Testeo de aplicaciones

Symfony se integra con la librería independiente PHPUnit proporcionando un potente framework de testeo. En este curso no se explicará la librería PHPUnit.

Lo primero de todo es instalar el componente **PHPUnit Bridge** que añade funcionalidades adicionales a las propias de PHPUnit.

> *composer require --dev symfony/phpunit-bridge*

Cada test -no importa si es funcional o unitario- debe estar situado en el directorio */tests/* de la aplicación.

Si seguimos esta sencilla regla, todos nuestros tests se ejecutarán con el siguiente comando:

> *./vendor/bin/simple-phpunit*

La configuración de PHPUnit se encuentra en el archivo phpunit.xml.dist de nuestro proyecto.

Por ejemplo, se puede configurar una base de datos diferente para testeo:

```
<?xml version="1.0" charset="utf-8" ?>
<phpunit>
```

```
    <php>
        <!-- the value is the Doctrine connection string i
n DSN format -->
        <env name="DATABASE_URL" value="mysql://USERNAME:P
ASSWORD@127.0.0.1/DB_NAME" />
    </php>
    <!-- ... -->
</phpunit>
```

Enlace a la documentación de PHPUnit:

https://phpunit.readthedocs.io/es/latest/

Tests Unitarios

Los tests unitarios no tienen nada de especial en symfony.

Veámoslo con un ejemplo que testea el método *add()* de una supuesta clase *App|Util|Calculator*:

```
// tests/Util/CalculatorTest.php
namespace App\Tests\Util;

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

class CalculatorTest extends TestCase
{
    public function testAdd()
    {
        $calculator = new Calculator();
        $result = $calculator->add(30, 12);

        // assert that your calculator added the numbers c
orrectly!
        $this->assertEquals(42, $result);
    }
}
```

Nuestra clase de test es una clase situada dentro del directorio **tests** y que extiende de la clase **TestCase**.

TestCase no es más que la clase de PHPUnit que contiene todos los métodos/assertions.

El comando para ejecutar los tests que conocimos en el vídeo anterior tiene algunas variantes:

```
# Ejecuta todos los tests
./vendor/bin/simple-phpunit

# Ejecuta los tests del directorio Util
./vendor/bin/simple-phpunit tests/Util

# Ejecuta los tests de la clase Util/CalculatorTest.php
./vendor/bin/simple-phpunit tests/Util/CalculatorTest.php
```

Tests Funcionales

Los tests funcionales no se diferencian demasiado de los tests unitarios en cuanto a PHPUnit se refiere. La diferencia es que se necesita un flujo específico de trabajo:

- Realizar una petición
- Testear la respuesta
- Hacer click en un link, o en un botón, o enviar un formulario…
- Testear la respuesta
- …

Para realizar tests funcionales necesitamos algunas utilidades extra:

> *composer require --dev symfony/browser-kit symfony/css-selector*

Un ejemplo de test funcional podría ser el siguiente:

```
// tests/Controller/PostControllerTest.php
namespace App\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class PostControllerTest extends WebTestCase
{
    public function testShowPost()
    {
```

```
        $client = static::createClient();

        $client->request('GET', '/post/hello-world');

        $this->assertEquals(200, $client->getResponse()->g
etStatusCode());
    }
}
```

En este caso, las clases de test extienden de **WebTestCase**.

El método **createClient()** devuelve un cliente http que se comporta como un navegador y permite moverse por la aplicación:

```
$crawler = $client->request('GET', '/post/hello-world');
```

El método **request()** devuelve un objeto **Crawler**que se puede utilizar para seleccionar elementos de la respuesta, hacer click en enlaces, enviar formularios...

El crawler utiliza selectores css para seleccionar elementos de la página:

```
$this->assertGreaterThan(
    0,
    $crawler->filter('html:contains("Hello World")')->coun
t()
```

```
);
```

También puede interaccionar con la página haciendo click en un enlace:

```
$link = $crawler
    ->filter('a:contains("Ir a login")')
    ->eq(1) // select the second link in the list
    ->link()
;

// and click it
$crawler = $client->click($link);
```

O enviando un formulario:

```
$form = $crawler->selectButton('submit')->form();

$form['name'] = 'Lucas';
$form['form_name[subject]'] = 'Hola';

// submit the form
$crawler = $client->submit($form);
```

Aserciones más utilizadas en tests funcionales

```php
use Symfony\Component\HttpFoundation\Response;

// ...

// asserts that there is at least one h2 tag
// with the class "subtitle"
$this->assertGreaterThan(
    0,
    $crawler->filter('h2.subtitle')->count()
);

// asserts that there are exactly 4 h2 tags on the page
$this->assertCount(4, $crawler->filter('h2'));

// asserts that the "Content-Type" header is "application/
json"
$this->assertTrue(
    $client->getResponse()->headers->contains(
        'Content-Type',
        'application/json'
    ),
    'the "Content-Type" header is "application/json"' // o
ptional message shown on failure
);

// asserts that the response content contains a string
$this->assertContains('foo', $client->getResponse()->getCo
```

```
ntent());
// ...or matches a regex
$this->assertRegExp('/foo(bar)?/', $client->getResponse()-
>getContent());

// asserts that the response status code is 2xx
$this->assertTrue($client->getResponse()->isSuccessful(),
'response status is 2xx');
// asserts that the response status code is 404
$this->assertTrue($client->getResponse()->isNotFound());
// asserts a specific 200 status code
$this->assertEquals(
    200, // or Symfony\Component\HttpFoundation\Response::
HTTP_OK
    $client->getResponse()->getStatusCode()
);

// asserts that the response is a redirect to /demo/contac
t
$this->assertTrue(
    $client->getResponse()->isRedirect('/demo/contact')
    // if the redirection URL was generated as an absolute
 URL
    // $client->getResponse()->isRedirect('http://localhos
t/demo/contact')
);
// ...or simply check that the response is a redirect to a
ny URL
```

```
$this->assertTrue($client->getResponse()->isRedirect());
```

El Test Client

El objeto **client** simula un cliente HTTP similar a un navegador y permite hacer peticiones y acceder a la respuesta:

```
$crawler = $client->request('GET', '/post/hello-world');
```

El método **request()** devuelve una instancia de un objeto **crawler**. La lista de argumentos que adminte el objeto request es la siguiente:

```
request(
    $method,
    $uri,
    array $parameters = array(),
    array $files = array(),
    array $server = array(),
    $content = null,
    $changeHistory = true
)
```

El argumento de tipo array $server es el array de valores al que accedemos con la variable superglobal de PHP $_SERVER.

```
$client->request(
    'GET',
    '/post/hello-world',
```

```
    array(),
    array(),
    array(
        'CONTENT_TYPE'           => 'application/json',
        'HTTP_REFERER'           => '/foo/bar',
        'HTTP_X-Requested-With' => 'XMLHttpRequest',
    )
);
```

El crawler se utiliza para encontrar elementos del DOM en la respuesta. Estos elementos se pueden utilizar para hacer click en ellos, para enviar formularios, etc.

```
$link = $crawler->selectLink('Go elsewhere...')->link();
$crawler = $client->click($link);

$form = $crawler->selectButton('validate')->form();
$crawler = $client->submit($form, array('name' => 'Fabien'
));
```

Los métodos **click()** y **submit()** devuelven a su vez un objeto crawler.

```
// hacer submit de un formulario (es más fácil con el obje
to crawler)
$client->request('POST', '/submit', array('name' => 'Fabie
n'));
```

```php
// submits a raw JSON string in the request body
$client->request(
    'POST',
    '/submit',
    array(),
    array(),
    array('CONTENT_TYPE' => 'application/json'),
    '{"name":"Fabien"}'
);

// Form submission with a file upload
use Symfony\Component\HttpFoundation\File\UploadedFile;

$photo = new UploadedFile(
    '/path/to/photo.jpg',
    'photo.jpg',
    'image/jpeg',
    123
);
$client->request(
    'POST',
    '/submit',
    array('name' => 'Fabien'),
    array('photo' => $photo)
);

// Perform a DELETE request and pass HTTP headers
$client->request(
```

```
    'DELETE',
    '/post/12',
    array(),
    array(),
    array('PHP_AUTH_USER' => 'username', 'PHP_AUTH_PW' =>
'pa$$word')
);
```

También tiene métodos de navegación

```
$client->back();
$client->forward();
$client->reload();

// clears all cookies and the history
$client->restart();
```

Acceso a objetos internos

```
$history = $client->getHistory();
$cookieJar = $client->getCookieJar();

// the HttpKernel request instance
$request = $client->getRequest();

// the BrowserKit request instance
$request = $client->getInternalRequest();
```

```
// the HttpKernel response instance
$response = $client->getResponse();

// the BrowserKit response instance
$response = $client->getInternalResponse();

$crawler = $client->getCrawler();
```

Y aunque no es recomendable en tests funcionales, a veces necesitaremos acceder directamente a algún servicio

```
$container = $client->getContainer();
```

Accessing the Profiler Data

En cada petición, se puede habilitar el profiler de symfony para recoger datos. Por ejemplo, podría servir para testear que una determinada página ejecuta menos de X consultas a la base de datos.

Pra habilitar el profiler en una petición hay que llamar al método **enableProfiler()** antes de realizar dicha petición.

```
$client->enableProfiler();

$crawler = $client->request('GET', '/producto');
```

```
$profile = $client->getProfile();
```

Redirecciones

Cuando una petición devuelve una respuesta de tipo redirección, el cliente de test no sigue la redirección automáticamente. Se puede examinar la respuesta y después forzar al cliente a que siga la redirección con el método **followRedirect()**.

```
$crawler = $client->followRedirect();
```

También existe el método **followRedirects()** (en plural) para forzar al cliente a que siga automáticamente todas las redirecciones. Se utilizaría antes de realizar la petición.

```
$client->followRedirects();
```

Si se le pasa *false* a este método, el cliente dejaría de seguir las redirecciones.

```
$client->followRedirects(false);
```

El objeto Crawler

Cada vez que realizamos una petición con el objeto client, nos devuelven una instancia de Crawler.

El crawler nos permite analizar y acceder a los elementos del DOM de la respuesta.

Acceso a elementos del DOM

De forma similar a jQuery, el crawler tiene métodos para acceder a elementos del DOM.

En el siguiente ejemplo, el crawler accede a todos los elementos input[type=submit], selecciona el último de ellos, y selecciona posteriormente al elemento padre de dicho elemento.

```
$newCrawler = $crawler->filter('input[type=submit]')
    ->last()
    ->parents()
    ->first()
;
```

Hay disponibles muchos métodos para acceder a los elementos del DOM:

- filter(‘h1.title’)

  Devuelve elementos en base a un selector CSS.

- filterXpath(‘h1’)

  Devuelve elementos en base a una expresión XPath.

- eq(1)

  Devuelve el elemento que ocupa la posición dada.

- first()

  Devuelve el primer elemento.

- last()

  Devuelve el último elemento.

- siblings()

  Todos los elementos “hermanos”.

- nextAll()

  Todos los hermanos siguientes.

- previousAll()

  Todos los hermanos anteriores.

- parents()

  Padres.

- children()

  Hijos.

- reduce(callable)

  Elementos para los cual el *callable* no devuelve *false*.

Todos los métodos anteriores devuelven una instancia de crawler, por lo que se pueden encadenar.

```
$crawler
    ->filter('h1')
    ->reduce(function ($node, $i) {
        if (!$node->getAttribute('class')) {
            return false;
        }
    })
    ->first()
;
```

Existe otro método **count()** que no devuelve un crawler sino que devuelve el número de elementos del crawler.

Extracción de información

```
// devuelve el valor del atributo dado del primer elemento
$crawler->attr('class');
```

```
// devuelve el valor del primer elemento
$crawler->text();

// extrae un array de atributos para todos los nodos
// devuelve un array por cada elemento del crawler con el class y el href
$info = $crawler->extract(array('class', 'href'));

// ejecuta un callable para cada nodo y devuelve un array con los resultados
$data = $crawler->each(function ($node, $i) {
    return $node->attr('href');
});
```

Enlaces

Para seleccionar un enlace, podemos utilizar las funciones de acceso anteriores o un método especial **selectLink()**.

```
$crawler->selectLink('Click here');
```

Este método selecciona todos los enlaces que contengan el texto dato o imágenes clickables cuyo atributo *alt* contenga dicho texto. Este método devuelve otro objeto crawler.

Esos elementos tienen un objeto especial, que se puede obtener con el método **link()**.

```
$link = $crawler->selectLink('Click here')->link();
```

Este objeto especial tiene métodos **getMethod()** y **getUri()**.

```
$method = $link->getMethod();
$url = $link->getUri();
```

Para hacer click en el enlace, se utiliza el método **click()** del client pasándole el objeto sobre el que se quiere hacer click.

```
$link = $crawler->selectLink('Click here')->link();

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

Forms

La forma de seleccionar un formulario con el crawler es un tanto curiosa.

Los formularios se pueden seleccionar utilizando sus botones, que a su vez se pueden seleccionar con el método **selectButton()** de forma similar a los enlaces:

```
$buttonCrawlerNode = $crawler->selectButton('submit');
```

Está hecho así porque un formulario puede tener varios botones, pero un botón solamente puede pertenecer a un formulario.

El método selectButton() busca por los siguientes conceptos:

- El valor del atributo value;
- El valor del atribuot id o del atributo alt para imágenes.
- El valor del atributo id o del atributo name para etiquetas button.

Una vez tenemos un crawler que representa a un botón, podemos llamar al método **form()** para obtener una instancia del objeto Form asociado:

```
$form = $buttonCrawlerNode->form();
```

Al llamar al método form(), podemos pasar un array de valores para los campos del formulario, que sobrescribirán a los valores que hubiera por defecto:

```
$form = $buttonCrawlerNode->form(array(
    'name'               => 'Carlos',
    'my_form[subject]'   => 'Este curso mola',
));
```

El segundo argumento del método form es para pasar un método HTTP específico:

```
$form = $buttonCrawlerNode->form(array(), 'DELETE');
```

El objeto client puede hacer submit de instancias de objetos Form a través de su método **submit()**.

```
$client->submit($form);
```

Al método submit también pueden pasársele los valores de los campos como segundo argumento:

```
$client->submit($form, array(
    'name'               => 'Carlos',
    'my_form[subject]'   => 'Este curso mola',
));
```

Y siempre podemos utilizar el objeto Form como si fuera un array para establecer los valores de los campos que queramos:

For more complex situations, use the Form instance as an array to set the value of each field individually:

```
$form['name'] = 'Carlos';
$form['my_form[subject]'] = 'Este curso mola';
```

Según cada campo, tenemos también algunos otros métodos:

```
// selects an option or a radio
$form['country']->select('France');

// ticks a checkbox
$form['like_symfony']->tick();

// uploads a file
$form['photo']->upload('/path/to/lucas.jpg');
```

Por último, el objeto Form tiene un método **getValues()** y un método getFiles() que devuelven los valores de los campos y de los archivos. Los dos métodos tienen una versión que devuelve los valores en formato php: **getPhpValues()** y **getPHPFiles()**.

Añadir o quitar elementos a un Form

Añadir campos:

```
// obtiene el form
$form = $crawler->filter('button')->form();

// obtiene the valores
$values = $form->getPhpValues();

// añade campos al array $values
```

```
$values['task']['tags'][0]['name'] = 'foo';
$values['task']['tags'][1]['name'] = 'bar';

// Hace submit del formulario
$crawler = $client->request($form->getMethod(), $form->get
Uri(), $values,  $form->getPhpFiles());
```

Eliminar campos:

```
$form = $crawler->filter('button')->form();
$values = $form->getPhpValues();

unset($values['task']['tags'][0]);

$crawler = $client->request($form->getMethod(), $form->get
Uri(), $values, $form->getPhpFiles());
```

Fixtures

> *composer require --dev doctrine/doctrine-fixtures-bundle*

Los fixtures son clases PHP en donde creamos objetos y los persistimos en la base de datos. Estas clases deben extender de **Doctrine\Bundle\FixturesBundle\Fixture** y deben implementar el método **load()**.

El siguiente ejemplo muestra un fixtures que crea 20 productos y los persiste:

```
// src/DataFixtures/AppFixtures.php
namespace App\DataFixtures;

use App\Entity\Producto;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;

class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        // Creamos y persistimos 20 productos
        for ($i = 0; $i < 20; $i++) {
            $product = new Producto();
            $product->setNombre('product '.$i);
```

```
            $product->setPrecio(mt_rand(10, 100));
            $manager->persist($product);
        }

        $manager->flush();
    }
}
```

El comando para cargar la base de datos con fixtures es:

> *bin/console doctrine:fixtures:load*

Este comando, primero purga la base de datos. Si queremos añadir los fixtures sin borrar previamente la base de datos, podemos añadir la opción *–append* al comando.

> *bin/console doctrine:fixtures:load --append*

Dividir los fixtures en ficheros separados

En ocasiones, la clase de fixtures crece demasiado y decidimos separarla en 2 o más clases. Symfony soluciona los dos asuntos más comunes que se dan en estos casos: compartir objetos entre las clases fixture y cargar los fixtures en orden.

Compartir objetos entre Fixtures

Symfony nos proporciona dos métodos **addReference()** y **getReference()** para compartir objetos entre fixtures.

```
// src/DataFixtures/UserFixtures.php
// ...
class UserFixtures extends Fixture
{
    public const ADMIN_USER_REFERENCE = 'admin-user';

    public function load(ObjectManager $manager)
    {
        $userAdmin = new User('admin', 'pass_1234');
        $manager->persist($userAdmin);
        $manager->flush();

        // other fixtures can get this object using the Us
erFixtures::ADMIN_USER_REFERENCE constant
        $this->addReference(self::ADMIN_USER_REFERENCE, $u
serAdmin);
    }
}
```

```
// src/DataFixtures/GroupFixtures.php
// ...
class GroupFixtures extends Fixture
```

```
{
    public function load(ObjectManager $manager)
    {
        $userGroup = new Group('administrators');
        // this reference returns the User object created
in UserFixtures
        $userGroup->addUser($this->getReference(UserFixtur
es::ADMIN_USER_REFERENCE));

        $manager->persist($userGroup);
        $manager->flush();
    }
}
```

Lo único que queda por solucionar es que estas clases deben ejecutarse en un orden determinado.

Cargar los ficheros de fixtures en orden

Una clase fixture que dependa de otra, debe implementar el interfaz **DependentFixtureInterface** y añadir un método **getDependencies()** en el que se indican las clases de las que depende.

```
// src/DataFixtures/UserFixtures.php
namespace App\DataFixtures;
```

```
// ...
class UserFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        // ...
    }
}
```

```
// src/DataFixtures/GroupFixtures.php
namespace App\DataFixtures;
// ...
use App\DataFixtures\UserFixtures;
use Doctrine\Common\DataFixtures\DependentFixtureInterface
;

class GroupFixtures extends Fixture implements DependentFi
xtureInterface
{
    public function load(ObjectManager $manager)
    {
        // ...
    }

    public function getDependencies()
    {
        return array(
```

```
            UserFixtures::class,
        );
    }
}
```

Symfony ejecutará las clases en un orden que respete las dependencias.

Testeo de comandos de consola

Symfony proporciona una clase **CommandTester** para facilitar el testeo de comandos. Esta clase permite lanzar comandos desde un test pasando argumentos y opciones y devolviéndonos la salida que se generaría.

```
// tests/Command/CreateUserCommandTest.php
namespace App\Tests\Command;

use App\Command\CreateUserCommand;
use Symfony\Bundle\FrameworkBundle\Console\Application;
use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
use Symfony\Component\Console\Tester\CommandTester;

class CreateUserCommandTest extends KernelTestCase
{
    public function testExecute()
    {
        $kernel = self::bootKernel();
        $application = new Application($kernel);

        $application->add(new CreateUserCommand());

        $command = $application->find('app:create-user');
```

```
        $commandTester = new CommandTester($command);
        $commandTester->execute(array(
            'command'  => $command->getName(),

            // pass arguments to the helper
            'username' => 'Wouter',

            // prefix the key with two dashes when passing
 options,
            // e.g: '--some-option' => 'option_value',
        ));

        // the output of the command in the console
        $output = $commandTester->getDisplay();
        $this->assertContains('Username: Wouter', $output)
;

        // ...
    }
}
```

Testeo de Repositorios de Doctrine

Para testear un repositorio de doctrine lo más sencillo es que nuestra clase de testeo extienda de **KernelTestCase**.

De esta forma, podemos acceder al objeto $kernel de forma muy sencilla con *self::bootKernel()* y a través del kernel obtener el *entityManager*, del cual podemos obtener el repositorio que queramos.

Veamos un ejemplo:

```
// tests/Repository/ProductRepositoryTest.php
namespace App\Tests\Repository;

use App\Entity\Product;
use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;

class ProductRepositoryTest extends KernelTestCase
{
    /**
     * @var \Doctrine\ORM\EntityManager
     */
    private $entityManager;

    /**
```

```
     * {@inheritDoc}
     */
    protected function setUp()
    {
        $kernel = self::bootKernel();

        $this->entityManager = $kernel->getContainer()
            ->get('doctrine')
            ->getManager();
    }

    public function testSearchByCategoryName()
    {
        $products = $this->entityManager
            ->getRepository(Product::class)
            ->searchByCategoryName('foo')
        ;

        $this->assertCount(1, $products);
    }

    /**
     * {@inheritDoc}
     */
    protected function tearDown()
    {
        parent::tearDown();
```

```
        $this->entityManager->close();
        $this->entityManager = null; // avoid memory leaks
    }
}
```