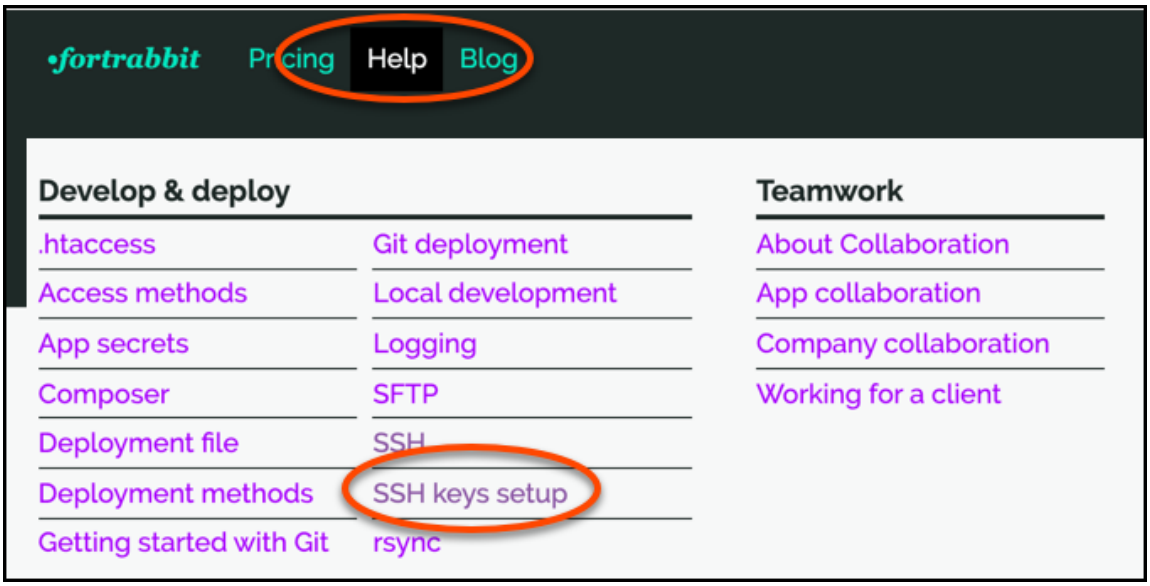

Figure 7.1: Fortrabbit ssh setup documentation.

7.2 Create a new Fortrabbit Symfony project

It's very straightforward to setup a new Symfony project on Fortrabbit:

1. Create a new PHP app in Fortrabbit. See Figure 7.2.
2. Choose Symfony project type. See Figure 7.3.
3. Choose European data centre. See Figure 7.4.

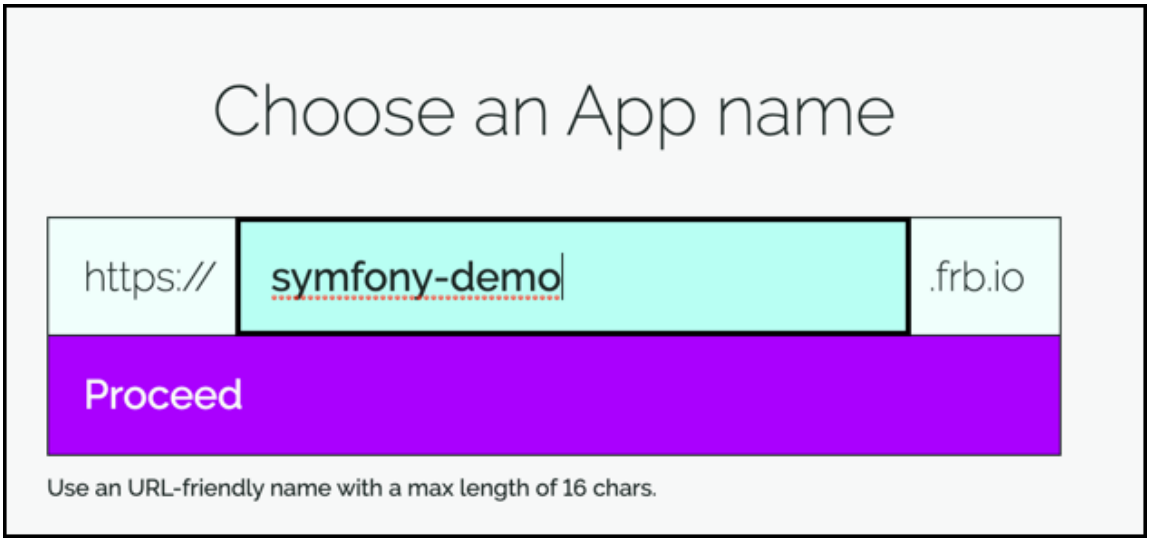

Figure 7.2: Create new app.

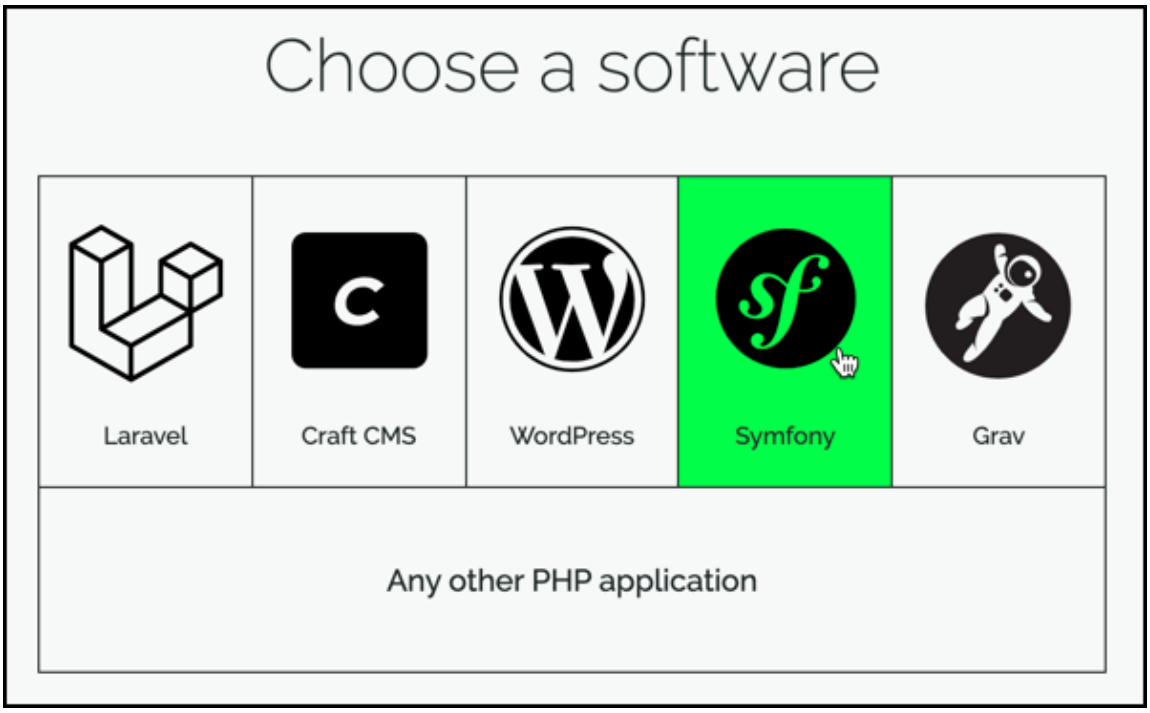

Figure 7.3: Choose Symfony project type.

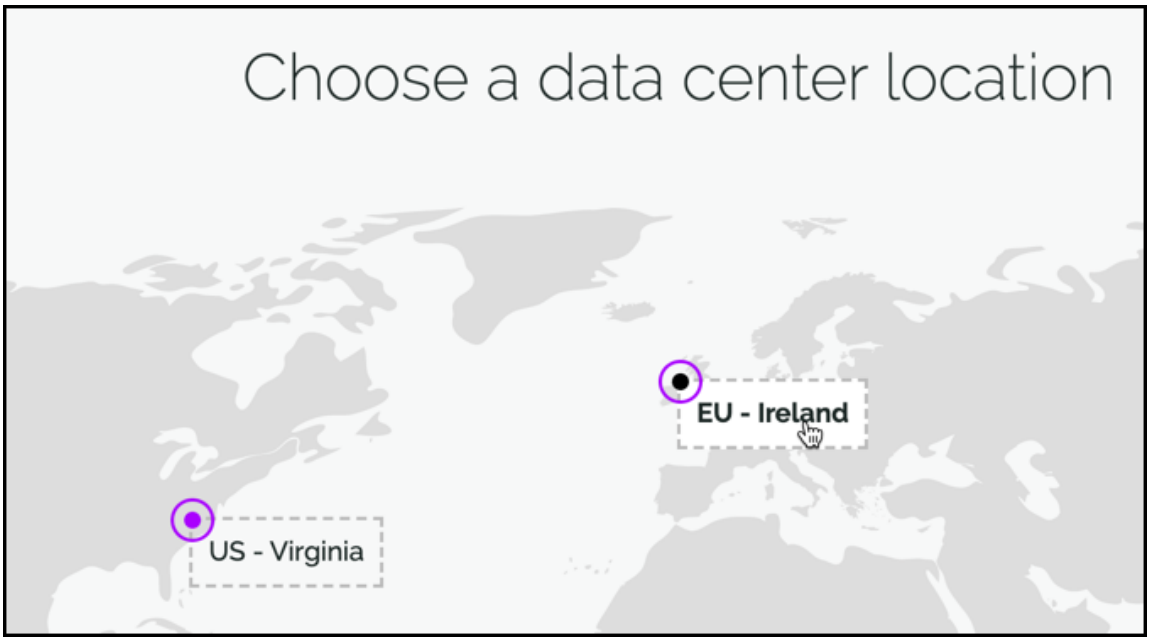

Figure 7.4: Choose EU data centre.

7.3 Choose plan - the Trial is free!

There are several plans available. At the time of writing they offered:

- Light at €5 per month
- Standard at €15 per month
- Plus at €30 per month

AND there is also a "free trial" option - this trial app will be available for 24-48 hours - long enough for testing …

7.4 Temporarily set project environment to `dev` so we can load DB fixtures

There is an issue in that Fortrabbit sets the Symfony environment as `prod`, for production, i.e. a running live website. This excludes things like the Symfony profiler, tests and so on. However, it also stops us from being able to load **fixtures** via Doctrine. This makes sense, since we don't want to reset the database for a live system.

While there are several ways to allow us to load fixtures, the simplest is to temporarily change the Fortrabbit project to the `dev` environnent. We do this by editing the projects **Custome ENV variable** `APP_ENV` from `prod` to `dev`. See Figures 7.5 and 7.6.

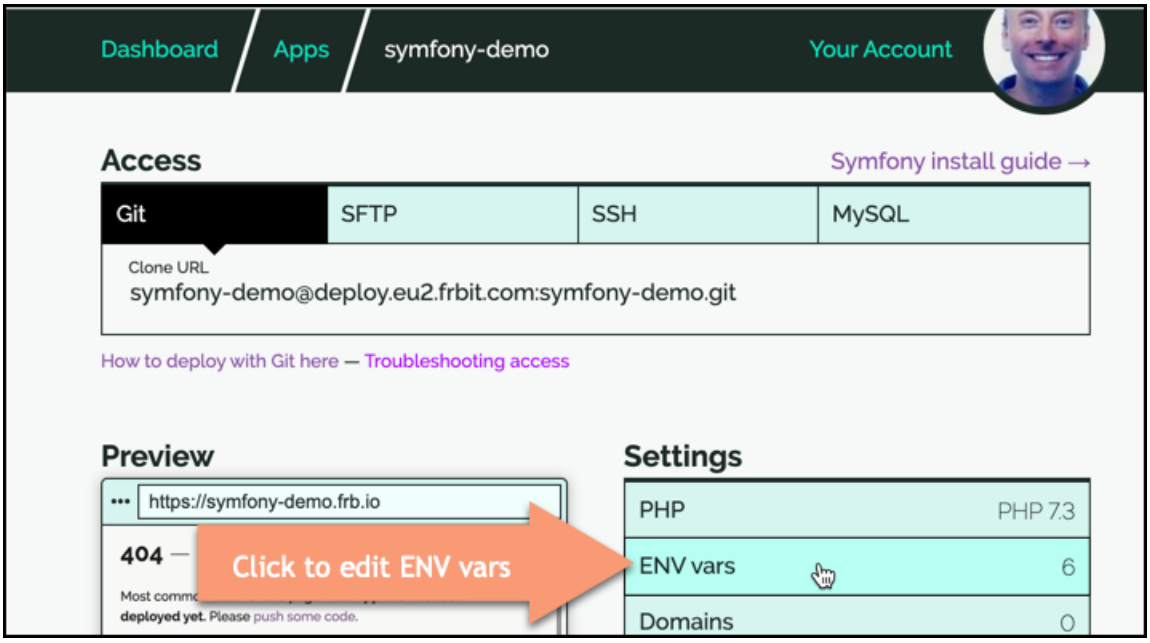

Figure 7.5: Edit ENV vars for Fortrabbit project.

NOTE: Don't forget to change this back to `prod` after you have loaded fixtures …

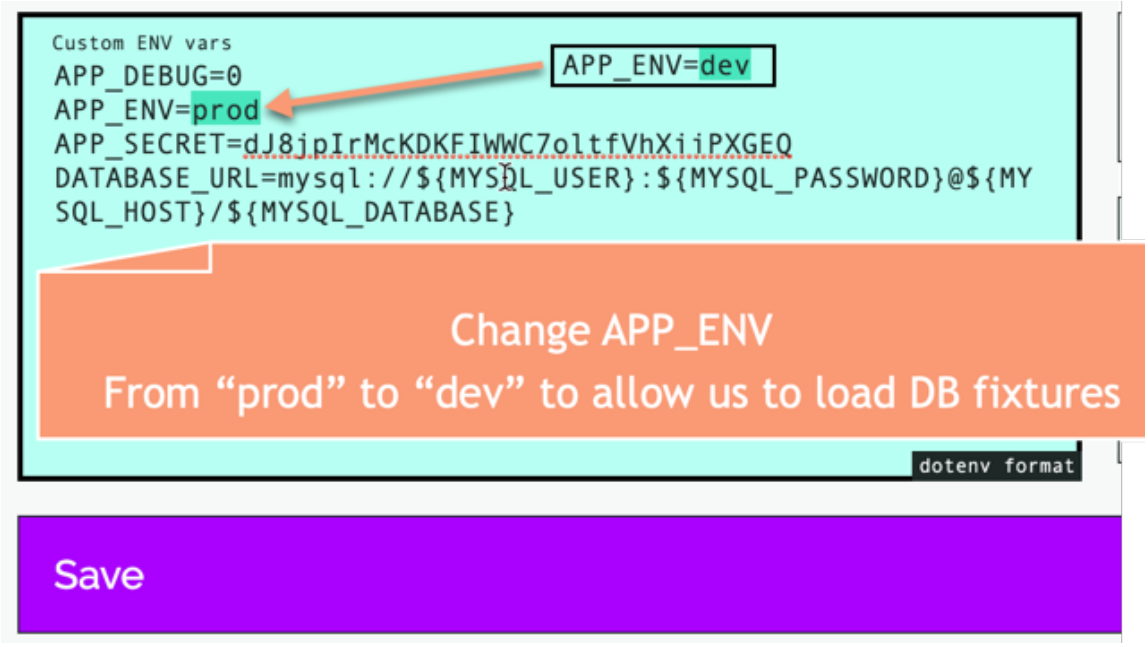

Figure 7.6: Changing environment from `prod` to `dev`.

7.5 Getting a linked Git project on your local computer

1. clone the repo to your local machine. See Figure 7.7.
   - NOTE: This will be an **empty** repository folder, apart from the hidden `.git` folder - you'll get a message warning you about this when you clone it to your computer
   - you'll copy your project files **into** this empty folder, to push back up to the Fortrabbit repo
1. copy your project files into the newly created cloned project folder
2. add all the new files to the current snapshot:

   ```
   git add .
   ```

3. Create the commit snapshot with a short message (the message doesn't matter):

   ```
   git commit -m "added files to project"
   ```

4. Push all these new files and folders up to the Fortrabbit repository:

   ```
   git push
   ```

NOTE: You'll see some Fortrabbit output when it responds to the new commit to the repositories `master` branch - CD (Contiuous Deployment) in action:

```
$ git push
    Enumerating objects: 92, done.
```

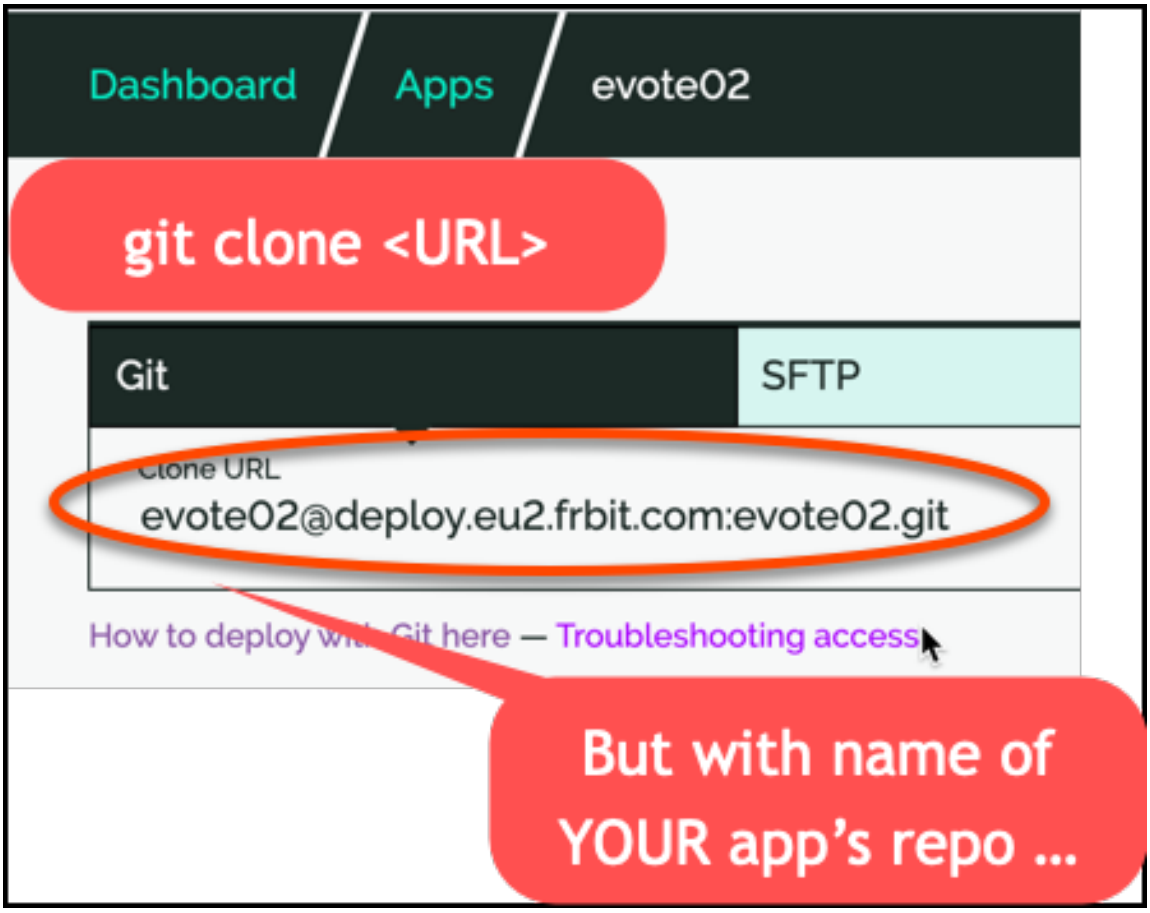

Figure 7.7: Clone git repo.

```
Counting objects: 100% (92/92), done.
Delta compression using up to 12 threads
Compressing objects: 100% (81/81), done.
Writing objects: 100% (92/92), 49.53 KiB | 3.10 MiB/s, done.
Total 92 (delta 3), reused 0 (delta 0)

Commit received, starting build of branch master

-----------------------  f  -----------------------

B U I L D
Checksum:
  814af6dc566dad405a8d4f13bc990a5e6dda6be7

Deployment file:
  not found

Pre-script:
  not found
  0ms

Composer:
  - - -
  Loading composer repositories with package information
  Installing dependencies (including require-dev) from lock file
  Package operations: 109 installs, 0 updates, 0 removals
    - Installing ocramius/package-versions (1.5.1): Downloading (100%)
    - Installing symfony/flex (v1.6.2): Downloading (100%)

  Prefetching 107 packages
    - Downloading (100%)

    - Installing doctrine/lexer (1.2.0): Loading from cache
    ... lots of installing for the first push ...
    ..........................................

  Executing script cache:clear [OK]
  Executing script assets:install public [OK]
  - - -
  10s 839ms
```

```
Post-script:
  not found
  0ms

R E L E A S E

Packaging:
  3s 796ms

Revision:
  1585658858201186125.814af6dc566dad405a8d4f13bc990a5e6dda6be7

Size:
  6.9 MB

Uploading:
  217ms

Build & release done in 14s 863ms, now queued for final distribution.

-----------------------  f  -----------------------

To deploy.eu2.frbit.com:symfony-demo.git
 * [new branch]      master -> master
```

7.6 Visit site to see if published (although may be DB errors)

Now visit your website - via link at Fortrabbit. See Figure 7.8.

- NOTE: Since the database isn't setup yet, you'll get an error if your homepage tries to list any DB data

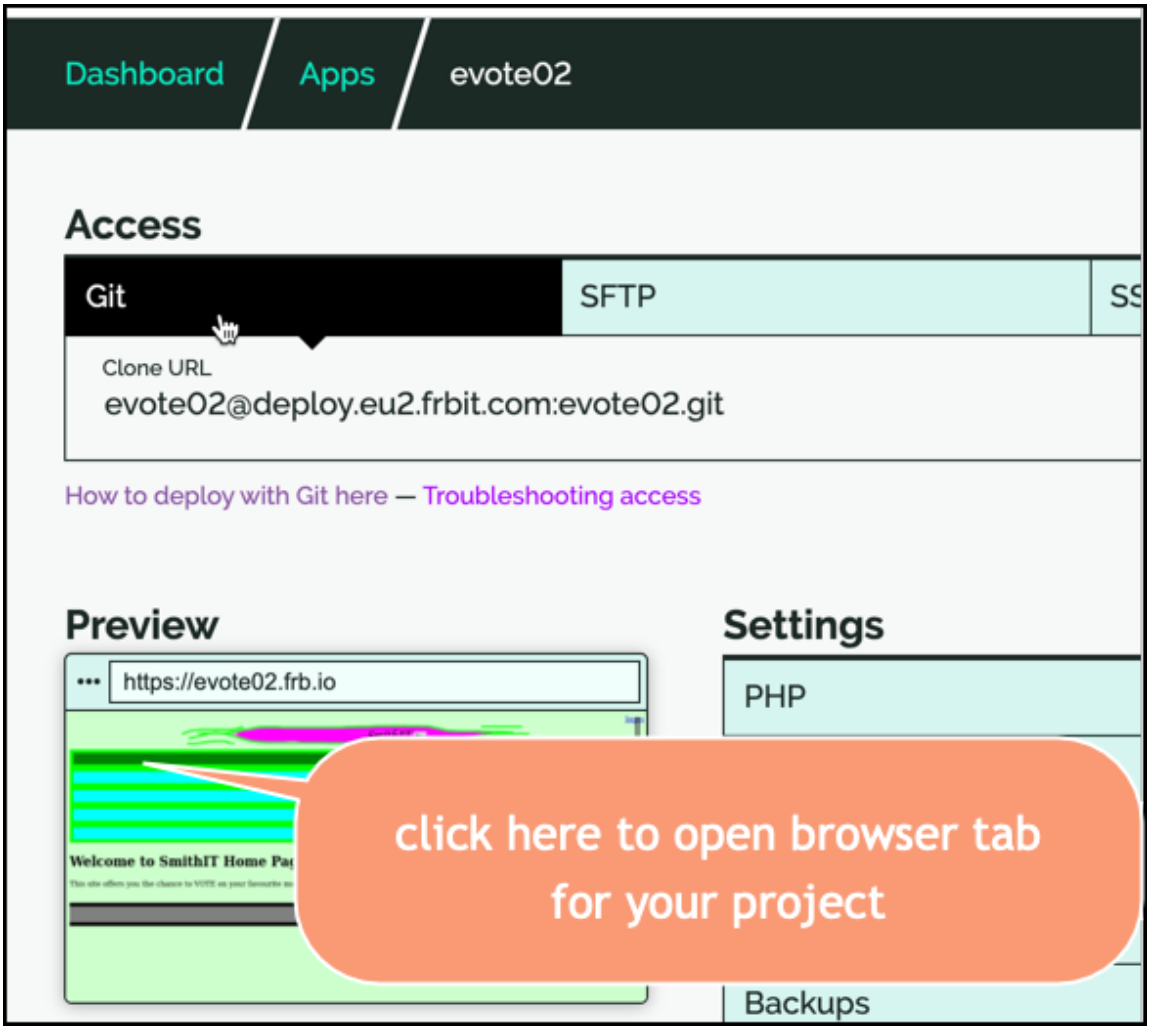

Figure 7.8: Visit published website link.

7.7 Connect command-line terminal to Fortrabbit project via SHH

Via an SSH terminal we can run the migration and load fixtures for our Fortrabbit app. Note, the database has already been created for us, so don't try to run a `database:create` Doctrine command …

Use the provided SSH connection command to connect to Fortrabbvit projet in a terminal. See Figure 7.9.

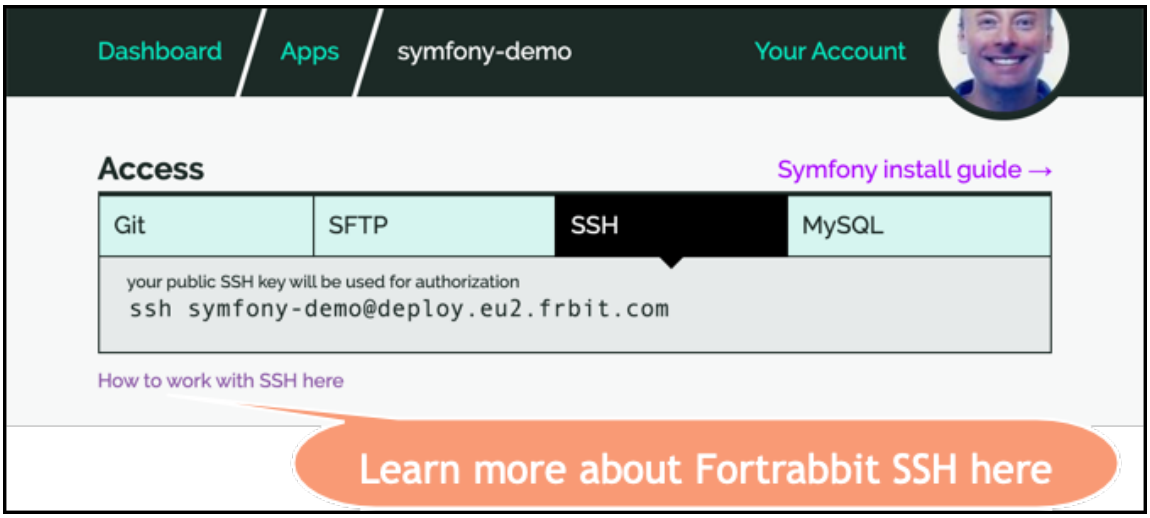

Figure 7.9: SSH connect command.

7.8 Use SSH to run DB migrations

We can now run the migration command `doctrine:migrations:migrate`. See Figure 7.10.

```
matt — -bash — 100×22
Last login: Tue Mar 31 13:52:34 on ttys000
matts-MacBook-Pro-2:~ matt$ ssh symfony-demo@deploy.eu2.frbit.com

symfony-demo:~$ php bin/console doctrine:migrations:migrate

                    Application Migrations

WARNING! You are about to execute a database migration that could result in schema changes and data loss. Are
you sure you wish to continue? (y/n)y
Migrating up to 20200331125138 from 0

  ++ migrating 20200331125138

     -> CREATE TABLE user (id INT AUTO_INCREMENT NOT NULL, email VARCHAR(180) NOT NULL, roles JSON NOT NULL, p
assword VARCHAR(255) NOT NULL, UNIQUE INDEX UNIQ_8D93D649E7927C74 (email), PRIMARY KEY(id)) DEFAULT CHARACTER
SET utf8mb4 COLLATE `utf8mb4_unicode_ci` ENGINE = InnoDB

  ++ migrated (took 39ms, used 12M memory)

  ------------------------

  ++ finished in 41.5ms
  ++ used 12M memory
  ++ 1 migrations executed
  ++ 1 sql queries
```

Figure 7.10: Running migration in SSH terminal.

7.9 Use SSH to load fixtures

We can now run the migration command `doctrine:fixtures:load`. See Figure 7.11.

```
matt — -bash — 100×22
Last login: Tue Mar 31 13:55:03 on ttys000
matts-MacBook-Pro-2:~ matt$ ssh symfony-demo@deploy.eu2.frbit.com

symfony-demo:~$ php bin/console doctrine:fixtures:load

 Careful, database "symfony-demo" will be purged. Do you want to continue? (yes/no) [no]:
 > y

   > purging database
   > loading App\DataFixtures\UserFixtures
```

Figure 7.11: Loading fixtures in SSH terminal.

NOTE: - if you change fixtures, you'll need to repeat this after pushing the updated code to the Fortrabbit repo

- don't forget to change the project environment back to `prod` if you want a secure, efficient running web application

7.10 Use SSH to clear the Symfony cache

It's a good idea to CLEAR the CACHE after making changes to your project.

Connected through SSH in the terminal and run the cache clear command:

```
php bin/console cache:clear
```

7.11 Use Doctrine query to check DB contents

We can use the Doctrine `doctrine:query:sql "<SQL>"` command in an SSH terminal to check the contents of the database. See Figure 7.12.

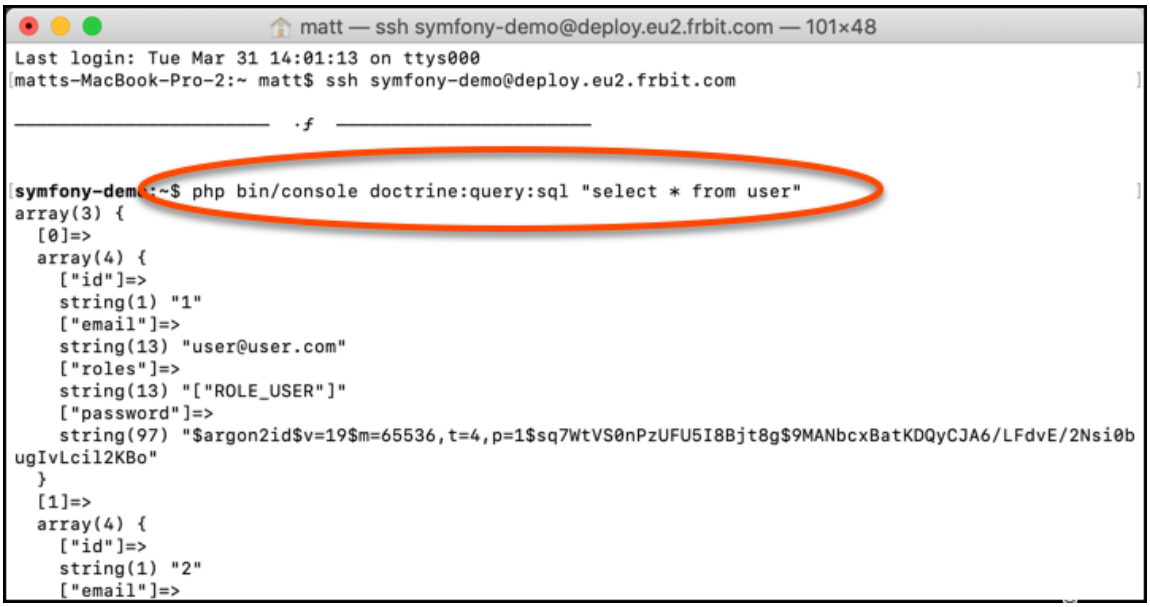

Figure 7.12: SSH terminal running SQL query via `doctrine:query:sql`.

7.12 MySQL queries using SSH tunnel ...

You can use an SSH tunnel to use your local MySQL terminal to connect to and query the remote Fortrabbit database

- follow the MySQL help steps from the Fortrabbit App dashboard

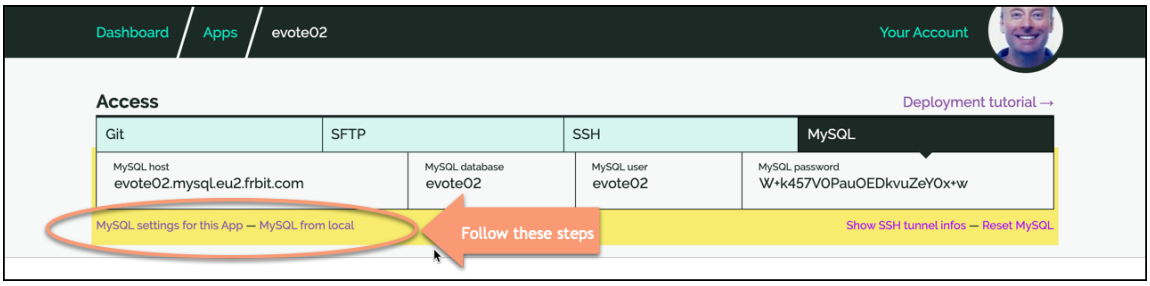

Figure 7.13: Fortrabbt MySQL help.

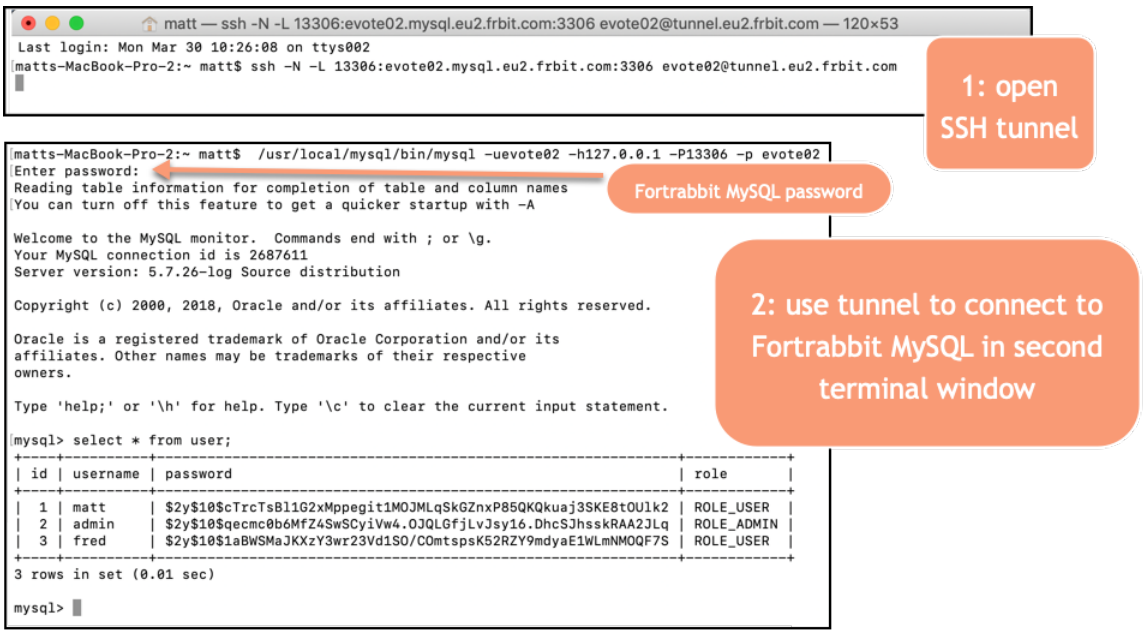

Figure 7.14: Remote SSH MySQL terminal connection.

7.13 Published website

If all has gone well, you should now have a live published Symfony website.

NOTE: If you can see the Symfony profiler debug footer, then you've forgotton to change the envrionment back to `prod` !!!!

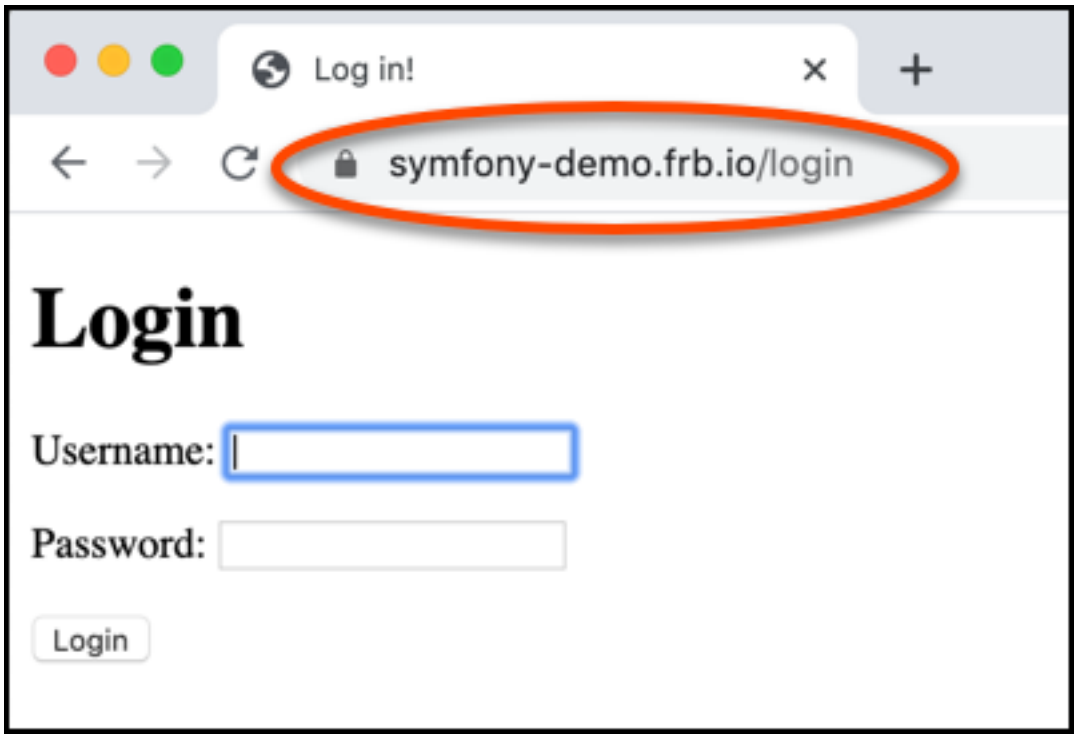

Figure 7.15: Screenshot of published website.

List of References