AN INTRODUCTION TO SYMFONY 4
(for people that already know OO-PHP and some MVC stuff)

by

Dr. Matt Smith

mattsmithdev.com

goryngge.com

https://github.com/dr-matt-smith

Acknowledgements

Thanks to ...

Table of Contents

Part I

Introduction to Symfony

1 Introduction

1.1 What is Symfony 4?

It's a PHP 'framework' that does loads for you, if you're writing a secure, database-drive web application.

1.2 What to I need on my computer to get started?

I recommend you install the following:

- PHP 7 (download/install from php.net)
- a MySQL database server - e.g. MySQLWorkbench Community is free and cross-platform
- a good text editor (I like PHPStorm, but then it's free for educational users…)
- Composer (PHP package manager - a PHP program)

or … you could use something like Cloud9, web-based IDE. You can get started on the free version and work from there …

Learn more about the software needed for Symfony developmnent in Appendix A. For steps in installing PHP and the other software, see Appendices B and C.

1.3 How to I get started with a new Symfony project

In a CLI (Command Line Interface) terminal window, `cd` into the directory where you want to create your Symfony project(s). Then create a new Symfony 4 empty project, named `project01` (or whatever you wish) by typing:

```
$ composer create-project symfony/skeleton project01
```

You should see the following, if all is going well:

```
Installing symfony/skeleton (v4.0.5)
  - Installing symfony/skeleton (v4.0.5): Loading from cache
Created project in my-project
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 21 installs, 0 updates, 0 removals
  - Installing symfony/flex (v1.0.66): Downloading (100%)
  - Installing symfony/polyfill-mbstring (v1.6.0): Loading from cache
  ...

  * Run your application:
    1. Change to the project directory
    2. Execute the php -S 127.0.0.1:8000 -t public command;
    3. Browse to the http://localhost:8000/ URL.

       Quit the server with CTRL-C.
       Run composer require server for a better web server.
```

NOTE: - If the first line does not show a Symfony version starting with `v4` then you may have an old version of PHP installed. You need PHP 7.1.3 minimum to run Symfony 4.

Another way to get going quickly with Symfomy is to download one of the projects accompanying this book …

1.4 Where are the projects accompanying this book?

All the projects in this book are freely available, as public repositories on Github as follows:

- https://github.com/dr-matt-smith/php-symfony4-book-codes

To retrieve and setup a sample project follow these steps:

1. download the project to your local computer (e.g. `git clone URL`)
2. change (`cd`) into the created directory
3. type `composer update` to download any required 3rd-party packages into a `/vendor` folder

4. Then run your web server (see below) and explore via a web browser

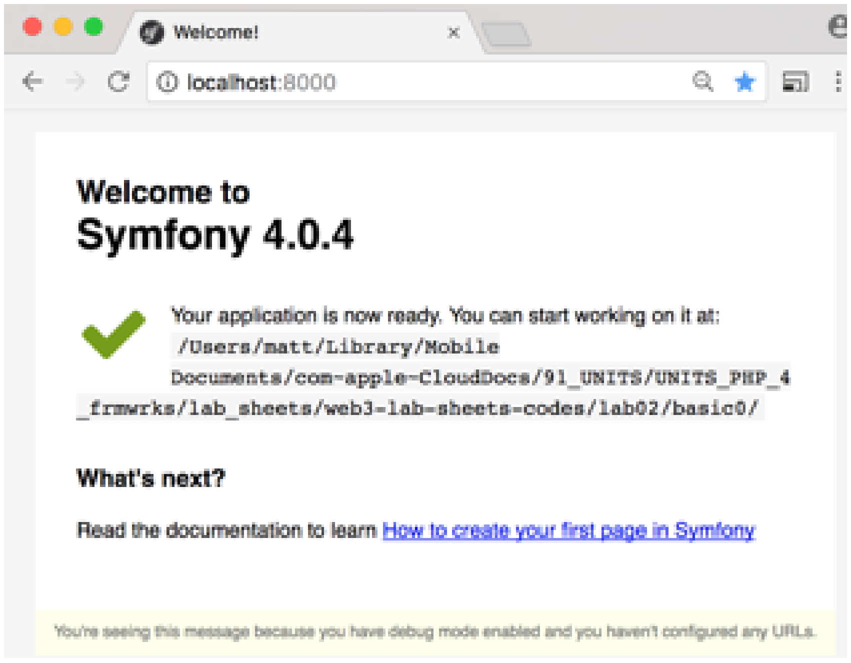
1.5 How to I run a Symfony webapp?

1.5.1 From the CLI with PHP built-in web server

At the CLI (command line terminal) ensure you are at the base level of your project (i.e. the same directory that has your `composer.json` file), and type the following to run the PHP built-in web server:

```
$ php -S localhost:8000 -t public
```

See Figure 1.1 for a screenshot of the default Symfony 4 home page.



Welcome to
Symfony 4.0.4

Your application is now ready. You can start working on it at:
/Users/matt/Library/Mobile Documents/com~apple~CloudDocs/91_UNITS/UNITS_PHP_4_frmwrks/lab_sheets/web3-lab-sheets-codes/lab02/basic0/

What's next?

Read the documentation to learn How to create your first page in Symfony

You're seeing this message because you have debug mode enabled and you haven't configured any URLs.

Figure 1.1: Screenshot default Symfony 4 home page.

1.5.2 From the CLI with Symfony's web server

However Symfony offers its own (better!) server, which is easily installed and run.

To install the Symfony server component just type the following at the CLI (having changed into the project directory):

```
$ composer req --dev server
```

Check this vanilla, empty project is all fine by running the web sever and visit website root at `http://localhost:8000/`:

To run the server:

```
$ php bin/console server:run
 [OK] Server listening on http://127.0.0.1:8000
 // Quit the server with CONTROL-C.
```

Then open a web browser and visit the website root at `http://localhost:8000`.

1.5.3 From a Webserver application (like Apache or XAMPP)

If you are running a webserver (or combined web and database server like XAMPP or Laragon), then point your web server root to the project's `/public` folder - this is where public files go in Symfony projects.

1.6 It isn't working! (Problem Solving)

If you have trouble with running Symfony, take a look at Appendix E, which lists some common issues and how to solve them.

1.7 Can I see a demo project with lots of Symfony features?

Yes! There is a full-featured Symfony demo project. Checkout Appendix D for details of downloading and running the demo and its associated automated tests.

1.8 Any free videos about SF4 to get me going?

Yes! Those nice people at KNP-labs/university have released a bunch of free videos all about Symfony 4.

So plug in your headphones and watch them, or read the transcripts below the video if you're no headphones. A good rule is to watch a video or two **before** trying it out yourself.

You'll find the video tutorials at:

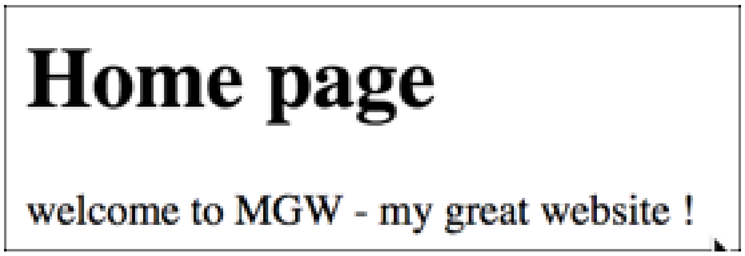
- https://knpuniversity.com/screencast/symfony

2

First steps

2.1 What we'll make (`basic00`)

See Figure 2.1 for a screenshot of the new homepage we'll create in our first project.



Home page

welcome to MGW - my great website !

Figure 2.1: New home page.

There are 3 things Symfony needs to serve up a page:

1. a route
2. a controller class and method
3. a Response object to be returned to the web client

The first 2 can be combined, through the use of 'Annotation' comments, which declare the route in a comment immediately before the controller method defining the 'action' for that route. See this example:

```
/**
 * @Route("/", name="homepage")
```

```
     */
    public function indexAction()
    {
        ... build and return Response object here ...
    }
```

For example the code below defines:

- an annotation Route comment for URL pattern `/` (i.e. website route)

  – `@Route("/", name="homepage")`

  – the Symfony "router" system attemptes to match pattern `/` in the URL of the HTTP Request received by the server

- controller method `indexAction()`

  – this method will be involved if the route matches

  – controller method have the responsibility to create and return a Symfony `Response` object

- note, Symfony allows us to declare an internal name for each route (in the example above `homepage`)

  – we can use the internal name when generating URLs for links in out templating system

  – the advantage is that the route is only defined once (in the annotation comment), so if the route changes, it only needs to be changed in one place, and all references to the internal route name will automatically use the updated route

  – for example, if this homepage route was changed from `/` to `/default` all URls generated using the `homepage` internal name would now generated `/default`

2.2 Create a new Symfony project

1. Create new Symfony 4 project (and then `cd` into it):

   ```
   $ composer create-project symfony/skeleton basic00
   Installing symfony/skeleton (v4.0.5)
     - Installing symfony/skeleton (v4.0.5): Loading from cache

   ... etc. ...

   $ cd basic1
   ```

2. Add the Symfony local development server:

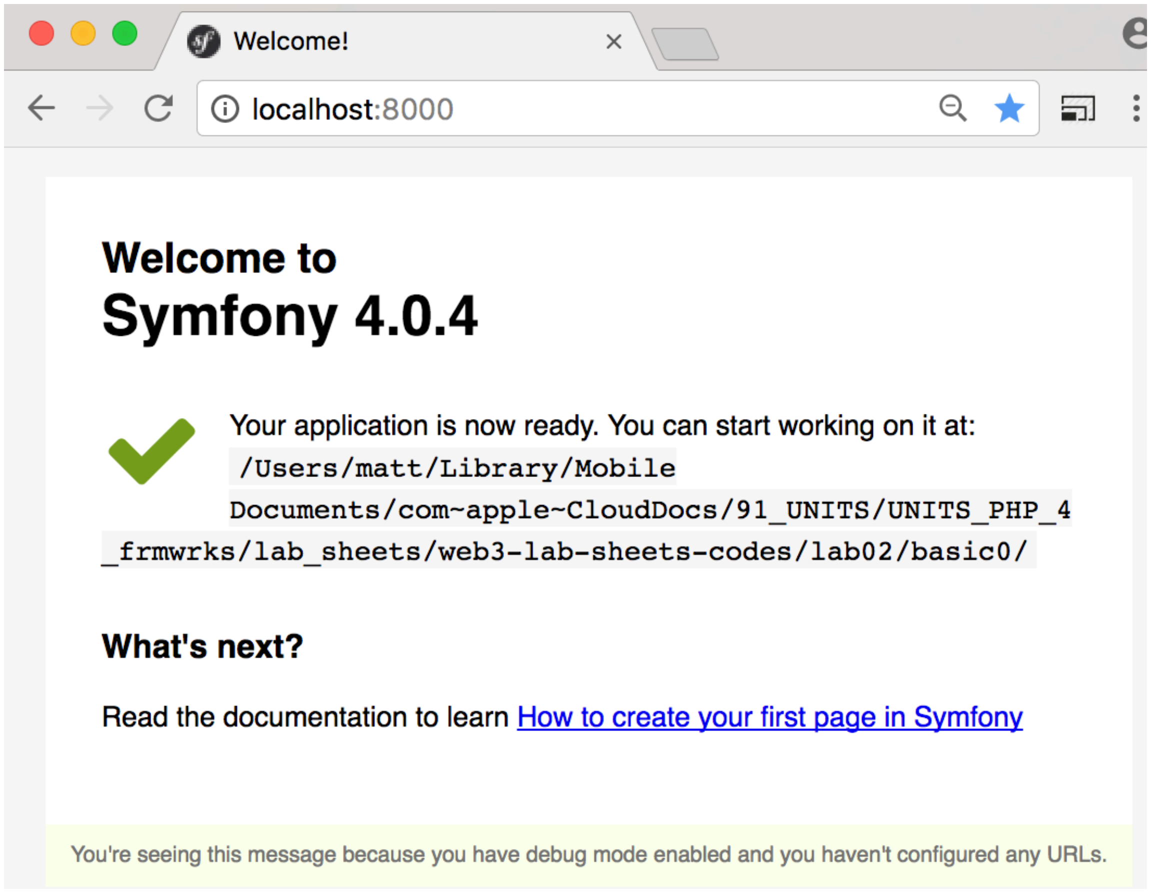
   ```
   composer req --dev server
   ```

NOTE: To **remove** a package use `composer rem <package>`, e.g. `composer rem server`.

Check this vanilla, empty project is all fine by running the web sever and visit website root at `http://localhost:8000/`:

```
$ php bin/console server:run
 [OK] Server listening on http://127.0.0.1:8000
 // Quit the server with CONTROL-C.
```

Figure 2.2 shows a screenshot of the default page for the web root (path `/`), when we have no routes set up and we are in development mode (i.e. our `.env` file contains `APP_ENV=dev`).



Welcome to
Symfony 4.0.4

Your application is now ready. You can start working on it at:
/Users/matt/Library/Mobile Documents/com~apple~CloudDocs/91_UNITS/UNITS_PHP_4_frmwrks/lab_sheets/web3-lab-sheets-codes/lab02/basic0/

What's next?

Read the documentation to learn How to create your first page in Symfony

You're seeing this message because you have debug mode enabled and you haven't configured any URLs.

Figure 2.2: Screenshot default Symfony 4 page for web root (when no routes defined).

2.3 List the routes

There should not be any routes yet - but let's check at the console:

```
    $ php bin/console debug:router
      Name    Method    Scheme    Host    Path
```

```
------ -------- -------- ------ ------
```

2.4 Add the annotations bundle

Since we'll be defining routes using annotation comments, we need to ask Composer to download the annotations bundle into our `/vendor` directory (and register the bundle, and update the autoloader etc.):

1. Add Annotations :

    ```
    $ composer req annotations
    Using version ^5.1 for sensio/framework-extra-bundle
    ./composer.json has been updated
    Loading composer repositories with package information
    ...
    Some files may have been created or updated to configure your new packages.
    Please review, edit and commit them: these files are yours.
    ```

2.5 Create a controller

We could write a new class for our homepage controller, but … let's ask Symfony to make it for us. Typical pages seen by non-logged-in users like the home page, about page, contact details etc. are often referred to as 'default' pages, and so we'll name the controller class for these pages our `DefaultController`.

1. First we need to add the `make` bundle to our console tool (for our development environment):

    ```
    $ composer req --dev make
    Using version ^1.0 for symfony/maker-bundle
    ./composer.json has been updated
    Loading composer repositories with package information
    ...
    ```

2. Now let's ask Symfony to create a new homepage (default) controller:

```
$ php bin/console make:controller Default
created: src/Controller/DefaultController.php

Success!
Next: Open your new controller class and add some pages!
```

NOTE: Symfony controller classes are stored in directory `/src/Controller`.

Look inside the generated class `/src/Controller/DefaultController.php`. It should look something like this:

```
namespace App\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    /**
     * @Route("/default", name="default")
     */
    public function index()
    {
        return new Response('Welcome to your new controller!');
    }
}
```

Learn more about the Maker bundle:

- https://symfony.com/blog/introducing-the-symfony-maker-bundle

2.6 Run web server to visit new default route

Run the web sever and visit the home page at `http://localhost:8000/`:

```
Sorry, the page you are looking for could not be found.
(2/2) NotFoundHttpException

No route found for "GET /"
```

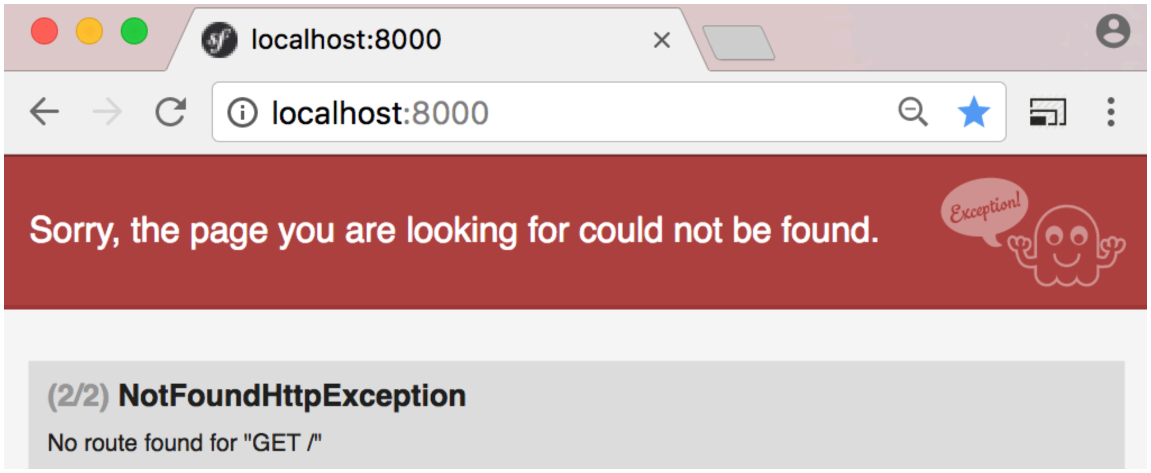
Oh no! We get a 404 (not found) error!. Figure 2.3 shows a screenshot of error page.

If we read it, it tells us `No route found for "GET /"`. Since we **have** defined a route, we don't get the default page any more. However, since we named our controller `Default`, then this is the route that was defined for it:

```
  Name                          Method   Scheme   Host   Path
 -------------------------- -------- -------- ------ -----------------------------------
  default                       ANY      ANY      ANY    /default
```

If we look more closely at the generated code, we can see this route `/default` in the **annotation** comment preceding controller method `index()` in `src/Controllers/DefaultController.php`

```
@Route("/default", name="default")
```



Sorry, the page you are looking for could not be found.

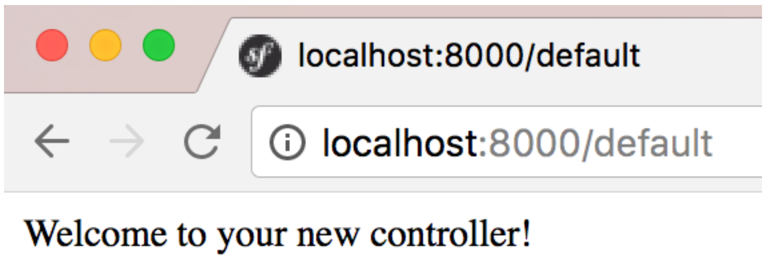
(2/2) NotFoundHttpException

No route found for "GET /"

Figure 2.3: Screenshot of 404 error for URL path /.

So visit `http://localhost:8000/default` instead, to see the page generated by our `DefaultController->index()` method.

Figure 2.4 shows a screenshot of the message created from our generated default controller method.



Welcome to your new controller!

Figure 2.4: Screenshot of generated page for URL path `/default`.

3 Twig templating

3.1 Add the debug bundle (`basic01`)

When developing we want all the error/warning/debugging information we can get. Let's add the Symfony profiler, which tells us lots about how thing are, and are not working with out site in development mode.

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

NOTE: The debug bundle makes use of (requires) the Twig templating bundle. This will impact:

- the look of error pages
- the code generated for new controllers

Try this (now we have Twig added):

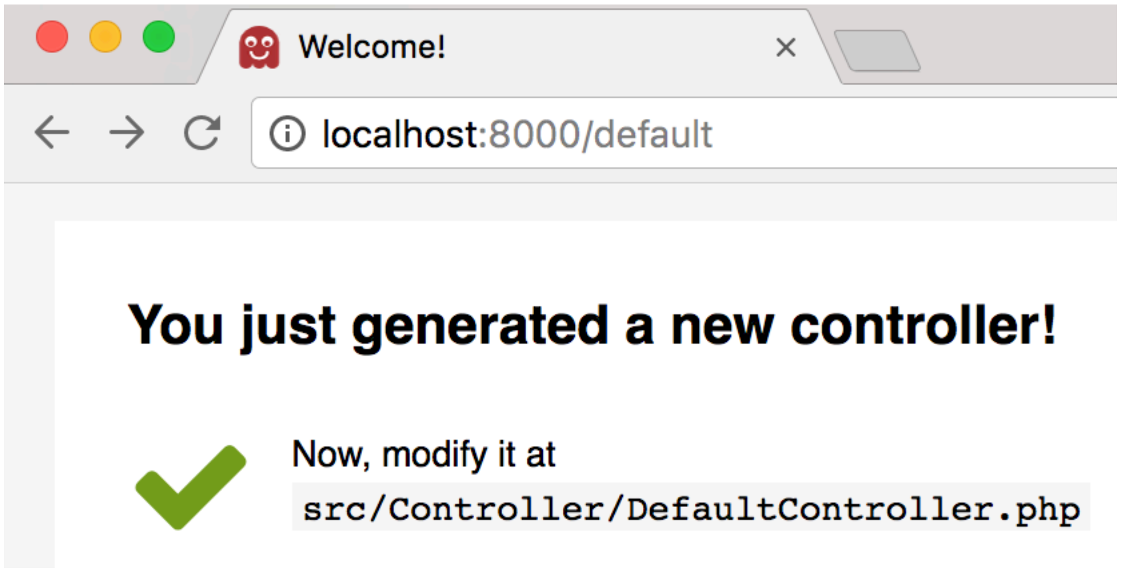
1. Delete the controller class file `/src/Controller/DefaultController.php`[1]
2. Generate a new Default controller class with `php bin/console make:controller Default`
3. Look at the generated code:

[1] That's one great thing about working with generated code - we can delete it and regenerate it with little or no work.

```
    /**
     * @Route("/default", name="default")
     */
    public function index()
    {
        // replace this line with your own code!
        return $this->render('@Maker/demoPage.html.twig', [
            'path' => str_replace($this->getParameter('kernel.project_dir').'/', '', __FILE
        ]);
    }
```

As you can see, the controller method now returns the output of method `$this->render(...)` rather than directly creating a `Response` object. With the Twig bundle added, each controller class now has access to the Twig `render(...)` method.

Figure 3.1 shows a screenshot of the message created from our generated default controller method with Twig.



You just generated a new controller!

Now, modify it at `src/Controller/DefaultController.php`

Figure 3.1: Screenshot of generated page for URL path `/default`.

3.2 View the routes added by the profiler

View the route list now - since our profile has added some (with the underscore _ prefix):

```
 Name                         Method    Scheme    Host   Path
 --------------------------- --------- --------- ------ -------------------------------------
 default                      ANY       ANY       ANY    /default
```

```
  _twig_error_test           ANY      ANY      ANY    /_error/{code}.{_format}
  _wdt                       ANY      ANY      ANY    /_wdt/{token}
  _profiler_home             ANY      ANY      ANY    /_profiler/
  _profiler_search           ANY      ANY      ANY    /_profiler/search
  _profiler_search_bar       ANY      ANY      ANY    /_profiler/search_bar
  _profiler_phpinfo          ANY      ANY      ANY    /_profiler/phpinfo
  _profiler_search_results   ANY      ANY      ANY    /_profiler/{token}/search/results
  _profiler_open_file        ANY      ANY      ANY    /_profiler/open
  _profiler                  ANY      ANY      ANY    /_profiler/{token}
  _profiler_router           ANY      ANY      ANY    /_profiler/{token}/router
  _profiler_exception        ANY      ANY      ANY    /_profiler/{token}/exception
  _profiler_exception_css    ANY      ANY      ANY    /_profiler/{token}/exception.css
```

3.3 Specific URL path and internal name for our default route method

Let's change the URL path to the website root (`/`) and name the route `homepage` by editing the annotation comments preceding method `index()` in `src/Controllers/DefaultController.php`.

While we're at it, let's also rename the method with the `Action` suffix, since that's a common convention…

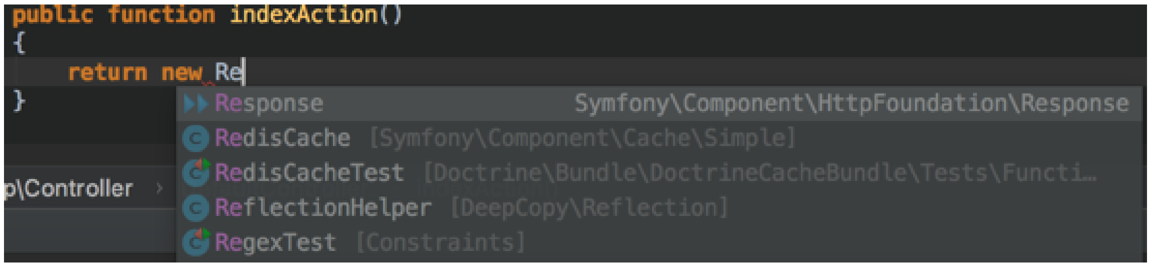
```
class DefaultController extends Controller
{
    /**
     * @Route("/", name="homepage")
     */
    public function indexAction()
```

Now the route is:

```
  Name                         Method   Scheme   Host   Path
 ---------------------------- -------- -------- ------ -----------------------------------
  homepage                     ANY      ANY      ANY    /
```

Finally, let's replace that default message with an HTTP response that **we** have created - how about the message `hello there!`. We can generate an HTTP response by creating an instance of the `Symfony\Component\HttpFoundation\Response` class.

Luckily, if we are using a PHP-friendly editor like PHPStorm, as we start to type the name of a class, the IDE will popup a suggestion of namespaced classes to choose from. Figure 3.2 shows a screenshot of PHPStorm offering up a list of suggested classes after we have typed the letters `new Re`. If we accept a suggested class from PHPStorm, then an appropriate `use` statement will be inserted before the class declaration for us.



```
public function indexAction()
{
    return new Re
}
```

Response Symfony\Component\HttpFoundation\Response
RedisCache [Symfony\Component\Cache\Simple]
RedisCacheTest [Doctrine\Bundle\DoctrineCacheBundle\Tests\Functi…
ReflectionHelper [DeepCopy\Reflection]
RegexTest [Constraints]
p\Controller

Figure 3.2: Screenshot of PHPStorm IDE suggesting namespaces classes.

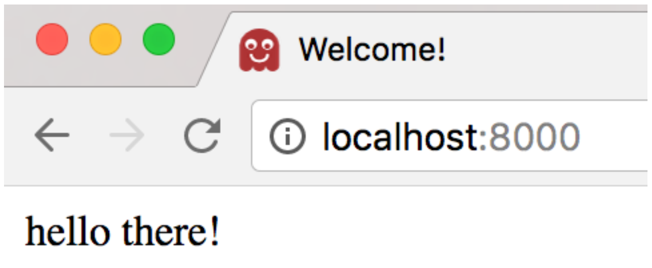
Here is a complete `DefaultController` class:

```
namespace App\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    /**
     * @Route("/", name="homepage")
     */
    public function indexAction()
    {
        return new Response('Hello there!');
    }
}
```

Figure 3.3 shows a screenshot of the message created from our `Response()` object.



hello there!

Figure 3.3: Screenshot of page seen for `new Response('hello there!')`.

3.4 Adding Twig directly

If we hadn't added the profiler, we could have added just the Twig bundle as follows:

- Add Twig :

  ```
  composer req twig
  ```

3.5 Secruty checker bundle

Another general bundle to always include is the Symfony security checkr:

1. Add the Symfony security checker - a good one to **always** have

   ```
   composer req sec-checker
   ```

3.6 Development Symfony 4 recipes

Libaries installed with `--dev` are only for use in our development setup - that software isn't used (or installed) for public deloployment of our `production` website that will actually run live on the internet.

Here is the list of the most common development libraries we'll need:

1. the Server recipe

   ```
   composer req --dev server
   ```

2. the Maker recipe (for development setup):

   ```
   composer req --dev make
   ```

3. Add the Symfony PHPUnit bridge(for development setup):

   ```
   composer req --dev phpunit
   ```

4. Add the Symfony web profiler (with great `dump()` functions!)

   ```
   composer req --dev profiler
   ```

5. Add the Symfony debugging libraries

   ```
   composer req --dev debug
   ```

3.7 Install multple libraries in a single `composer` commad

We can install all our non-development libraries with one command:

```
composer req twig annotations sec-checker
```

and all our development libraries with another command (with the `--dev` option):

```
composer req --dev server make phpunit debug profiler
```

3.8 Let's create a nice Twig hompage (`basic02`)

We are (soon) going to create Twig template in `templates/default/homepage.html.twig`. So we need to ask the `Twig` object in our Symfony project to create an HTTP response via its `render()` method. Part of the 'magic' of PHP Object-Orienteted inheritance (and the **Dependancy Injection** design pattern), is that since our controller class is a subclass of `Symfony\Bundle\FrameworkBundle\Controller\Controller`, then objects of our controller automatically have access to a `render(...)` method for an automatically generated Twig object.

In a nutshell, to output an HTTP response generated from Twig, we just have to specify the Twig template name, and relative location[2], and supply an array of any parameters we want to pass to the template.

So we can simply write the following to ask Symfony to generate an HTTP response from Twig's text output from rendering the template that can (will soon!) be found in `/tempaltes/default/homepage.html.twig`:

```
/**
 * @Route("/", name="homepage")
 */
public function indexAction()
{
    $template = 'default/homepage.html.twig';
    $args = [];
    return $this->render($template, $args);
}
```

Now let's create that Twig template in `/templates/default/homepage.html.twig`:

1. Create new directory `/templates/default`
2. Create new file `/templates/default/homepage.html.twig`:

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

{% block body %}
    <h1>home page</h1>

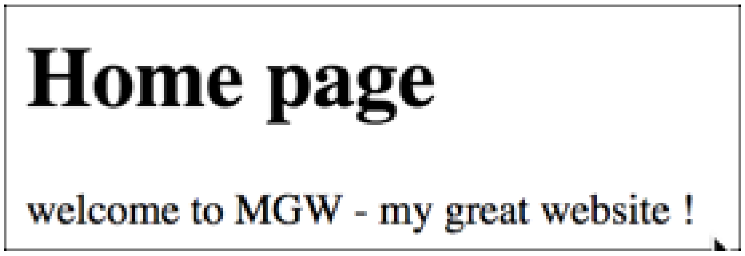
    <p>
```

[2]The 'root' of Twig template locations is, by default, `/templates`. To keep files well-organised, we should create subdirectories for related pages. For example, if there is a Twig template `/templates/admin/loginForm.html.twig`, then we would need to refer to its location (relative to `/templates`) as `admin/loginForm.html.twig`.

```
            welcome to the home page
        </p>
    {% endblock %}
```

Note that Twig paths searches from the Twig root location of `/templates`, not from the location of the file doing the inheriting, so do **NOT** write `{% extends 'default/base.html.twig' %}`...

Figure 3.4 shows a screenshot our Twig-generated page in the web browser.



Home page

welcome to MGW - my great website !

Figure 3.4: Screenshot of page from our Twig template.

4 Creating our own classes

4.1 Goals

Our goals are to:

- create a simple Student entity class
- create a route / controller / template to show one student on a web page
- create a repository class, to manage an array of Student objects
- create a route / controller / template to list all students as a web page
- create a route / controller / template to show one student on a web page for a given Id

4.2 Let's create an Entity `Student` (`basic03`)

Entity classes are declared as PHP classes in `/src/Entity`, in the namespace `App\Entity`. So let's create a simple `Student` class:

```
<?php
namespace App\Entity;

class Student
{
    private $id;
    private $firstName;
```

```
        private $surname;
    }
```

That's enough typing - use your IDE (E.g. PHPStorm) to generate a public constructor (taking in values for all 3 properties), and also public getters/setters for each property.

4.3 Create a StudentController class

Generate a StudentController class:

```
$ php bin/console make:controller Student
```

It should look like this (`/src/Controller/StudentController.php`):

```
<?php

namespace App\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class StudentController extends Controller
{
    /**
     * @Route("/student", name="student")
     */
    public function index()
    {
        ... default code here ...
    }
}
```

NOTE!!!!: When adding new routes, it's a good idea to **CLEAR THE CACHE**, otherwise Symfony may not recognised the new or changed routes ... Either manually delete the `/var/cache` directory, or run the `cache:clear` console command:

```
$ php bin/console cache:clear

// Clearing the cache for the dev environment with debug true
[OK] Cache for the "dev" environment (debug=true) was successfully cleared.
```

Let's make this create a student (`1`, `matt`, `smith`) and display it with a Twig template (which we'll write next!). We will also improve the route internal name, changing it to `student_show`, and change the method name to `showAction()`. So your class (with its `use` statements, especially for `App\Entity\Student`) looks as follows now:

```
namespace App\Controller;

use App\Entity\Student;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class StudentController extends Controller
{
    /**
     * @Route("/student", name="student_show")
     */
    public function showAction()
    {
        $student = new Student(1, 'matt', 'smith');

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

NOTE:: Ensure your code has the appropriate `use` statement for the `App\Entity\Student` class - a nice IDE like PHPStorm will add this for you...

4.4 The show student template `/templates/student/show.html.twig`

Create the directory `/templates/student`. In that directory create a new Twig template named `show.html.twig`. Write the following Twig code for the template:

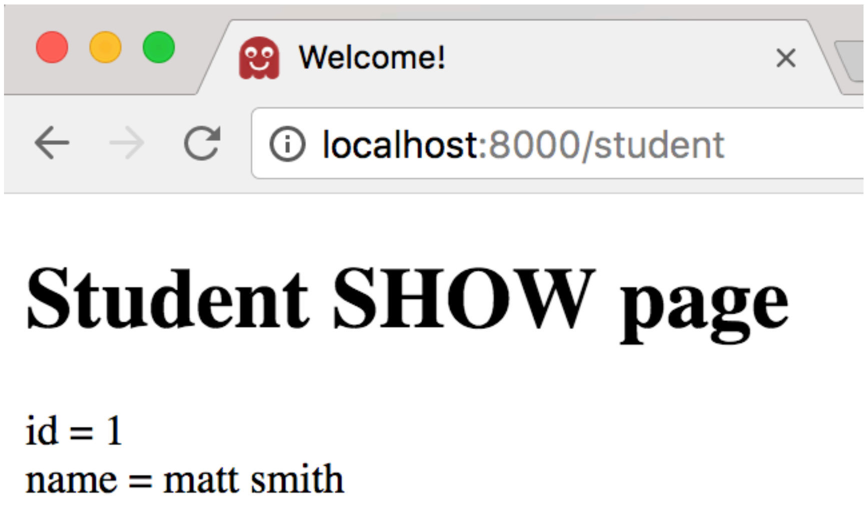
```
{% extends 'base.html.twig' %}

{% block body %}
    <h1>Student SHOW page</h1>

    <p>
        id = {{ student.id }}
        <br>
        name = {{ student.firstName }} {{ student.surname }}
    </p>
{% endblock %}
```

Run the web server and visit `/student`, and you should see our student details displayed as a nice HTML page.

Figure 4.1 shows a screenshot our student details web page.



Student SHOW page

id = 1
name = matt smith

Figure 4.1: Screenshot of student show page.

4.5 Creating an Entity Repository (`basic04`)

Let's create a repository class to work with collections of Student objects. So let's create class `StudentRepository` in a new directory `/src/Repository`:

```
<?php
namespace App\Repository;

use App\Entity\Student;

class StudentRepository
{
    private $students = [];

    public function __construct()
    {
        $id = 1;
        $s1 = new Student($id, 'matt', 'smith');
        $this->students[$id] = $s1;
        $id = 2;
        $s2 = new Student($id, 'joelle', 'murphy');
        $this->students[$id] = $s2;
        $id = 3;
        $s3 = new Student($id, 'frances', 'mcguinness');
        $this->students[$id] = $s3;
    }

    public function findAll()
    {
        return $this->students;
    }
}
```

4.6 The student list controller method

Now we have a repository that can supply a list of students, let's created a new route `/student/list` that will retrieve the array of student records from an instance of `StudentRepository`, and pass that array to a Twig template, to loop through and display each one. We'll give this route the internal name `student_list` in our annotation comment.

Add method `listAction()` to the controller class `StudentController`:

```
use App\Repository\StudentRepository;

...

/**
 * @Route("/student/list", name="student_list")
 */
public function listAction()
{
    $studentRepository = new StudentRepository();
    $students = $studentRepository->findAll();

    $template = 'student/list.html.twig';
    $args = [
        'students' => $students
    ];
```

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

We should see this new route in our list of routes:

```
 -------------------------- -------- -------- ------ -----------------------------------
  Name                       Method   Scheme   Host   Path
 -------------------------- -------- -------- ------ -----------------------------------
  homepage                   ANY      ANY      ANY    /
  student_show               ANY      ANY      ANY    /student
  student_list               ANY      ANY      ANY    /student/list
  ... and the debug / profile routes ...
```

4.7 The list student template `/templates/student/list.html.twig`

In directory `/templates/student`create a new Twig template named `list.html.twig`. Write the following Twig code for the template:

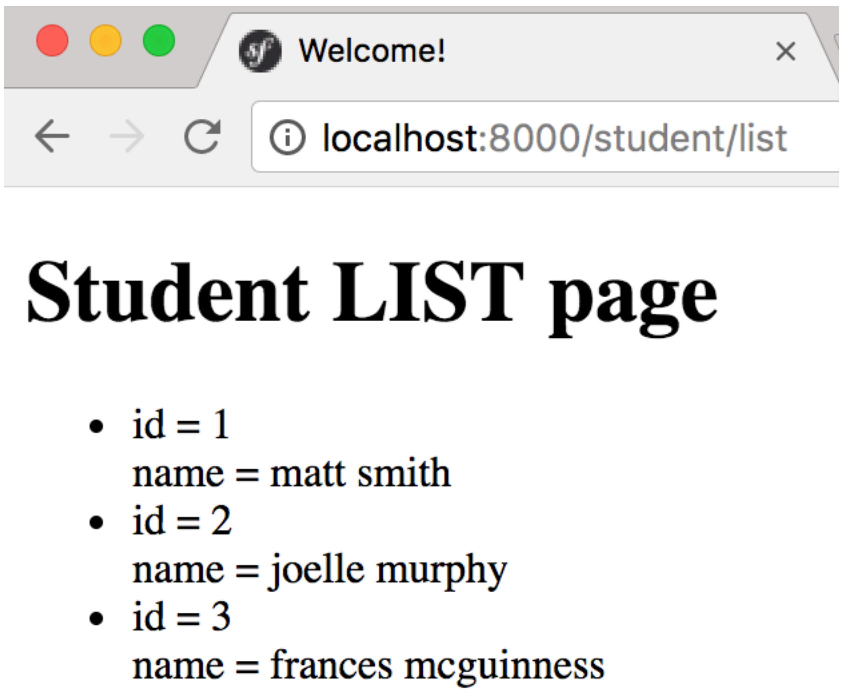
```
    {% extends 'base.html.twig' %}

    {% block body %}
        <h1>Student LIST page</h1>

        <ul>
            {% for student in students %}
                <li>
                    id = {{ student.id }}
                    <br>
                    name = {{ student.firstName }} {{ student.surname }}
                </li>
            {% endfor %}
        </ul>
    {% endblock %}
```

Run the web server and visit `/student/list`, and you should see a list of all student details displayed as a nice HTML page.

Figure 4.2 shows a screenshot our list of students web page.



Student LIST page

- id = 1
  name = matt smith
- id = 2
  name = joelle murphy
- id = 3
  name = frances mcguinness

Figure 4.2: Screenshot of student list page.

4.8 Refactor show action to show details of one Student object (project `basic05`)

The usual convention for CRUD is that the **show** action will display the details of an object given its `id`. So let's refactor our method `showAction()` to do this, and also we'll need to add a `findOne(...)` method to our repository class, that returns an object given an id.

The route we'll design will be in the form `/student/{id}`, where `{id}` will be the integer `id` of the object in the repository we wish to display. And, conincidentally, this is just the correct syntax for routes with parameters that we write in the annotation comments to define routes for controller methods in Symfony ...

```
/**
 * @Route("/student/{id}", name="student_show")
 */
public function showAction($id)
{
    $studentRepository = new StudentRepository();
    $student = $studentRepository->find($id);

    // we are assuming $student is not NULL....
```

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

While we are at it, we'll change the route for our list action, to make a list of students the default for a URL path starting with `/student`:

```
        /**
         * @Route("/student", name="student_list")
         */
        public function listAction()
        {
            ... as before
        }
```

We can check these routes via the console:

- `/student/{id}` will invoke our `showAction($id)` method
- `/student` will invoke our `listAction()` method

```
 -------------------------- -------- -------- ------ -----------------------------------
  Name                       Method   Scheme   Host   Path
 -------------------------- -------- -------- ------ -----------------------------------
  homepage                   ANY      ANY      ANY    /
  student_show               ANY      ANY      ANY    /student/{id}
  student_list               ANY      ANY      ANY    /student
```

If you have issues of Symfony not finding a new route you've added via a controller annotation comment, try the following.

It's a good idea to **CLEAR THE CACHE** when addeding/changing routes, otherwise Symfony may not recognised the new or changed routes ... Either manually delete the `/var/cache` directory, or run the `cache:clear` console command:

```
    $ php bin/console cache:clear

    // Clearing the cache for the dev environment with debug true
    [OK] Cache for the "dev" environment (debug=true) was successfully cleared.
```

Symfony caches (stores) routing data and also rendered pages from Twig, to speed up response time. But if you have changed controllers and routes, sometimes you have to manually delete the cache to ensure all new routes are checked against new requests.

4.9 Make each item in list a link to show

Let's link our templates together, so that we have a clickable link for each student listed in the list template, that then makes a request to show the details for the student with that id.

In our list template `/templates/student/list.html.twig` we can get the id for the current student with `student.id`. The internal name for our show route is `student_show`. We can use the `url(...)` Twig function to generate the URL path for a route, and in this case an `id` parameter.

So we update `list.html.twig` to look as follows, where we add a list `(details)` that will request a student's details to be displayed with our show route:

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

{% block body %}
    <h1>Student LIST page</h1>

    <ul>
        {% for student in students %}
            <li>
                id = {{ student.id }}
                <br>
                name = {{ student.firstName }} {{ student.surname }}
                <br>
           <a href="{{ url('student_show', {id : student.id} ) }}">(details)</a>
            </li>
        {% endfor %}
    </ul>
{% endblock %}
```

As we can see, to pass the `student.id` parameter to the `student_show` route we write a call to Twig function `url(...)` in the form:

```
url('student_show', {<name:value-parameter-list>} )
```

We can represent a key-value array in Twig using the braces (curly brackets), and colons. So the PHP associative array (map):

```
$daysInMonth = [
    'jan' => 31,
    'feb' => 28
];
```

could be represented in Twig as:

```
set daysInMonth = {'jan':31, 'feb':28}
```

Thus we can pass an array of parameter-value pairs to a route in Twig using the brace (curly bracket) syntax, as in:

```
url('student_show', {id : student.id} )
```

4.10 Adding a `find($id)` method to the student repository

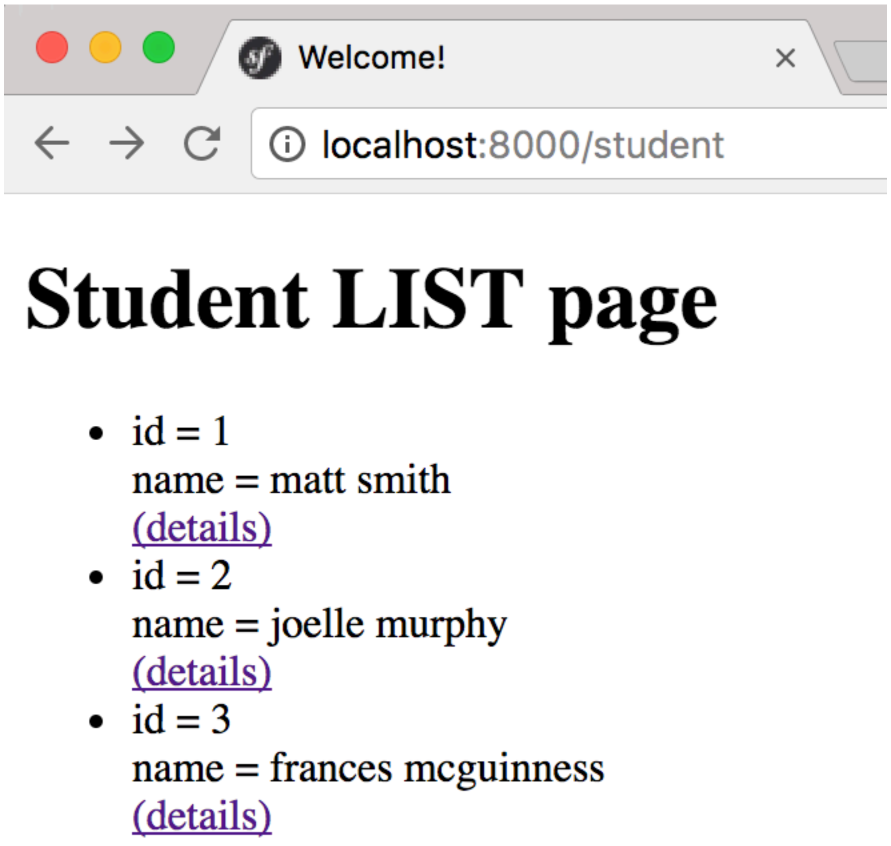
Let's add the find-one-by-id method to class `StudentRepository`:

```
public function find($id)
{
    if(array_key_exists($id, $this->students)){
        return $this->students[$id];
    } else {
        return null;
    }
}
```

If an object can be found with the key of `$id` it will be returned, otherwise `null` will be returned.

NOTE: At this time our code will fail if someone tries to show a student with an Id that is not in our repository array ...

Figure 4.3 shows a screenshot our list of students web page, with a `(details)` hypertext link to the show page for each individual student obbject.



Student LIST page

- id = 1
  name = matt smith
  (details)
- id = 2
  name = joelle murphy
  (details)
- id = 3
  name = frances mcguinness
  (details)

Figure 4.3: Screenshot of student list page, with links to show page for each student object.

4.11 Dealing with not-found issues

If we attempted to retrieve a record, but got back `null`, we might cope with it in this way in our controller method, i.e. by throwing a Not-Found-Exception (which generates a 404-page in production):

```
if (!$student) {
    throw $this->createNotFoundException(
        'No product found for id '.$id
    );
}
```

Or we could simply create an error Twig page, and display that to the user, e.g.:

```
public function showAction($id)
{
    $studentRepository = new StudentRepository();
    $student = $studentRepository->find($id);

    $template = 'student/show.html.twig';
    $args = [
        'student' => $student
    ];

    if (!$student) {
        $template = 'error/404.html.twig';
    }

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

and a Twig template `/templates/error/404.html.twig` looking like this:

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

{% block body %}
    <h1>Whoops! something went wrong</h1>

    <h2>404 - no found error</h2>

    <p>
        sorry - the item/page you were looking for could not be found
    </p>
{% endblock %}
```

Part II

Symfony and Databases

5
Doctrine the ORM

5.1 What is an ORM?

The acronym ORM stands for:

- O: Object
- R: Relational
- M: Mapping

In a nutshell projects using an ORM mean we write code relating to collections of related **objects**, without having to worry about the way the data in those objects is actually represented and stored via a database or disk filing system or whatever. This is an example of 'abstraction' - adding a 'layer' between one software component and another. DBAL is the term used for separating the database interactions completed from other software components. DBAL stands for:

- DataBase
- Abstraction
- Layer

With ORMs we can interactive (CRUD[1]) with persistent object collections either using methods of the object repositories (e.g. `findAll()`, `findOneById()`, `delete()` etc.), or using SQL-lite languages. For example Symfony uses the `Doctrine` ORM system, and that offers `DQL`, the Doctrine Query Language.

You can read more about ORMs and Symfony at:

[1] CRUD = Create-Read-Update-Delete

- Doctrine project's ORM page
- Wikipedia's ORM page
- Symfony's Doctrine help pages

5.2 Adding Doctrine DB package (project `db01`)

First we need to install the `doctrine` Symfony package:

```
$ composer req doctrine
Using version ^1.0 for symfony/orm-pack
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 14 installs, 0 updates, 0 removals
  - Installing ocramius/package-versions (1.2.0): Loading from cache
... lots of installs ...

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

 Next: Configuration

  * Modify your DATABASE_URL config in .env

  * Configure the driver (mysql) and
    server_version (5.7) in config/packages/doctrine.yaml
```

5.3 Setting the database connection URL for MySQL

Edit file `.env` to change the default database URL to one that will connect to MySQL server running at port 3306, with username `root` and password `pass`, and working with databse schema `web3`:

So change this line in `.env` from:

```
DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name
```

to

```
DATABASE_URL=mysql://root:pass@127.0.0.1:3306/web3
```

NOTE: If you prefer to parametize the database connection, use environment variables and then `${VAR}` in your URL:

```
DB_USER=root
DB_PASSWORD=pass
DB_HOST=127.0.0.1
DB_PORT=3306
DB_NAME=web3
DATABASE_URL=mysql://${DB_USER}:${DB_PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}
```

5.4 Setting the database connection URL for SQLite

If you want a non-MySQL database setup for now, then just use the basic SQLite setup:

So change this line in `.env` from:

```
DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name
```

to

```
DATABASE_URL=sqlite:///%kernel.project_dir%/var/data.db
```

This will work with SQLite database file `data.db` in directory `/var`.

5.5 Quick start

Once you've learnt how to work with Entity classes and Doctrine, these are the 3 commands you need to know (executed from the CLI console `php bin/console ...`):

1. `doctrine:database:create`
2. `doctrine:migrations:diff`
3. `doctrine:migrations:migrate` (or possibly `doctrine:schema:update --force`)
4. `doctrine:schema:validate`
5. `doctrine:fixtures:load`
6. `doctrine:query:sql`

This should make sense by the time you've reached the end of this database introduction.

6 Working with Entity classes

6.1 A `Student` DB-entity class (project `db01`)

Doctrine expects to find entity classes in a directory named `/src/Entity`, and corresponding repository classes in `/src/Repository`. We already have our `Student` and `StudentRepository` classes in the right places!

Although we'll have to make some changes to these classes of course.

6.2 Using annotation comments to declare DB mappings

We need to tell Doctrine what table name this entity should map to, and also confirm the data types of each field. We'll do this using annotation comments (although this can be also be declare in separate YAML or XML files if you prefer). We need to add a `use` statement and we define the namespace alias `ORM` to keep our comments simpler.

Our first comment is for the class, stating that it is an ORM entity and mapping it to ORM repository class `StudentRepository`.

```
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
```

```
 * @ORM\Entity(repositoryClass="App\Repository\StudentRepository")
 */
class Student
{
```

6.3 Declaring types for fields

We now use annotations to declare the types (and if appropriate, lengths) of each field.

```
/**
 * @ORM\Id
 * @ORM\GeneratedValue
 * @ORM\Column(type="integer")
 */
private $id;

/**
 * @ORM\Column(type="string")
 */
private $firstName;

/**
 * @ORM\Column(type="string")
 */
private $surname;
```

6.4 Validate our annotations

We can now validate these values. This command performs 2 actions, it checks our annotation comments, it also checks whether these match with the structure of the table the database system. Of course, since we haven't yet told Doctrine to create the actual database schema and tables, this second check will fail at this point in time.

```
$ php bin/console doctrine:schema:validate
```

The output should be something like this (if our comments are valid):

```
Mapping
-------
[OK] The mapping files are correct.

Database
```

```
--------
[ERROR] The database schema is not in sync with the current mapping file.
```

6.5 The StudenRepository class (`/src/Repository/StudentRepository`)

We need to change our repository class to be one that works with the Doctrine ORM. Unless we are writing special purpose query methods, all we really need for an ORM repository class is to ensure is subclasses `DoctrineBundle\Repository\ServiceEntityRepository` and its constructor points it to the corresponding entity class.

Change class `StudentRepository` as follows:

- remove all methods
- add `use` statements for:

  ```
  use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
  use Symfony\Bridge\Doctrine\RegistryInterface;
  ```

- make the class extend class `ServiceEntityRepository`

  ```
  class StudentRepository extends ServiceEntityRepository
  ```

- add a constructor method:

  ```
  public function __construct(RegistryInterface $registry)
  {
      parent::__construct($registry, Student::class);
  }
  ```

So the full listing for `StudentRepository` is now:

```
namespace App\Repository;

use App\Entity\Student;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Symfony\Bridge\Doctrine\RegistryInterface;

class StudentRepository extends ServiceEntityRepository
{
    public function __construct(RegistryInterface $registry)
    {
        parent::__construct($registry, Student::class);
    }
}
```

6.6 Create migrations `diff` file

We now will tell Symfony to create the a PHP class to run SQL migration commands required to change the structure of the existing database to match that of our Entity classes:

```
$ php bin/console doctrine:migrations:diff

Generated new migration class to
".../src/Migrations/Version20180213082441.php" from schema differences.
```

A migrations SQL file should have been created in `/src/Migrations/...php`:

```
namespace DoctrineMigrations;

use Doctrine\DBAL\Migrations\AbstractMigration;
use Doctrine\DBAL\Schema\Schema;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
class Version20180213082441 extends AbstractMigration
{
    public function up(Schema $schema)
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql',
        'Migration can only be executed safely on \'mysql\'.');

        $this->addSql('CREATE TABLE student (id INT AUTO_INCREMENT NOT NULL,
        first_name VARCHAR(255) NOT NULL, surname VARCHAR(255) NOT NULL, PRIMARY KEY(id)) DE
```

6.7 Run the migration to make the database structure match the entity class declarations

Run the `migrate` command to execute the created migration class to make the database schema match the structure of your entity classes, and enter `y` when prompted - if you are happy to go ahead and change the database structure:

```
$ 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 lost. Are you sure you wish to continue? (y/n)y
Migrating up to 20180201223133 from 0

  ++ migrating 20180201223133

     -> CREATE TABLE product (id INT AUTO_INCREMENT NOT NULL,
     description VARCHAR(100) NOT NULL, price NUMERIC(10, 2) DEFAULT NULL,
     PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci ENGINE = InnoDB

  ++ migrated (0.14s)

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

  ++ finished in 0.14s
  ++ 1 migrations executed
  ++ 1 sql queries
```

You can see the results of creating the database schema and creating table(s) to match your ORM entities using a database client such as MySQL Workbench. Figure 6.1 shows a screenshot of MySQL Workbench showing the database's `student` table to match our `Student` entity class.

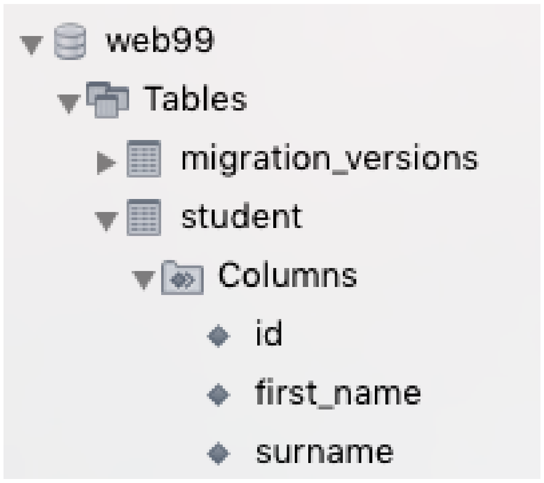


Figure 6.1: Screenshot MySQL Workbench and generated schema and product table.

6.8 Re-validiate our annotations

We should get 2 "ok"s if we re-validate our schema now:

```
$ php bin/console doctrine:schema:validate
```

The output should be something like this (if our comments are valid):

```
Mapping
-------
[OK] The mapping files are correct.

Database
--------
[OK] The database schema is in sync with the mapping files.
```

6.9 Generating entities from an existing database

Doctrine allows you to generated entities matching tables in an existing database. Learn about that from the Symfony documentation pages:

- Symfony docs on inferring entites from existing db tables

6.10 Note - use maker to save time (project `db02`)

We could have create our Student entity and StudentRepository classes from scratch, using the `make` package:

```
$ php bin/console make:entity Student

 created: src/Entity/Student.php
 created: src/Repository/StudentRepository.php


  Success!


 Next: Add more fields to your entity and start using it.
 Find the documentation at https://symfony.com/doc/current/doctrine.html#creating-an-entity-
```

Then we would have had to add the `firstName` and `surname` properties (and their annotation comments):

```
/**
 * @ORM\Column(type="string")
 */
```

```
    private $firstName;

    /**
     * @ORM\Column(type="string")
     */
    private $surname;
```

Finally we would have had to generate getters and setters for these 2 fields, and migrate to the database.

That's it!

7
Symfony approach to database CRUD

7.1 Creating new student records (project `db03`)

Let's add a new route and controller method to our `StudentController` class. This will define the `createAction()` method that receives parameter `$name` extracted from the route `/students/create/{name}`.

We need to add `use` statements, so our controller class can work with `Student` and `StudentRepository` objects.

Update the class declaration as follows:

```
namespace App\Controller;

use App\Entity\Student;
use App\Repository\StudentRepository;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;
```

Creating a new `Student` object is straightforward, given `$firstName` and `$surname` from the URL-encoded GET name=value pairs:

```
$student = new Student();
$student->setFirstName($firstName);
$student->setSurame($surname);
```

Then we see the Doctrine code, to get a reference to the ORM `EntityManager`, to tell it to store (`persist`) the object `$product`, and then we tell it to finalise (i.e. write to the database) any entities waiting to be persisted:

```
$em = $this->getDoctrine()->getManager();
$em->persist($student);
$em->flush();
```

So the code for our create action is:

```
/**
 * @Route("/student/create/{firstName}/{surname}")
 */
public function createAction($firstName, $surname)
{
    $student = new Student();
    $student->setFirstName($firstName);
    $student->setSurame($surname);

    // entity manager
    $em = $this->getDoctrine()->getManager();

    // tells Doctrine you want to (eventually) save the Product (no queries yet)
    $em->persist($student);

    // actually executes the queries (i.e. the INSERT query)
    $em->flush();

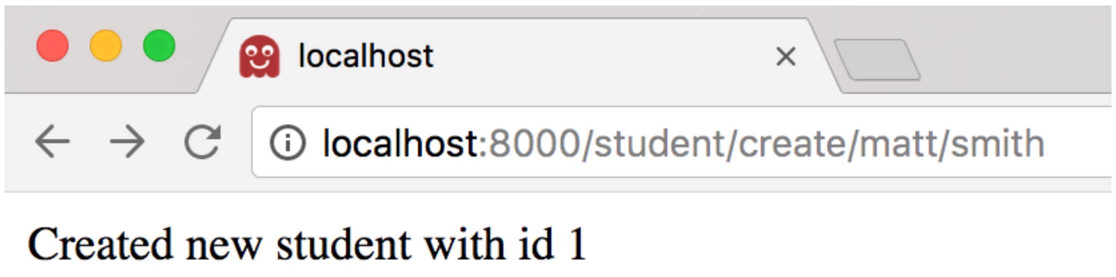
    return new Response('Created new student with id '.$student->getId());
}
```

The above now means we can create new records in our database via this new route. So to create a record with name `matt smith` just visit this URL with your browser:

```
http://localhost:8000/students/create/matt/smith
```

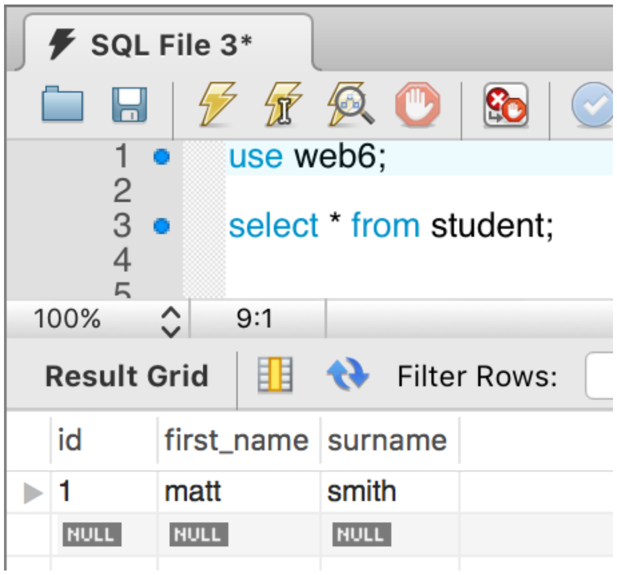
Figure 7.1 shows how a new record `matt smith` is added to the database table via route `/students/create/{firstName}/{surname}`.

We can see these records in our database. Figure 7.2 shows our new `students` table created for us.



Created new student with id 1

Figure 7.1: Creating new student via route `/students/create/{firstName}/{surname}`.



```
use web6;

select * from student;
```

id	first_name	surname
1	matt	smith
NULL	NULL	NULL

Figure 7.2: Controller created records in PHPMyAdmin.

7.2 Query database with SQL from CLI server

The `doctrine:query:sql` CLI command allows us to run SQL queries to our database directly from the CLI. Let's request all `Product` rows from table `product`:

```
$ php bin/console doctrine:query:sql "select * from student"

.../vendor/doctrine/common/lib/Doctrine/Common/Util/Debug.php:71:
array (size=1)
  0 =>
    array (size=3)
      'id' => string '1' (length=1)
      'first_name' => string 'matt' (length=4)
      'surname' => string 'smith' (length=5)
```

7.3 Updating the listAction() to use Doctrine

Doctrine creates repository objects for us. So we change the first line of method listAction() to the following:

```
$studentRepository = $this->getDoctrine()->getRepository('App:Student');
```

Doctrine repositories offer us lots of useful methods, including:

```
// query for a single record by its primary key (usually "id")
$student = $repository->find($id);

// dynamic method names to find a single record based on a column value
$student = $repository->findOneById($id);
$student = $repository->findOneByFirstName('matt');

// find *all* products
$students = $repository->findAll();

// dynamic method names to find a group of products based on a column value
$products = $repository->findBySurname('smith');
```

So we need to change the second line of our method to use the findAll() repository method:

```
$students = $studentRepository->findAll();
```

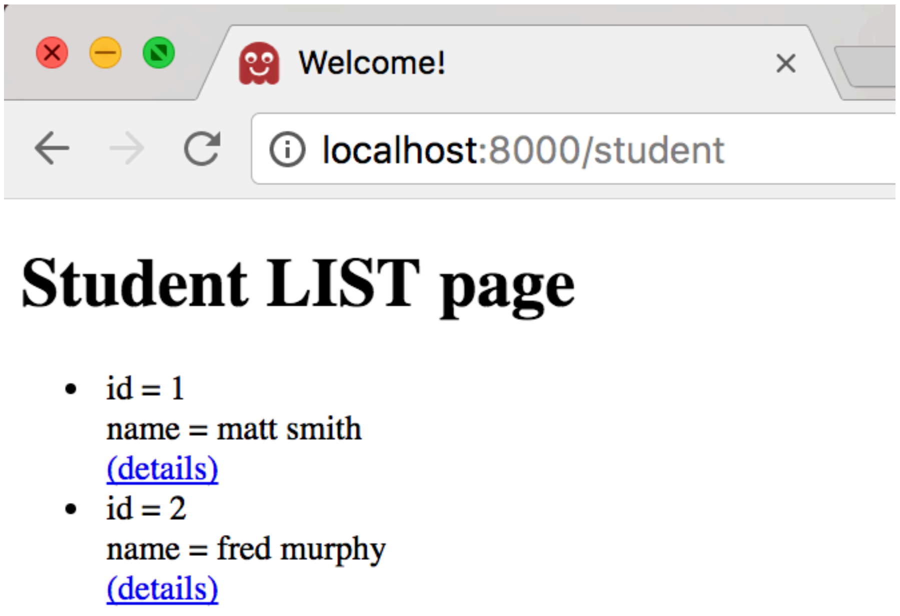
Our listAction() method now looks as follows:

```
public function listAction(Request $request)
{
    $studentRepository = $this->getDoctrine()->getRepository('App:Student');
    $students = $studentRepository->findAll();

    $argsArray = [
        'students' => $students
    ];

    $templateName = 'students/list';
    return $this->render($templateName . '.html.twig', $argsArray);
}
```

Figure 7.3 shows Twig HTML page listing all students generated from route `/student`.



Student LIST page

- id = 1
  name = matt smith
  (details)
- id = 2
  name = fred murphy
  (details)

Figure 7.3: Listing all database student records with route `/student`.

7.4 Deleting by id

Let's define a delete route `/student/delete/{id}` and a `deleteAction()` controller method. This method needs to first retreive the object (from the database) with the given ID, then ask to remove it, then flush the changes to the database (i.e. actually remove the record from the database). Note in this method we need both a reference to the entity manager `$em` and also to the student repository object `$studentRepository`:

```
    /**
     * @Route("/student/delete/{id}")
     */
    public function deleteAction($id)
    {
        // entity manager
        $em = $this->getDoctrine()->getManager();
        $studentRepository = $this->getDoctrine()->getRepository('App:Student');

        // find thge student with this ID
        $student = $studentRepository->find($id);

        // tells Doctrine you want to (eventually) delete the Student (no queries yet)
        $em->remove($student);

        // actually executes the queries (i.e. the DELETE query)
        $em->flush();

        return new Response('Deleted student with id '.$id);
    }
```

7.5 Updating given id and new name

We can do something similar to update. In this case we need 3 parameters: the id and the new first and surname. We'll also follow the Symfony examples (and best practice) by actually testing whether or not we were successful retrieving a record for the given id, and if not then throwing a 'not found' exception.

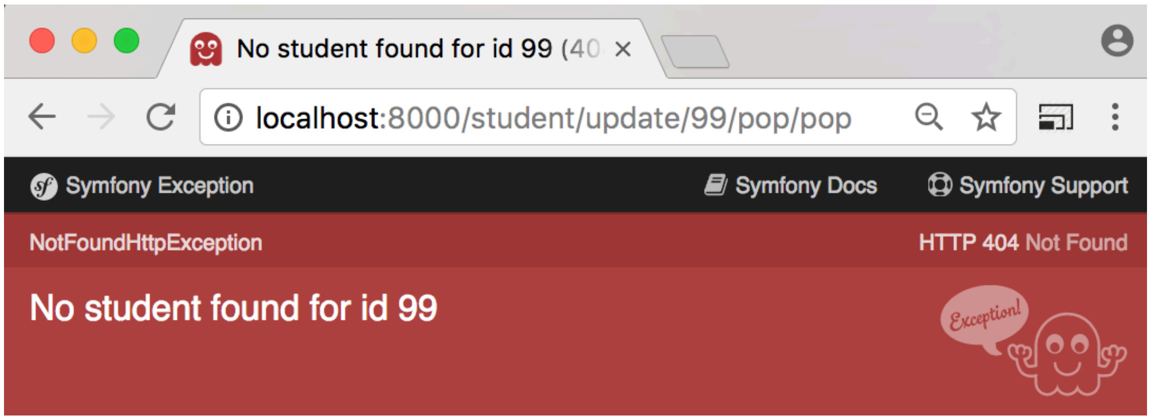
```
    /**
     * @Route("/student/update/{id}/{newFirstName}/{newSurname}")
     */
    public function updateAction($id, $newFirstName, $newSurname)
    {
        $em = $this->getDoctrine()->getManager();
        $student = $em->getRepository('App:Student')->find($id);

        if (!$student) {
            throw $this->createNotFoundException(
                'No student found for id '.$id
            );
        }

        $student->setFirstName($newFirstName);
        $student->setSurname($newSurname);
        $em->flush();

        return $this->redirectToRoute('homepage');
    }
```

Until we write an error handler we'll get Symfony style exception pages, such as shown in Figure 7.4 when trying to update a non-existent student with id=99.


No student found for id 99 (40

localhost:8000/student/update/99/pop/pop

Symfony Exception

Symfony Docs

Symfony Support

NotFoundHttpException

HTTP 404 Not Found

No student found for id 99

Figure 7.4: Listing all database student records with route `/students/list`.

Note, to illustrate a few more aspects of Symfony some of the coding in `updateAction()` has been written a little differently:

- we are getting the reference to the repository via the entity manager `$em->getRepository('App:Student')`
- we are 'chaining' the `find($id)` method call onto the end of the code to get a reference to the repository (rather than storing the repository object reference and then invoking `find($id)`). This is an example of using the 'fluent' interface[1] offered by Doctrine (where methods finish by returning an reference to their object, so that a sequence of method calls can be written in a single statement.
- rather than returning a `Response` containing a message, this controller method redirect the webapp to the route named `homepage`

We should also add the 'no student for id' test in our `deleteAction()` method ...

7.6 Updating our show action

We can now update our code in our `showAction(...)` to retrieve the record from the database:

```
    public function showAction($id)
    {
```

[1] read about it at Wikipedia

```
        $em = $this->getDoctrine()->getManager();
        $student = $em->getRepository('App:Student')->find($id);
```

So our full method for the show action looks as follows:

```
    /**
     * @Route("/student/{id}", name="student_show")
     */
    public function showAction($id)
    {
        $em = $this->getDoctrine()->getManager();
        $student = $em->getRepository('App:Student')->find($id);

        $template = 'student/show.html.twig';
        $args = [
            'student' => $student
        ];

        if (!$student) {
            $template = 'error/404.html.twig';
        }

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

We could, if we wish, throw a 404 error exception if no student records can be found for the given id, rather than rendering an error Twig template:

```
    if (!$student) {
        throw $this->createNotFoundException(
            'No student found for id '.$id
        );
    }
```

7.7 Redirecting to show after create/update

Keeping everything nice, we should avoid creating one-line and non-HMTL responses like the following in `ProductController->createAction(...)`:

```
    return new Response('Saved new product with id '.$product->getId());
```

Let's go back to the list page after a create or update action. Tell Symfony to redirect to the `student_show` route for

```
return $this->redirectToRoute('student_show', [
    'id' => $student->getId()
]);
```

e.g. update the `createAction()` method to be as follows:

```
/**
 * @Route("/student/update/{id}/{newFirstName}/{newSurname}")
 */
public function updateAction($id, $newFirstName, $newSurname)
{
    $em = $this->getDoctrine()->getManager();
    $student = $em->getRepository('App:Student')->find($id);

    if (!$student) {
        throw $this->createNotFoundException(
            'No student found for id '.$id
        );
    }

    $student->setFirstName($newFirstName);
    $student->setSurname($newSurname);
    $em->flush();

    return $this->redirectToRoute('student_show', [
        'id' => $student->getId()
    ]);
}
```

7.8 Given `id` let Doctrine find Product automatically (project `basic5`)

One of the features added when we installed the `annotations` bundle was the **Param Converter**. Perhaps the most used param converter is when we can substitute an entity `id` for a reference to the entity itself.

We can simplify our `showAction(...)` from:

```
/**
 * @Route("/student/{id}", name="student_show")
 */
public function showAction($id)
```

```
    {
        $em = $this->getDoctrine()->getManager();
        $student = $em->getRepository('App:Student')->find($id);

        $template = 'student/show.html.twig';
        $args = [
            'student' => $student
        ];

        if (!$student) {
            $template = 'error/404.html.twig';
        }

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

to just:

```
    /**
     * @Route("/student/{id}", name="student_show")
     */
    public function showAction(Student $student)
    {
        $template = 'student/show.html.twig';
        $args = [
            'student' => $student
        ];

        if (!$student) {
            $template = 'error/404.html.twig';
        }

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

The Param-Converter will use the Doctrine ORM to go off, find the `ProductRepository`, run a `find(<id>)` query, and return the retrieved object for us!

Note - if there is no record in the database corresponding to the `id` then a 404-not-found error page will be generated.

Learn more about the Param-Converter on the Symfony documentation pages:

- https://symfony.com/doc/current/doctrine.html#automatically-fetching-objects-paramconverter

- http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html

Likewise for delete action:

```
    /**
     * @Route("/student/delete/{id}")
     */
    public function deleteAction(Student $student)
    {
        // entity manager
        $em = $this->getDoctrine()->getManager();

        // store ID before deleting, so can report ID later
        $id = $student->getId();

        // tells Doctrine you want to (eventually) delete the Student (no queries yet)
        $em->remove($student);

        // actually executes the queries (i.e. the DELETE query)
        $em->flush();

        return new Response('Deleted student with id = '. $id);
    }
```

Likewise for update action:

```
/**
 * @Route("/student/update/{id}/{newFirstName}/{newSurname}")
 */
public function updateAction(Student $student, $newFirstName, $newSurname)
{
    $em = $this->getDoctrine()->getManager();

    $student->setFirstName($newFirstName);
    $student->setSurname($newSurname);
    $em->flush();

    return $this->redirectToRoute('student_show', [
        'id' => $student->getId()
    ]);
}
```

NOTE - we will now get ParamConverter errors rather than 404 errors if no record matches ID through …

7.9 Creating the CRUD controller automatically from the CLI

Here is something you might want to look into …

```
$ composer req security-csrf form
$ composer req --dev koff/crud-maker-bundle
```

Then use the make crud command:

```
$ php bin/console make:crud Student
```

8 Fixtures - setting up a database state

8.1 Initial values for your project database (project `db06`)

Fixtures play two roles:

- inserting initial values into your database (e.g. the first `admin` user)
- setting up the database to a known state for **testing** purposes

Doctrine provides a Symfony fixtures **bundle** that makes things very straightforward.

Learn more about Symfony fixtures at:

- Symfony website fixtures page

8.2 Installing and registering the fixtures bundle

8.2.1 Install the bundle

Use Composer to install the bunder in the the `/vendor` directory:

```
composer req --dev doctrine/doctrine-fixtures-bundle
```

8.3 Writing the fixture classes

We need to locate our fixtures in our `/src` directory, inside a `/DataFixtures` directory. The path for our data fixtures classes should be `/src/DataFixtures/`.

Fixture classes need to implement the interfaces, `Fixture`.

NOTE: Some fixtures will also require your class to include the `ContainerAwareInterface`, for when our code also needs to access the container,b y implement the `ContainerAwareInterface`.

Let's write a class to create 3 objects for entity `App\Entity\Student. The class will be declared in file`/src/DataFixtures/ORM/LoadStudentData.php'.

Ensure your class extends the Doctrine `Fixture` class:

```
namespace App\DataFixtures;

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

class LoadStudents extends Fixture
{
```

We need a `load(...)` method, that gets invoked when we are loading fixtures from the CLI. This method creates objects for the entities we want in our database, and the saves (persists) them to the database. Finally the `flush()` method is invoked, forcing the database to be updated with all queued new/changed/deleted objects:

In the code below, we create 3 `Student` objects and have them persisted to the database.

```
public function load(ObjectManager $manager)
{
    $s1 = new Student();
    $s1->setFirstName('matt');
    $s1->setSurname('smith');
    $s2 = new Student();
    $s2->setFirstName('joe');
    $s2->setSurname('bloggs');
    $s3 = new Student();
    $s3->setFirstName('joelle');
    $s3->setSurname('murph');

    $manager->persist($s1);
    $manager->persist($s2);
    $manager->persist($s3);
```

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

8.4 Loading the fixtures

WARNING Fixtures **replace** existing DB contents - so you'll lose any previous data when you load fixtures...

Loading fixtures involves deleting all existing database contents and then creating the data from the fixture classes - so you'll get a warning when loading fixtures. At the CLI type:

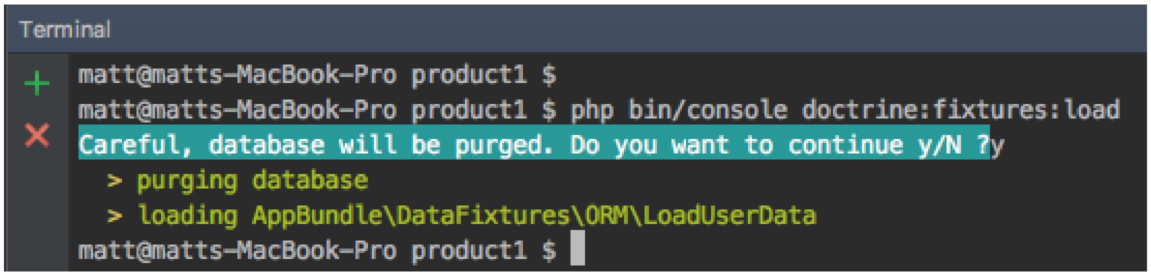
```
php bin/console doctrine:fixtures:load
```

You should then be asked to enter `y` (for YES) if you want to continue:

```
$ php bin/console doctrine:fixtures:load

Careful, database will be purged. Do you want to continue y/N ?y
  > purging database
  > loading App\DataFixtures\LoadStudents
```

Figure 8.1 shows an example of the CLI output when you load fixtures (in the screenshot it was for initial user data for a login system...)



```
matt@matts-MacBook-Pro product1 $
matt@matts-MacBook-Pro product1 $ php bin/console doctrine:fixtures:load
Careful, database will be purged. Do you want to continue y/N ?y
  > purging database
  > loading AppBundle\DataFixtures\ORM\LoadUserData
matt@matts-MacBook-Pro product1 $
```

Figure 8.1: Using CLI to load database fixtures.

Alternatively, you could execute an SQL query from the CLI using the `doctrine:query:sql` command:

```
$ php bin/console doctrine:query:sql "select * from student"

/.../db06_fixtures/vendor/doctrine/common/lib/Doctrine/Common/Util/Debug.php:71:
array (size=3)
  0 =>
    array (size=3)
      'id' => string '13' (length=2)
      'first_name' => string 'matt' (length=4)
      'surname' => string 'smith' (length=5)
  1 =>
    array (size=3)
      'id' => string '14' (length=2)
      'first_name' => string 'joe' (length=3)
      'surname' => string 'bloggs' (length=6)
  2 =>
    array (size=3)
      'id' => string '15' (length=2)
      'first_name' => string 'joelle' (length=6)
      'surname' => string 'murph' (length=5)
```

8.5 User Faker to generate plausible test data

For testing purposes the Faker library is fantastic for generating plausible, random data.

Let's install it and generate some random students in our Fixtures class:

1. use Composer to add the Faker package to our /vendor/ directory:

   ```
   $ composer req fzaninotto/faker
   Using version ^1.7 for fzaninotto/faker
   ./composer.json has been updated
   Loading composer repositories with package information
   ...
   Executing script assets:install --symlink --relative public [OK]
   ```

2. refactor our load() method in /src/DataFixtures/LoadStudents.php to create a Faker 'factory', and loop to generate names for 10 male students, and insert them into the database:

   ```
   public function load(ObjectManager $manager) {
       $faker = \Faker\Factory::create();

       $numStudents = 10;
       for ($i=0; $i < $numStudents; $i++) {
           $firstName = $faker->firstNameMale;
           $surname = $faker->lastName;

           $student = new Student();
           $student->setFirstName($firstName);
           $student->setSurname($surname);

           $manager->persist($student);
   ```

```
        }

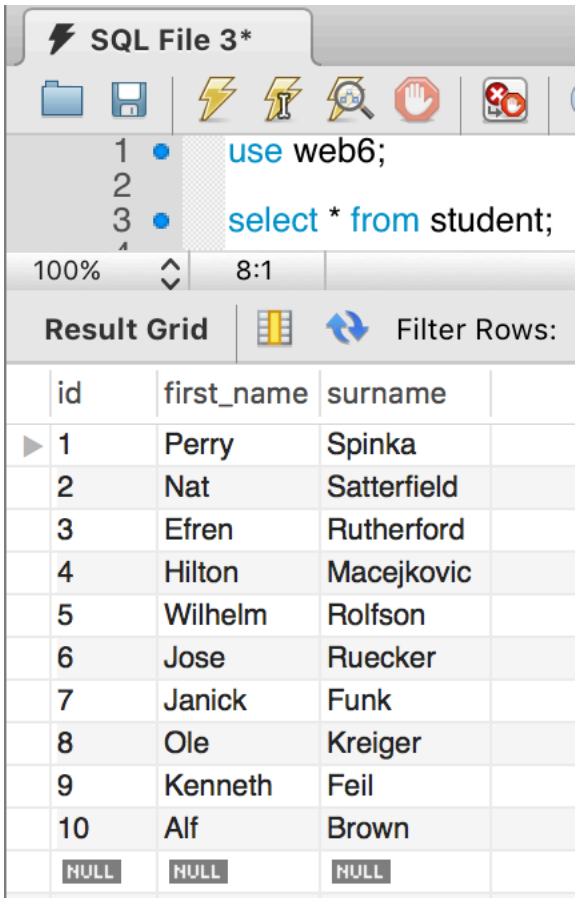
        $manager->flush();
    }
```

3. use the CLI Doctrine command to run the fixtures creation method:

```
$ php bin/console doctrine:fixtures:load
Careful, database will be purged. Do you want to continue y/N ?y
  > purging database
  > loading App\DataFixtures\LoadStudents
```

That's it - you should now have 10 'fake' students in your database.

Figure 8.2 shows a screenshot of the DB client showing the 10 created 'fake' students.



```
use web6;

select * from student;
```

id	first_name	surname
1	Perry	Spinka
2	Nat	Satterfield
3	Efren	Rutherford
4	Hilton	Macejkovic
5	Wilhelm	Rolfson
6	Jose	Ruecker
7	Janick	Funk
8	Ole	Kreiger
9	Kenneth	Feil
10	Alf	Brown
NULL	NULL	NULL

Figure 8.2: Ten fake students inserted into DB.

Learn more about the `Faker` class at its Github project page:

- https://github.com/fzaninotto/Faker

Part III

Froms and form processing

9
DIY forms

9.1 Adding a form for new Student creation (`form01`)

Let's create a DIY (Do-It-Yourself) HTML form to create a new student. We'll need:

- a controller method (and template) to display our new student form
  - route `/student/new`
- a controller method to process the submitted form data
  - route `/student/processNewForm`

The form will look as show in Figure 9.1.

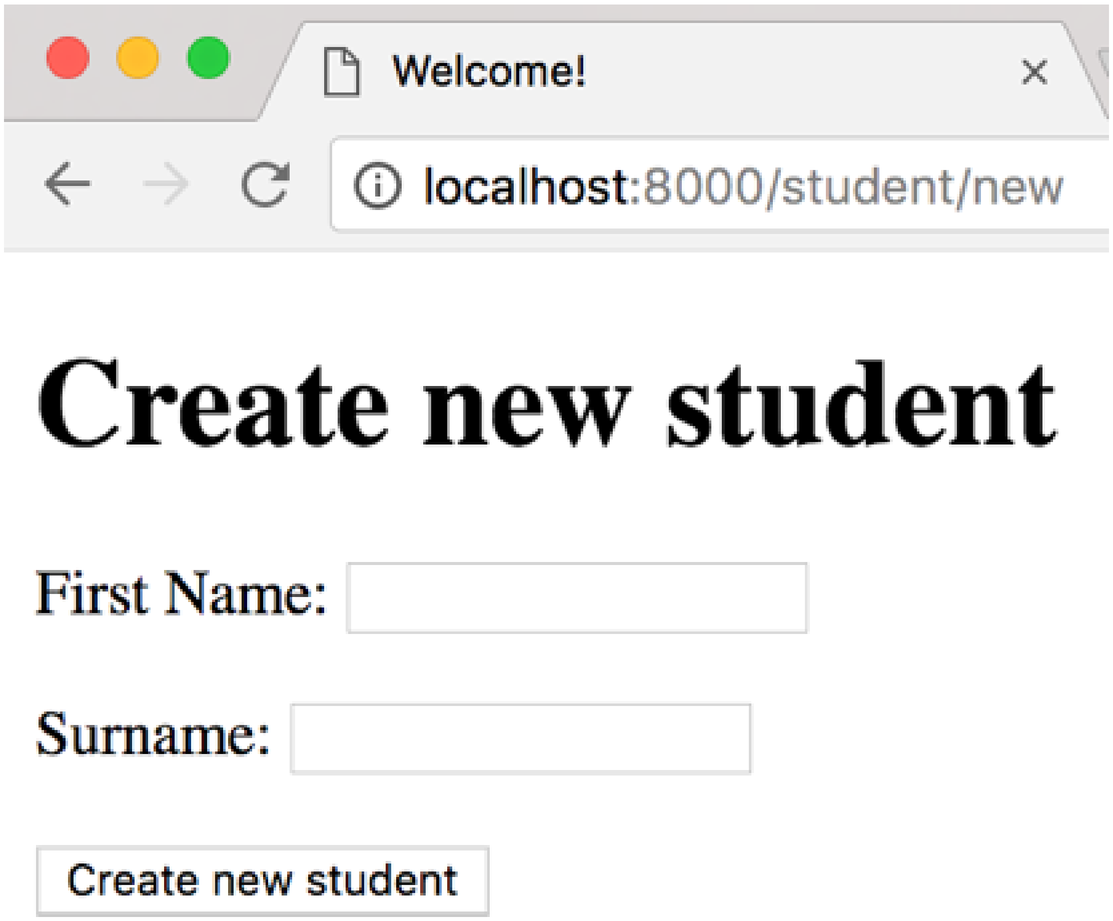


Figure 9.1: Form for a new student

9.2 Twig new student form

Here is our new student form '/templates/student/new.html.twig':

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

{% block pageTitle %}new student form{% endblock %}

{% block body %}
    <h1>Create new student</h1>

    <form action="/student/processNewForm" method="POST">
            First Name:
            <input type="text" name="firstName">
        <p>
            Surname:
            <input type="text" name="surname">
        <p>
            <input type="submit" value="Create new student">
    </form>
{% endblock %}
```

9.3 Controller method (and annotation) to display new student form

Here is our `StudentController` method to display our Twig form:

```
/**
 * @Route("/student/new", name="student_new_form")
 */
public function newFormAction()
{
    $argsArray = [
    ];

    $templateName = 'student/new';
    return $this->render($templateName . '.html.twig', $argsArray);
}
```

NOTE: Ensure the 'show' action is last in our controller. Since we don't want `/student/new` being treated as `/student/{id = 'new'}`, so our new form action method should be placed before our show action.

We'll also add a link to this form route in our list of students page. So we add to the end of `/templates/student/list.html.twig` the following link:

```
        (... existing Twig code to show list of students here ...)

        <hr>
        <a href="{{ path('student_new_form')}}">
            create NEW student
        </a>
    {% endblock %}
```

9.4 Controller method to process POST form data

We can access POST submitted data using the following expression:

```
$request->request->get(<POST_VAR_NAME>)
```

So we can extract and store in `$firstName` and `$surname` the POST `firstName` and `surname` parameters by writing the following:

```
$firstName = $request->request->get('firstName');
$surname = $request->request->get('surname');
```

We will need access to the HTTP request, so we must declare a method parameter of `Request $request`. Symfony will now automatically provide this method with access to an object `$request`, which we can interrogate for things like the HTTP method of the request, and any name/value variables received in the request:

```
public function processNewFormAction(Request $request)
{
```

Note: We have not **namespaced** class `Request`, so, at the top of our controller class declaration, we need to add an appropriate `uses` statement, so PHP knows **which** `Request` class we are referring to. So we need to add the following before the class declaration:

```
use Symfony\Component\HttpFoundation\Request;
```

Our full listing for `StudentController` method `processNewForm()` looks as follows:

```
/**
 * @Route("/student/processNewForm", name="student_process_new_form")
 */
public function processNewFormAction(Request $request)
{
    // extract name values from POST data
    $firstName = $request->request->get('firstName');
    $surname = $request->request->get('surname');
```

```
        // forward this to the createAction() method
        return $this->createAction($firstName, $surname);
    }
```

Note that we then invoke our existing `createAction(...)` method, passing on the extracted `$firsName` and `$surname` strings.

9.5 Validating form data, and displaying temporary 'flash' messages in Twig

What should we do if an empty name string was submitted? We need to **validate** form data, and inform the user if there was a problem with their data.

Symfony offers a very useful feature called the 'flash bag'. Flash data exists for just 1 request and is then deleted from the session. So we can create an error message to be display (if present) by Twig, and we know some future request to display the form will no have that error message in the session any more.

9.6 Three kinds of flash message: notice, warning and error

Typically we create 3 different kinds of flash notice:

- notice
- warning
- error

Our Twig template would style these differntly (e.g. pink background for errors etc.). Here is how to creater a flash message and have it stored (for 1 request) in the session:

```
    $this->addFlash(
            'error',
            'Your changes were saved!'
        );
```

In Twig we can attempt to retrieve flash messages in the following way:

```
    {% for flash_message in app.session.flashBag.get('notice') %}
        <div class="flash-notice">
            {{ flash_message }}
        </div>
    {% endfor %}
```

9.7 Adding flash display (with CSS) to our Twig template (project `form02`)

First let's create a CSS stylesheet and ensure it is always loaded by adding its import into our `base.html.twig` template.

First create the directory `css` in `/web` - remember that `/web` is the Symfony public folder, where all public images, CSS, javascript and basic front controllers (`app.php` and `app_dev.php`) are served from).

Now create CSS file `/public/css/flash.css` containing the following:

```
.flash-error {
    padding: 1rem;
    margin: 1rem;
    background-color: pink;
}
```

Next we need to edit our `/templates/base.html.twig` so that every page in our webapp will have imported this CSS stylesheet. Edit the `<head>` element in `base.html.twig` as follows:

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

        <style>
            @import '/css/flash.css';
        </style>
        {% block stylesheets %}{% endblock %}
    </head>
```

9.8 Adding validation logic to our form processing controller method

Our form data is valid if **neither** name received was empty:

```
$isValid = !empty($firstName) && !empty($surname);
```

Now we can add the empty string test (and flash error message) to our `processNewFormAction()` method as follows:

```
public function processNewFormAction(Request $request)
{
    // extract name values from POST data
    $firstName = $request->request->get('firstName');
    $surname = $request->request->get('surname');

    // valid if neither value is EMPTY
    $isValid = !empty($firstName) && !empty($surname);

    if(!$isValid){
        $this->addFlash(
            'error',
            'student firstName/surname cannot be an empty string'
        );

        // forward this to the createAction() method
        return $this->newFormAction($request);
    }

    // forward this to the createAction() method
    return $this->createAction($firstName, $surname);
}
```

So if the `$name` we extracted from the POST data is an empty string, then we add an `error` flash message into the session 'flash bag', and forward on processing of the request to our method to display the new student form again.

Finally, we need to add code in our new student form Twig template to display any error flash messages it finds. So we edit `/templates/student/new.html.twig` as follows:

```
{% extends '_base.html.twig' %}
{% block pageTitle %}new student form{% endblock %}

{% block body %}

    <h1>Create new student</h1>

    {% for flash_message in app.session.flashBag.get('error') %}
        <div class="flash-error">
            {{ flash_message }}
        </div>
    {% endfor %}
    (... show HTML form as before ...)
```

9.9 Postback logic (project `form03`)

A common approach (and used in CRUD auto-generated code) is to combine the logic for displaying a form, and processing its submission, in a single method. The logic for this is that if any of the submitted data was invalid (or missing), then the default form processing can go back to re-displaying the form (with an appropriate 'flash' error message) to the user.

This approach is known as a 'postback' - i.e. that the submission of the form is POSTEd back to the same method that displayed the form.

The logic usually goes something like this:

1. Define a controller method route for both `GET` and `POST` HTTP methods
2. Attempt to find values in the `POST` request body
3. If the form was submitted by the `POST` method AND the data was all valid THEN
    - invoke the method to create the object/process the data and retuurn an appropriate success response
4. (else) If POST submitted but NOT valid THEN
    - create an appropriate flash error message in the session
5. return a response showing the form via Twig `render(...)` method
    - passing values, if we want a 'sticky' form remembering partly valid form values

Let's name our combined show form & process form controller method `newAction(...)`, name its internal route as `student_new`, and declare that only `POST` and `GET` HTTP requests are to be routed to this method[1]:

```
/**
 * @Route("/student/new", name="student_new",  methods={"POST", "GET"})
 */
public function newAction(Request $request)
{
```

Remember, we will need access to the `Request` object to get access to the POST values, and to check with HTTP method the request was sent via.

The simplest request will be for the new student form to be displayed, the logic for that is from our old `newFormAction()`:

[1]By default a controller method that does not declare any specific HTTP methods will be used for **any** HTTP method matching the route pattern. So it is good practice to start limiting our controller methods to only those HTTP methods that are valid for how we wish our web application to behave...

```
    // render the form for the user
    $template = 'student/new.html.twig';
    $argsArray = [
    ];

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

The rest of the logic in this method will related to when the HTTP request is POST-submission of the form, and its validation. We can check whether the HTTP request was received as follows:

```
    $isSubmitted = $request->isMethod('POST');
```

We can attempt to retrieve values from a POST submitted form as follows:

```
    // attempt to find values in POST variables
    $firstName = $request->request->get('firstName');
    $surname = $request->request->get('surname');
```

Note: If there was no named variable in the POST data, the variables `$firstName` and `$surnamne` will return `null` (and so will register as `true` when tested with `isEmpty(...)`).

If our form validation logic is simply that neither name can be an empty string (or null), then we can write an expression to check that neither is empty as follows:

```
    $isValid = !empty($firstName) && !empty($surname);
```

Our core logic for this controller is that **if** the request was an HTTP `POST` method **and** the values received were value, then we are happy to accept the form data and go off an create a new object (and return an appropriate response). We can write this as follows:

```
    // if SUBMITTED & VALID - go ahead and create new object
    if ($isSubmitted && $isValid) {
        return $this->createAction($firstName, $surname);
    }
```

NOTE: Since our method is invoking a `return`, then no further processing of statements in the method will occur. I.e. we can locate our logic for (re)displaying the form after this `if`-test.

If it was a `POST` submitted form but the data was **not** valid, then we should create a 'flash' error message in the session:

```
   if ($isSubmitted && !$isValid) {
        $this->addFlash(
            'error',
            'student firstName/surname cannot be an empty string'
        );
    }
```

We can now simply replace the previous 2 methods `processNewFormAction()` and `newFormAction()` with our new single postback method `newAction(...)` as follows:

```
/**
 * @Route("/student/new", name="student_new",  methods={"POST", "GET"})
 */
public function newAction(Request $request)
{
    // attempt to find values in POST variables
    $firstName = $request->request->get('firstName');
    $surname = $request->request->get('surname');

    // valid if neither value is EMPTY
    $isValid = !empty($firstName) && !empty($surname);

    // was form submitted with POST method?
    $isSubmitted = $request->isMethod('POST');

    // if SUBMITTED & VALID - go ahead and create new object
    if ($isSubmitted && $isValid) {
        return $this->createAction($firstName, $surname);
    }

    // if submitted and NOT valid, add a FLASH ERROR message
    if ($isSubmitted && !$isValid) {
        $this->addFlash(
            'error',
            'student firstName/surname cannot be an empty string'
        );
    }

    // render the form for the user
    $template = 'student/new.html.twig';
    $argsArray = [
        'firstName' => $firstName,
        'surname' => $surname
    ];

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

Finally (!) we can achieve a 'sticky' form by passing any value in `$firstName` and `$surname` to our

Twig template in its argument array:

```
$argsArray = [
    'firstName' => $firstName,
    'surname' => $surname
];
```

These will either be null, or have the string values from the POST submited form attempt. We re-display these values (if no null) by adding `value=""` attributed in our Twig form template `/templates/student/new.html.twig` as follows:

```
<form action="/student/processNewForm" method="POST">
        First Name:
        <input type="text" name="firstName">
    <p>
        Surname:
        <input type="text" name="surname">
    <p>
        <input type="submit" value="Create new student">
</form>
```

9.10 Extra notes

Here is how to work with Enum style drop-down combo-boxes:

- Articled on Symfony Enums in forms frmo Maxence POUTORD

10

Automatic forms generated from Entities

10.1 Using the Symfony form generator (project `form04`)

Given an object of an Entity class, Symfony can analyse its property names and types, and generate a form (with a little help). That's what we'll do in this chapter.

However, first, let's simplify something for later, we'll make our `createAction()` expect to be given a reference to a `Student` object (rather than expect 2 string parameters `firstName` and `surname`):

```
public function createAction($student)
{
    $em = $this->getDoctrine()->getManager();
    $em->persist($student);
    $em->flush();

    return $this->listAction($student->getId());
}
```

10.2 The form generator

First ensure your project has this package:

```
$ composer req form
```

In a controller we can create a `$form` object, and pass this as a Twig variable to the template `form`. Twig offers 4 special functions for rendering (displaying) forms, these are:

- `form()` :: display the whole form (i.e. display the whole thing in one line!)
- `form_start()` :: display the beginning of the form
- `form_widget()` :: display all the fields etc.
- `form_end()` :: display the end of the form

So we can simplify the `body` block of our Twig template (`/app/Resources/views/students/new.html.twig`) for the new `Student` form to the following:

```
{% block body %}
    <h1>Create new student</h1>
    {{ form(form) }}
{% endblock %}
```

That's it! No `<form>` element, no `<input>`s, no submit button, no labels! Even flash messages (relating to form validation errors) will be displayed by this function Twig function (global form errors at the top, and field specific errors by each form field).

The 'magic' happens in the controller method…

10.3 Updating `StudentController->newFormAction()`

First, our controller method will need to pass a Twig variable `form` to the `render()` method. This will be created for us by the `createView()` method of a Symfony form object. So `newAction()` will end as follows:

```
$argsArray = [
    'form' => $form->createView(),
];

$templateName = 'students/new';
return $this->render($templateName . '.html.twig', $argsArray);
```

Our method will use Symfony's FormBuilder to create the form for us, based on an instance of class `Student`. First we create a new, empty `Student` object, and then use Symfony's `createFormBuilder()` method to create a form based on the Entity class of our `$student` object:

```
public function newFormAction(Request $request)
{
    // create a new Student object
    $student = new Student();

    // create a form with 'firstName' and 'surname' text fields
```

```
        $form = $this->createFormBuilder($student)
            ->add('firstName', TextType::class)
            ->add('surname', TextType::class)
            ->add('save', SubmitType::class, array('label' => 'Create Student'))->getForm();
```

Note - for the above code to work we also need to add two `use` statements so that PHP knows about the classes `TextType` and `SubmitType`. These can be found in the form extension Symfony component:

```
    use Symfony\Component\Form\Extension\Core\Type\TextType;
    use Symfony\Component\Form\Extension\Core\Type\SubmitType;
```

We ask Symfony to 'handle' the request for us. If the HTTP request was a POST submission, then the submitted values will be used to populate our `$student` object. Otherwise, if GET method, the form will be an empty, new form.

```
    // if was POST submission, extract data and put into '$student'
    $form->handleRequest($request);
```

Forms have basic validation. The default for text entity properties is `NOT NULL`, so both name fields will be validated this way - both through HTML 5 validation and on the server side. If the form was submitted (via POST) and is valid, then we'll go ahead and create a new `Student` object as before:

```
    // if SUBMITTED & VALID - go ahead and create new object
    if ($form->isSubmitted() && $form->isValid()) {
        return $this->createAction($student);
    }
```

If not submitted (or not valid), then the logic falls through to displaying the form via Twig. The full listing for our improved `newAction()` method is as follows:

```
    /**
     * @Route("/student/new", name="student_new",  methods={"POST", "GET"})
     */
    public function newAction(Request $request)
    {
        // create a task and give it some dummy data for this example
        $student = new Student();

        // create a form with 'firstName' and 'surname' text fields
        $form = $this->createFormBuilder($student)
            ->add('firstName', TextType::class)
            ->add('surname', TextType::class)
            ->add('save', SubmitType::class, array('label' => 'Create Student'))->getForm();
        // if was POST submission, extract data and put into '$student'
        $form->handleRequest($request);

        // if SUBMITTED & VALID - go ahead and create new object
        if ($form->isSubmitted() && $form->isValid()) {
            return $this->createAction($student);
        }

        // render the form for the user
        $template = 'student/new.html.twig';
        $argsArray = [
            'form' => $form->createView(),
        ];

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

We can see that the method does the following:

1. creates a new (empty) `Student` records ‘$students
2. creates a new form builder, passing in `$student`, and stating that we want it to create a HTML form input element for the `name` field, and also a submit button (`SubmitType`) with the label `Create Student`. We chain these method calls in sequence, making use of the form builder’s ‘fluent’ interface, and store the created form object in PHP variable `$form`.
3. Finally, we create a Twig argument array, passing in the form object `$form` with Twig variable name `form`, and tell Twig to render the template `student/new.html.twig`.

Figure 10.1 shows a screenshot of the resulting form.

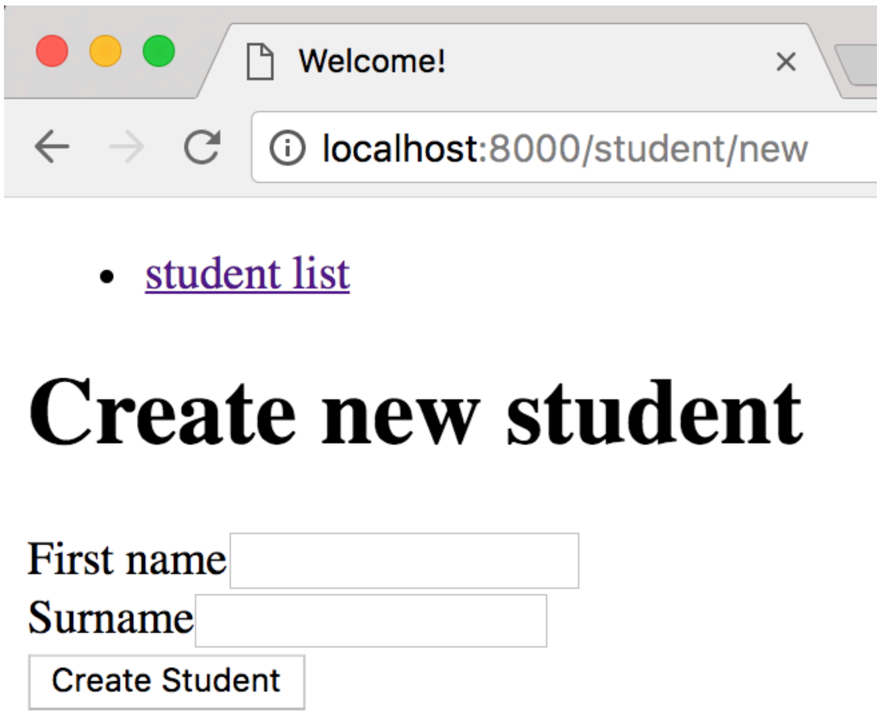
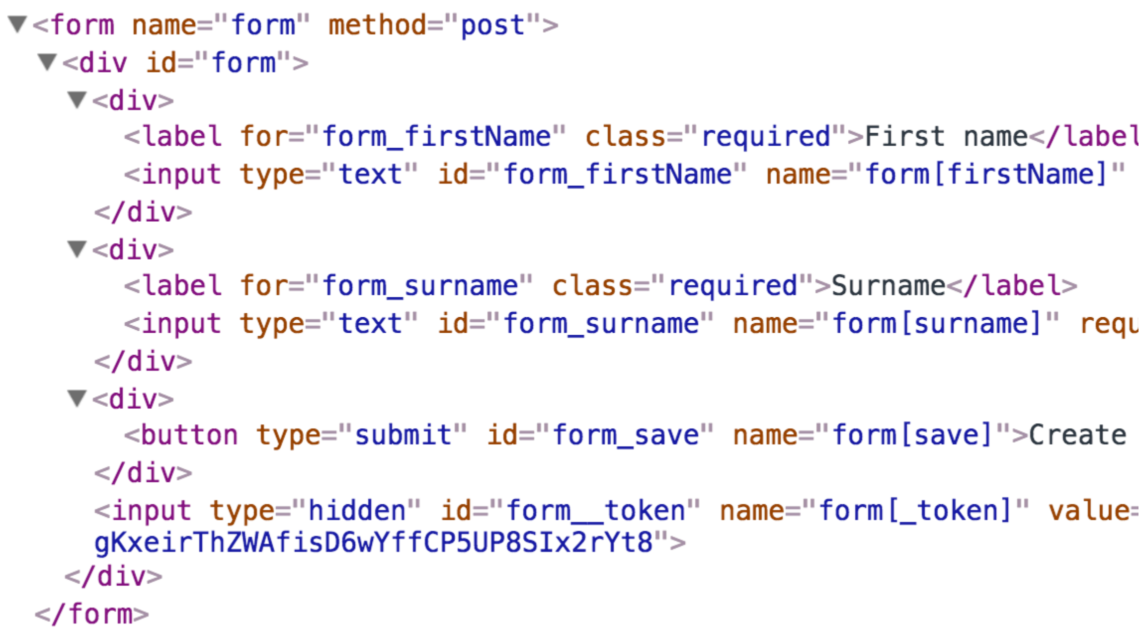


Figure 10.1: Symfony generated new student form.

10.4 Postback - form submits to same URL

If we look at the HTML in the source of our web page (see Figure 10.2), we can see that the form has no `action` attribute, which means that when POST submitted, it will be submitted to the same URL (i.e. a our `newAction()`). However, since we've already written our logic to process a **post-back** like this, then our code will work :-)



```
▼<form name="form" method="post">
  ▼<div id="form">
    ▼<div>
       <label for="form_firstName" class="required">First name</label
       <input type="text" id="form_firstName" name="form[firstName]"
     </div>
    ▼<div>
       <label for="form_surname" class="required">Surname</label>
       <input type="text" id="form_surname" name="form[surname]" requ
     </div>
    ▼<div>
       <button type="submit" id="form_save" name="form[save]">Create
     </div>
     <input type="hidden" id="form__token" name="form[_token]" value=
     gKxeirThZWAfisD6wYffCP5UP8SIx2rYt8">
   </div>
 </form>
```

Figure 10.2: HTML source of generated form

10.5 Using form classes (project `form05`)

Although simple forms can be created inside a controller method as above, it's good practice to create a separate from 'type' class to create each form.

Rather than write one from scratch, some of the work can be done for us using the **maker** bundle. To create class `/src/Form/StudentType.php` we can write CLI command:

```
$ php bin/console make:form Student
```

We should see something like this:

```
$ php bin/console make:form Student


 created: src/Form/StudentType.php
  Success!


 Next: Add fields to your form and start using it.
 Find the documentation at https://symfony.com/doc/current/forms.html
```

If we look inside `/src/Forn/StudentType.php` we see a skeleton class as follows:

"'php namespace App;

```
use App\Entity\Student;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class StudentType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('field_name')
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        ...
    }
}
```
```` ``` ````

We need to:

- add `uses` statements for field types we want to use

`php use Symfony\Component\Form\Extension\Core\Type\TextType; use Symfony\Component\Form\Extension\Core\Type\SubmitType;`

- write statements to add our `firstName`, `surname` and submit buttons to the form:

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('firstName', TextType::class)
        ->add('surname', TextType::class)
        ->add('save', SubmitType::class, array('label' => 'Create Student'))
    ;

}
```

That's our `StudentType` form class complete. We now can have a much simpler controller method, which can create the form in a single statement:

```
$form = $this->createForm(StudentType::class, $student);
```

So our refactored `newAction()` methods looks as follows:

```
/**
 * @Route("/student/new", name="student_new",  methods={"POST", "GET"})
 */
```
````

```
public function newAction(Request $request)
{
    // create a task and give it some dummy data for this example
    $student = new Student();

    // create form from form class StudentType
    $form = $this->createForm(StudentType::class, $student);

    // if was POST submission, extract data and put into '$student'
    $form->handleRequest($request);

    // if SUBMITTED & VALID - go ahead and create new object
    if ($form->isSubmitted() && $form->isValid()) {
        return $this->createAction($student);
    }

    // render the form for the user
    $template = 'student/new.html.twig';
    $argsArray = [
        'form' => $form->createView(),
    ];

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

10.6 Video tutorials about Symfony forms

Here are some video resources on this topic:

- Code Review form validation with `@Assert`

11 Customising the display of generated forms

11.1 First let's Bootstrap this project (project `form06`)

The DIY way to add Bootstrap to a project is as follows:

1. copy any custom CSS and fonts into your `/public` directory (see example code for `/fonts` and `/css` which we modified from a great KNPUniversity free tutorial)

2. tell Symfony to generate forms using the Bootstrap theme - by adding `form_themes: ['bootstrap_4_layout.html.twig']` to `/config/packages/twig.yml` file as follows:

```
twig:
    paths: ['%kernel.project_dir%/templates']
    debug: '%kernel.debug%'
    strict_variables: '%kernel.debug%'
    form_themes: ['bootstrap_4_layout.html.twig']
```

3. Change `/base.html.twig` to read the required Javascript, and also wrap Bootstrap div's around parts of our page. Here is an updated `base.html.twig` (get this from sample codes `/teamplates/base.html.twig`:

   - the beginning of `base.html.twig` is the same:

```
<!doctype html>
<html lang="en">

<head>
```

```
    <title>{% block title %}Welcome to the SpaceBar{% endblock %}</title>
    <meta charset="utf-8">
```

- then we need to declare some stylesheets:

```
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    {% block stylesheets %}
        <link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/css/bootstrap.min.css"
    integrity="sha384-PsH8R72JQ3SOdhVi3uxftmaW6Vc51MKb0q5P2rRUpPvrszuE4W1povHYgTpBfshb"
        <link rel="stylesheet" href="/css/font-awesome.css">
        <link rel="stylesheet" href="/css/styles.css">
    {% endblock %}
</head>
```

- we use Bootstrap divs for navigation in `<body>` (links for student list and new student form):

```
<body>
<nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
  <a style="margin-left: 75px;" class="navbar-brand space-brand" href="#">
        My Great Website !
    </a>
    <button class="navbar-toggler" type="button" data-toggle="collapse"
    data-target="#navbarNavDropdown" aria-controls="navbarNavDropdown"
    aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse" id="navbarNavDropdown">
        <ul class="navbar-nav ml-auto">
            <li class="nav-item">
         <a style="color: #fff;" class="nav-link" href="{{ url('student_list') }}">
                student list
                </a>
            </li>
            <li class="nav-item">
         <a style="color: #fff;" class="nav-link" href="{{ url('student_new') }}">
                Create NEW student
                </a>
            </li>
        </ul>
    </div>
</nav>
```

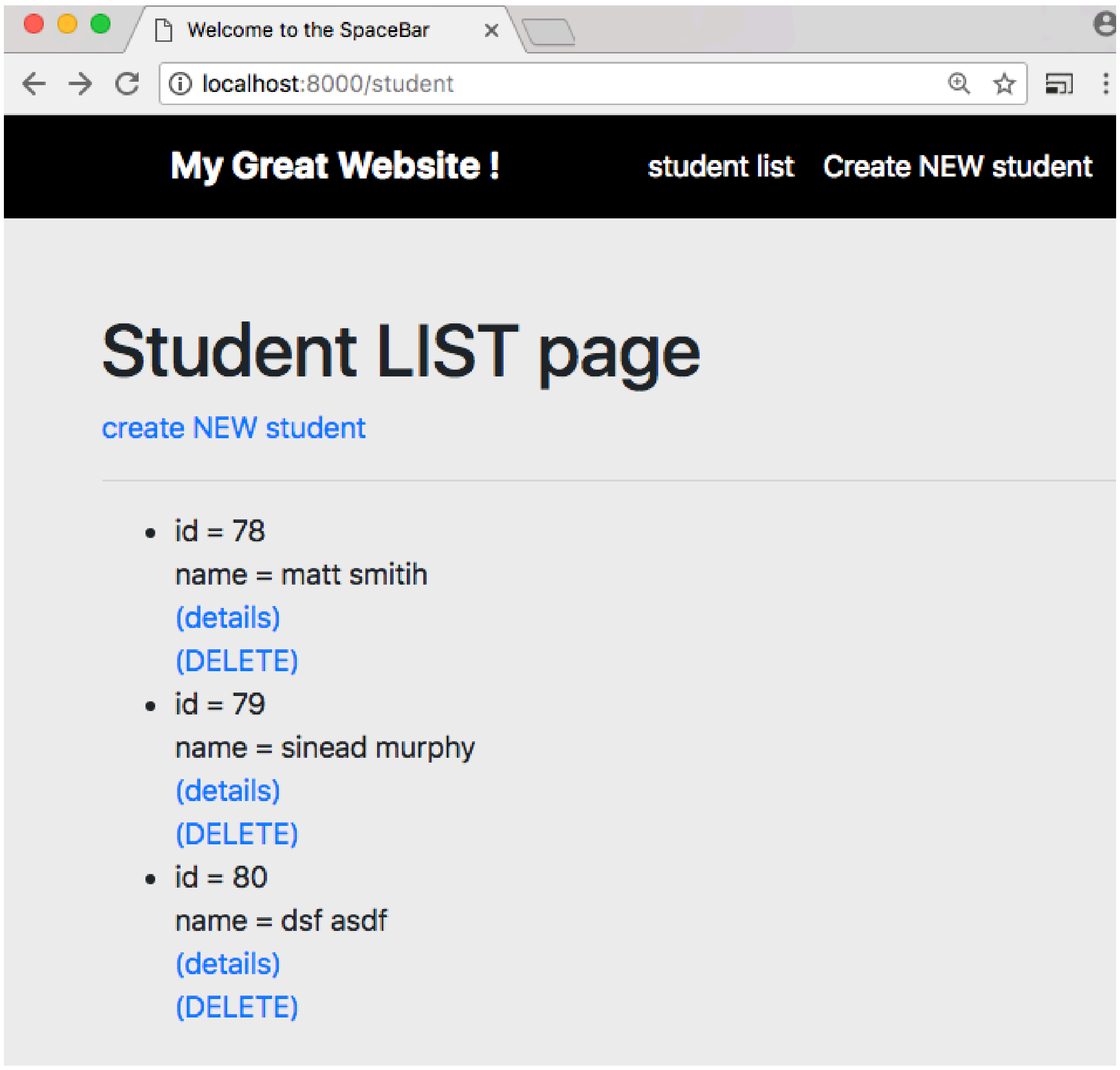
- we wrap Bootstrap container and row divs around our `body` Twig block:

```
<div class="container">
    <div class="row">
        <div class="col-sm-12">
{% block body %}
{% endblock %}
        </div>
    </div>
</div>
```

- finally we need to add some Javascrpts to be executed after the rest of the page has been loaded:

```
{% block javascripts %}
    <script src="https://code.jquery.com/jquery-3.2.1.min.js"
  integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4=" crossorigin="anonymous"></scr
  <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.3/umd/popper.min.js"
  integrity="sha384-vFJXuSJphROIrBnz7yo7oB41mKfc8JzQZiCq4NCceLEaO4IHwicKwpJf9c9IpFgh" crossorig
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/js/bootstrap.min.js"
  integrity="sha384-alpBpkh1PFOepccYVYDB4do5UnbKysX5WZXm3XxPqe5iKTfUKjNkCk9SaVuEZflJ" crossorig
    <script>
        $('.dropdown-toggle').dropdown();
    </script>
{% endblock %}
</body>
</html>
```

That's it (although you may need to clear your cache). See Figure 11.1 shows a screenshot of the rnew home page.



Student LIST page

create NEW student

- id = 78
  name = matt smitih
  (details)
  (DELETE)
- id = 79
  name = sinead murphy
  (details)
  (DELETE)
- id = 80
  name = dsf asdf
  (details)
  (DELETE)

Figure 11.1: Website with Bootstrap theme.

11.2 Understanding the 3 parts of a form (project `form07`)

In a controller we create a `$form` object, and pass this as a Twig variable to the template `form`. Twig renders the form in 3 parts:

- the opening `<form>` tag
- the sequence of form fields (with labels, errors and input elements)
- the closing `</form>` tag

This can all be done in one go (using Symfony/Twig defaults) with the Twig `form()` function, or we can use Twigs 3 form functions for rendering (displaying) each part of a form, these are:

- `form_start()`
- `form_widget()`
- `form_end()`

So we could write the `body` block of our Twig template (`/app/Resources/views/students/new.html.twig`) for the new `Student` form to the following:

```
{% block body %}
    <h1>Create new student</h1>
    {{ form_start(form) }}
    {{ form_widget(form) }}
    {{ form_end(form) }}
{% endblock %}
```

Although since we're not adding anything between these 3 Twig functions' output, the result will be the same form as before.

11.3 Using a Twig form-theme template

Symfony provides several useful Twig templates for common form layouts.

These include:

- wrapping each form field in a `<div>`
  - form_div_layout.html.twig
- put form inside a table, and each field inside a table row `<tr>` element
  - form_table_layout.html.twig
- Boostrap CSS framework div's and CSS classes
  - bootstrap_4_layout.html.twig

For example, to use the `div` layout we can declare this template be used for all forms in the `/config/packages/twig.yml` file as follows:

```
twig:
    paths: ['%kernel.project_dir%/templates']
    debug: '%kernel.debug%'
    strict_variables: '%kernel.debug%'
    form_themes: ['bootstrap_4_layout.html.twig']
```

11.4 DIY (Do-It-Yourself) form display customisations

Each form field can be rendered all in one go in the following way:

```
{{ form_row(form.<FIELD_NAME>) }}
```

For example, if the form has a field `name`:

```
{{ form_row(form.name) }}
```

So we could display our new student form this way:

```
{% block body %}
    <h1>Create new student</h1>
    {{ form_start(form) }}

    {{ form_row(form.firstName) }}
    {{ form_row(form.surname) }}
    {{ form_row(form.save) }}

    {{ form_end(form) }}
{% endblock %}
```

11.5 Customising display of parts of each form field

Alternatively, each form field can have its 3 constituent parts rendered separately:

- label (the text label seen by the user)
- errors (any validation error messages)
- widget (the form input element itself)

For example:

```
<div>
    {{ form_label(form.name) }}

    <div class="errors">
    {{ form_errors(form.name) }}
    </div>

    {{ form_widget(form.name) }}
```

```
</div>
```

So we could display our new student form this way:

```
{% block body %}
    <h1>Create new student</h1>
    {{ form_start(form) }}

<div>
    <div class="errors">
    {{ form_errors(form.name) }}
    </div>

    {{ form_label(form.name) }}

    {{ form_widget(form.name) }}
</div>

<div>
    {{ form_row(form.save) }}
</div>

    {{ form_end(form) }}
{% endblock %}
```

The above would output the following HTML (if the errors list was empty):

```html

<div>
    <div class="errors">

    </div>

    <label for="form_name" class="required">Name</label>

    <input type="text" id="form_name" name="form[name]" required="required" />
</div>
```

NOTE: If we have the Bootstrap template, we need to use appropriate classes for our DIVs to get that nice form layout …

Learn more at:

- The Symfony form customisation page

11.6 Specifying a form's method and action

While Symfony forms default to `POST` submission and a postback to the same URL, it is possible to specify the method and action of a form created with Symfony's form builder. For example:

```
$formBuilder = $formFactory->createBuilder(FormType::class, null, array(
    'action' => '/search',
    'method' => 'GET',
));
```

Learn more at:

- Introduction to the Form component

Part IV

Symfony code generation

12
CRUD controller and templates generation

12.1 Symfony's CRUD generator

Symfony 3 used to offer a very powerful CRUD generator command. However, it isn't (yet) available as part of the core (Sensio Labs) published Symfony bundles.

Open Source Developers to the rescue. There has been some ongoing discussion about providing a CRUD generator recipe for Symfony 4:

- https://github.com/symfony/maker-bundle/issues/3

Vladimir Sadicovhas created a Symfony `make:crud` recipe, that can be installed via Composer.

- https://github.com/sadikoff/crud-maker-bundle

Which (at the time of writing - Feb 2018) is currently an open **Pull Request** to the Symfony **Maker Bundle**:

- https://github.com/symfony/maker-bundle/pull/113

(note, Ryan Weaver, from KnpLabs - main author of the scripts and codes for KnpUniversity, is an active Symfony developer - the PHP/Symfony world is a small one …)

It did have an issue with its `newAction()` - but, thanks to Git and pull-requests, I managed to published a fix and have it accepted within a few hours. So, if Vladimir's code is accepted as a pull request to the Symfony codebase, I'll then be a contributor to Symfony as well :-)

So what you need to do to work with the draft (unofficial) CRUD generator is:

```
$ composer req security-csrf form validator
$ composer req --dev make koff/crud-maker-bundle
```

You can now create Symfony CRUD for a given entity as follows (in this example the `Category` entity is used):

```
$ php bin/console make:crud Category
```

With the single command above Symfony will generate a CRUD controller (`CategoryController`) and also create a directory containing Twig templates (`/templates/category/index.html.twig` etc.). The list of new files is:

```
/src/Controller/CategoryController.php

/src/Form/CategoryType.php

/templates/category/_form.html.twig
/templates/category/_delete_form.html.twig
/templates/category/edit.html.twig
/templates/category/index.html.twig
/templates/category/new.html.twig
/templates/category/show.html.twig
```

The list of new (annotation-defined routes):

```
/category --> CategoryController->indedxAction()
/category/new --> CategoryController->newAction()
/category/show/{id} --> CategoryController->showAction(Category $category)
/category/delete/{id} --> CategoryController->deleteAction(Category $category)
```

12.2 The generated routes

Let's see the new routes generated magically for us:

```
 -------------------------- ---------- -------- ------ -------------------------------------
  Name                       Method     Scheme   Host   Path
 -------------------------- ---------- -------- ------ -------------------------------------
  category_index             ANY        ANY      ANY    /category/
  category_new               GET|POST   ANY      ANY    /category/new
  category_show              GET        ANY      ANY    /category/{id}
  category_edit              GET|POST   ANY      ANY    /category/{id}/edit
  category_delete            DELETE     ANY      ANY    /category/{id}
  ...
```

NOTE: The **sequence** of these routes is important (this was the error I fixed for this project). a

`GET` requrest with a URL looking like this: `/category/1` should be matched to the show action, i.e. `/category/{id = 1}`. But a URL like this `/category/new` we want to match with the action. If the show route attempts to match **before** the new route, then `/category/new` is matched to the show route as `/category/{id = new}`, which will then throw a 404 error, since `new` is not a valid `id` for a database `Category` object.

Once solution is to ensure the new action method appears **before** the show action method in our controller class. If you don't like this solution (I'm not sure myself), then another solution is to design URL routes that cannot get mixed up like this - but that means adding **verbs** for every every route. E.g. our category routes could be defined as follows:

```
 -------------------------- ---------- -------- ------ ---------------------------------------
  Name                       Method     Scheme   Host   Path
 -------------------------- ---------- -------- ------ ---------------------------------------
  category_index             ANY        ANY      ANY    /category/list
  category_new               GET|POST   ANY      ANY    /category/new
  category_show              GET        ANY      ANY    /category/show/{id}
  category_edit              GET|POST   ANY      ANY    /category/edit/{id}
  category_delete            DELETE     ANY      ANY    /category/delete/{id}
  ...
```

So, for each request, the entity `category` is followed by its action verb (`list`, `show`, `new` etc.), and then finally, if required an `{id}` parameter. This approach has the advantage of being simple and unambiguous (the sequence of methods in our controller class no longer matters), But (a) it breaks with common conventions in routes in Symfony projects, and (b) it means the URLs are getting longer, and simple, short URLs are one aim (benefit!) of a well designed web framework.

But for **your** personal projects, choose a route pattern scheme that **you** prefer, so this **verb** approach might be something you are happier with. It would also mean you could use `GET` metod for **delete** requests, rather than emulating a `DELETE` HTTP request …

12.3 The generated CRUD controller

A controller class was geneated for Category objects in `/src/Controller/CategoryController.php`. Let's first look at the namespaces and class declaration line:

```
namespace App\Controller;

use App\Entity\Category;
use App\Form\CategoryType;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

/**
 * @Route("/category", name="category_")
 */
class CategoryController extends Controller
```

Above we see a set of `use` statements, and then an interesting class comment. The `@Route` annotation comment declares a route 'prefix' which will at the beginning of any `@Route` annotations for individual controller methods. So, for example, the new action will have the route `/category/new`.

Notice also that there is a **prefix** defined for the internal names of routes, of `category_`. This means that for each route method in this class, the internal route name will begin with `category_`, for example the new route will be `category_new` and so on.

If we look in directory `/templates/category/` we'll see the following generated main templates:

```
edit.html.twig
index.html.twig
new.html.twig
show.html.twig
```

and the 2 partial templates (that are included in other pages):

```
_form.html.twig
_delete_form.html.twig
```

Note that all these generated templates extend Twig class `base.html.twig`.

12.4 The generated index (a.k.a. list) controller method

Below we can see the code for `indexAction()` that retrieves and then passes an array of `Category` objects to template 'category/index.html.twig.

```
/**
 * @Route("/", name="index")
 *
 * @return Response
 */
public function index()
{
    $categories = $this->getDoctrine()
        ->getRepository(Category::class)
        ->findAll();
    return $this->render('category/index.html.twig', ['categories' => $categories]);
}
```

If you prefer, you can re-write the last statement in the more familiar form:

```
    $argsArray = [
        'categories' => $categories,
    ];

    $template = 'category/index.html.twig';
    return $this->render($template, $argsArray);
```

Twig template `category/index.html.twig` loops through array `categories`, wrapping HTML table row tags around each entity's content:

```
    {% for category in categories %}
        <tr>
            <td>{{ category.id }}</td>
            <td>{{ category.name }}</td>
                            <td>
                <a href="{{ path('category_show', {'id':category.id}) }}">show</a>
                <a href="{{ path('category_edit', {'id':category.id}) }}">edit</a>
            </td>
        </tr>
    {% else %}
        <tr>
            <td colspan="3">no records found</td>
        </tr>
    {% endfor %}
```

Let's create a CSS file for table borders and padding in a new file `/public/css/table.css`;

```
    table, tr, td {
        border: 0.1rem solid black;
        padding: 0.5rem;
    }
```

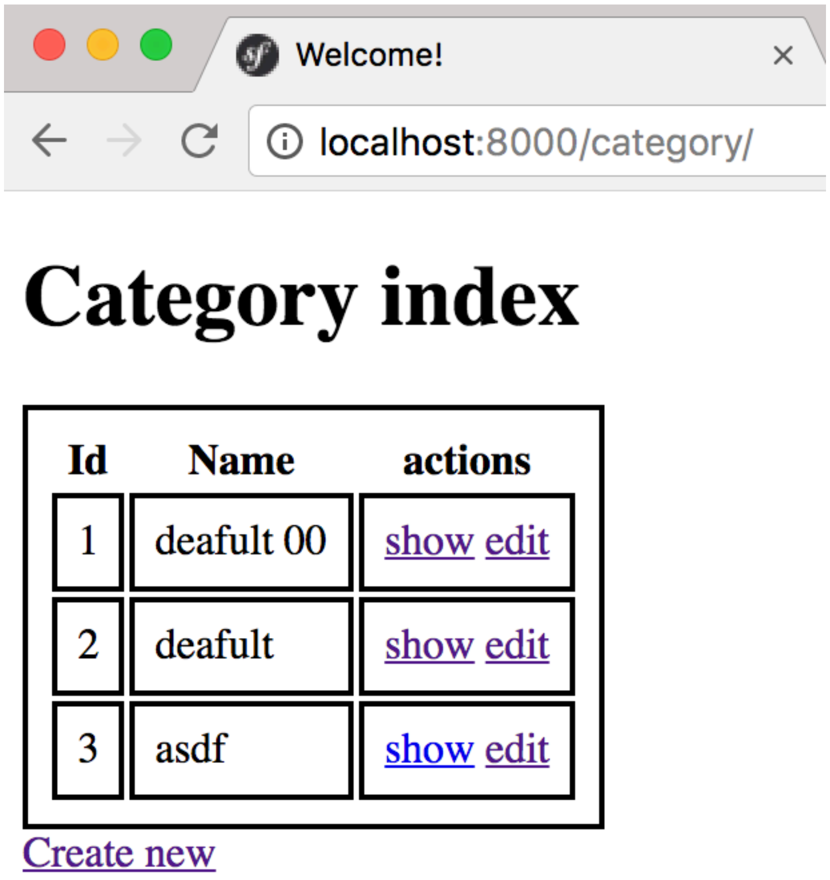
Remember in `/templates/base.html.twig` there is a block for style sheets:

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

So now we can edit template `category/index.html.twig` to add a stylesheet block import of this CSS stylesheet:

```
{% block stylesheets %}
    <style>
        @import '/css/table.css';
    <style>
{% endblock %}
```

Figure 12.1 shows a screenshot of how our list of categories looks,rendered by the `categories/index.html.twig` template.



Category index

Id	Name	actions
1	deafult 00	show edit
2	deafult	show edit
3	asdf	show edit

Create new

Figure 12.1: List of categories in HTML table.

12.5 The generated `newAction()` method

The method and Twig template for a new `Category` work just as you might expect. For `GET` requests (and invalid `POST` submissions) a form will be displayed. Upon valid `POST` submission the

`$category` object populated swith the form data will be persisted to the database, and then the user will be redirected to the `edit` action form for the newly created entity.

```
/**
* @Route("/new", name="new")
* @Method({"GET", "POST"})
*
* @param Request $request
*
* @return Response
*/
public function new(Request $request)
{
    $category = new Category();
    $form = $this->createForm(CategoryType::class, $category);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $em = $this->getDoctrine()->getManager();
        $em->persist($category);
        $em->flush();

        return $this->redirectToRoute('category_edit', ['id' => $category->getId()]);
    }

    return $this->render('category/new.html.twig', [
        'category' => $category,
        'form' => $form->createView(),
    ]);
}
```

Note that it redirects to the edit method (`category_edit`) after a successful object creation and saving to the database. You could change this to redirect to the show route if you wished, by writing:

```
return $this->redirectToRoute('category_show', ['id' => $category->getId()]);
```

12.6 The generated `showAction()` method

Initially, the generated 'show' method looks just as we might write ourselves:

```
/**
 * @Route("/{id}", name="show")
```

```
     * @Method("GET")
     *
     * @param Category $category The Category entity
     *
     * @return Response
     */
    public function show(Category $category)
    {
        $deleteForm = $this->createDeleteForm($category);

        return $this->render('category/show.html.twig', [
            'category' => $category,
            'delete_form' => $deleteForm->createView(),
        ]);
    }
```

But looking closely, we see that while the route specifies parameter `{id}`, the method declaration species a parameter of `Category $category`. Also the code in the method makes no reference to the `Category` entity repository. So by some **magic** the numeric 'id' in the request path has used to retrieve the corresponding `Category` record from the database!

This magic is the work of the Symfony 'param converter'. Also, of course, if there is no record found in table `category` that corresponds to the received 'id', then a 404 not-found-exception will be thrown.

Learn more about the 'param converter' at the Symfony documentation pages:

-

12.7 The generated `editAction()` and `deleteAction()` methods

The 'edit' and 'delete' generated methods are as you might expect. The edit method creates a form, and also include code to process valid submission of the edited entity.

Note that it redirects to itself upon successful save of edits. You could change this to redirect to the show route as described above for the new action.

```
    /**
     * @Route("/{id}/edit", name="edit")
     * @Method({"GET", "POST"})
     *
     * @param Request $request
     * @param Category  $category  The Category entity
     *
     * @return Response
     */
    public function edit(Request $request, Category $category)
    {
        $form = $this->createForm(CategoryType::class, $category);
        $form->handleRequest($request);

        $deleteForm = $this->createDeleteForm($category);

        if ($form->isSubmitted() && $form->isValid()) {
            $this->getDoctrine()->getManager()->flush();

            return $this->redirectToRoute('category_edit', ['id' => $category->getId()]);
        }

        return $this->render('category/edit.html.twig', [
            'category' => $category,
            'form' => $form->createView(),
            'delete_form' => $deleteForm->createView(),
        ]);
    }
```

The ‘delete’ method deletes the entity and redirects back to the list of categories for the ‘index’ action. Notice that an annotation comment states that this controller method is in response to `DELETE` method requests (more about this below).

```
    /**
     * @Route("/{id}", name="delete")
     * @Method("DELETE")
     *
     * @param Request $request
     * @param Category  $category  The Category entity
     *
     * @return Response
     */
    public function delete(Request $request, Category $category)
    {
        $form = $this->createDeleteForm($category);
        $form->handleRequest($request);
        if ($form->isSubmitted() && $form->isValid()) {
            $em = $this->getDoctrine()->getManager();
            $em->remove($category);
            $em->flush();
        }

        return $this->redirectToRoute('category_index');
    }
```

12.8 The generated method `createDeleteForm()`

To avoid the delete method becoming too long and complicated, a separate method `createDeleteForm()` was generated that creates and returns a Symfony form-builder form with a 'DELETE' button simulating an HTTP DELETE method.

```
    /**
     * Creates a form to delete a Category entity.
     *
     * @param Category $category The Category entity
     *
     * @return \Symfony\Component\Form\FormInterface The form
     */
    private function createDeleteForm(Category $category)
    {
        return $this->createFormBuilder()
            ->setAction($this->generateUrl('category_delete', ['id' => $category->getId()]))
            ->setMethod('DELETE')
            ->getForm()
        ;
    }
```

If we actually look at the HTML source of this button-form, we can see that it is actually submitted with the HTTP `post` action, along with a hidden form field named `_method` with the value `DELETE`. This kind of approach means we can write our controllers as if they are responding to the full range of HTTP methods (`GET`, `POST`, `PUT`, `DELETE` and perhaps `PATCH`).

```
    <form name="form" method="post" action="/category/3">
        <input type="hidden" name="_method" value="DELETE" />
        <input type="submit" value="Delete">
    </form>
```

Part V

Sessions

13 Introduction to Symfony sessions

13.1 Create a new project from scratch (project `sessions01`)

Let's start with a brand new project to learn about Symfony sessions:

```
$ composer create-project symfony/skeleton session01
```

Let's add to our project the Twig and annotations packages[1]:

```
$ composer req --dev server make
$ composer req twig annotations
```

13.2 Default controller - hello world

Create a new Default controller that renders a Twig template to say `Hello World` to us.

So the controller should look as this (you can speed things up using `make` and then editing the created file):

```
namespace App\Controller;

use Symfony\Component\Routing\Annotation\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
```

[1]And add the server package to `--dev` if that's how you are testing locally.

```
class DefaultController extends Controller
{
    /**
     * @Route("/", name="default")
     */
    public function indexAction()
    {
        $template = 'default/index.html.twig';
        $args = [];
        return $this->render($template, $args);
    }
}
```

Our home page default template `default/index.html.twig` can be this simple[2]:

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

{% block body %}
<p>
    Hello World
</p>
{% endblock %}
```

13.3 Remembering foreground/background colours in the session (`sessions02`)

Let's start out Symfony sessions learning with the ability to store (and remember) foreground and background colours[3]. First let's add some HTML in our `default/index.html.twig` page to display the value of our 2 stored values.

We will assume we have 2 Twig variables:

- `colours` - an associative array in the form:

  ```
  colours = [
      'foreground' => 'black',
      'background' => 'white'
  ]
  ```

- `default_colours` - a Boolean (true/false) value, telling us whether or not our colours came from the session, or are defaults due to no array being found in the session

[2]If we have a suitable HTML skeleton base template.

[3]I'm not going to get into a colo(u)rs naming discussion. But you may prefer to just always use US-English spelling (*sans* 'u') since most computer language functions and variables are spelt the US-English way

So our controller needs to create these variables and pass them to Twig:

```
public function index()
{
    $colours = [
        'foreground' => 'black',
        'background' => 'white'
    ];
    $default_colours = true;

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

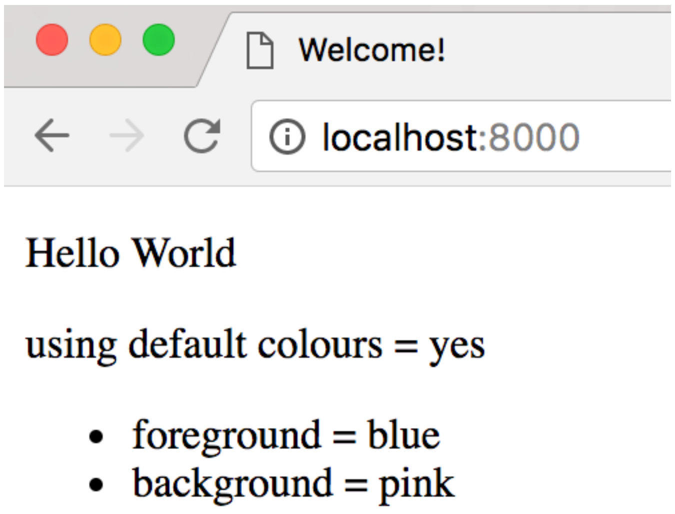
here is the Twig HTML to output the values of these variables:

```
    using default colours =
    {% if default_colours %}
        yes
    {% else %}
        no
    {% endif %}
<ul>
    {% for property, colour in colours %}
        <li>
            {{ property }} = {{ colour }}
        </li>
    {% endfor %}
</ul>
```

Note that Twig offers a key-value array loop just like PHP, in the form:

```
{% for <key>, <value> in <array> %}
```

Figure 13.1 shows a screenshot of our home page listing these Twig variables.



Hello World

using default colours = yes

- foreground = blue
- background = pink

Figure 13.1: Screenshot of home page listing Twig colour variables.

13.4 Twig function to retrieve values from session (project `session03`)

Twig offers a function to attempt to retrieve a named value in the session:

```
app.session.get('<attribute_key>')
```

If fact the `app` Twig variable allows us to read lots about the Symfony, including:L

- request (`app.request`)
- user (`app.user`)
- session (`app.session`)
- environment (`app.environment`)
- debug mode (`app.debug`)

Read more about Twig `app` in the Symfony documentation pages:

- https://symfony.com/doc/current/templating/app_variable.html

13.5 Attempt to read `colors` array property from the session

We can store values in Twig variables using the `set <var> = <expression>` statement. So let's try to read an array of colours from the session named `colors`, and store in a local Twig variable names `colors`:

```
{% set colours = app.session.get('colours') %}
```

After this statement, `colors` either contains the array retrieved from the session, or it is `null` if no such variable was found in the session.

So we can test for `null`, and if found we can set `colors` to our default values:

```
{% set colours = app.session.get('colours') %}

{% if colours is null %}
    {% set default_colours = true %}

    {% set colours = {
        'foreground': 'black',
        'background': 'white'
       }
    %}
{% endif %}
```

So at the point we know `colours` contains an array, either from the session or our default values.

13.6 Storing whether colours from session or not

If we wish to know whether the values inside `colours` are from the session or not, we can do so with a second (flag) variable. E.g.

- set flag to false (assume we will find values in the session)
- attempt to read values from session
- THEN null (no values found in session) THEN
  - set `colours` to values from session
  - set flag to true

With this logic, our flag is true/false based on whether or not we used default values due to finding nothing in the session.

Here it is written in Twig:

```
{# ------ assume using values from session ---- #}
{% set default_colours = false %}

{# ------ attempt to read 'colours' from session ----- #}
{% set colours = app.session.get('colours') %}

{# ------ if 'null' then no found in session ----- #}
{% if colours is null %}
```

```
        {# ------ set default flag to TRUE ----- #}
        {% set default_colours = true %}

        {# ------ set our default colours array ----- #}
        {% set colours = {
            'foreground': 'black',
            'background': 'white'
          }
        %}
    {% endif %}
```

We can now **remove** from our controller the setting of any Twig variables - since all the work is done in the Twig template. So our Default homepage controller method reverts to an empty argument array for Twig:

```
    public function indexAction()
    {
        $template = 'default/index.html.twig';
        $args = [
        ];
        return $this->render($template, $args);
    }
```

The full listing for our Twig tempalte `default/index.html.twig` looks as follows: first part logic testing session, second part outputting details about the variables:

```
    {# ------ assume using values from session ---- #}
    {% set default_colours = false %}

    {# ------ attempt to read 'colors' from session ----- #}
    {% set colours = app.session.get('colours') %}

    {# ------ if 'null' then no found in session ----- #}
    {% if colours is null %}
        {# ------ set default flag to TRUE ----- #}
        {% set default_colours = true %}

        {# ------ set our default colours array ----- #}
        {% set colours = {
            'foreground': 'black',
            'background': 'white'
        }
        %}
```

```
{% endif %}

<p>Hello World</p>
using default colours =
{% if default_colours %}
    yes
{% else %}
    no
{% endif %}
<ul>
    {% for property, colour in colours %}
        <li>
            {{ property }} = {{ colour }}
        </li>
    {% endfor %}
</ul>
```

13.7 Working with sessions in Symfony Controller methods (project session04)

All we need to write to work with the current session object in a Symfony controller method is the following statement:

```
$session = new Session();
```

Note, you also need to add the following `use` statement for the class using this code:

```
use Symfony\Component\HttpFoundation\Session\Session;
```

Note - do **not** use any of the standard PHP command for working with sessions. Do all your Symfony work through the Symfony session API. So, for example, do not use either of these PHP functions:

```
session_start(); // ----- do NOT use this in Symfony -------
session_destroy(); // ----- do NOT use this in Symfony -------
```

You can now set/get values in the session by making reference to `$session`.

Note: You may wish to read about **how to start a session in Symfony**[4].

[4]While a session will be started automatically if a session action takes places (if no session was already started), the Symfony documentation recommends your code starts a session if one is required. Here is the code to do so: `$session->start()`, but to be honest it's simpler to rely on Symfony to decide when to start a new session, since sometimes integrating this into your controller logic can be tricky (especially with controller redirects). You'll get errors if you try to start an already started session ...

13.8 Symfony's 2 session 'bags'

We've already met sessions - the Symfony 'flash bag', which stores messages in the session for one request cycle.

Symfony also offers a second kind of session storage, session 'attribute bags', which store values for longer, and offer a namespacing approach to accessing values in session arrays.

We store values in the attribute bag as follows using the `session->set()` method:

```
$session->set('<key>', <value>);
```

Here's how we store our colours array in the Symfony application session from our controllers:

```
// create colours array
$colours = [
    'foreground' => 'blue',
    'background' => 'pink'
];

// store colours in session 'colours'
$session = new Session();
$session->set('colours', $colours);
```

Note - also learn how to 'unset' values when you learn to set them. We can clear everything in a session by writing:

```
$session = new Session();
$session->clear();
```

13.9 Storing values in the session in a controller action

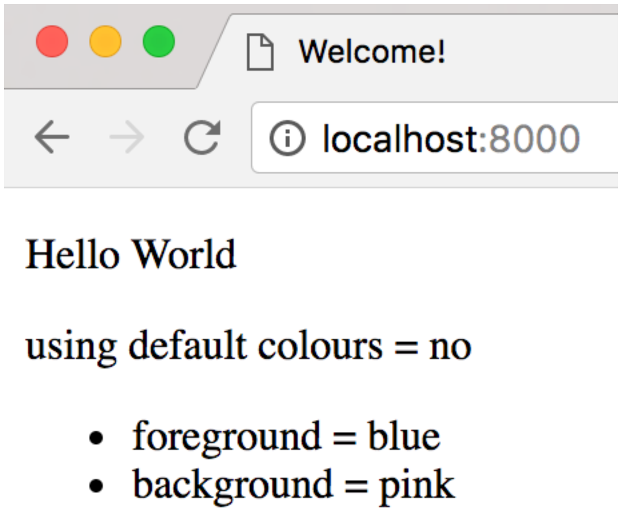
Let's create a new route `/pinkblue` with a new `DefaultController` method `pinkBlueAction()`, which will store colours in the session and then re-direct to the home page:

```
/**
 * @Route("/pinkblue", name="pinkblue")
 */
public function pinkBlueAction()
{
    // create colours array
    $colours = [
        'foreground' => 'blue',
        'background' => 'pink'
    ];
```

```
        // store colours in session 'colours'
        $session = new Session();
        $session->set('colours', $colours);

        return $this->redirectToRoute('default');
    }
```

Figure 13.1 shows a screenshot of our home page listing after visiting `/pinkblue`.



Hello World

using default colours = no

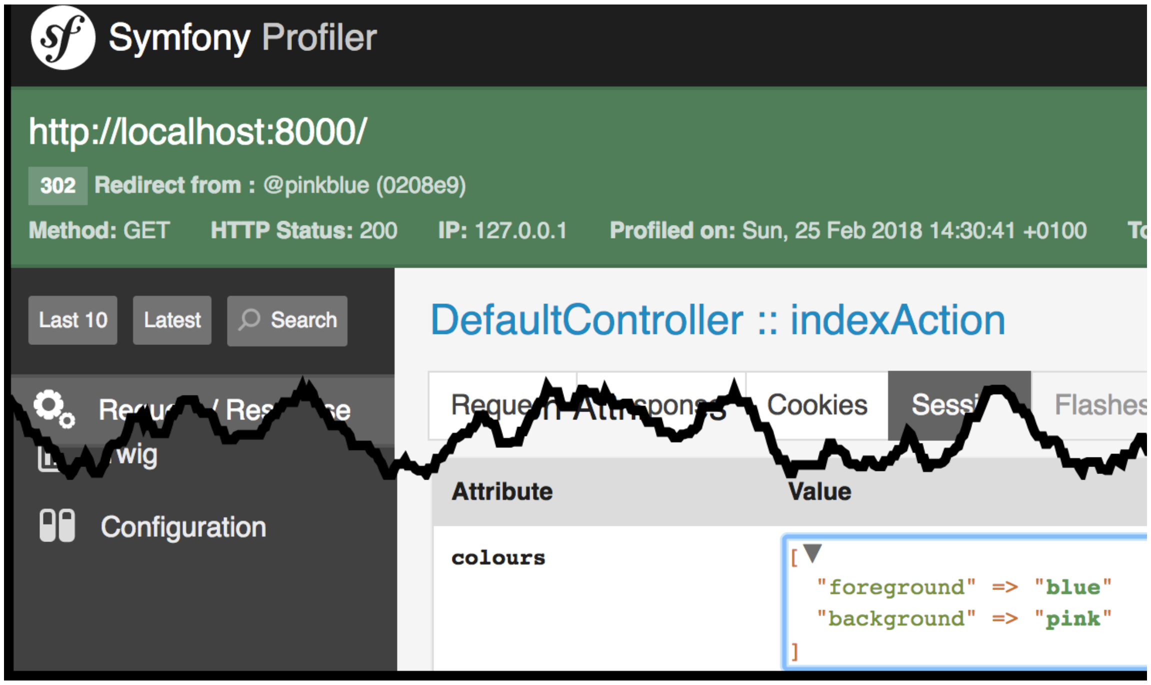
- foreground = blue
- background = pink

Figure 13.2: Screenshot of home page showing values from session.

If you add the Symfony Profiler (`composer req --dev profiler`) you can view session values in its session tab,, as show in Figure 13.3.

Learn more at about Symfony sessions at:

- Symfony and sessions


Symfony Profiler
http://localhost:8000/
302 Redirect from : @pinkblue (0208e9)
Method: GET HTTP Status: 200 IP: 127.0.0.1 Profiled on: Sun, 25 Feb 2018 14:30:41 +0100
Last 10
Latest
Search
Configuration
DefaultController :: indexAction
Cookies
Flashes
Attribute
Value

```
colours   [
            "foreground" => "blue"
            "background" => "pink"
          ]
```

Figure 13.3: Homepage with session colours applied via CSS.

13.10 Applying colours in HTML head `<style>` element (project `session06`)

Since we have an array of colours, let's finish this task logically by moving our code into `base.html.twig` and creating some CSS to actually set the foreground and background colours using these values.

So we remove the Twig code from template `index.html.twig` and paste it, slighly edited, into `base.html.twig` as follows.

Add the following **before** we start the HTML doctype etc.

```
{% set colours = app.session.get('colours') %}

{# default = blue #}
{% if colours is null %}
    {% set colours = {
        'foreground': 'black',
        'background': 'while'
      }
    %}
{% endif %}
```

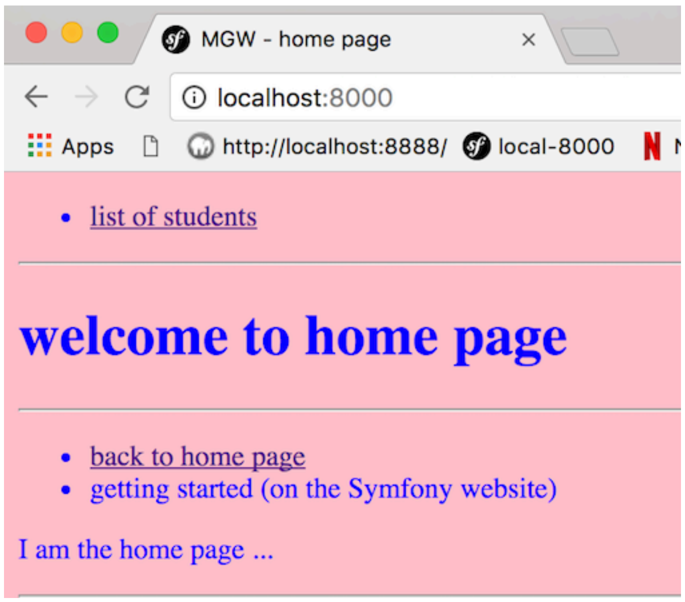
So now we know we have our Twig variable `colours` assigned values (either from the session, or from the defaults. Now we can update the `<head>` of our HTML to include a new `body {}` CSS rule, that pastes in the values of our Twig array `colours['foreground']` and `colours['background']`:

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

    <style>
        @import '/css/flash.css';
        {% block stylesheets %}
        {% endblock %}

        body {
            color: {{ colours['foreground'] }};
            background-color: {{ colours['background'] }};
        }
    </style>
</head>
```

Figure 13.4 shows our text and background colours applied to the CSS of the website homepage.



- list of students

welcome to home page

- back to home page
- getting started (on the Symfony website)

I am the home page ...

Figure 13.4: Homepage with session colours applied via CSS.

13.11 Clearing the colours from the session

We might as well add another route `/reset` to clear the colours in the session and reset things to the defaults:

```
/**
 * @Route("/reset", name="reset")
 */
public function resetAction()
{
    $session = new Session();
    $session->clear();

    return $this->redirectToRoute('default');
}
```

13.12 Testing whether an attribute is present in the current session

Before we work with a session attribute in a PHP controller method, we may wish to test whether it is present. We can test for the existance of an attribute in the session bag as follows:

```
if($session->has('<key>')){
     //do something
 }
```

13.13 Removing an item from the session attribute bag

To remove an item from the session attribute bag write the following:

```
$session->remove('<key>');
```

13.14 Clearing all items in the session attribute bag

To remove all items from the session attribute bag write the following:

```
$session->clear();
```

14 Working with a session ‘basket’ of electives

14.1 Shopping cart of products (project `session06`)

When you’re leaning sessions, you need to build a ‘shopping cart’! Let’s create CRUD for some Products and then offer a shopping baset.

We will have an `basket` item in the session, containing an array of `Product` objects adding the the basket. This array will be indexed by the `id` property of each Product (so we won’t add the same Product twice to the array), and items are easy to remove by unsetting.

14.2 Create a Product entity & generate its CRUD

Let’s use the Symfony make tool. First ensure you have these packages:

```
$ composer req doctrine security-csrf validator form
$ composer req --dev make koff/crud-maker-bundle
```

Make a new `Product` entity:

```
$ php bin/console make:entity Product
```

Then add some private properties for `description` (text), `image` (text) and `price` (float):

```
class Product
{
    /**
```

```
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string")
     */
    private $description;

    /**
     * @ORM\Column(type="string")
     */
    private $image;

    /**
     * @ORM\Column(type="float")
     */
    private $price;

    ... etc.
```

Then add accessor methods (getters and setters) for all properties.

Configure your `.env` database settings:

```
    DB_USER=root
    DB_PASSWORD=pass
    DB_HOST=127.0.0.1
    DB_PORT=3306
    DB_NAME=web7
   DATABASE_URL=mysql://${DB_USER}:${DB_PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}
```

Generate the database, and migrations and migrate:

```
    $ php bin/console doctrine:database:create
    $ php bin/console doctrine:migrations:diff
    $ php bin/console doctrine:migrations:migrate
```

Then generate CRUD for this entity (i.e. a ProductController and some templates in `/templates/product/`):

```
    $ php bin/console make:crud Product
```

14.3 Homepage - link to products home

Make the Twig template for the default homepage be a link to generated route `product_index`:

```
<p>
    Hello World
</p>

<a href="{{ url('product_index') }}">list of products</a>
```

Run the server and use your CRUD to add a few products into the database, e.g.:

```
Id  Description      Image          Price
1   hammer           hammer.png     5.99
2   ladder           ladder.png     19.99
3   bucket of nails  nails.png       0.99
```

14.4 Basket index: list basket contents (project sessions07)

We’ll write our code in a new controller class `BasketController.php` in directory `/src/Controller/`.

Generate our new controller:

```
$ php bin/console make:controller Basket
```

Here is our class (with a couple of changes to routes, and a `use` statement for `Session` class we need to use in a minute):

```
namespace App\Controller;

use Symfony\Component\HttpFoundation\Session\Session;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

/**
 * @Route("/basket")
 */
class BasketController extends Controller
{
    /**
     * @Route("/", name="_index")
     */
    public function index()
    {
        $template = 'basket/index.html.twig';
```

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

Note:

- we have added the `@Route` prefix `/basket` to all controller actions in this class by writing a `@Route` annotation comment for the class declaration.
- the basket index controller action is very simple, since all the work extracting values from the session will be done by our Twig template. So our index action simply returns the Twig rendering of template `basket/index.html.twig`

14.5 Controller method - `clearAction()`

Let's write another simple method next - a method to remove any `basket` attribute from the session. We can achieve this with the statement `$session->remove('basket')`:

```
    /**
     * @Route("/clear", name="_clear")
     */
    public function clearAction()
    {
        $session = new Session();
        $session->remove('basket');

        return $this->redirectToRoute('basket_index');
    }
```

Let's see how each route is prefixed with `/basket` and each route name is prefixed with `basket_` by listing routes at the CLI:

```
    $ php bin/console debug:router
```

```
 -------------------------- ---------- -------- ------ -----------------------------------
  Name                       Method     Scheme   Host   Path
 -------------------------- ---------- -------- ------ -----------------------------------
  basket_index               ANY        ANY      ANY    /basket/
  basket_clear               ANY        ANY      ANY    /basket/clear
  default                    ANY        ANY      ANY    /
  pinkblue                   ANY        ANY      ANY    /pinkblue
  reset                      ANY        ANY      ANY    /reset
  product_index              ANY        ANY      ANY    /product/
```

```
product_new                GET|POST   ANY      ANY     /product/new
product_show               GET        ANY      ANY     /product/{id}
product_edit               GET|POST   ANY      ANY     /product/{id}/edit
product_delete             DELETE     ANY      ANY     /product/{id}
```

14.6 Debugging sessions in Twig

As well as the Symfony profiler, there is also the powerful Twig function `dump()`. This can be used to interrogate values in the session.

Add the Symfony Twig dumper to your project using Composer:

```
$ composer req --dev var-dumper
```

You can either dump **every** variable that Twig can see, with `dump()`. This will list arguments passed to Twig by the controller, plus the `app` variable, containing session data and other applicatiton object properties.

Or you can be more specific, and dump just a particular object or variable. For example we'll be building an attribute stack session array named `basket`, and the contents of this array can be dumped in Twig with the following statement:

```
<hr>
contents of session 'basket'

{{ dump(app.session.get('basket')) }}
```

You might put this at the bottom of the HTML

element in your `base.html.twig` main template while debugging this shopping basket application.

Here is an example of what we'd see in the web page from this Twig `dump()` statement if there is one item (Product 1) in the session basket:

```
contents of session 'basket'

...Twig/Extension/Debug.php:50:
array (size=1)
  1 =>
    object(App\Entity\Product)[473]
      private 'id' => int 1
      private 'description' => string 'hammer' (length=6)
      private 'image' => string 'hammer.png' (length=10)
      private 'price' => float 5.99
```

14.7 Adding a object to the basket

The logic to add an object into our session `basket` array requires a little work.

We’ll make things easy for ourselves - using the Symfony Param-Converter. So a product `id` is in the URL `/add/<id>`, but for our method declaration we say we are expecting a reference to a `Product` record `$product`. Symfony will go off and retrieve the row from the database corresonding to the `id`, and return us a reference to a `Product` object containing the properties from the database.

First we need to get a PHP array `$products`, that is either what is currently in the session, or a new empty array if no such array was found in the session:

```
/**
 * @Route("/add/{id}", name="add")
 */
public function addToBasket(Product $product)
{
    // default - new empty array
    $products = [];

    // if 'products' array in session, retrieve and store in $products
    $session = new Session();
    if ($session->has('basket')) {
        $products = $session->get('basket');
    }
```

Note above, that we are relying on the ‘magic’ of the Symfony param-converter here, so that the integer ‘id’ received in the request is converted into its corresponding Elective object for us.

Next we get the ‘id’ of the Elective object, and see whether it can be found already in array `$electives`. If if is not already in the array, then we add it to the array (with the ‘id’ as key), and store the updated array in the session under the attribute bag key `basket`:

```
    // get ID of product
    $id = $product->getId();

    // only try to add to array if not already in the array
    if (!array_key_exists($id, $products)) {
        // append $product to our list
        $products[$id] = $product;

        // store updated array back into the session
        $session->set('basket', $products);
    }
```

Finally (whether we changed the session `basket` or not), we redirect to the basket index route:

```
        return $this->redirectToRoute('basket_index');
    }
```

14.8 The delete action method

The delete action method is very similar to the add action method. In this case we never need the whole `Product` object, so we can keep the integer `id` as the parameter for the method.

We start (as for add) by ensuring we have a PHP variable array `$products`, whether or not one was found in the session.

```
    /**
     * @Route("/delete/{id}", name="delete")
     */
    public function deleteAction(int $id)
    {
        // default - new empty array
        $products = [];

        // if 'products' array in session, retrieve and store in $products
        $session = new Session();
        if ($session->has('basket')) {
            $products = $session->get('basket');
        }
```

Next we see whether an item in this array can be found with the key `$id`. If it can, we remove it with `unset` and store the updated array in the session attribute bag with key `basket`.

```
        // only try to remove if it's in the array
        if (array_key_exists($id, $products)) {
            // remove entry with $id
            unset($products[$id]);

            if (sizeof($products) < 1) {
                return $this->redirectToRoute('basket_clear');
            }

            // store updated array back into the session
            $session->set('basket', $products);
        }
```

Note - if there are no items left in the baset, we redirect to the clear action to remove the basket attribute completely from the session.

Finally (whether we changed the session `basket` or not), we redirect to the basket index route:

```
        return $this->redirectToRoute('basket_index');
    }
```

14.9 The Twig template for the basket index action

The work extracting the array of electives in the basket and displaying them is the task of template `index.html.twig` in `/templates/basket/`.

First, we attempt to retrieve item `basket` from the session:

```
    {% set basket = app.session.get('basket') %}
```

Next we have a Twig `if` statement, displaying an empty basket message if `basket` is null, i.e.:

```
    {% if basket is null %}
        <p>
            you have no electives in your basket
        </p>
```

The we have an `else` statement (for when we did retrieve an array), that loops through creating an unordered HTML list of the basket items:

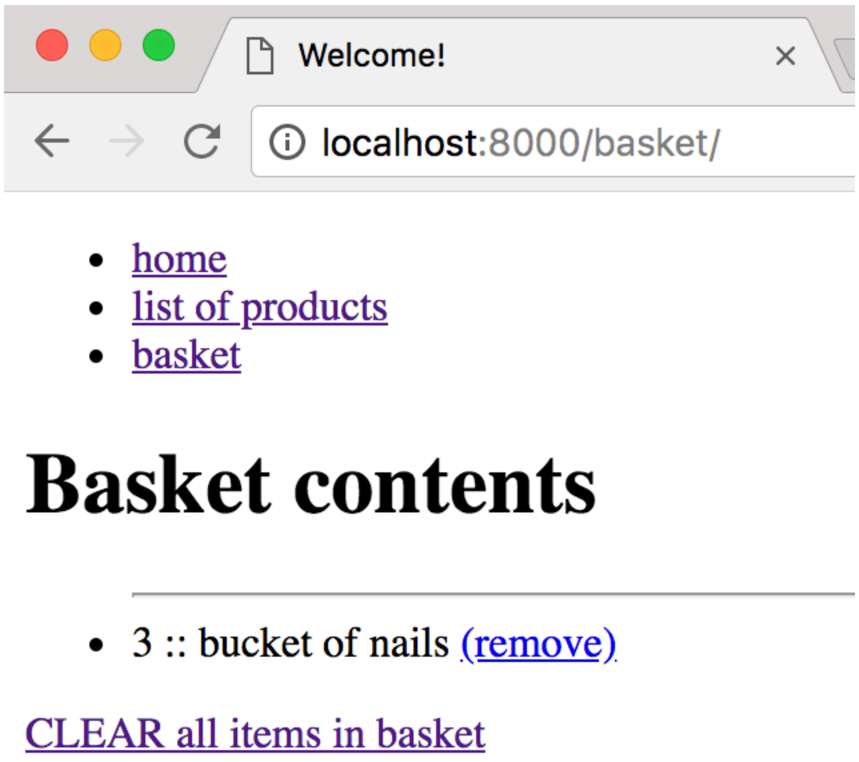
```
    {% else %}
        <ul>
            {% for product in basket %}
                <li>
                    <hr>
                    {{ product.id }} :: {{ product.description }}
             <a href="{{ path('basket_delete', { 'id': product.id }) }}">(remove)</a>
                </li>
            {% endfor %}
        </ul>
    {% endif %}
```

Note that a link to the `delete` action is offered at the end of each list item as text `(remove)`.

Finally, a paragraph is offered, containing a list to clear all items from the basket:

```
    <p>
        <a href="{{ path('basket_clear') }}">CLEAR all items in basket</a>
    </p>
```

Figure 14.1 shows a screenshot of the basket index page - listing the basket contents.



- home
- list of products
- basket

Basket contents

- 3 :: bucket of nails (remove)

CLEAR all items in basket

Figure 14.1: Shopping basket of elective modules.

14.10 Adding the 'add to basket' link in the list of electives

To link everything together, we can now add a link to 'add to basket' in our products CRUD index template. So when we see a list of electives we can add one to the basket, and then be redirected to see the updated basket of elective modules. We see below an extra list item for path `basket_add` in template `index.html.twig` in directory `/templates/product/`.

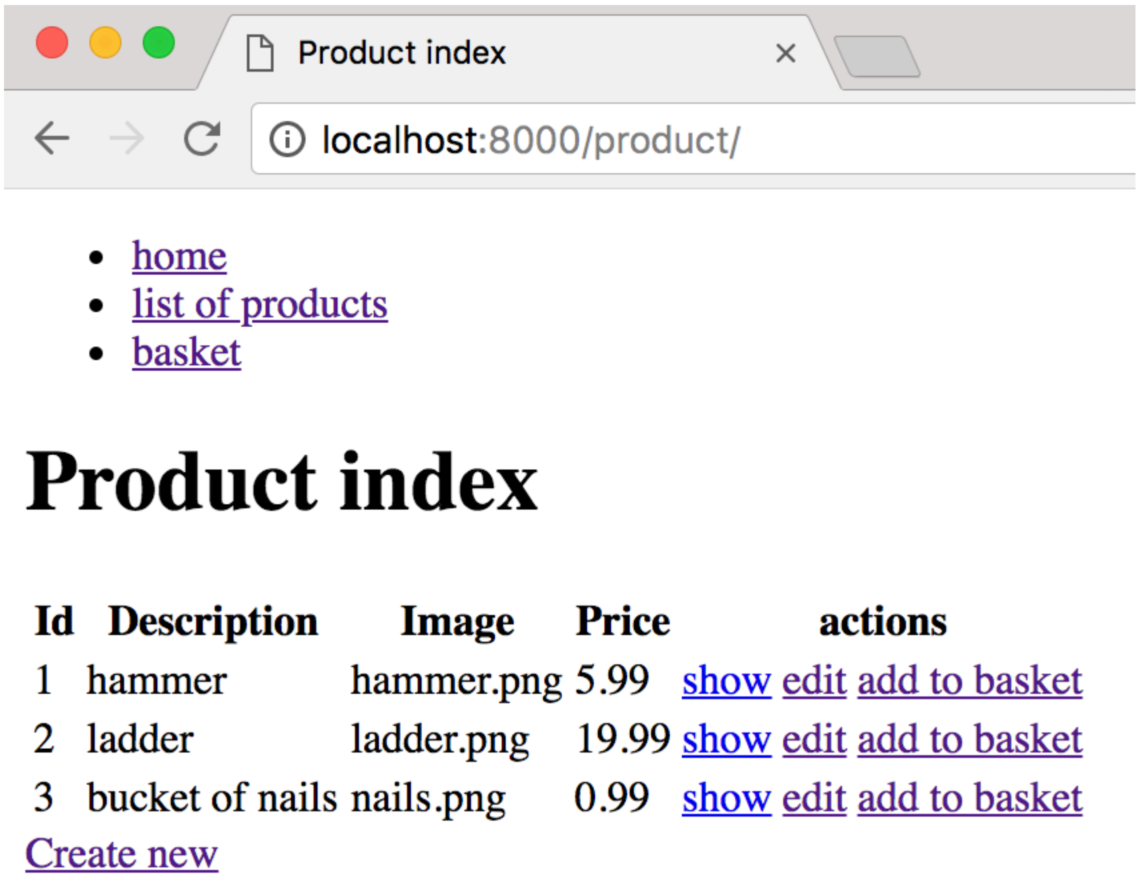
We add this line:

```
<a href="{{ path('basket_add', { 'id': product.id }) }}">add to basket</a>
```

to the end of the table cell displaying each `Product`

```
{% for product in products %}
    <tr>
        <td>{{ product.id }}</td>
        <td>{{ product.description }}</td>
        <td>{{ product.image }}</td>
        <td>{{ product.price }}</td>
        <td>
            <a href="{{ path('product_show', {'id': product.id}) }}">show</a>
            <a href="{{ path('product_edit', {'id': product.id}) }}">edit</a>
      <a href="{{ path('basket_add', { 'id': product.id }) }}">add to basket</a>
        </td>
    </tr>
{% else %}
    <tr>
        <td colspan="5">no records found</td>
    </tr>
{% endfor %}
```

Figure 14.2 shows a screenshot of the list of products page, each with an 'add to basket' link.



- home
- list of products
- basket

Product index

Id	Description	Image	Price	actions
1	hammer	hammer.png	5.99	show edit add to basket
2	ladder	ladder.png	19.99	show edit add to basket
3	bucket of nails	nails.png	0.99	show edit add to basket

Create new

Figure 14.2: List of Products with 'add to basket' link.

Part VI

Security and Authentication

15

Quickstart Symfony security

15.1 Learn about Symfony security

There are several key Symfony reference pages to read when starting with security. These include:

- Introduction to security
- How to build a traditional login form
- Using CSRF protection

15.2 Create a new project from scratch (project `security01`)

Create a new project, adding the usual packages for database and CRUD generation:

- server
- make
- twig
- annotations
- doctrine
- form
- validation
- annotations
- security-csrf
- koff/crud-maker-bundle

- profiler

Also add following 3 packages to allow us to work with security and its annotations:

- security
- sensio/framework-extra-bundle
- symfony/expression-language

```
$ composer req security sensio/framework-extra-bundle symfony/expression-language
```

15.3 Make a Default controller

Let's make a Default controller `/src/Controller/DefaultController.php`:

```
$ php bin/console make:controller Default
```

Edit the route to be `/` and the internal name to be `homepage`:

```
/**
 * @Route("/", name="homepage")
 */
public function indexAction()
{
    $template = 'default/index.html.twig';
    $args = [];
    return $this->render($template, $args);
}
```

Change the template to be something like:

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

{% block body %}
    welcome to the home page
{% endblock %}
```

This will be accessible to everyone.

15.4 Make a secured Admin controller

Let's make a Admin controller:

```
$ php bin/console make:controller Admin
```

This will be accessible to only to users logged in with `ROLE_ADMIN` security.

Edit the new `AdminController` in `/src/Controller/AdminController.php`. Add a `use` statement, to let us use the `@Security` annotation:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
```

Now we'll restrict access to the index action of our Admin controller using the `@Security` annotation:

```
/**
 * @Route("/admin", name="admin")
 * @Security("has_role('ROLE_ADMIN')")
 */
public function index()
{
    $template = 'admin/index.html.twig';
    $args = [];
    return $this->render($template, $args);
}
```

Change the template to be something like:

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

{% block body %}
    <h1>Admin home</h1>

    here is the secret code to the safe:
    007123
{% endblock %}
```

That's it!

Run the web sever:

- visiting the Default page at `/` is fine, even though we have not logged in ag all
- however, visiting the the `/admin` page should result in a 500 (server error) due to insufficient authentication. See Figure 15.1.

Of course, we now need to add a way to login and define different user credentials etc…

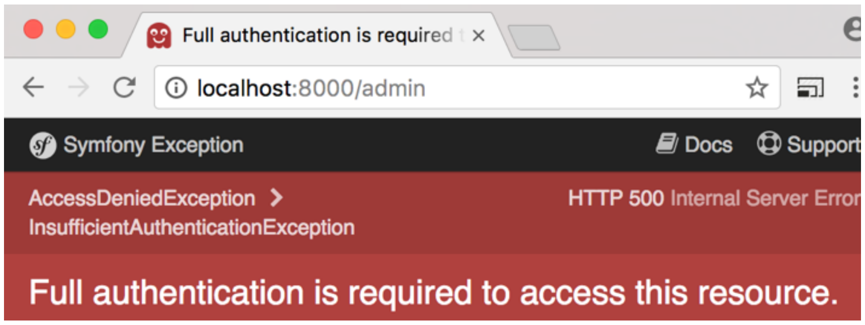

Full authentication is required
localhost:8000/admin
Symfony Exception
Docs
Support
AccessDeniedException > InsufficientAuthenticationException
HTTP 500 Internal Server Error
Full authentication is required to access this resource.

Figure 15.1: Screenshot of error attempting to visit `/admin`.

15.5 Core features about Symfony security

There are several related features and files that need to be understood when using the Symnfony security system. These include:

- **firewalls**
- **providers** and **encoders**
- **route protection** (we met this with `@Security` controller method annotation comment above...)
- user **roles** (we met this as part of `@Security` above `has_role('ROLE_ADMIN')` ...)

Core to Symfony security are the **firewalls** defined in `/config/packages/security.yml`. Symfony firewalls declare how route patterns are protected (or not) by the security system. Here is its default contents (less comments - lines starting with hash `#` character):

```
security:
    providers:
      in_memory: { memory: ~ }

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: true
```

Symfony considers **every** request to have been authenticated, so if no login action has taken place then the request is considered to have been authenticated to be **anonymous** user `anon`. We can see in this `anon` user in Figure 15.2 this looking at the user information from the Symfony debug bar when visiting the default home page.

A Symfony **provider** is where the security system can access a set of defined users of the web application. The default is simply `in_memory` - although usually larger applications have users in a database or from a separate API. We see that the `main` firewall simply states that users are

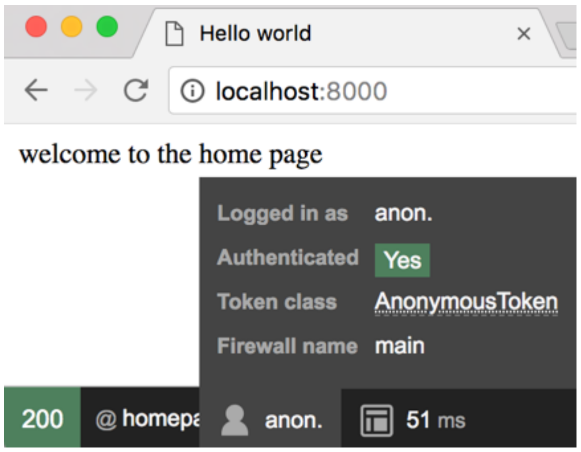


Figure 15.2: Symfony profiler showing anonymous user authentication.

permitted (at present) any request route pattern, and anonymous authenticated users (i.e. ones who have not logged in) are permitted.

NOTE In some Symfony documentation you'll see `default` instead of `main` for the default firewall. Both seem to work the same way (i.e. as the default firewall settings). So choose one and stick with it. Since my most recent new Symfony project called this `main` in the `security.yml` file I'll stick with that one for now …

The `dev` firewall allows Symfony development tools (like the profiler) to work without any authentication required. Leave it in `security.yml` and just ignore the `dev` firewall from this point onwards.

15.6 Defining some users and HTTP basic login (project `security02`)

We control security through file '/config/packages/security.yml'.

Let's define 3 users[1]:

- `user` has password `user`, and the security role ROLE_USER
- `admin` has password `admin`, and the security role ROLE_ADMIN
- `matt` has password `smith`, and the security role ROLE_ADMIN

The simplest way is to define them in `security.yml` as 'in memory' users:

replace:

```
    #       in_memory: { memory: ~ }
```

[1]Ensure you always prefix security roles with `ROLE_`, to ensure they are processed by Symfony's security system.

with:

```
        in_memory:
            memory:
                users:
                    user:
                        password: user
                        roles: 'ROLE_USER'
                    admin:
                        password: admin
                        roles: 'ROLE_ADMIN'
                    matt:
                        password: smith
                        roles: 'ROLE_ADMIN'
```

We also must state how these user's passwords are encoded (or not):

```
    encoders:
        Symfony\Component\Security\Core\User\User: plaintext
```

Finally, we need some kind of login form. The simplest is the basic HTTP login form built into web browsers. So remove the hash # comment in line:

```
            http_basic: true
```

See Figure 15.3 to see the Chrome browser's basic HTTP built-in login form.

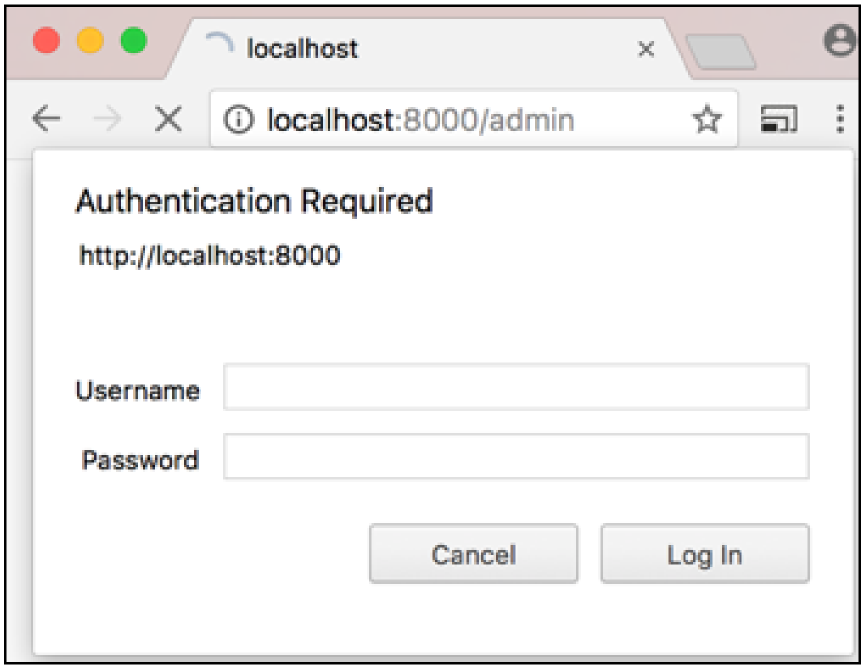
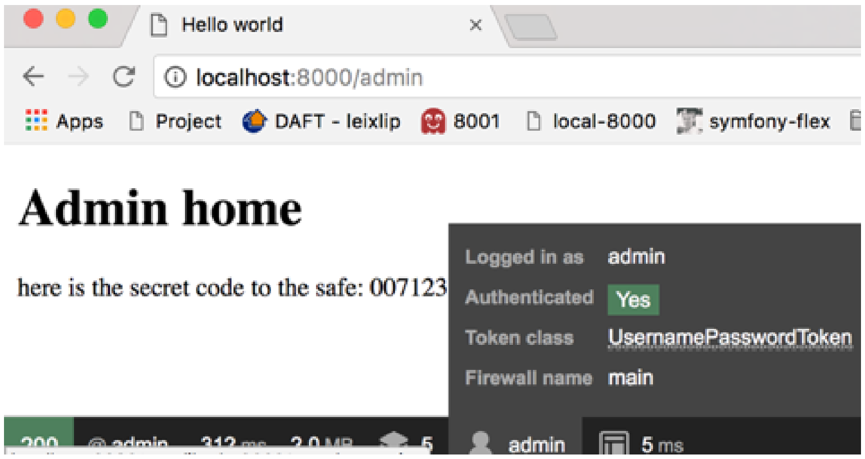


Figure 15.3: Screenshot of Chrome HTTP basic login form.

See Figure 15.4 to see the logged-in user in the Symfony page footer profiler.



Admin home

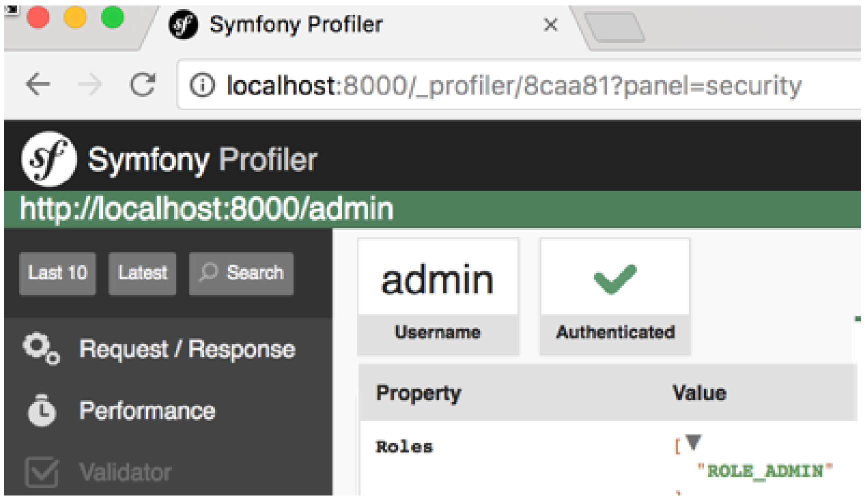
here is the secret code to the safe: 007123

Logged in as	admin
Authenticated	Yes
Token class	UsernamePasswordToken
Firewall name	main

Figure 15.4: Screenshot of Profiler showing admin login.

15.7 Click user in Profile bar to see ROLE

If you click the user in the Symfony profiler footer, the Profiler will show you details of the logged-in user, including their ROLE_. See Figure 15.5.



Property	Value
Roles	["ROLE_ADMIN"]

Figure 15.5: Screenshot of Profiler showing user role details.

16

Custom login page and a logout route

16.1 Custom login form controller ((project `security03`)

The Symfony documentation tell's us how to create a custom login form, with CSRF protection, so let's do that.

- How to build a Traditional Login form
- CSRF protection in the Login form (NOTE the default settings work fines for this - we just need to make sure any Twig templates we write display the appropriate hidden CSRF form fields…)

First we need to replace out `http_basic` login authentication with our own, custom login form. We do this by replacing `http_basic: ~` in our `main` firewall with the a `form_login` entry.

NOTE Below I have commented-out the `http_basic` entry, to make it clear where we are replacing its entry:

```
security:
    encoders:
        Symfony\Component\Security\Core\User\User: plaintext

    providers:
        in_memory:
            memory:
                users:
                    user:
                        password: user
                        roles: 'ROLE_USER'
                    admin:
                        password: admin
                        roles: 'ROLE_ADMIN'
                    matt:
                        password: smith
                        roles: 'ROLE_ADMIN'

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        main:
            anonymous: true
#            http_basic: true
            form_login:
                login_path: login
                check_path: login
```

The above declares that the route for an authentication login form is named `login` (we'll add a controller method naming that route next). We are defining 2 important properties for the security system:

- `login_path` - this is the route users will be redirected to if they attempt to access a resource but are do not have the authentication permitted to do so
- `check_path` - this is the route which the login form will submit a POST request to

You can read more about these paths, and other customisable features of the Symfony login system in the Symfony documentation:

- Symfony `login_path` and `check_path` reference

Let's create a new `SecurityController` in `/src/Controller/` which declares the login route, and also tells our application to render a Twig custom login form template.

```
    namespace App\Controller;

    use Symfony\Component\Routing\Annotation\Route;
    use Symfony\Bundle\FrameworkBundle\Controller\Controller;
    use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;
    use Symfony\Component\HttpFoundation\Request;
```

```php
class SecurityController extends Controller
{
    /**
     * @Route("/login", name="login")
     */
    public function login(Request $request, AuthenticationUtils $authUtils)
    {
            // logic to show login form goes here
    }
}
```

NOTE We have broken-down the final steps of naming the Twig template and building the Twig argument array (simplifying the one-liner code from the Symfony documentation):

```php
public function login(Request $request, AuthenticationUtils $authUtils)
{
    // get the login error if there is one
    $error = $authUtils->getLastAuthenticationError();

    // last username entered by the user
    $lastUsername = $authUtils->getLastUsername();

    $template = 'security/login.html.twig';
    $args = [
        'last_username' => $lastUsername,
        'error'         => $error,
    ];

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

Looking at the above we can note the following:

- the method parameters give us a reference to the `Request` object, and a reference to the Symfony security utilities service `$authUtils`
- if there was a previous login attempt, then the error and username are retrieved
    - `$error` is set to any error `$error` (from the last login attempt -)
    - `$lastUsername` is the username `$last_username` (for repeated login attempts)
- finally we have our Twig statements, declaring that the login template is template directory `security`, and building and then passing to Twig an arguments array containing the error and last username

We also can see that there is no logic in this method to **process** the submission of the form. The

Symfony security system will process login form submission by looking through all its **providers** to see if it can match with a username/password pair, and acting accordingly.

16.2 Creating the login form Twig template

Let's write our Twig template for the login form (copied from the Symfony documentation pages). Here is `/templates/security/login.html.twig`:

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

{% block body %}
    {% if error %}
        <div>{{ error.messageKey|trans(error.messageData, 'security') }}</div>
    {% endif %}

    <form action="{{ path('login') }}" method="post">
        <label for="username">Username:</label>
        <input type="text" id="username" name="_username" value="{{ last_username }}" />

        <label for="password">Password:</label>
        <input type="password" id="password" name="_password" />

        <button type="submit">login</button>
    </form>
{% endblock %}
```

Above we can see the following in our Login Twig template:

- display of any Twig `error` variable received
- the HTML `<form>` open tag, which we see submits via HTTP `POST` method to the route named `login`
- the `username` label and text input field (and value of the `last_username` if any)
- the `password` label and password input field
- the submit button named `login`

16.3 Custom login form when attempting to acess `/admin`

See Figure 16.1 to see our custom login form in action.

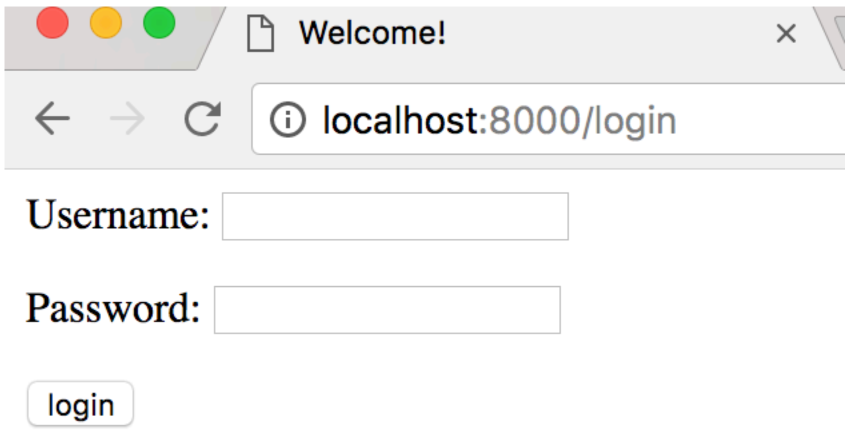


Figure 16.1: Screenshot of custom login form.

16.4 Adding a `/logout` route

We can define a route to logout very easily in Symfony, with no need for any controller method. In `/config/routes.yaml` we add our login route, and its redirect to the website home page `/`. These are the only 2 lines we need in this file (for our simple demo application):

```
logout:
    path: /logout
```

We also need to define the logout route as part of our security firewall. So in `security.yml` we add the following `logout` route to our `main` firewall, immediately after our `form_login` routes:

```
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        anonymous: true
        form_login:
            login_path: login
            check_path: login
        logout:
            path:   /logout
            target: /
```

Figure 16.2 shows that we can see the logout route is available from the Symfony profile toolbar. We can, of course, also enter the route directly in the browser address bar, e.g. via URL:

```
http://localhost:8000/logout
```

In either case we'll logout any currently logged-in user, and return the anonymously authenticated

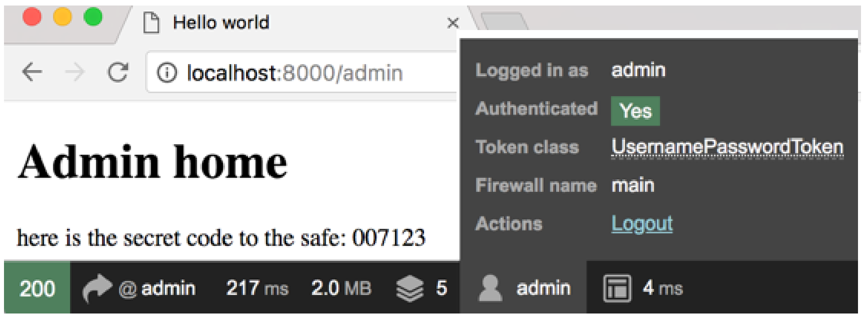


Figure 16.2: Symfony profiler user logout action.

user `anon` with no defined authentication roles.

16.5 CSRF protection

CSRF = Cross Site Request Forgery

NOTE: For any public **production** site you should always implement CSRF protection. This is implemented using CSRF ‘tokens’ created on the server and exchanged with the web client and form sumissions. CSRF tokens help protect web applications against cross-site scripting request forgery attacks and forged login attacks.

Learn more about CSRF threats and security:

- Symfony CSRF protection
- Wikipedia

17

Security users from database

17.1 Create a `User` entity (project `security04`)

We need to create a `User` entity, that implements the Symfony Security User Interface.

Create Entity class `/src/Entity/User.php`:

```
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;

/**
 * @ORM\Table(name="app_users")
 * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
 */
class User implements UserInterface, \Serializable
{
    // properties and methods go here ...
}
```

Here are all the properties needed:

```
/**
 * @ORM\Column(type="integer")
 * @ORM\Id
```

```
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=25, unique=true)
     */
    private $username;

    /**
     * @ORM\Column(type="string", length=64)
     */
    private $password;

    /**
     * @ORM\Column(type="json_array")
     */
    private $roles = [];
```

We need 2 special methods relating to security:

```
    public function getSalt()
    {
        // no salt needed since we are using bcrypt
        return null;
    }

    public function eraseCredentials()
    {
    }
```

We need 2 special methods relating to serialization:

```
    /** @see \Serializable::serialize() */
    public function serialize()
    {
        return serialize(array(
            $this->id,
            $this->username,
            $this->password,
        ));
    }
```

```
/** @see \Serializable::unserialize() */
public function unserialize($serialized)
{
    list (
        $this->id,
        $this->username,
        $this->password,
    ) = unserialize($serialized);
}
```

We need 2 special methods relating to user ROLEs:

```
public function getRoles()
{
    $roles = $this->roles;
    // ensure always contains ROLE_USER
    $roles[] = 'ROLE_USER';

    return array_unique($roles);
}

public function setRoles($roles)
{
    $this->roles = $roles;
    return $this;
}
```

You can now generate standard getters and setters for all remaining properties.

17.2 Create database and migrate

Now you can create a database based on your `.env` database settings, and migrate the Entity structure to your database schema:

```
$ php bin/console doctrine:database:create
$ php bin/console doctrine:migrations:diff
$ php bin/console doctrine:migrations:migrate
```

17.3 Configure security to use DB users as 'provider'

In `/config/packages/security.yml` we specify our firewalls, and also:

- where our security users are provided from `providers`

- how passwords are hashed/encrypted for security users `encoders`

We will replace our in memory provider with a database provider of objects of our `App\Entity\User` class:

```
providers:
    our_db_provider:
        entity:
            class: App\Entity\User
            property: username
```

We will replace the plaintext encoder with a `bcrypt` algorithm for objects of our `App\Entity\User` class:

```
encoders:
    App\Entity\User:
        algorithm: bcrypt
```

So our complete `security.yml` file now reads as follows:

```
security:
    providers:
        our_db_provider:
            entity:
                class: App\Entity\User
                property: username

    encoders:
        App\Entity\User:
            algorithm: bcrypt

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: true
            provider: our_db_provider
            form_login:
                login_path: login
                check_path: login

            logout:
                path:   /logout
                target: /
```

17.4 Initial values for your project database

Fixtures play two roles:

- inserting initial values into your database (e.g. the first `admin` user)
- setting up the database to a known state for **testing** purposes

Doctrine provides a Symfony fixtures **bundle** that makes things very straightforward.

Learn more about Symfony fixtures at:

- Symfony website fixtures page

17.5 Instal Doctrine fixtures package

Use Composer to add the Doctrine fixtures package to the project:

```
composer req doctrine/doctrine-fixtures-bundle
```

17.6 Writing the fixture classes

We need to locate our fixtures in directory `/src/DataFixtures`.

We need to write a class that extends the Doctrine `Fixtures` class.

We now need to create class `LoadUsers` in file `/src/DataFixtures/LoadUsers.php`:

```
namespace App\DataFixtures;

use App\Entity\User;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;

class LoadUsers extends Fixture
{
    // properties and methods go here ...
}
```

We need a private property `$encoder` that will allow us to hash passwords before their storage:

```
/**
 * @var UserPasswordEncoderInterface
 */
private $encoder;
```

```
    public function __construct(UserPasswordEncoderInterface $encoder)
    {
        $this->encoder = $encoder;
    }
```

We need a `load(...)` method, that gets invoked when we are loading fixtures from the CLI. This method creates objects for the entities we want in our database, and the saves (persists) them to the database:

```
    public function load(ObjectManager $manager)
    {
        // create objects
        $userUser = $this->createUser('user', 'user');
        $userAdmin = $this->createUser('admin', 'admin', ['ROLE_ADMIN']);
        $userMatt = $this->createUser('matt', 'smith', ['ROLE_ADMIN']);

        // store to DB
        $manager->persist($userUser);
        $manager->persist($userAdmin);
        $manager->persist($userMatt);
        $manager->flush();
    }
```

Rather than put all the work in the `load(...)` method, we can create a helper method to create each new object. Method `createUser(...)` creates and returns a reference to a new `User` object given some parameters:

```
    private function createUser($username, $plainPassword, $roles = ['ROLE_USER']):User
    {
        $user = new User();
        $user->setUsername($username);
        $user->setRoles($roles);

        // password - and encoding
        $encodedPassword = $this->encodePassword($user, $plainPassword);
        $user->setPassword($encodedPassword);

        return $user;
    }
```

NOTE: The default role is `ROLE_USER` if none is provided.

Finally, to make it clear how we are encoding the password, we have method `encodePassword(...)`, returning an encoded password given a `User` object and a plain text password:

```
private function encodePassword($user, $plainPassword):string
{
    $encodedPassword = $this->encoder->encodePassword($user, $plainPassword);
    return $encodedPassword;
}
```

17.7 Loading the fixtures

Loading fixtures involves deleting all existing database contents and then creating the data from the fixture classes - so you'll get a warning when loading fixtures. At the CLI type:

```
php bin/console doctrine:fixtures:load
```

That's it!

You should now be able to access `/admin` with either the `matt/smith` or `admin/admin` users. You will get an Access Denied exception if you login with `user/user`, since that only has `ROLE_USER` privileges, and `ROLE_ADMIN` is required to visit `/admin`.

See Figure 18.1 to see the default Symfony (dev mode) Access Denied exception page.

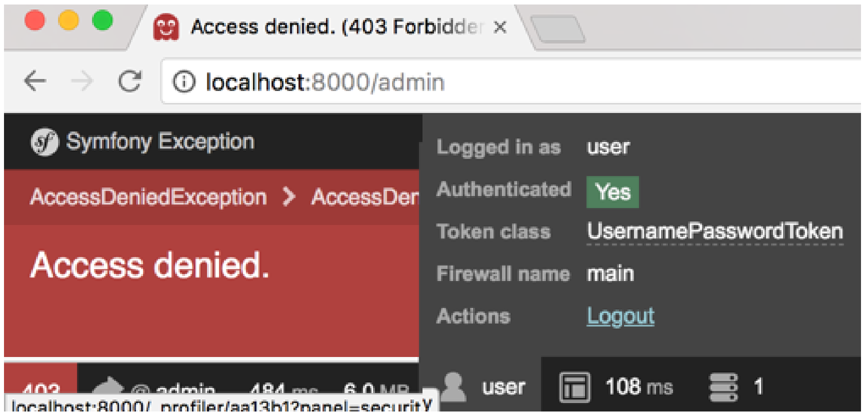

Access denied. (403 Forbidde
localhost:8000/admin
Symfony Exception
AccessDeniedException > AccessDen
Access denied.
Logged in as user
Authenticated Yes
Token class UsernamePasswordToken
Firewall name main
Actions Logout
user
108 ms
1
localhost:8000/_profiler/aa13b1?panel=security

Figure 17.1: Screenshot of Default Symfony access denied page.

The next chapter will show you how to deal with (and log) access denied exceptions ...

17.8 Using SQL from CLI to see users in DB

To double check your fixtures have been created correctly in the database, you could run an SQL query from the CLI:

```
$ php bin/console doctrine:query:sql "SELECT * FROM app_users"

/../security04_db_users/vendor/doctrine/common/lib/Doctrine/Common/Util/Debug.php:71:
```

```
array (size=3)
  0 =>
    array (size=6)
      'id' => string '1' (length=1)
      'username' => string 'user' (length=4)
      'password' => string '$2y$13$uLxKuVGLJnnKzXmlmCizf.scKM5rm87w9WPlatk2g8KXrCDOtSIvy' (l
      'roles' => string '["ROLE_USER"]' (length=13)
  1 =>
    array (size=6)
      'id' => string '2' (length=1)
      'username' => string 'admin' (length=5)
      'password' => string '$2y$13$xTIs6Fmt9ZPeKUORUWWIkO6Wt9SlFZEZnrhbbE3yw5BCx5aLwgE1a' (l
      'roles' => string '["ROLE_ADMIN"]' (length=14)
  2 =>
    array (size=6)
      'id' => string '3' (length=1)
      'username' => string 'matt' (length=4)
      'password' => string '$2y$13$wSciYVsT5HwAws69wwe//ObWfj3RufGVuhw01hvjLkkSqCR5hWaha' (l
      'roles' => string '["ROLE_ADMIN"]' (length=14)
```

18

Custom AccessDeniedException handler

18.1 Declaring our handler (project `security05`)

In `/config/packages/security.yml` we need to declare that the class we'll write below will handle access denied exceptions.

So we add this line to the end of our `main` firewall in `security.yml`:

```
    access_denied_handler: App\Security\AccessDeniedHandler
```

So the full listing for our `security.yml` is now:

```
security:
    encoders:
        App\Entity\User:
            algorithm: bcrypt

    providers:
        our_db_provider:
            entity:
                class: App\Entity\User
                property: username

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: true
            provider: our_db_provider
            form_login:
                login_path: login
                check_path: login
            logout:
                path:   /logout
                target: /
            access_denied_handler: App\Security\AccessDeniedHandler
```

18.2 The exception handler class

Now we needs to write our exception handler class in `/src/Security`.

Create new class `AccessDeniedHandler` in file `/src/Security/AccessDeniedHandler.php`:

```
namespace App\Security;

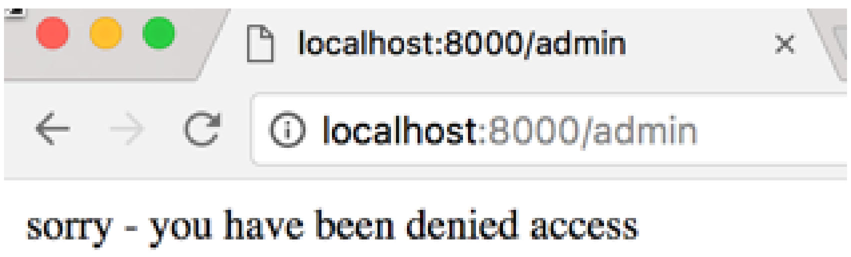
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Symfony\Component\Security\Http\Authorization\AccessDeniedHandlerInterface;

class AccessDeniedHandler implements AccessDeniedHandlerInterface
{
    public function handle(Request $request, AccessDeniedException $accessDeniedException)
    {
        return new Response('sorry - you have been denied access', 403);
    }
}
```

That's it!

Now if you try to access `/admin` with `user/user` you'll see the message 'sorry - you have been denied access' on screen. See Figure 18.1.

Although it won't be generated through the Twig templating system - we'll learn how to do that next …



sorry - you have been denied access

Figure 18.1: Screenshot of Custom Twig access denied page.

18.3 Twig and logging (project security06)

Let's improved our Access Denied exception handler in 2 ways:

- display a nice Twig template
- log the exception using the standard Monolog logging system

First add Monolog to our project with Composer:

```
$ composer req logger
```

Now we will refactor class `AccessDeniedHandler` to

```
namespace App\Security;

use Psr\Log\LoggerInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Symfony\Component\Security\Http\Authorization\AccessDeniedHandlerInterface;

class AccessDeniedHandler implements AccessDeniedHandlerInterface
{
    private $twig;
    private $logger;

    public function __construct(ContainerInterface $container, LoggerInterface $logger)
    {
        $this->twig = $container->get('twig');
        $this->logger = $logger;
    }

}
```

Now we can re-write method `handle(...)` to log an error message, and

```
public function handle(Request $request, AccessDeniedException $accessDeniedException)
{
    $this->logger->error('access denied exception');

    $template = 'error/accessDenied.html.twig';
    $args = [];
    $html = $this->twig->render($template, $args);
    return new Response($html);
}
```

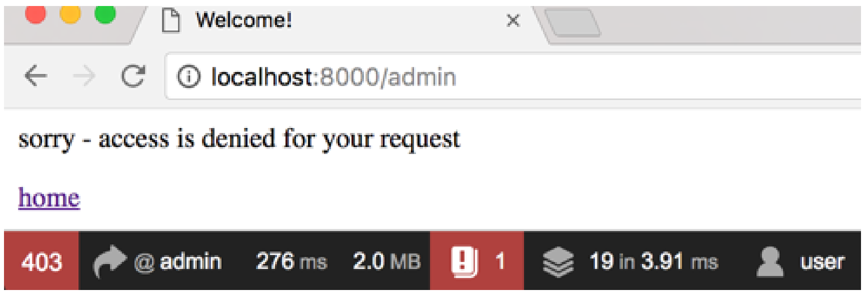
18.4 The Twig page

Create Twig error page `/templates/error/accessDenied.html.twig`:

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

{% block body %}
    sorry - access is denied for your request
    <p>
        <a href="{{ url('homepage') }}">home</a>
    </p>
{% endblock %}
```

See Figure 18.2 to see the error log register in the Symfony profiler footer, at the bottom of our custom error page.

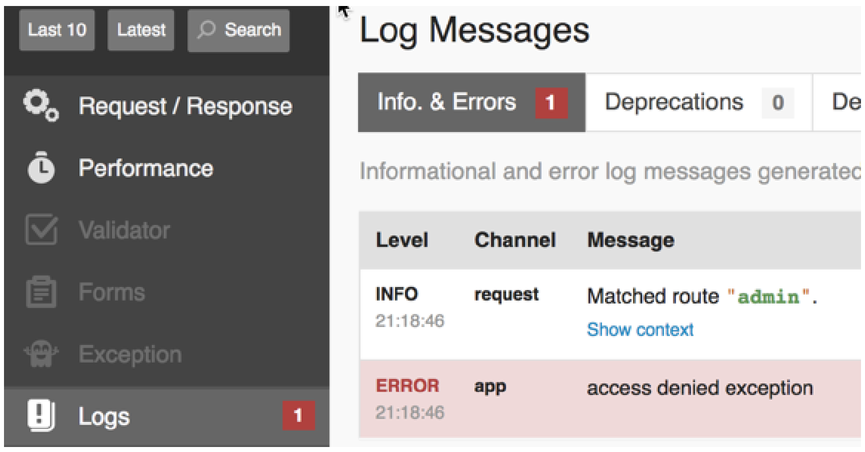


sorry - access is denied for your request

home

Figure 18.2: Screenshot of Custom Twig access denied page.

If you click on the red error you'll see details of all logged messages during the processing of this request. See Figure 18.3.



Informational and error log messages generatec

Level	Channel	Message
INFO 21:18:46	request	Matched route "admin". Show context
ERROR 21:18:46	app	access denied exception

Figure 18.3: Screenshot of Profiler log entries.

18.5 Learn more about logger and exceptions

Learn more about Symfony and the Monolog logger:

- Logging with Monolog

Learn more about custom exception handlers and error pages:

- Access Denied Handler
- Custom Error pages

19

User roles and role hierarchies

19.1 Simplifying roles with a hierarchy (project `security07`)

Let's avoid repeating roles in our program logic (e.g. IF `ROLE_USER` OR `ROLE_ADMIN`) by creating a hierarchy, so we can give `ROLE_ADMIN` all properties of `ROLE_USER` as well. We can easily create a role hierarchy in `/config/packages/security.yml`:

```
security:
    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER

    ... rest of 'security.yml' as before ...
```

In fact let's go one further - let's create a 3rd user role (`ROLE_SUPER_ADMIN`) and define that as having all `ROLE_ADMIN` privileges plus the ab

```
security:
    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: ROLE_ADMIN

    ... rest of 'security.yml' as before ...
```

Now if we log in as a user with `ROLE_SUPER_ADMIN` we also get `ROLE_ADMIN` and `ROLE_USER` too!

19.2 Modify fixtures

Now we can modify our fixtures to make user `matt` have `ROLE_SUPER_ADMIN`.

Change `/src/DataFixtures/LoadUsers.php` as follows:

```
    public function load(ObjectManager $manager)
    {
        ...

        $userMatt = $this->createActiveUser(
            'matt',
            'smith',
            ['ROLE_SUPER_ADMIN']);

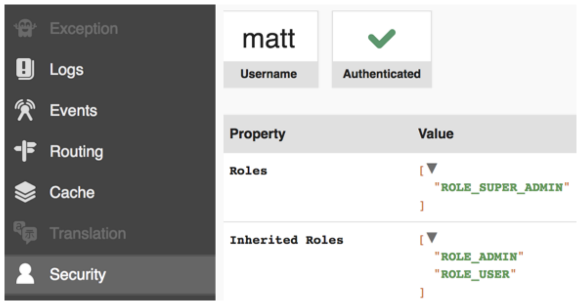
        ...
```

19.3 Removing default adding of `ROLE_USER` if using a hierarchy

If we are using a hierarchy, we don't need always add `ROLE_USER` in code, so we can simplify our getter in our `User` Entity in `/src/Entity/User.php`:

~~~
```php
    public function getRoles()
    {
        return $this->roles;
    }
```
~~~

We'll still see `ROLE_USER` for admin and super users, but in the list of **inherited** roles from the hierarchy. This is show in Figure 19.1.



Property	Value
Roles	["ROLE_SUPER_ADMIN"]
Inherited Roles	["ROLE_ADMIN" "ROLE_USER"]

Figure 19.1: Super admin user inheriting ROLE_USER.

Learn about user role hierarchies at:

- Symfony hierarchical roles

19.4 Allowing easy switching of users when debugging

If you wish to speed up testing, you can allow easy switching between users just by adding a but at the end of your request URL, **if** you add the following to your firewall:

```
switch_user: true
```

Now you can switch users bu adding the following at the end of the URL:

```
?_switch_user=<username>
```

You stop impersonating users by adding `?_switch_user=_exit` to the end of a URL.

For example to visit the home page as user `user` you would write this URL:

```
http://localhost:8000/?_switch_user=user
```

In your Twig you can allow this user to see special content (e.g. a link to exit impersonation) by testing for the special (automatically created role) `ROLE_PREVIOUS_ADMIN`:

```
{% if is_granted('ROLE_PREVIOUS_ADMIN') %}
  <a href="{{ path('admin_index', {'_switch_user': '_exit'}) }}">Exit impersonation & return to admin ho
{% endif %}
```

Learn more at:

- Impersonating users

20 Customising view based on logged-in user

20.1 Twig nav links when logged in (project `security08`)

The Symfony security docs give us the Twig code for a conditional statement for when the current user has logged in:

```
{% if is_granted('IS_AUTHENTICATED_FULLY') %}
    <p>Username: {{ app.user.username }}</p>
{% endif %}
```

We can also test for which **role** a user may have granted when logged-in, e.g.:

```
{% if is_granted('ROLE_ADMIN') %}
      Welcome to the Admin home page ...
{% endif %}
```

We can use such conditionals in 2 useful and common ways:

1. Confirm the login username and offer a `logout` link for users who are logged in
2. Have navbar links revealed only for logged-in users (of particular roles)

So let's add such code to our `base.html.twig` master template (in `/templates`).

First, let's add a `<header>` element to either show the username and a logout link, or a link to login if the user is not logged-in yet:

```
<header>
    {% if is_granted('IS_AUTHENTICATED_FULLY') %}
```

```
            Username:
            <strong>{{ app.user.username }}</strong>
            <br>
            <a href="{{ url('logout') }}">logout</a>
        {% else %}
            <a href="{{ url('login') }}">login</a>
        {% endif %}
    </header>
```

We can right align it and have a black bottom border with a little style in the `<head>`:

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

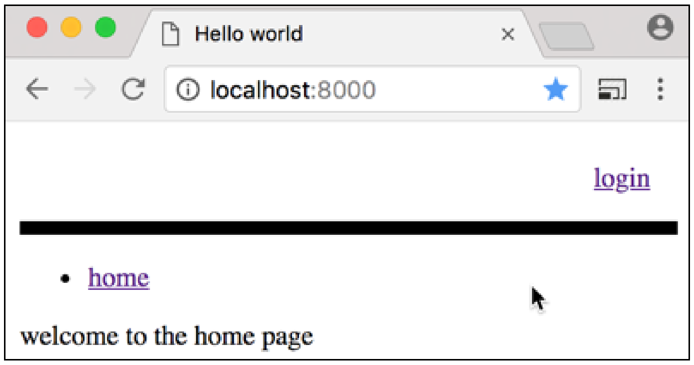
            <style>
                header {
                    text-align: right;
                    border-bottom: 0.5rem solid black;
                    padding: 1rem;
                }
            </style>
```

Next, let's define a `<nav>` element, so that **all** users see a link to the homepage on every page on the website (at least those that extend `base.html.twig`). We will also add a conditinal navigation link - to that users logged-in with `ROLE_ADMIN` can also see a link to the admin home page:

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

            {% if is_granted('ROLE_ADMIN') %}
                <li>
                    <a href="{{ url('admin_home') }}">admin home</a>
                </li>
            {% endif %}
        </ul>
    </nav>
```

So when a user first visits our website homepage, they are not logged-in, so will see a `login` link in the header, and the navigation bar will only show a link to this homepage. See Figure 20.1.



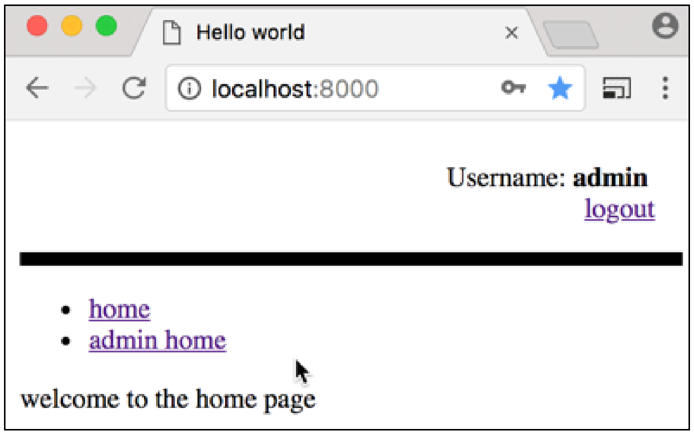
login

- home

welcome to the home page

Figure 20.1: Screenshot of homepage before logging-in.

If the user has successfully logged-in with a `ROLE_ADMIN` privilege account, they will now see their userame and a `logout` link in the header, and they will also see revealed a link to the admin home page. See Figure 20.2.



Username: **admin**
logout

- home
- admin home

welcome to the home page

Figure 20.2: Screenshot of homepage after `ROLE_ADMIN` has logged-in.

Part VII

Entity associations (one-to-many relationships etc.)

21 Database relationships (Doctrine associations)

21.1 Information about Symfony 4 and databases

Learn about Doctrine relationships and associates at the Symfony documentation pages:

- https://symfony.com/doc/current/doctrine.html#relationships-and-associations
- https://symfony.com/doc/current/doctrine/associations.html

21.2 Create a new project from scratch (project `associations01`)

Create a new project, adding the usual packages for database and CRUD generation:

- server
- make
- twig
- annotations
- doctrine
- form
- validation
- annotations
- security-csrf
- koff/crud-maker-bundle

21.3 Categories for Products

Let's have 2 categories of Product:

- large items
- small items

So we need to generate a Category Entity:

```
$ php bin/console make:entity Category

 created: src/Entity/Category.php
 created: src/Repository/CategoryRepository.php
```

We'll add a `name` field and generate getters and setters:

```
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\CategoryRepository")
 */
class Category
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string")
     */
    private $name;

    ... getters and setters
```

21.4 Create a Product entity

Generate a Product Entity:

```
$ php bin/console make:entity Product

 created: src/Entity/Product.php
 created: src/Repository/ProductRepository.php
```

Then add some private properties for `description` (text), `image` (text) and `price` (float):

```
class Product
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string")
     */
    private $description;

    /**
     * @ORM\Column(type="string")
     */
    private $image;

    /**
     * @ORM\Column(type="float")
     */
    private $price;

    ... getters and setters
```

21.5 Defining the many-to-one relationship from Product to Category

We now edit our `Product` entity, declaring a property `category` that has a many-to-one relationship with entity `Category`. I.e., many products relate to one category.

Add the following field and setter in `/src/Entity/Product.php`:

```
class Product
{

    ... properties and accessor methods for descrition / image / price ...

    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\Category", inversedBy="products")
     * @ORM\JoinColumn(nullable=true)
     */
    private $category;

    public function getCategory(): Category
    {
        return $this->category;
    }

    public function setCategory(Category $category)
    {
        $this->category = $category;
    }
```

21.6 How to allow `null` for a Product's category

We need to allow `null` for a Product's category:

- when it is first created (to generate a form the easy way)
- to allow a category to be removed from a Product

We need to allow our 'getter' to return `null` or a reference to a `Category`, so we change the return type to `?Category`:

```
// allow null - ?Category vs Category
public function getCategory(): ?Category
{
    return $this->category;
}
```

We need set the default value for our 'setter' to `null`:

```
// default Category to null
public function setCategory(Category $category = null)
{
    $this->category = $category;
}
```

This 'nullable' parameter/return value is one of the new features from PHP 7.1 onwards:

- PHP.net guide to migrating to PHP 7.1

21.7 Adding the optional one-to-many relationship from Category to Product

Each `Category` relates to many `Products`. Symfony with Doctrine makes it very easy to get an array of `Product` objects for a given `Category` object, without having to write any queries. We just declare a `products` property in gthe `Category` entity class, and use annotations to declare that it is the recipricol one-to-many relationship with entity `Product`.

Add the following field and setter in `/src/Entity/Product.php` (don't forget to add the `use` statement for class `ArrayCollection`):

```
use Doctrine\ORM\Mapping as ORM;
use Doctrine\Common\Collections\ArrayCollection;

/**
 * @ORM\Entity(repositoryClass="App\Repository\CategoryRepository")
 */
class Category
{
    ... properties and accessor methods for name ...

    /**
    /**
     * @ORM\OneToMany(targetEntity="App\Entity\Product", mappedBy="category")
     */
    private $products;

    public function __construct()
    {
        $this->products = new ArrayCollection();
    }
```

21.8 Create and migrate DB schema

Configure your `.env` database settings:

```
    DB_USER=root
    DB_PASSWORD=pass
    DB_HOST=127.0.0.1
    DB_PORT=3306
    DB_NAME=web7
  DATABASE_URL=mysql://${DB_USER}:${DB_PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}
```

Generate the database, and migrations and migrate:

```
$ php bin/console doctrine:database:create
$ php bin/console doctrine:migrations:diff
$ php bin/console doctrine:migrations:migrate
```

21.9 Generate CRUD for Product and Category

Then generate CRUD for this entity (i.e. a ProductController and some templates in `/templates/product/`):

```
$ php bin/console make:crud Product
$ php bin/console make:crud Category
```

21.10 Add Category selection in Product form

Our generated CRUD for Product creates a Symfony form using method `buildForm(...)` in generated form class `/src/Form/ProductType`:

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('description')
    ->add('image')
    ->add('price');
}
```

We need to add to this form builder a declaration that the Category to be associated with this Product can be set in the form. So we need to add the `category` property to our builder:

```
->add('category', EntityType::class, [
          // list objects from this class
          'class' => 'App:Category',
          // use the 'Category.name' property as the visible option string
          'choice_label' => 'name',
      ]);
```

This references the `EntityType` class, so we need to add a `use` statement for this class:

```
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
```

So the full listing for our updated `ProductType` class is:

```
namespace App\Form;

use App\Entity\Product;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;

class ProductType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('description')
        ->add('image')
        ->add('price')
        ->add('category', EntityType::class, [
                // list objects from this class
                'class' => 'App:Category',

                // use the 'Category.name' property as the visible option string
                'choice_label' => 'name',
            ]);
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Product::class,
        ]);
    }
}
```

21.11 Add some Categories using CRUD

Use your CRUD to create 'small items' and 'large items' Category objects at `/category/new`, and then you are ready to create new Products with their associated Categories.

Done - now we are automatically given a drop-down list of Categories to choose from when we visit `/product/new` to create a new `Product` object. See Figure 21.1.

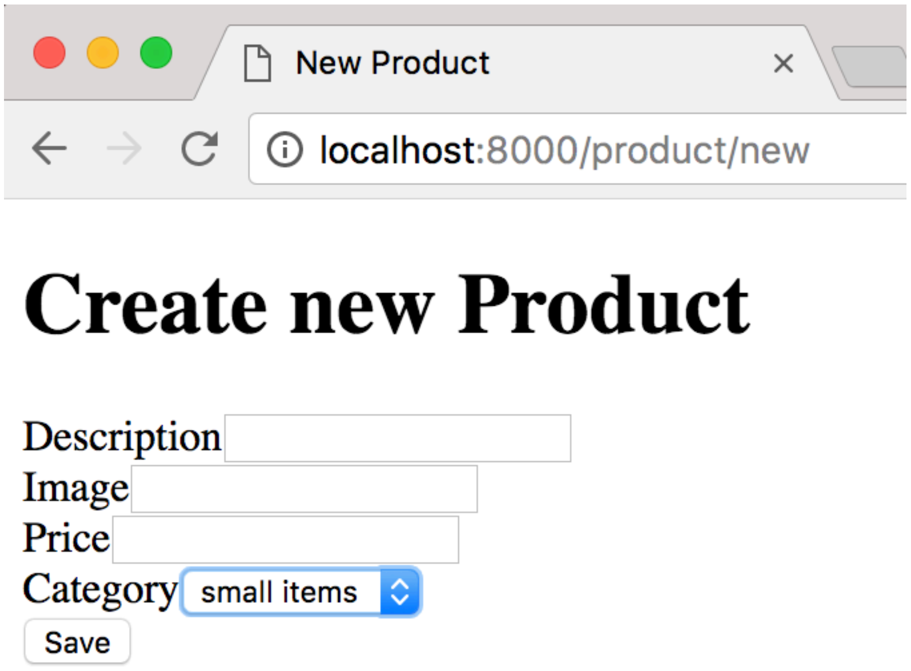


Figure 21.1: Screenshot of Category dropdown for new Product form.

21.12 Adding display of Category to list and show Product

If we have a reference to a `Product` object, in PHP we can get its `Category` as follows:

```
$category = $product->getCategory();

if(null != $category)
    // do something with $category
```

In Twig its even simpler, since the dot-syntax finds the public `getter` automatically:

```
Category = {{ product.category }}
```

So we can update the Product list Twig template to show Category as follows (`/templates/product/index.html.twi`

```
{% for product in products %}
    <tr>
        <td>{{ product.id }}</td>
        <td>{{ product.description }}</td>
        <td>{{ product.image }}</td>
        <td>{{ product.price }}</td>
        <td>{{ product.category.name }}</td>
        ...
```

and add a new column header:

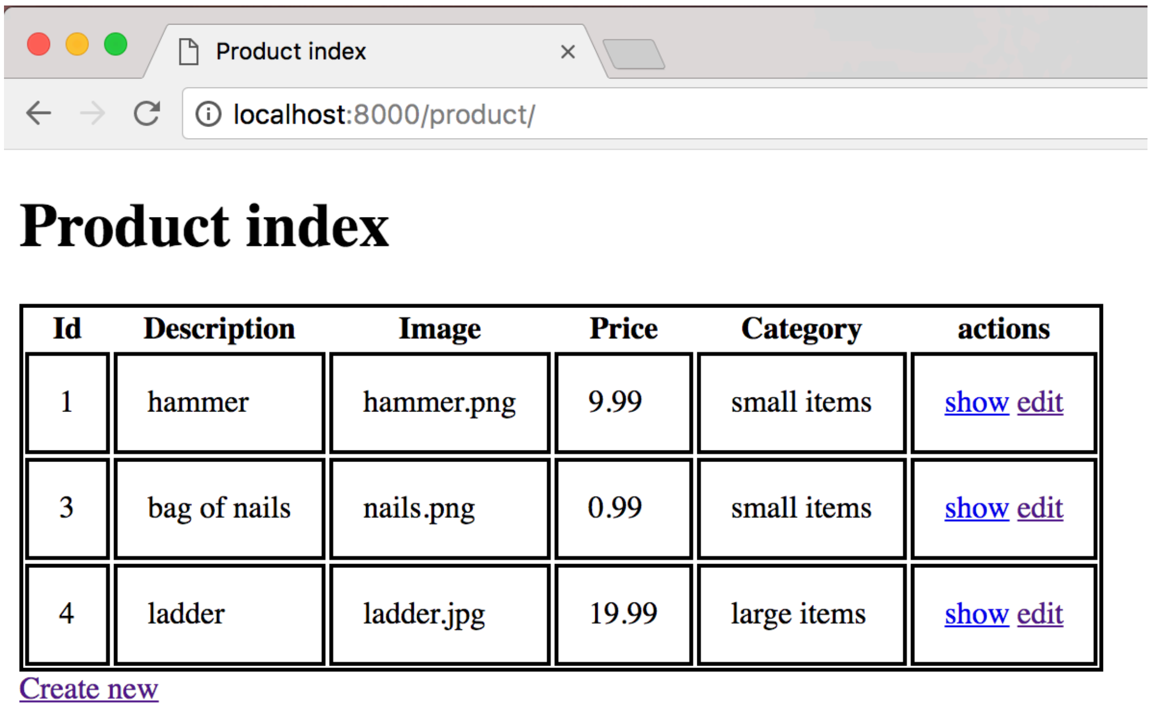
```
<tr>
    <th>Id</th>
    <th>Description</th>
    <th>Image</th>
    <th>Price</th>
    <th>Category</th>
    <th>actions</th>
```

And we can update the Product show Twig template to show Category as follows (`/templates/product/show.html.twig`):

```
<tr>
    <th>Id</th>
    <td>{{ product.id }}</td>
</tr>
<tr>
    <th>Description</th>
    <td>{{ product.description }}</td>
</tr>
<tr>
    <th>Image</th>
    <td>{{ product.image }}</td>
</tr>
<tr>
    <th>Price</th>
    <td>{{ product.price }}</td>
</tr>

<tr>
    <th>Cateogry</th>
    <td>{{ product.category.name }}</td>
</tr>
```

See Figure 21.1 to see Category for each Product in the list.



Product index

Id	Description	Image	Price	Category	actions
1	hammer	hammer.png	9.99	small items	show edit
3	bag of nails	nails.png	0.99	small items	show edit
4	ladder	ladder.jpg	19.99	large items	show edit

Create new

Figure 21.2: Screenshot of list of Products with their Category names.

22

One-to-many (e.g. Products for a single Category)

22.1 Basic list products for current Category (project `associations02`)

First we'll do the minimum to add a list of all the Projects associated with a single Category, then later we'll do it in a nicer way ...

22.2 Add `getProducts()` for Entity Category

We need to add a getter for products in `/src/Entity/Category.php`:

```
public function getProducts()
{
    return $this->products;

}
```

22.3 Add a `__toString()` for Entity Products

We need to add a 'magic method' `__toString()` to Entity Product, since our form builder will need a string for each Product in its list to display:

Add `__toString()`to `/src/Entity/Product.php`. We'll just list `id` and `description`:

```
public function __toString()
{
    return $this->id . ': ' . $this->getDescription();
}
```

22.4 Make Category form type add `products` property

Earlier we added the special `products` property to entity Category, which is the 'many' link to all the Products for the current Category object. We will now add this property to our Category form class `CategoryType`, so that the form created will display all Products found by automatically following that relationship[1].

In `/src/Form/CategoryType.php` add `add('products')` to our `buildForm(```)` method:

```
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name')
            ->add('products')
        ;
    }
```

If we visit the 'edit' page for a Category now, we can see a read-only multiple value list box displayed for `products`, with all Products for the current Category selected.

While it doesn't look very nice, our inverse-relationship is all working fine.

See Figure 22.1.

[1]I.e. Doctrine will magically run something like 'SELECT * FROM product WHERE product.category = category.id' for the current Category object.

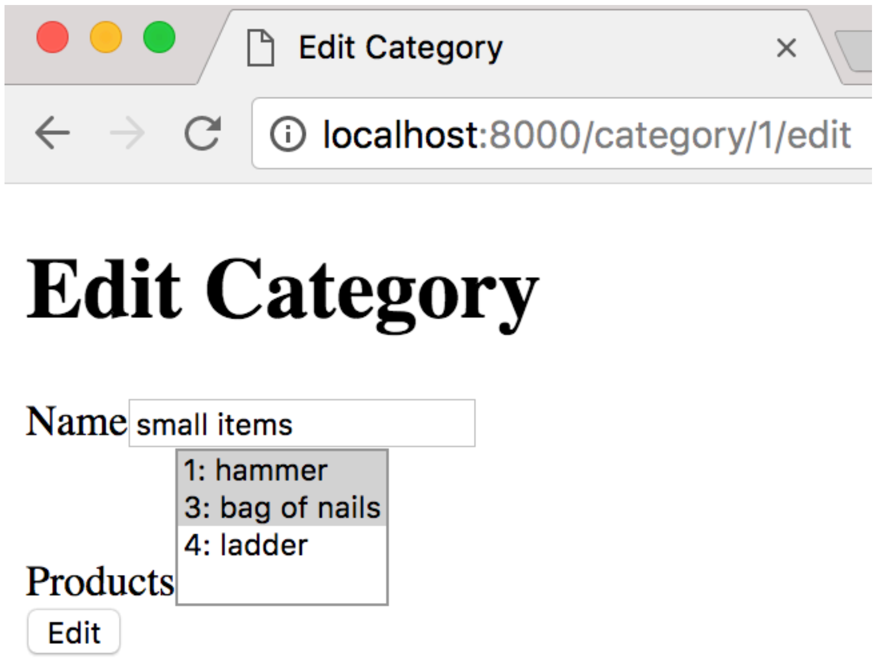


Figure 22.1: Screenshot of products list for Category edit.

22.5 Adding a nicer list of Products for Category show page

Let's add a **nice** list of Products for a Category on the Category show page.

In the Twig `/src/templates/show.html.twig` we have a reference to the `category` object. We can get an array of associated `Product` objects by writing simply `category.products`, so we can loop through this:

```
{% for product in category.products %}
    {{ product.id }} :: {{ product.description }}
    <br>
{% else %}
    (no products for this category)
{% endfor %}
```

This will output some HTML like this:

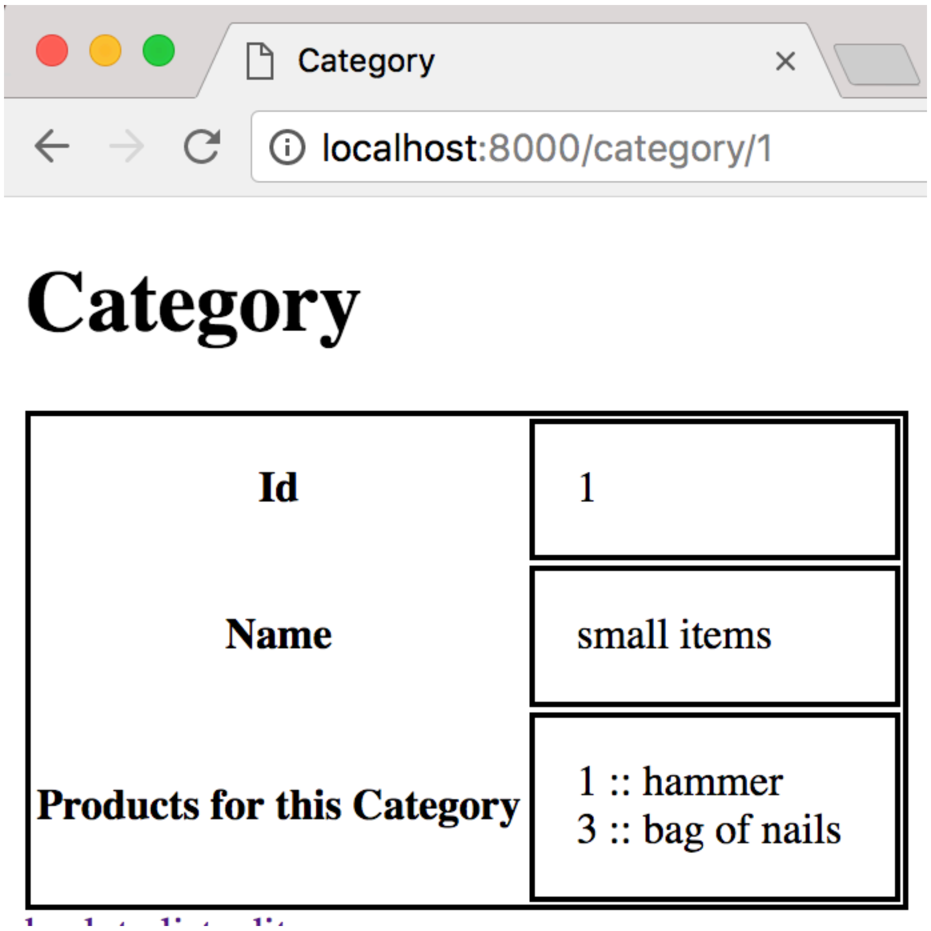
```
1 :: hammer
<br>
3 :: bag of nails
<br>
```

So we can simply add a new HTML table row in our `show.html.twig` template, listing Products as follows:

```
        <tr>
        <th>Id</th>
        <td>{{ category.id }}</td>
    </tr>
    <tr>
        <th>Name</th>
        <td>{{ category.name }}</td>
    </tr>

    <tr>
        <th>Products for this Category</th>
        <td>
            {% for product in category.products %}
                {{ product.id }} :: {{ product.description }}
                <br>
            {% else %}
                (no products for this category)
            {% endfor %}
        </td>
    </tr>
```

See Figure 22.2.



Category

Id	1
Name	small items
Products for this Category	1 :: hammer 3 :: bag of nails

Figure 22.2: Screenshot of improved Category show page.

22.6 Improving the Edit form (project `associations03`)

That multi-selection form element was not very nice for our Edit/New forms.

Let's refactor template `/templates/category/_form.html.twig` to display the list of products for a Category in a nicer way.

Our Twig form did contain:

```
{{ form_start(form) }}
    {{ form_widget(form) }}

    <button>{{ button_label|default('Save') }}</button>
{{ form_end(form) }}
```

Since we want to customise how form elements are displayed, we need to replace `{{ form_widget(form) }}` with our own form elements and HTML.

We can still use the default rendering of the `name` property, so replace `{{form_widget(form) }}` with `{{ form_row(form.name) }}`.

Now we have to decide how to render the `products` array. Let's do something very similar to our show form:

```
<div>
    Products for this Category:
    <ul>
    {% for product in form.vars.value.products %}
        <li>
            <a href="{{ url('product_show', {'id':product.id}) }}">
                {{ product.id }} :: {{ product.description }}
            </a>
        </li>
    {% else %}
        <li>
            (no products for this category)
        </li>
    {% endfor %}
    </ul>
</div>
```

As you can see, we can access the array `products` of our Category object with expression:

```
form.vars.value.products
```

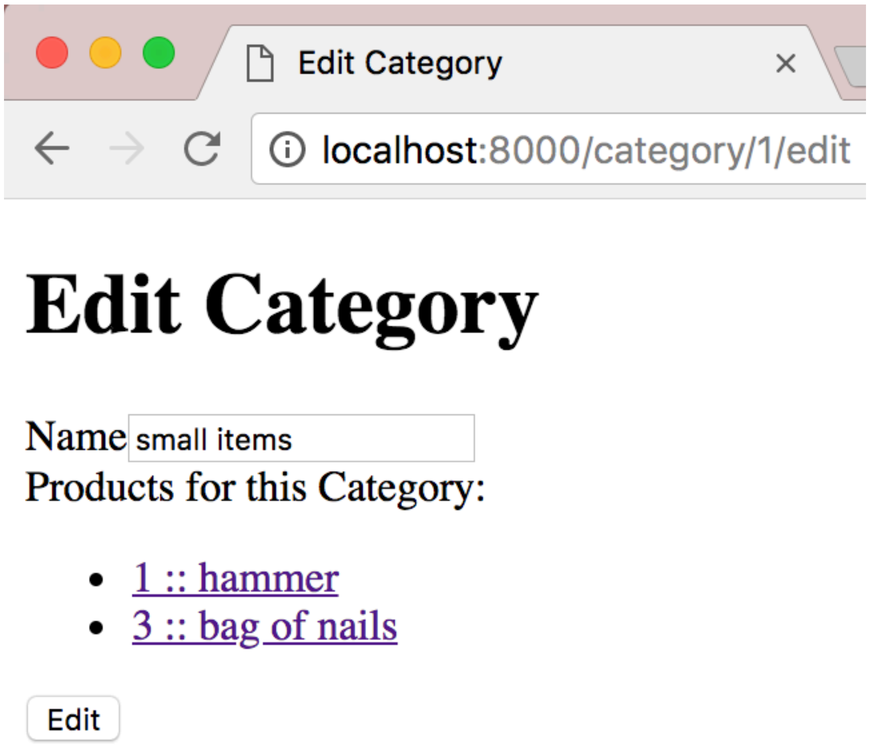
So we can write a `for`-loop around this array.

Note - we still need to render the `products` form widget, otherwise Symfony will end the form

HTML with all properties not yet rendered. So we can **hide** the default rendering for a selection element by wrapping an HTML comment around the default HTML `select` form element:

```
<!--
{{ form_widget(form.products) }}
-->
```

Figure **??** shows a screenshot of our customised Edit form.



Edit Category

Name small items
Products for this Category:

- 1 :: hammer
- 3 :: bag of nails

Edit

Part VIII

PHPDocumentor (2)

23
PHPDocumentor

23.1 Why document code?

There are several reasons to document your code:

1. It makes you **think** about the code

   You might even improve the code having thought about it

2. Writing about code **before** writing the code may lead to better code design
3. It makes you remember **your** code may be used / read by other people
4. It means you don't have to **remember** what things do or why
5. Automated tools can help check for missing documentation
6. It's a handy way to scan your code for TODO comments .

23.2 PHPDocumentor 2

There are several automated tools for supporting PHP code documentation, one of the most popular is PHPDocumentor 2, which is the one described in this chapter.

Learn more about PHPDocumentor 2 at their website:

- phpdoc.org

23.3 Installing PHPDocumentor 2

THe Composer install is a bit big, so it is recommended to just add the PHAR (PHP Archive) either to your project, or globally (somewhere in your system path).

Download the PHAR from their website:

- phpdoc.org/phpDocumentor.phar

23.4 DocBlock comments

The PHPDocumentor is driven by analysing 'DocBlock' comments in your code. A DocBlock comment looks as follows:

```
/**
 * This is a DocBlock.
 */
public function indexController()
{
}
```

They are multi-line comments that start with a double asterisk `/**`.

23.5 Generating the documentation

The PHPDocumentor needs to know at least 2 things:

- where is the PHP source code containing the documentation comments
- where do you want the documentation files to be output

These are specified using the `-d` (PHP source directory), and `-t` (output director) as follows.

So, for example, so analyse **all** files in directory `/src` and output to `/docs` write:

```
$ php phpdoc -d src -t docs
```

To limited the code analysed to just `/src/Controller`, `/src/Util` and `/src/Entity` we would give 3 `-d` arguemnts as follows:

```
php phpdoc.phar -d src/Controller -d src/Entity -d src/Util -t docs
```

Part IX

Symfony Testing

24

Unit testing in Symfony

24.1 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

24.2 Installing Simple-PHPUnit (project `test01`)

Symfony has as special 'bridge' to work with PHPUnit. Add this to your project as follows:

```
$ composer req --dev simple-phpunit
```

You should now see a `/tests` directory created. 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
namespace App\Test;

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`
- 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\Test`
  - 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\Controller\Test`
  - so our testing class architecture directly matches our source code architecture

24.3 Completing the installation

The first time you run Simple-PHPUnit it will probably need to install some more files.

There is an executable file in `/vendor/bin` named `simple-phpunit`. To run it just type `vendor/bin/simple-phpunit` (or for Windows, to run the BATch file, type `vendor\bin\simple-phpunit` - with backslashes since this is a Windows file path):

```
$ vendor/bin/simple-phpunit
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies
Package operations: 19 installs, 0 updates, 0 removals
  - Installing sebastian/recursion-context (2.0.0): Loading from cache
  ...
  - Installing symfony/phpunit-bridge (dev-master): Symlinking from /Users/matt/Library/Mobile Docu
Writing lock file
Generating optimized autoload files
```

NOTE: Error message about missing `ext-mbstring`:

- if you get a message about "ext mbstring" required - when trying to work in Windows with Simple PHP Unit or make:
- simple solution - in your php.ini file
  - Just as you did for pdo_mysql, remove the semi-colon in front of the statement in the php.ini file:
  - e.g. change `;extension=mbstring` to: `extension=mbstring`

24.4 Running Simple-PHPUnit

Let's run the tests (using the default configuration settings, in `phpunit.dist.xml`):

```
$ vendor/bin/simple-phpunit
PHPUnit 5.7.27 by Sebastian Bergmann and contributors.
Testing Project Test Suite
.                                                                  1 / 1 (100%)

Time: 93 ms, Memory: 4.00MB
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%.

24.5 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 subtractingt the second from the first

so our `Calculator` class is as follows:

```
<?php
namespace App\Util;

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

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

24.6 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/Calculator.php`. And since the namespace of our source code class was `App\Util` then the namespace of our testing class will be `App\Util\Test`. 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 class `CalculatorTest`:

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

```
namespace App\Util\Test;

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:

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

```
Testing Project Test Suite
...                                                                 3 / 3 (100%)

Time: 64 ms, Memory: 4.00MB
OK (3 tests, 3 assertions)
```

24.7 Using a data provider to test with multiple datasets (project `test03`)

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)
```

24.8 Configuring testing reports (project `test04`)

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>
    <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>
```

Figure 24.1 shows a screenshot of the contents of the created `/build` directory after Simple-PHPUnit has been run.

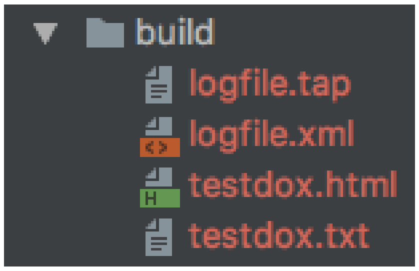


Figure 24.1: Contents of directory `/build`.

The `.txt` file version of **testdox** 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:

```
App\Test\Simple
 [x] One plus one equals two

App\Util\Test\Calculator
 [x] Can create object
 [x] Add one and one
 [x] Additions with provider
```

Another easy to understand logging format is the TAP (Test-Anywhere Protocol). Although official deprecated by PHPUnit it still seems to work. What is nice about the TAP format is that repeated invocations of test methods iterating through a data-provider are enumerated, with the values. So we can see how many times, and their successes, a method was invoked with test data. This file is named (by our XML configuration above) `logfile.tap`:

```
  TAP version 13
  ok 1 - App\Test\SimpleTest::testOnePlusOneEqualsTwo
  ok 2 - App\Util\Test\CalculatorTest::testCanCreateObject
  ok 3 - App\Util\Test\CalculatorTest::testAddOneAndOne
ok 4 - App\Util\Test\CalculatorTest::testAdditionsWithProvider with data set #0 (1, 1, 2)
ok 5 - App\Util\Test\CalculatorTest::testAdditionsWithProvider with data set #1 (2, 2, 4)
ok 6 - App\Util\Test\CalculatorTest::testAdditionsWithProvider with data set #2 (0, 1, 1)
  1..6
```

24.9 Testing for exceptions (project `test07`)

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:

```php
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`:

```php
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...`:

```php
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 5.7.27 by Sebastian Bergmann and contributors.

Warning:       Deprecated TAP test listener used

Testing Project Test Suite
.........E                                                        10 / 10 (100%)

Time: 1.21 seconds, Memory: 10.00MB

There was 1 error:

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

.../src/Util/Calculator.php:21
/Users/matt/Library/Mobile Documents/com~apple~CloudDocs/11_Books/symfony/php-symfony4-book-

ERRORS!
Tests: 10, Assertions: 9, Errors: 1.
```

And our logs will confirm the failure:

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

App\Test\Simple
 [x] One plus one equals two

App\Util\Test\Calculator
 [x] Can create object
 [x] Add one and one
 [x] Additions with provider
```

```
[x] Divide one and one
[ ] Divide one and zero
```

24.10 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:

```
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 5.7.27 by Sebastian Bergmann and contributors.

Warning:       Deprecated TAP test listener used

Testing Project Test Suite
..........                                                        10 / 10 (100%)
```

24.11 PHPUnit annotation comment `@expectedException`

PHPUnit allows us to use an annotation comment to state that we expect an exception to be thrown during the execution of a particular test. This is a nice way to keep our test logic simple.

Since annotation comments are declared immediately **before** the method, some programmers (I do!) prefer the annotation way of declaring that we expect a test method to result in an exception being thrown:

```php
/**
 * @expectedException \InvalidArgumentException
 */
public function testDivideOneAndZeroAnnotation()
{
    // Arrange
    $calculator = new Calculator();
    $num1 = 1;
    $num2 = 0;

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

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

NOTE: You must ensure the exception class is fully namespaced in the annotation comment (no `::class` shortcuts!).

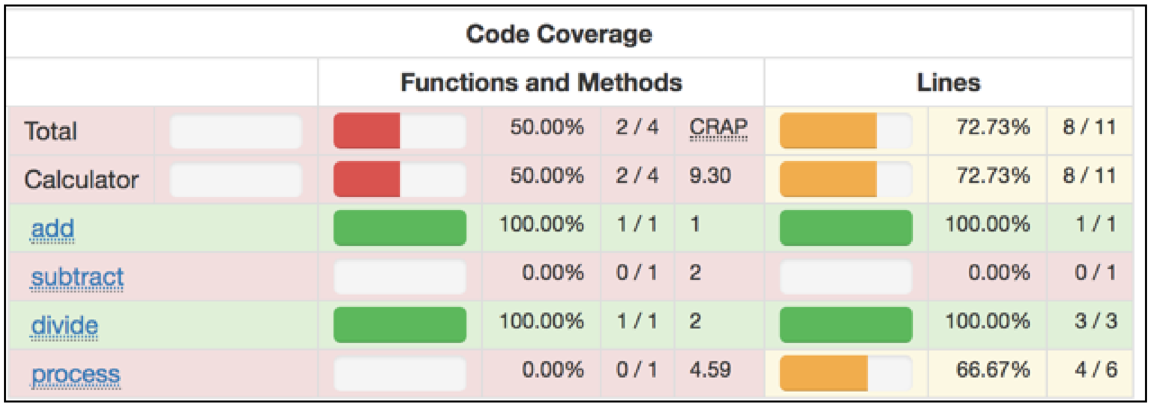
25

Code coverage and xDebug

25.1 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 2.2 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.



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 25.1: Screenshot of HTML coverage report.

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 P for these stesp.

25.2 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 25.2.

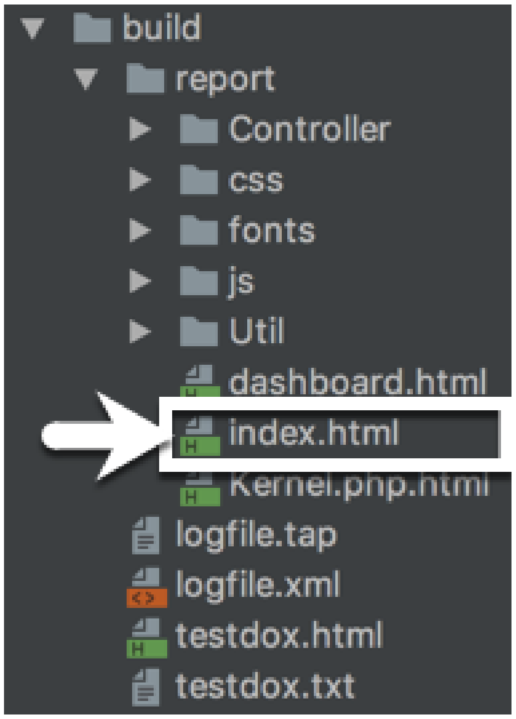


Figure 25.2: Build files showing `index.html` in `/build/report`.

25.3 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>
```

26 Web testing

26.1 Testing controllers with `WebTestCase` (project `test05`)

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

First we need to add 2 packages to the project development environment:

```
composer req --dev browser-kit css-selector
```

Note - these next steps assume your project has Twig, annotations and the Symfony maker packages available, so you may need to add these to your project as well:

```
composer req twig annotations make
```

Let's make a new `DefaultController` class:

```
php bin/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' %}

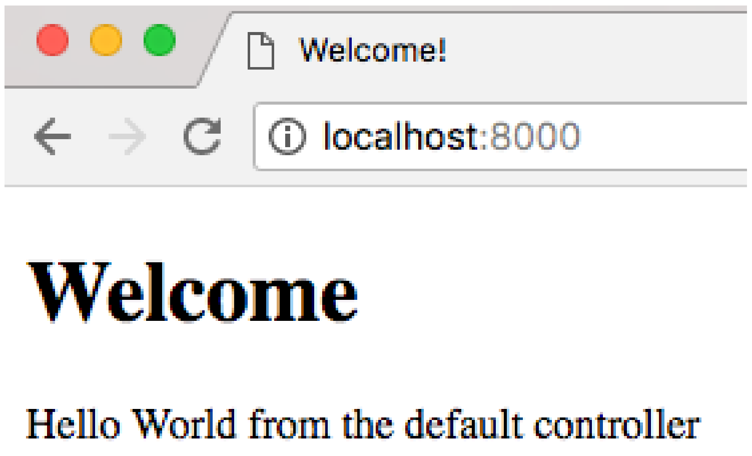
{% block body %}
<h1>Welcome</h1>

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

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

```
class DefaultController extends Controller
{
    /**
     * @Route("/", name="default")
     */
    public function index()
    {
        return $this->render('default/index.html.twig', [
            'controller_name' => 'DefaultController',
        ]);
    }
}
```

If we run a web server and visit the home page we should see our 'hello world' message in a browser - see Figure 26.1.



Welcome

Hello World from the default controller

Figure 26.1: Contents of directory /build.

26.2 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 Reponse 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 Reponse code is as follows:

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

    // Assert
    $client->request($httpMethod, $url);

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

We see how a web client object `$client` is created and makes a GET requerst to `/`. We see how we can interrogate the contents of the HTTP Response recevied 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 200 OK Reponse code is as follows:

```
public function testHomepageContentContainsHelloWorld()
{
    // Arrange
    $url = '/';
    $httpMethod = 'GET';
    $client = static::createClient();
    $searchText = 'Hello World';

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

```
        // Assert
        $this->assertContains(
            $searchText,
            $client->getResponse()->getContent()
        );
    }
```

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

When we run Simple-PHPUnit we can see success both from the full-stops at the CLI, and in our log files, e.g.:

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

...
```

26.3 Normalise content to lowercase (project `test06`)

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

This kind of problem can be avoided if we **normalise** the content from the Response, e.g. making all letters **lower-case**. This only makes sense if you are happy (at this stage) to not worry about the case of text content in your pages (you could always write some specific spelling / grammar checker tests for that …)

The solution is to use the built-in PHP function `strtolower()`:

```
public function testHomepageContentContainsHelloWorld()
{
    // Arrange
    $url = '/';
    $httpMethod = 'GET';
    $client = static::createClient();
    $searchText = 'Hello World';

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

```
        // to lower case
        $searchTextLowerCase = strtolower($searchText);
        $contentLowerCase = strtolower($content);

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

26.4 Test multiple pages with a data provider

Avoid duplicating code when only the values change, by writing a testing method fed by arrays of test input / expected values from a data provider method:

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

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

        // to lower case
        $contentLowerCase = strtolower($content);

        // Assert - status code 200
        $this->assertSame(Response::HTTP_OK, $statusCode);

        // Assert - expected content
        $this->assertContains(
            $exepctedLowercaseText,
            $contentLowerCase
        );
    }

    public function basicPagesTextProvider()
    {
        return [
            ['/', 'home page'],
            ['/about', 'about'],
        ];
    }
```

26.5 Testing links (project test08)

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

  ```
  $content = $client->getResponse()->getContent();

  // set $expectedText to what should in page when link has been followed ...
  $this->assertContains(
      $exepctedText,
      $content
  );
  ```

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 aboutAction()
    {
        $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 %}
            {% block javascripts %}{% 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();
    $searchText = 'about page';
    $linkText = 'about';

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

    // to lower case
    $searchTextLowerCase = strtolower($searchText);
    $contentLowerCase = strtolower($content);

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

27 Testing web forms

27.1 Testing forms (project `test09`)

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("/calc", name="calc_home")
     */
```

```
public function indexAction()
{
    return $this->render('calc/index.html.twig', []);
}
```

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

```
/**
 * @Route("/calc/process", name="calc_process")
 */
public function processAction(Request $request)
{
    // 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);

    return $this->render(
        'calc/result.html.twig',
        [
            'n1' => $n1,
            'n2' => $n2,
            'operator' => $operator,
            'answer' => $answer
        ]
    );
}
```

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

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

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

    <form method="post" action="{{ url('calc_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 }}
```

27.2 Test we can get a reference to the form

Let's test that can see the form page

```
public function testHomepageResponseCodeOkay()
{
    // Arrange
    $url = '/calc';
    $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 = '/calc';
    $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">
```

27.3 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 = '/calc';
    $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 & make lower case
    $content = $client->getResponse()->getContent();
    $contentLowerCase = strtolower($content);

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

27.4 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 = '/calc';
    $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();
    $contentLowerCase = strtolower($content);

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

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

27.5 Testing we get the correct result via form submission

Assuming all our `Calculator`, methods have been inidividually **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 = '/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

    // (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
```

```
        $this->assertContains($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.

27.6 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
        $this->assertContains($expectedResultString, $content);
    }
```

Part X

Appendices

A

Software required for Symfony development

A.1 Don't confuse different software tools

Please do not confuse the following:

- Git and Github
- PHP and PHPStorm

Here is a short description of each:

- Git: A version control system - can run locally or on networked computer. There are several website that support Git projects, including:
    - Github (perhaps the most well known)
    - Gitlab
    - Bitbucket
    - you can also create and run your own Git web server …
- Github: A commercial (but free for students!) cloud service for storing and working with projects using the Git version control system
- PHP: A computer programming language, maintained by an international Open Source community and published at `php.net`
- PHPStorm: A great (and free for student!) IDE - Interactive Development Enviroment. I.e. a really clever text editor created just for working with PHP projects. PHPStorm is one of the professional software tools offered by the **Jetbrains** company.

So in summary, Git and PHP are open source core software. Github and PHPStorm are commercial (but free for students!) tools that support development using Git and PHP.

A.2 Software tools

Ensure you have the following setup for developing Symfony software on your local machine

- PHP 7.1.5 or later (free, open source)
- Composer (up-to-date with `composer self-update`)(free, open source - a PHP program!)
- PHPStorm (with educational free account if you're a student!) - or some other editor of your choice
- MySQL Workbench (Community Edition free)
- Git (free, open source)

See Appendix B for checking, and if necessary, installing PHP on your computer. See Appendix A for details about other software needed for working with PHP projects.

A.3 Test software by creating a new Symfony 4 project

Test your software by using PHP and Composer to create a new Symfony 4 project. We'll follow the steps at the Symfony setup web page.

Follow the steps in Appendix **??**.

B

PHP Windows setup

B.1 Check if you have PHP installed and working

You need PHP version 7.1.3 or later.

Check your PHP version at the command line with:

```
> php -v
PHP 7.1.5 (cli) (built: May  9 2017 19:49:10)
Copyright (c) 1997-2017 The PHP Group
Zend Engine v3.1.0, Copyright (c) 1998-2017 Zend Technologies
```

If your version is older than 7.1.5, or you get an error about command not understood, then complete the steps below.

B.1.1 Download the latest version of PHP

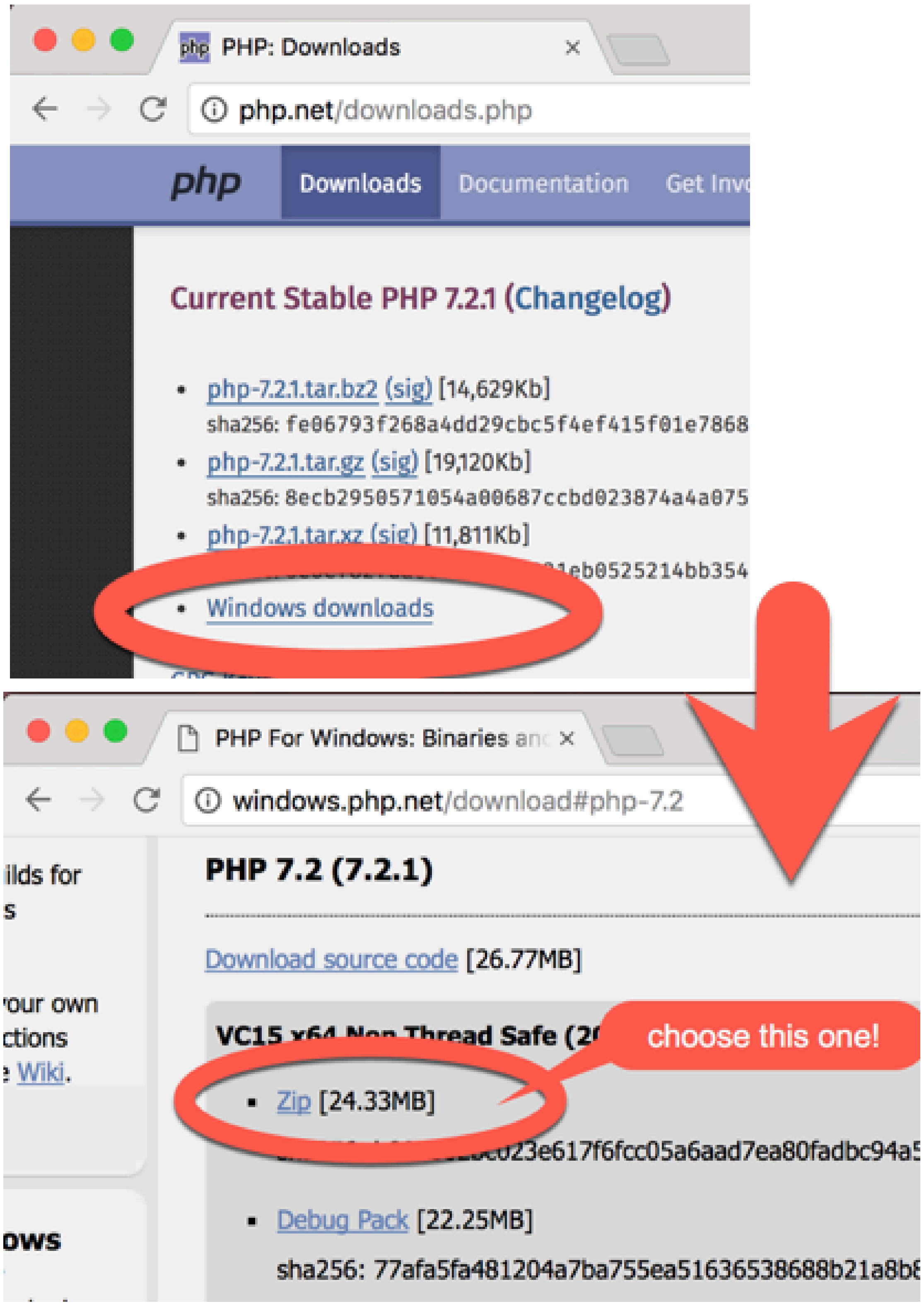
Get the latest (7.2.1 at the time of writing) PHP Windows ZIP from:

- php.net click the **Windows Downloads** link

Figure B.1 shows a screenshot of the `php.net` general and Windows downloads page. The `ZIP` file to download (containing `php.exe` … don't download the source code version unless you want to build the file from code …):

Do the following:

- unzip the PHP folder into: `C:\php`


PHP: Downloads

php.net/downloads.php

php Downloads Documentation Get Inv

Current Stable PHP 7.2.1 (Changelog)

- php-7.2.1.tar.bz2 (sig) [14,629Kb]
  sha256: fe06793f268a4dd29cbc5f4ef415f01e7868
- php-7.2.1.tar.gz (sig) [19,120Kb]
  sha256: 8ecb2950571054a00687ccbd023874a4a075
- php-7.2.1.tar.xz (sig) [11,811Kb]
  1eb0525214bb354
- Windows downloads

PHP For Windows: Binaries an

windows.php.net/download#php-7.2

PHP 7.2 (7.2.1)

Download source code [26.77MB]

VC15 x64 Non Thread Safe (2

- Zip [24.33MB]
  023e617f6fcc05a6aad7ea80fadbc94a5
- Debug Pack [22.25MB]
  sha256: 77afa5fa481204a7ba755ea51636538688b21a8b

Figure B.1: PHP.net / Windows ZIP download pages.

- so you should now have a file `php.exe` inside `C:\php`, along with lots of other files
- make a copy the file `C:\php\php.ini-development`, naming the copy `C:\php\php.ini`
- open a new terminal CLI window (so new settings are loaded) and run `php --ini` to confirm the location of the `php.ini` file that you've just created. Note the following for a Mac - for Windows it should (hopefully) tell you it found the ini file in `c:\php\php.ini`:

```
$ php --ini
Configuration File (php.ini) Path: /Applications/MAMP/bin/php/php7.1.8/conf
Loaded Configuration File:         /Applications/MAMP/bin/php/php7.1.8/conf/php.ini
Scan for additional .ini files in: (none)
Additional .ini files parsed:      (none)
```

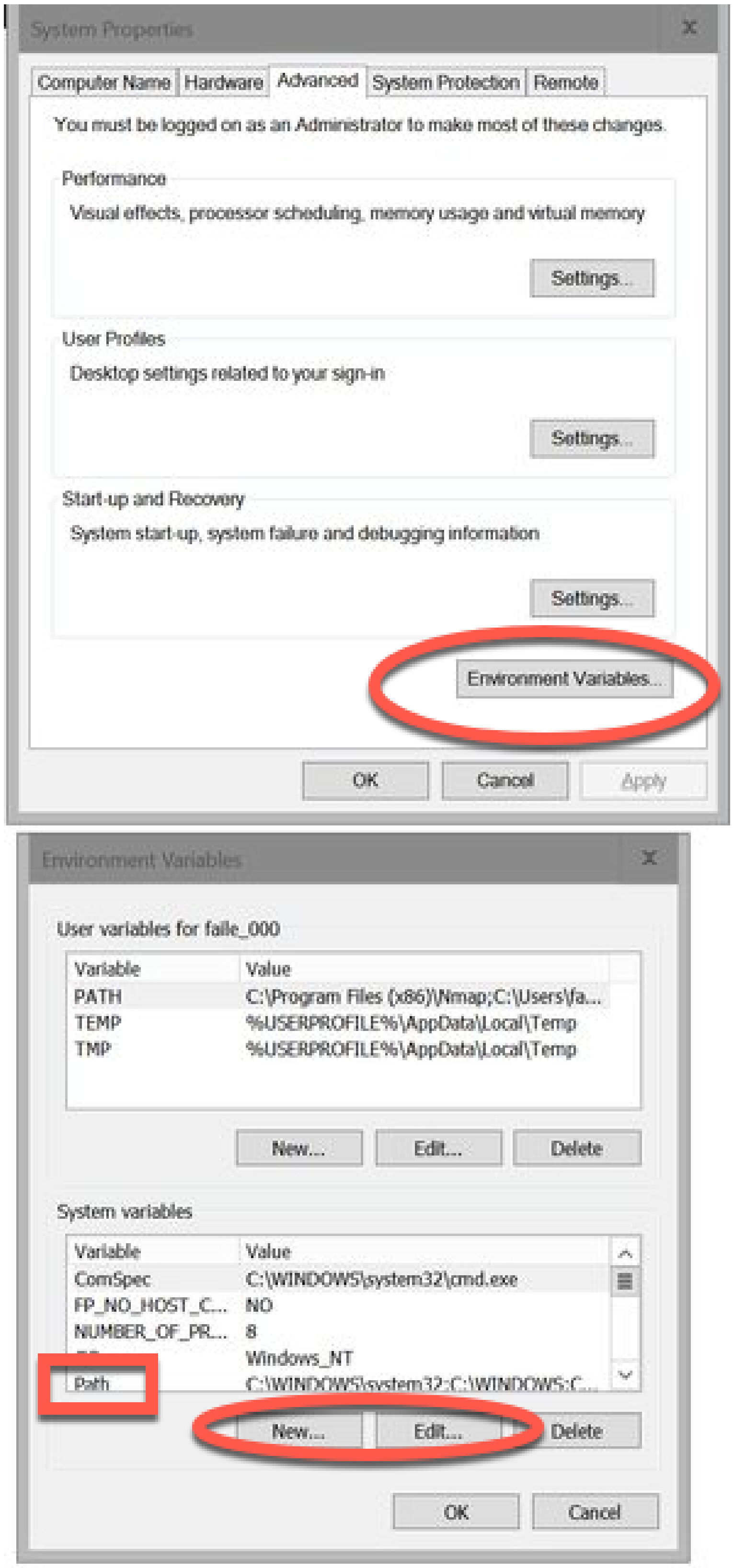
B.2 Add the path to `php.exe` to your System environment variables

Whenever you type a command at the CLI (Command Line Interface) Windows searches through all the directories in its `path` environment variable. In order to use PHP at the CLI we need to add `c:\php` to the `path` environment variable so the `php.exe` executable can be found.

Via the System Properties editor, open your Windows Evironment Variables editor. The **system** environment variablesa re in the lower half of the Environment Variables editor. If there is already a system variable named `Path`, then select it and click the **Edit** button. If none exists, then click the **New** button, naming the new variable **path**. Add a new value to the **path** variable with the value `c:\php`. Then click all the **Okay** buttons needed to close all these windows.

Now open a windows **Cmd** window and try the `php -v` - hopefully you'll see confirmation that your system now has PHP installed and in the **path** for CLI commands.

Figure B.2 shows a screenshot of the Windows system and environment variables editor.



System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage and virtual memory

Settings...

User Profiles

Desktop settings related to your sign-in

Settings...

Start-up and Recovery

System start-up, system failure and debugging information

Settings...

Environment Variables...

OK Cancel Apply

Environment Variables

User variables for faile_000

Variable	Value
PATH	C:\Program Files (x86)\Nmap;C:\Users\fa...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

New... Edit... Delete

System variables

Variable	Value
ComSpec	C:\WINDOWS\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_PR...	8
OS	Windows_NT
Path	C:\WINDOWS\system32;C:\WINDOWS;C...

New... Edit... Delete

OK Cancel

Figure B.2: The Windows Environment Variables editor.

B.3 PHP Info & SQL driver test

For database work we need to enable the PDO[1] options for MySQL and SQLite (see later database exercises for how to do this)

Although PHP may have been installed, and its SQL drivers too, they may have not been enabled. For this module we'll be using the SQLite and MySQL drivers for PHP – to talk to databases. The function `phpinfo()` is very useful since it displays many of the settings of the PHP installation on your computer / website.

1. In the current (or a temporary) direcotry, create file `info.php` containing just the following 2 lines of code:

   ```
   <?php
   print phpinfo();
   ```

2. At the CLI run the built-in PHP web server to serve this page, and visit: localhost:8000/info.php in your web browser

   ```
   php -S localhost:8000
   ```

In the PDO section of the web page (`CTL-F` and search for `pdo` ...) we are looking for **mysql** and **sqlite**. If you see these then great!

Figure B.3 shows a screenshot the Windows system and environment variables editor.

But, if you see "no value" under the PDO drivers section, then we'll need to edit file `c:\php\php.ini`:

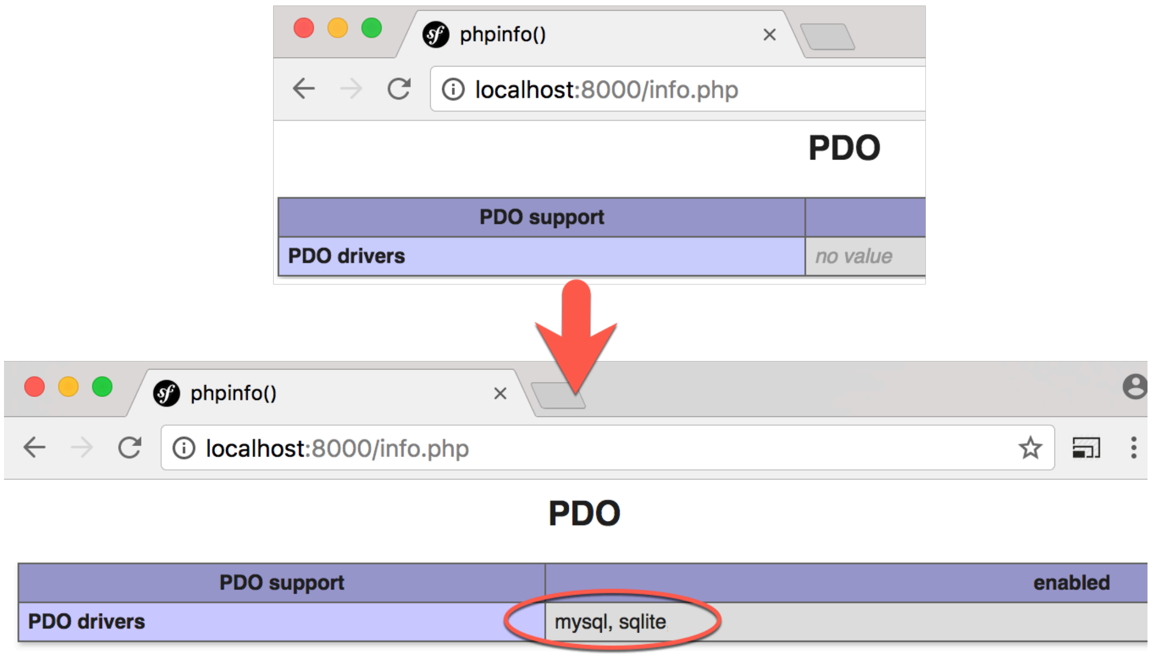
1. In a text editor open file `c:\php\php.ini` and locate the "Dynamic Extensions" section in this file (e.g. use the editor Search feature - or you could just search for `pdo`)

2. Now remove the semi-colon `;` comment character at the beginning of the lines for the SQLite and MySQL DLLs to enable them as shown here:

   ```
   ;;;;;;;;;;;;;;;;;;;;;;
   ; Dynamic Extensions ;
   ;;;;;;;;;;;;;;;;;;;;;;

   .. other lines here ...
   extension=php_pdo_mysql.dll <<<<<<<<<< here is the PDO MYSQL driver line
   ;extension=php_pdo_oci.dll
   ;extension=php_pdo_odbc.dll
   ```

[1]PDO = PHP Database Objects, the modern library for managing PHP program communications with databases. Avoid using old libries like `mysql` (security issues) and even `mysqli` (just for MySQL). PDO offers an object-oriented, standardized way to communicate with many different database systems. So a project could change the databse management system (e.g. from Oracle to MySQL to SQLite), and only the database connetion optins need to change - all other PDO code will work with the new database system!



PDO

PDO support	
PDO drivers	no value

PDO

PDO support	enabled
PDO drivers	mysql, sqlite

Figure B.3: The PDO section of the `phpinfo()` information page.

```
;extension=php_pdo_pgsql.dll
extension=php_pdo_sqlite.dll <<<<<<<<  here is the PDO SQLITE driver line
```

3. Save the file. Close your Command Prompt, and re-open it (to ensure new settings are used).

   – Run the webserver again and visit: `localhost:8000/info.php` to check the PDO drivers.

NOTE: Knowing how to view `phpinfo()` is very handy when checking server features.

C

Get/Update your software tools

NOTE: All the following are already available on the ITB college computers. All you may need to do is:

1. ensure that Composer is up to date by running:

   ```
   composer self-update
   ```

2. enable the PDO options for MySQL and SQLite (see Appendix B for how to do this by editing ther `c:\php\php.ini` file …)

C.1 Composer

The Composer tool is actually a **PHAR** (PHP Archive) - i.e. a PHP application packaged into a single file. So ensure you have PHP installed and in your environment **path** before attempting to install or use Composer.

Ensure you have (or install) an up-to-date version of the Composer PHP package manager.

```
composer self-update
```

C.1.1 Windows Composer install

Get the latest version of Composer from

- getcomposer.org

- run the provided **Composer-Setup.exe** installer (just accept all the default options - do NOT tick the developer mode)
  - https://getcomposer.org/doc/00-intro.md#installation-windows

C.2 PHPStorm editor

Ensure you have your free education Jetbrains licence from:

- Students form: https://www.jetbrains.com/shop/eform/students (ensure you use your ITB student email address)

Downdload and install PHPStorm from:

- https://www.jetbrains.com/phpstorm/download/

To save lots of typing, try to install the following useful PHPStorm plugins:

- Twig
- Symfony
- Annotations

C.3 MySQL Workbench

While you can work with SQLite and other database management systems, many ITB modules use MySQLWorkbench for database work, and it's fine, so that's what we'll use (and, of course, it is already installed on the ITB windows computers ...)

Download and install MySQL Workbench from:

- https://dev.mysql.com/downloads/workbench/

C.4 Git

Git is a fantastic (and free!) DVCS - Distributed Version Control System. It has free installers for Windows, Mac, Linus etc.

Check is Git in installed on your computer by typing `git` at the CLI terminal:

```
> git
usage: git [--version] [--help] [-C <path>] [-c name=value]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

```
These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

...

collaborate (see also: git help workflows)
   fetch      Download objects and refs from another repository
   pull       Fetch from and integrate with another repository or a local branch
   push       Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.

>
```

If you don't see a list of **Git** commands like the above, then you need to install Git on your computer.

C.5 Git Windows installation

Visit this page to run the Windows Git installer.

- https://git-scm.com/downloads

NOTE: Do **not** use a GUI-git client. Do all your Git work at the command line. It's the best way to learn, and it means you can work with Git on other computers, for remote terminal sessions (e.g. to work on remote web servers) and so on.

D The fully-featured Symfony 4 demo

D.1 Visit Github repo for full Symfony demo

Visit the project code repository on Github at: https://github.com/symfony/demo

D.2 Git steps for download (clone)

If you have Git setup on your computer (it is on the college computers) then do the following:

- copy the clone URL into the clipboard
- open a CLI (Command Line Interface) window
- navigate (using `cd`) to the location you wish to clone[1]
- use `git clone <url>` to make a copy of the project on your computer

```
lab01 $ git clone https://github.com/symfony/demo.git
Cloning into 'demo'...
remote: Counting objects: 7165, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 7165 (delta 4), reused 8 (delta 2), pack-reused 7150
Receiving objects: 100% (7165/7165), 6.79 MiB | 1.41 MiB/s, done.
Resolving deltas: 100% (4178/4178), done.
```

[1]For a throw-away exercise like this I just create a directry named `temp` (with `mkdir temp`) and `cd` into that...

```
lab01 $
```

D.3 Non-git download

If you don't have Git on your computer, just download and **unzip** the project to your computer (and make a note to get **Git** installed a.s.a.p.!)

D.4 Install dependencies

Install any required 3rd party components by typing `cd`ing into folder `demo` and typing CLI command `composer install`. A **lot** of dependencies will be downloaded and installed!

```
lab01/demo $ composer install
Loading composer repositories with package information
Installing dependencies (including require-dev) from lock file
Package operations: 89 installs, 0 updates, 0 removals
  - Installing ocramius/package-versions (1.2.0): Loading from cache
  - Installing symfony/flex (v1.0.65): Loading from cache
  ...
  - Installing symfony/phpunit-bridge (v4.0.3): Loading from cache
  - Installing symfony/web-profiler-bundle (v4.0.3): Loading from cache
  - Installing symfony/web-server-bundle (v4.0.3): Loading from cache
Generating autoload files
ocramius/package-versions:  Generating version class...
ocramius/package-versions: ...done generating version class
```

D.5 Run the demo

Run the demo with `php bin\console server:run`

(Windows) You may just need to type `bin\console  server:run` since I think there is a `.bat` file in `\bin`:

```
lab01/demo$ php bin/console server:run

 [OK] Server listening on http://127.0.0.1:8000

 // Quit the server with CONTROL-C.
```

```
PHP 7.1.8 Development Server started at Tue Jan 23 08:19:05 2018
Listening on http://127.0.0.1:8000
Document root is /Users/matt/Library/Mobile Documents/com~apple~CloudDocs/91_UNITS/UNITS_PHP_4_frmw
Press Ctrl-C to quit.
```

D.6 View demo in browser

Open a browswer to `localhost:8000` and play around with the website. Figure D.1 shows a screenshot of the default Symfony page for a new, empty project.

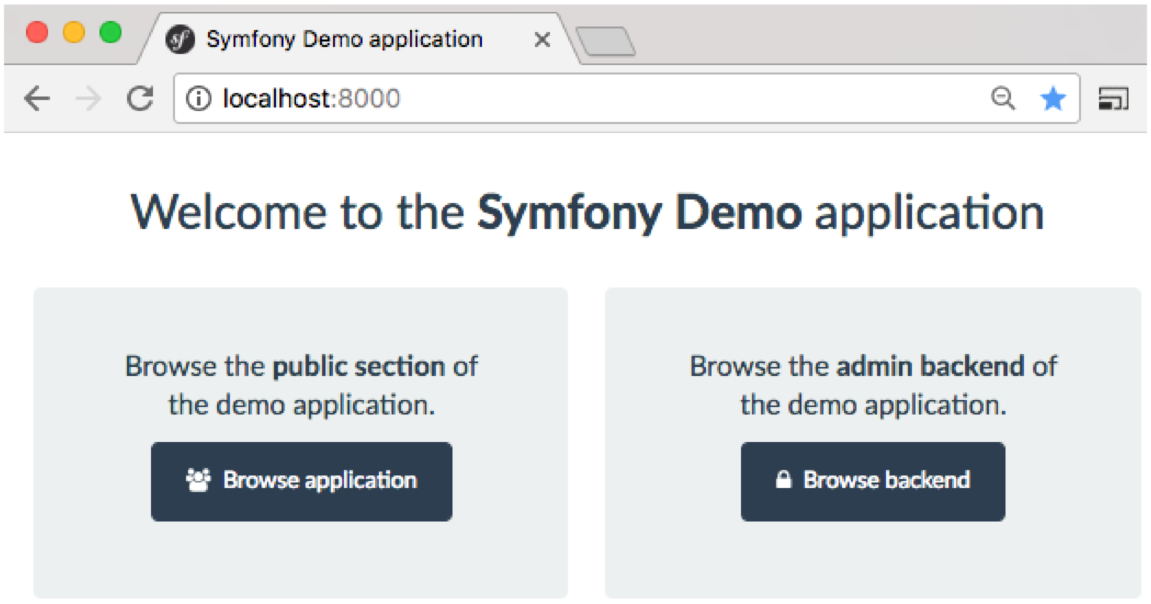

Symfony Demo application
localhost:8000
Welcome to the Symfony Demo application
Browse the public section of the demo application.
Browse application
Browse the admin backend of the demo application.
Browse backend

Figure D.1: Default Symfony 4 demo project

D.7 Explore the code in PHPStorm

Open the code for the project in PHPStorm, and look especially in the `/controllers` and `/templates` directories, to work out what is going on

D.8 Switch demo from SQLite to MySQL

At present the Symfony demo uses the SQLite driver, working with a databse 'file' in `/var/data`.

Let's change this project to work with a MySQL database schema named `demo`.

Do the following:

1. Run MySQL Workbench
2. Change the **URL** for the projects data in `.env` from:

   `DATABASE_URL=sqlite:///%kernel.project_dir%/var/data/blog.sqlite`

   to

   `DATABASE_URL="mysql://root:pass@127.0.0.1:3306/demo"`
3. Get the Symfony CLI to create the new database schema, type `php bin/console doctrine:database:create`:

   ```
   demo (master) $ php bin/console doctrine:database:create
   Created database `demo` for connection named default

   demo (master) $
   ```
4. Get the Symfony CLI to note any changes that need to happen to the databast to make it match Entites and relationships defined by the project's classes, by typing `php bin/console doctrine:migrations:diff`:

   ```
   demo (master) $ php bin/console doctrine:migrations:diff
   Generated new migration class to "/Users/matt/Library/Mobile Documents/com~apple~CloudDocs/

   demo (master) $
   ```

   A migration file has now been created.
5. Run the migration file, by typing `php bin/console doctrine:migrations:migrate` and then typing `y`:

"'bash demo (master) $ 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
    Migrating up to 20180127081633 from 0

      ++ migrating 20180127081633

      -> CREATE TABLE symfony_demo_comment (id INT AUTO_INCREMENT NOT NULL, post_id INT NOT NULL, au
      -> CREATE TABLE symfony_demo_post (id INT AUTO_INCREMENT NOT NULL, author_id INT NOT NULL, tit
      -> CREATE TABLE symfony_demo_post_tag (post_id INT NOT NULL, tag_id INT NOT NULL, INDEX IDX_6A
      -> CREATE TABLE symfony_demo_tag (id INT AUTO_INCREMENT NOT NULL, name VARCHAR(255) NOT NULL, 
      -> CREATE TABLE symfony_demo_user (id INT AUTO_INCREMENT NOT NULL, full_name VARCHAR(255) NOT 
      -> ALTER TABLE symfony_demo_comment ADD CONSTRAINT FK_53AD8F834B89032C FOREIGN KEY (post_id) 
      -> ALTER TABLE symfony_demo_comment ADD CONSTRAINT FK_53AD8F83F675F31B FOREIGN KEY (author_id
```

```
   -> ALTER TABLE symfony_demo_post ADD CONSTRAINT FK_58A92E65F675F31B FOREIGN KEY (author_id) REFERENC
   -> ALTER TABLE symfony_demo_post_tag ADD CONSTRAINT FK_6ABC1CC44B89032C FOREIGN KEY (post_id) REFERE
   -> ALTER TABLE symfony_demo_post_tag ADD CONSTRAINT FK_6ABC1CC4BAD26311 FOREIGN KEY (tag_id) REFEREN

   ++ migrated (0.44s)

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

   ++ finished in 0.44s
   ++ 1 migrations executed
   ++ 10 sql queries

demo (master) $
```

"'

D.9 Running the tests in the SF4 demo

The project comes with configuration for `simple-phpuit`. Run this once to download the depedencies:

```
lab01/demo $ vendor/bin/simple-phpunit
   ./composer.json has been updated
   Loading composer repositories with package information
   Updating dependencies
   Package operations: 19 installs, 0 updates, 0 removals
     - Installing sebastian/recursion-context (2.0.0): Loading from cache
     ...
     - Installing symfony/phpunit-bridge (5.7.99): Symlinking from /Users/matt/lab01/demo/vendor/sy
   Writing lock file
   Generating optimized autoload files

lab01/demo $
```

D.10 Run the tests

Run the tests, by typing `vendor\bin\simple-phpunit`[2]

[2]The backlash-forward slash thing is annoying. In a nutshell, for file paths for Windows machines, use backslashes, for everything else use forward slashes. So it's all forward slashes with Linux/Mac machines :-)

```
lab01/demo $ vendor/bin/simple-phpunit
PHPUnit 5.7.26 by Sebastian Bergmann and contributors.

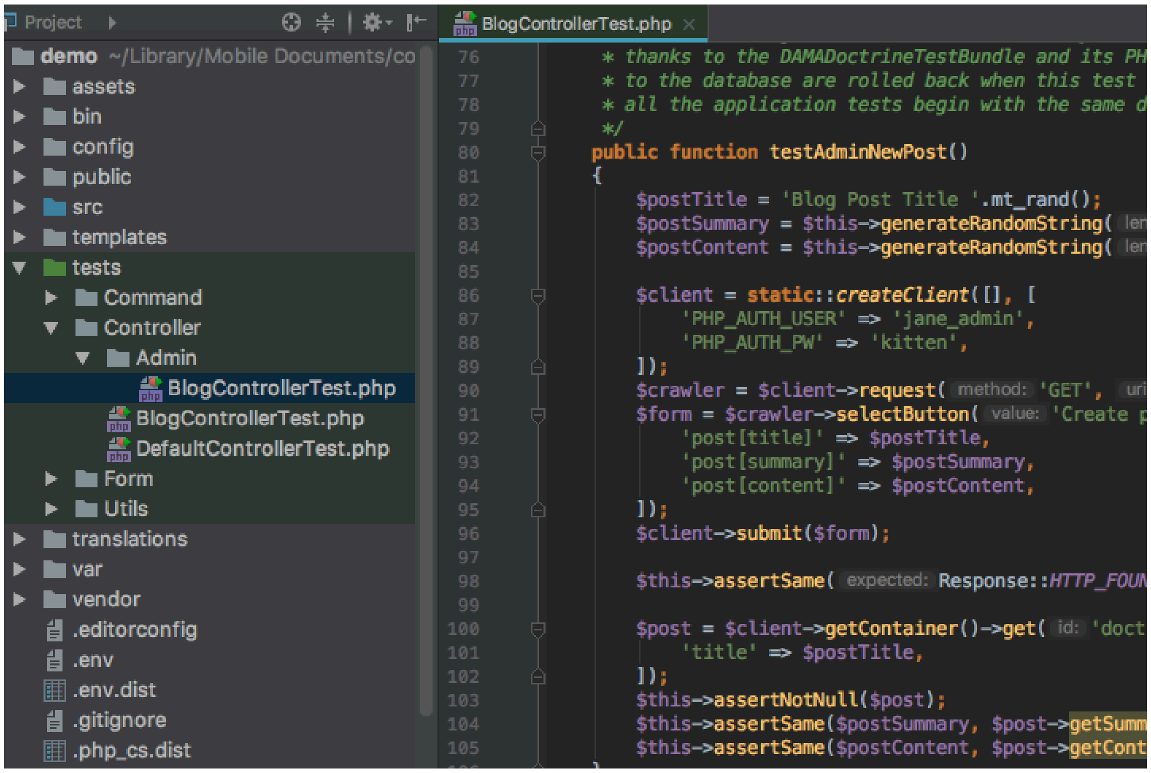
Testing Project Test Suite
.................................................                    49 / 49 (100%)

Time: 27.65 seconds, Memory: 42.00MB

OK (49 tests, 88 assertions)
matt@matts-MacBook-Pro demo (master) $
```

D.11 Explore directory /tests

Look in the `/tests` directory to see how those tests work. For example Figure D.2 shows a screenshot of the admin new post test in PHPStorm.



```
     * thanks to the DAMADoctrineTestBundle and its PH
     * to the database are rolled back when this test
     * all the application tests begin with the same d
     */
    public function testAdminNewPost()
    {
        $postTitle = 'Blog Post Title '.mt_rand();
        $postSummary = $this->generateRandomString( len
        $postContent = $this->generateRandomString( len

        $client = static::createClient([], [
            'PHP_AUTH_USER' => 'jane_admin',
            'PHP_AUTH_PW' => 'kitten',
        ]);
        $crawler = $client->request( method: 'GET',  uri
        $form = $crawler->selectButton( value: 'Create p
            'post[title]' => $postTitle,
            'post[summary]' => $postSummary,
            'post[content]' => $postContent,
        ]);
        $client->submit($form);

        $this->assertSame( expected: Response::HTTP_FOUN

        $post = $client->getContainer()->get( id: 'doct
            'title' => $postTitle,
        ]);
        $this->assertNotNull($post);
        $this->assertSame($postSummary, $post->getSumm
        $this->assertSame($postContent, $post->getCont
```

Figure D.2: The admin new post test in PHPStorm

D.12 Learn more

Learn more about PHPUnit testing and Symfony by visiting:

- https://symfony.com/doc/current/testing.html

E Solving problems with Symfony

E.1 No home page loading

Ensure web server is running (either from console, or a webserver application with web root of the project's `/public` directory).

If Symfony thinks you are in **production** (live public website) then when an error occurs it will throw a 500 server error (which a real production site would catch and display some nicely sanitised message for website visitors).

Since we are in **development** we want to see the **details** of any errors. We set the environment in the `.env` file. For development mode you should see the following in this file:

```
APP_ENV=dev
```

In development you should then get a much more detailed description of the error (including the class / line / template causing the problem etc.).

Also, if you know where your server error logs are stored, you can see the errors written to the log file. Symfony lots are usually create in `/var/log`.

E.2 "Route not Found" error after adding new controller method

If you have issues of Symfony not finding a new route you've added via a controller annotation comment, try the following.

It's a good idea to **CLEAR THE CACHE** when addeding/changing routes, otherwise Symfony may not recognised the new or changed routes … Either manually delete the `/var/cache` directory, or run the `cache:clear` console command:

```
$ php bin/console cache:clear

// Clearing the cache for the dev environment with debug true
[OK] Cache for the "dev" environment (debug=true) was successfully cleared.
```

Symfony caches (stores) routing data and also rendered pages from Twig, to speed up response time. But if you have changed controllers and routes, sometimes you have to manually delete the cache to ensure all new routes are checked against new requests.

E.3 Issues with timezone

Try adding the following construction to `/app/AppKernel.php` to solve timeszone problems:

```
public function __construct($environment, $debug)
{
    date_default_timezone_set( 'Europe/Dublin' );
    parent::__construct($environment, $debug);
}
```

E.4 Issues with Symfony 3 and PHPUnit.phar

Symfon 3.2 has issues with PHPUnit (it's PHPUnit's fault!). You can solve the problem with the Symphony PHPUnit `bridge` - which you install via Composer:

```
composer require --dev symfony/phpunit-bridge
```

You then execute your PHPUnit test with the `simple-phpunit` command in `/vendor/bin` as follows:

```
./vendor/bin/simple-phpunit
```

Source:

- Symfony Blog December 2016

E.5 PHPUnit installed via Composer

To install PHPUnit with Composer run the following Composer update CLI command:

```
composer require --dev phpunit/phpunit ^6.1
```

To run tests in directory `/tests` exectute the following CLI command:

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

Source:

- Stack overflow

As always you can add a shortcut script to your `composer.json` file to save typing, e.g.:

```
"scripts": {
    "run":"php bin/console server:run",
    "test":"./vendor/bin/phpunit tests",

    ...
}
```

F

Quick setup for new ‘blog’ project

F.1 Create a new project, e.g. ‘blog’

Use the Symfony command line installer (if working for you) to create a new project named ‘blog’ (or whatever you want!)

```
$ symfony new blog
```

Or use Composer:

```
$ composer create-project symfony/framework-standard-edition blog
```

Read more at:

- Symfony create project reference

F.2 Set up your localhost browser shortcut to for `app_dev.php`

Set your web browser shortcut to the `app_dev.php`, i.e.:

```
http://localhost:8000/app_dev.php
```

F.3 Add `run` shortcut to your Composer.json scripts

Make life easier - add a “run” Composer.json script shortcut to run web server from command line:

```
"scripts": {
    "run":"php bin/console server:run",
    ...
```

F.4 Change directories and run the app

Change to new project directory and run the app

```
/~user/$ cd blog
/~user/blog/$ composer run
```

Now visit: `http://localhost:8000/app_dev.php` in your browser to see the welcome page

F.5 Remove default content

If you want a **completely blank** Symfony project to work with, then delete the following:

```
/src/AppBundle/Controller/DefaultController.php
/app/Resources/views/default/
/app/Resources/views/base.html.twig
```

Now you have no controllers or Twig templates, and can start from a clean slate...

G

Steps to download code and get website up and running

G.1 First get the source code

First you need to get the source code for your Symfony website onto the computer you want to use

G.1.1 Getting code from a zip archive

Do the following:

- get the archive onto the desired computer and extract the contents
- if there is no `/vendor` folder then run CLI command `composer update`

G.1.2 Getting code from a Git respository

Do the following:

- on the computer to run the server `cd` to the web directory
- clone the repository with CLI command `git clone <REPO-URL>`
- populate the `/vendor` directory by running CLI command `composer update`

G.2 Once you have the source code (with vendor) do the following

- update `/app/config/parameters.yml` with your DB user credentials and name and host of the Database to be used
- start running your MySQL database server (assuming your project uses MySQL)
- create the database with CLI command `php bin/console doctrine:database:create`
- create the tables with CLI command `php bin/console doctrine:schema:update --force`

G.3 Run the webserver

Either run your own webserver (pointing web root to `/web`, or

- run the webserver with CLI command `php bin/console server:run`
- visit the website at `http://localhost:8000/`

H

About `parameters.yml` and `config.yml`

H.1 Project (and deployment) specific settings in (`/app/config/parameters.y`

Usually the project-specific settings are declared in this file:

```
/app/config/parameters.yml
```

These parameters are referred to in the more generic `/app/config/config.yml`.

For example the host of a MySQL database for the project woudl be defined by the following variable in `parameters.yml`:

```
parameters:
    database_host: 127.0.0.1
```

Note that this file (`parameters.yml`) is include in the `.gitignore`, so it is **not** archived in your Git folder. Usually we need different parameter settings for different deployments, so while on your local, development machine you'll have certain settings, you'll need different settings for your public production 'live' website. Plus you don't want to accidently publically expose your database credentials on a open source Github page :-)

If there isn't already a `parameters.yml` file, then you can copy the `parameters.yml.dist` file end edit it as appropriate.

H.2 More general project configuration (/app/config/config.yml)

The file `/app/config/config.yml` is actually the one used by Symfony when it looks up project settings. So the `config.yml` file uses references to the variables declared in the `/app/config/parameters.yml` file. For example the following lines in `config.yml` make a reference to the variable `database_path` that is declared in `parameters.yml`:

```
doctrine:
    dbal:
        driver:   pdo_mysql
        host:     "%database_host%"
```

For many projects we need to make **no changes** to the contents of `config.yml`. Although, since Symfony is setup with defaults for a MySQL database, if we are using SQLIte, for example then we do need to change the configuration settings, as well as declaring appropriate variables in `parameters.yml`. This is discussed in Appendix , describing how to set up a Symfony project to work with SQLite.

I Setting up for MySQL Database

I.1 Declaring the parameters for the database (`parameters.yml`)

Usually the project-specific settings are declared in this file:

```
/app/config/parameters.yml
```

These parameters are referred to in the more generic `/app/config/config.yml` - which for MySQL projects we don't need to touch.

The simplest way to connect your Symfony application to a MySQL database is by setting the following variables in `parameters.yml` (located in (`/app/config/`):

```
# This file is auto-generated during the composer install
parameters:
    database_host: 127.0.0.1
    database_port: null
    database_name: symfony_book
    database_user: root
    database_password: null
```

Note, you can learn move about `parameters.yml` and `config.yml` in Appendix H.

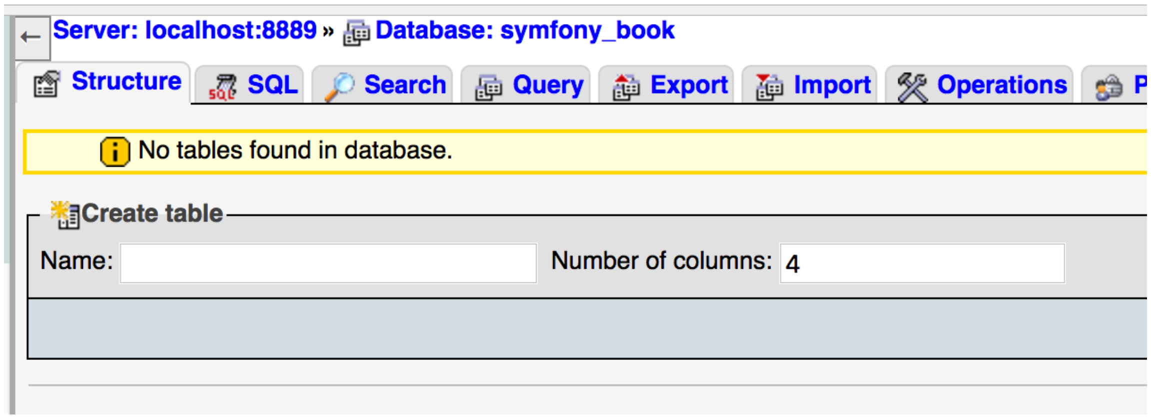
You can replace `127.0.0.1` with `localhost` if you wish. If your code cannot connect to the database check the 'port' that your MySQL server is running at (usually 3306 but may be different, for example my Mac MAMP server uses 8889 for MySQL for some reason). So my parameters look like this:

```
parameters:
    database_host:     127.0.0.1
    database_port:     8889
    database_name:     symfony_book
    database_user:     symfony
    database_password: pass
```

We can now use the Symfony CLI to **generate** the new database for us. You've guessed it, we type:

```
$ php bin/console doctrine:database:create
```

You should now see a new database in your DB manager. Figure I.1 shows our new `symfony_book` database created for us.



No tables found in database.

Create table

Name:

Number of columns: 4

Figure I.1: CLI created database in PHPMyAdmin.

NOTE Ensure your database server is running before trying the above, or you'll get an error like this:

```
[PDOException] SQLSTATE[HY000] [2002] Connection refused
```

now we have a database it's time to start creating tables and populating it with records …

J Setting up for SQLIte Database

J.1 NOTE regarding FIXTURES

If you are using the Doctrine Fixtures Bundle, install that first **before** changing paramaters and config for SQLite. The fixtures bundle assumes MySQL, and will overwrite some of the parameters during installation.

If that does happen, you'll just have to repeat the steps in this Appdendix to set things back to SQLite after fixtures installation.

J.2 SQLite suitable for most small-medium websites

For small/medium projects, and learning frameworks like Symfony, it's often simplest to just use a file-based SQLite database.

Learn more about SQLite at the project's website, and their discussion of when SQLite is a good choices, and when a networked DBMS like MySQL is more appropriate:

- SQLite website
- Appropriate Uses For SQLite

J.3 Create directory where SQLite database will be stored

Setting one up with Symfony is **very** easy. These steps assume you are gong to use an SQLite database file named `data.sqlite` located in directory `/var/data`.

Our first step to configuring a Symfony project to work with SQLite is to ensure the directory exists where the SQLIte file is to be created. The usual location for Symfony projects is `/var/data`. So create directory `data` in `/var` if it doesn't already exist in your project.

J.4 Declaring the parameters for the database (`parameters.yml`)

In `/app/parameters.yml` replace the default `database_host/name/user/password` parameters with a single parameter `database_path` as follows:

~~~~
```yaml
    parameters:
        database_path: ../var/data/data.sqlite
        mailer_transport: smtp
        mailer_host: 127.0.0.1
        etc.
```
~~~~

J.5 Setting project configuraetion to work with the SQLite database driver and path (`/app/config/config.yml`)

In `/app/config.yml` change the `doctrine` settings **from** these MySQL defaults:

~~~~
```yaml
    # Doctrine Configuration
    doctrine:
        dbal:
            driver:   pdo_mysql
            host:     "%database_host%"
            port:     "%database_port%"
            dbname:   "%database_name%"
            user:     "%database_user%"
            password: "%database_password%"
            charset:  UTF8
```
~~~~

to these SQLite settings:

~~~
```yaml
    # Doctrine Configuration
    doctrine:
        dbal:
            driver:   pdo_sqlite
            path:     "%kernel.root_dir%/%database_path%"
```
~~~

That's it! You can now tell Symfony to create your database with CLI command:

```
php bin/console doctrine:database:create
```

You'll now have an SQLite database file at `/var/data/data.sqlite`. You can even use the PHP-Storm to open and read the DB for you. See Figures J.1 and J.2.

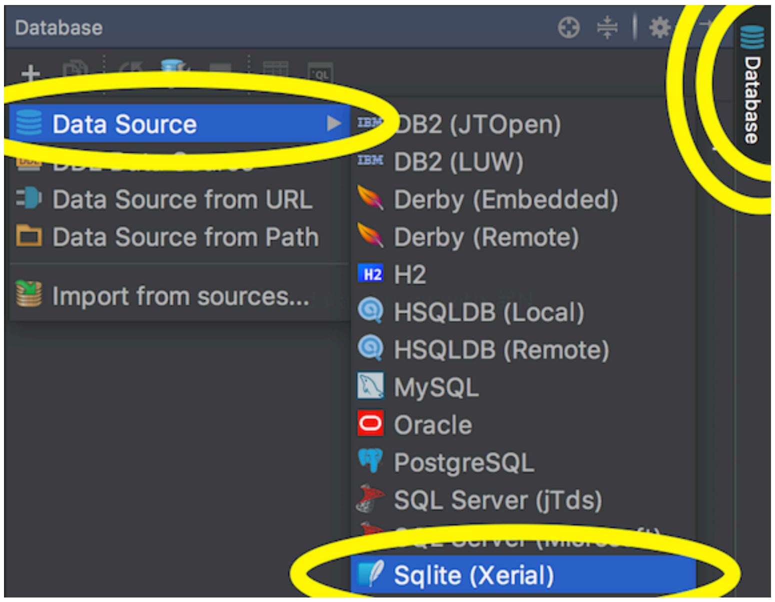


Figure J.1: Open SQLite view in PHPMyAdmin.

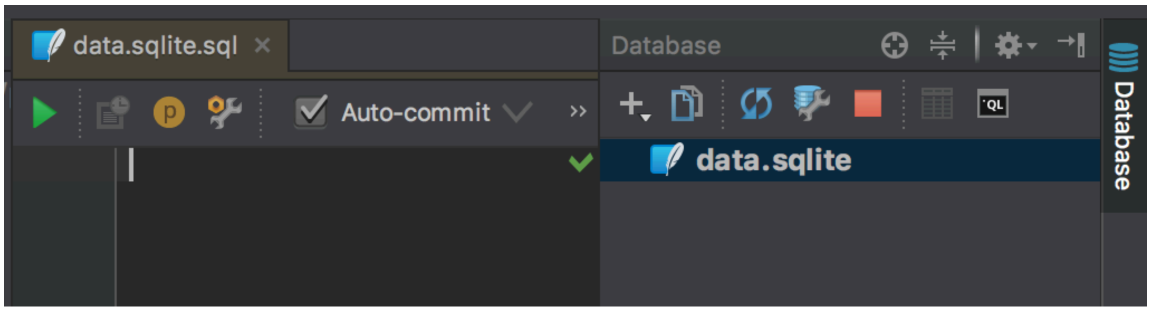


Figure J.2: Viewing `/var/data.sqlite` in PHPStorm.

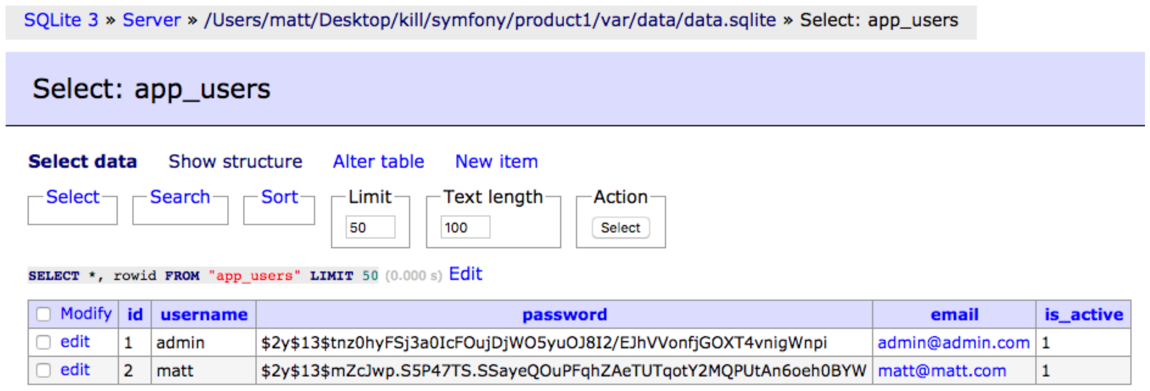
K

Setting up Adminer for DB GUI interaction

K.1 Adminer - small and simple DB GUI

Adminer is a lightweight PHP, web-based GUI for DB interaction. It supports both MySQL and SQLite.

Figure K.1 shows **Adminer** listing 2 user records created from Symofny Doctrine fixtures.



SQLite 3 » Server » /Users/matt/Desktop/kill/symfony/product1/var/data/data.sqlite » Select: app_users

Select: app_users

Select data Show structure Alter table New item

Select Search Sort Limit 50 Text length 100 Action Select

```
SELECT *, rowid FROM "app_users" LIMIT 50 (0.000 s) Edit
```

Modify	id	username	password	email	is_active
edit	1	admin	$2y$13$tnz0hyFSj3a0IcFOujDjWO5yuOJ8I2/EJhVVonfjGOXT4vnigWnpi	admin@admin.com	1
edit	2	matt	$2y$13$mZcJwp.S5P47TS.SSayeQOuPFqhZAeTUTqotY2MQPUtAn6oeh0BYW	matt@matt.com	1

Figure K.1: Using CLI to load database fixtures.

K.2 Getting Adminer

Download Adminer from the project website. I recommedn you get the English only version - it's smaller...

- Adminer.org website

K.3 Setting up

Extract the file to a suitable location. For example you could create an '/amdminer' directory in your current project:

```
.../project/adminer/adminer.php
```

To keep things simple, and also to remove the login requiremnt for SQLIte access, create file `index.php` in your Adminer directory, containing the following:

```
<?php
// index.php

function adminer_object()
{
    class AdminerSoftware extends Adminer
    {
        function login($login, $password) {
            return true;
        }
    }
    return new AdminerSoftware;
}

include __DIR__ . "/adminer.php";
```

K.4 Running Adminer

Since we have an `index.php` page, we just need to run a web server pointing its root to our Adminer directory. Perhaps the simplest way to do this is with the built-in PHP server, e.g.:

```
php -S localhost:3306 -t ./adminer
```

To save typing, you could add a script alias to your `composer.json` file:

```
"adminer":"php -S localhost:3306 -t adminer",
```

When run, choose the appropriate DBMS from the dropdown menu (e.g. **SQLite 3**), and enter the required credentials. For SQLite all we need to enter is the path to the location of the SQLite database file, e.g.:

```
/Users/matt/Desktop/kill/symfony/product1/var/data/data.sqlite
```

L

Avoiding issues of SQL reserved words in entity and property names

Watch out for issues when your Entity name is the same as SQL keywords.

Examples to **avoid** for your Entity names include:

- user
- group
- integer
- number
- text
- date

If you have to use certain names for Entities or their properties then you need to ‘escape’ them for Doctrine.

- Doctrine identifier escaping

You can ‘validate’ your entity-db mappings with the CLI validation command:

```
$ php bin/console doctrine:schema:validate
```

M

Transcript of interactive entity generation

The following is a transcript of an interactive session in the terminal CLI to create an `Item` entity class (and related `ItemRepository` class) with thse properties:

- title (string)
- price (float)

You start this interactive entity generation dialogue with the following console command:

```
php bin/console doctrine:generate:entity
```

Here is the full transcript (note all entites are automatically given an 'id' property):

```
$ php bin/console doctrine:generate:entity


  Welcome to the Doctrine2 entity generator


This command helps you generate Doctrine2 entities.

First, you need to give the entity name you want to generate.
You must use the shortcut notation like AcmeBlogBundle:Post.

The Entity shortcut name: AppBundle:Product/Item

Determine the format to use for the mapping information.

Configuration format (yml, xml, php, or annotation) [annotation]:
 Instead of starting with a blank entity, you can add some fields now.
 Note that the primary key will be added automatically (named id).

 Available types: array, simple_array, json_array, object,
 boolean, integer, smallint, bigint, string, text, datetime, datetimetz,
 date, time, decimal, float, binary, blob, guid.

 New field name (press <return> to stop adding fields): description
 Field type [string]:
 Field length [255]:
 Is nullable [false]:
 Unique [false]:

 New field name (press <return> to stop adding fields): price
 Field type [string]: float
 Is nullable [false]:
 Unique [false]:

 New field name (press <return> to stop adding fields):

   Entity generation

   created ./src/AppBundle/Entity/Product/
   created ./src/AppBundle/Entity/Product/Item.php
 > Generating entity class src/AppBundle/Entity/Product/Item.php: OK!
> Generating repository class src/AppBundle/Repository/Product/ItemRepository.php: OK!

   Everything is OK! Now get to work :).

 $
```

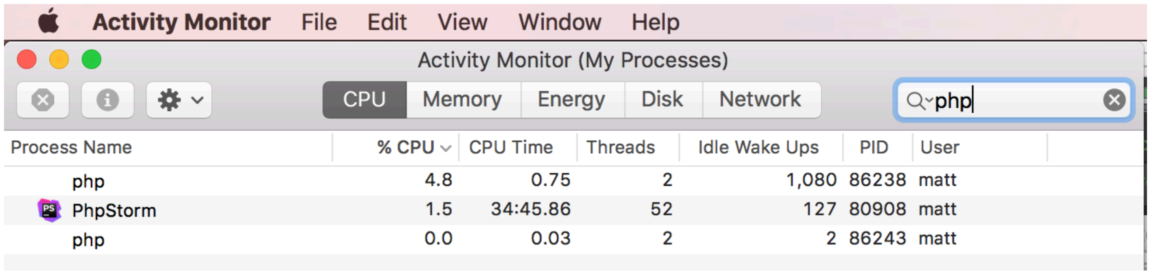
N

Killing 'php' processes in OS X

Do the following:

- run the **Activity Monitor**
- search for Process Names that are `php`
- double click them and choose `Quit` to kill them

voila!



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
php	4.8	0.75	2	1,080	86238	matt
PhpStorm	1.5	34:45.86	52	127	80908	matt
php	0.0	0.03	2	2	86243	matt

Figure N.1: Mac running `php` process.

O
Docker and Symfony projects

O.1 Setup

Start with your Symfony project directory

O.2 Dockerfile

Create a file `Dockerfile`, containing the steps to build a Docker Image of your virtual computer system.

This `Dockerfile` assumes your Symofny project code is in directory “admin-prototype”:

```
FROM php:7.1.7-apache

COPY admin-prototype /var/www

## Expose apache.
EXPOSE 80

## Copy this repo into place. - if /www/site is referred to in Apache conf file ...
#ADD admin-prototype/web /var/www/site

## Update the default apache site with the config we created.
ADD apache-config.conf /etc/apache2/sites-enabled/000-default.conf
```

```
####### fix Symfony var Cache issue ###############
# source: http://symfony.com/doc/current/setup/file_permissions.html
CMD HTTPDUSER=$(ps axo user,comm | grep -E '[a]pache|[h]ttpd|[_]www|[w]ww-data|[n]ginx' | grep -
CMD setfacl -dR -m u:"$HTTPDUSER":rwX -m u:$(whoami):rwX /var/www/var
CMD setfacl -R -m u:"$HTTPDUSER":rwX -m u:$(whoami):rwX /var/www/var

## Run symfony server
CMD php /var/www/bin/console server:run 0.0.0.0:80&
```

O.3 Build your Docker image

Build your Docker image:

```
$ docker build -t my-application .
```

O.4 Run a Container process based on your image (exposing HTTP port 80)

Now run your Docker **Image** as a Docker **Container** process. The `-p 80:80` option is to expose port 80 in the container as port 80 on your main computer system, so you can visit the web site via your web browser at `http://localhost`.

```
docker run -it -p 80:80 my-application
```

O.5 Alternative `Dockerfile` for a basic PHP application, using Apache

This `Dockerfile` assumes the PHP project files are in directory "game1":

```
FROM php:7.1.7-apache

COPY game1 /var/www

## Expose apache.
EXPOSE 80

## Copy this repo into place. - if /www/site is referred to in Apache conf file ...
#ADD game1/web /var/www/site
```

```
## Update the default apache site with the config we created.
ADD apache-config.conf /etc/apache2/sites-enabled/000-default.conf

## By default start up apache in the foreground, override with /bin/bash for interative.
CMD /usr/sbin/apache2ctl -D FOREGROUND
```

O.6 Create config file for Apache

Create a file `apache-config.conf`, containing the following:

```
<VirtualHost *:80>
  ServerAdmin me@mydomain.com
  DocumentRoot /var/www/web

  <Directory /var/www/web/>
      Options Indexes FollowSymLinks MultiViews
      AllowOverride All
      Order deny,allow
      Allow from all
  </Directory>

  ErrorLog ${APACHE_LOG_DIR}/error.log
  CustomLog ${APACHE_LOG_DIR}/access.log combined

</VirtualHost>
```

O.7 Other Docker reference stuff

O.7.1 Docker Images

List images on disk

```
docker images
```

See the details of an image:

```
docker history php:7.1-cli
```

(hint: `php:7.1-cli` = repository name : tag)

O.7.2 Containers

List currently running containers (processes):

```
docker ps
```

Run an image as a container process:

```
docker run -it repository:tag /bin/bash
```

Note:

- the "-it" means go into an INTERACTIVE TERMINAL
- the "/bin/bash" is the command to run - i.e. run our BASH shell

Kill a process:

```
docker kill NAME
```

E.g. if container name was `wonderful_wozniak` then you'd type:

```
docker kill wonderful_wozniak
```

O.7.3 New Image from current (changed) state of a running Container

To Save an updated filesystem in Container to a new Image do the following:

```
docker commit -m "comments" containerName
```

E.g. if container name was `nifty_hodgkin` and you'd installed, say, git and composer, then write:

```
$ docker commit -m "installed git and composer" nifty_hodgkin
sha256:7e555cc0df651a1b68593733a35cdac341175bed294084eb73b7fb23ebdc5bbd
$
```

Note that the SHA is output, the ID of the new Image.

You can then add a **tag** to the new image.

First look at the images, and note its short Image Id:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              7e555cc0df65        5 days ago          433 MB
phpd                latest              d0ee7be93033        5 days ago          372 MB
```

Now give a TAG to our image, e.g. `php_composer_git`:

```
$ docker tag 7e555cc0df65 php_composer_git
```

Now we see our nicely tagged Image:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
php_composer_git    latest              7e555cc0df65        5 days ago          433 MB
phpd                latest              d0ee7be93033        5 days ago          372 MB
```

O.7.4 Exposing HTTP ports for Containers running web application servers

We can use the option `-p PORT:PORT` to expose a port from the Container to our main computer system.

E.g. To expose Container port 80 as port 80 on our computer we add `-p 80:80`, as part of our `docker run` command:

```
$ docker run -it -p 80:80 php_composer_git /bin/bash
```

We can **Inspect** the details of a running Container with the `docker inspect` command:

```
docker inspect wonderful_wozniak
```

O.8 Useful reference sites

Some useful sites for Docker and PHP include:

- (Good overview)[http://odewahn.github.io/docker-jumpstart/docker-images.html]
- (Web Server Docker - with note about Mac IP)[https://writing.pupius.co.uk/apache-and-php-on-docker-44faef716150]
- (Nice intro for PHP)[https://semaphoreci.com/community/tutorials/dockerizing-a-php-application]

From the offical Docker documentation pages:

- (Introduction)[https://docs.docker.com/get-started/#conclusion]
- (Download Docker)[https://www.docker.com/community-edition#/download]

P

xDebug for Windows

P.1 Steps for Windows

To setup xDebug for Windows you need:

1. to download the appropriate DLL for your PHP system into `C:\php\ext` (or elsewhere if you installed PHP somewhere else on your system)
2. add/uncomment the following line at the end of your `php.ini` file:

   ```
   zend_extensions = C:\php\ext\php_xdebug-2-6-0-7.1-vc14-x86_64.dll
   ```

   NOTE: The location / name of this file will depend on your PHP installation (see Wizard steps below)

P.2 Steps for Linux/Mac

You can quickly confirm xDebug status with the following CLI command:

```
$ php -ini|grep 'xdebug support'
xdebug support => enabled
```

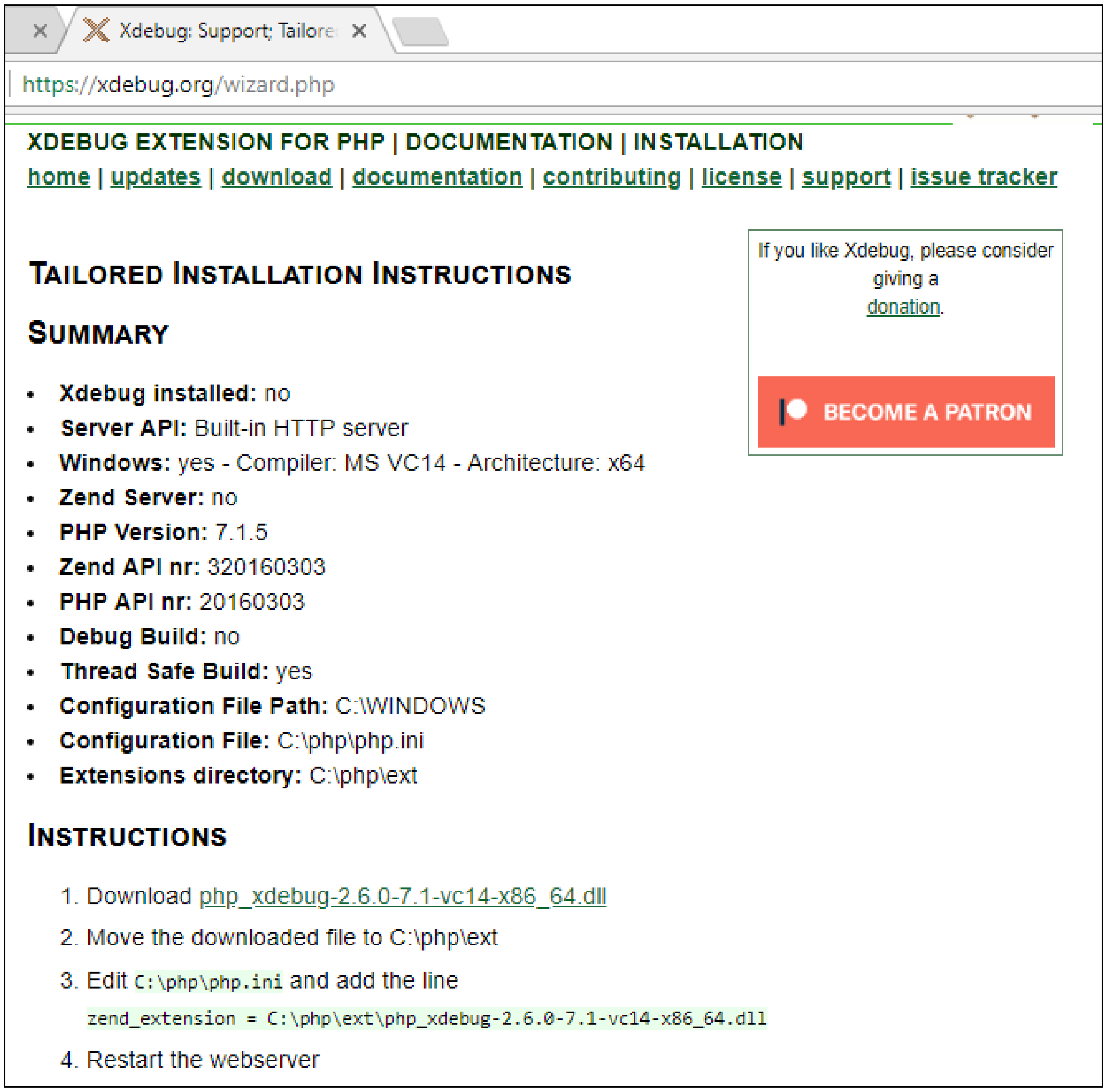
If you see 'enabled' then no further work is needed. Otherwise, the simplest way to get xDebug working is to use the wizard ...

P.3 Use the xDebug wizard!

Perhaps the easiest way to setup xDebug is to follow the steps recommended by their 'wizard' at:

- xDebug Windows wizard: https://xdebug.org/wizard.php

Figure ?? shows a screenshot of the xDebug wizard web page output.


Xdebug: Support; Tailore
https://xdebug.org/wizard.php
XDEBUG EXTENSION FOR PHP | DOCUMENTATION | INSTALLATION
home | updates | download | documentation | contributing | license | support | issue tracker
If you like Xdebug, please consider giving a donation.
BECOME A PATRON

TAILORED INSTALLATION INSTRUCTIONS

SUMMARY

- **Xdebug installed:** no
- **Server API:** Built-in HTTP server
- **Windows:** yes - Compiler: MS VC14 - Architecture: x64
- **Zend Server:** no
- **PHP Version:** 7.1.5
- **Zend API nr:** 320160303
- **PHP API nr:** 20160303
- **Debug Build:** no
- **Thread Safe Build:** yes
- **Configuration File Path:** C:\WINDOWS
- **Configuration File:** C:\php\php.ini
- **Extensions directory:** C:\php\ext

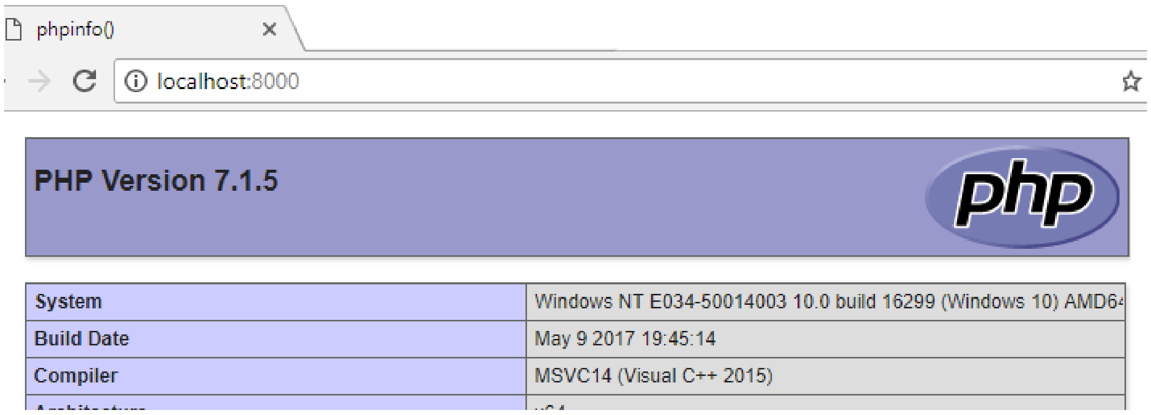
INSTRUCTIONS

1. Download php_xdebug-2.6.0-7.1-vc14-x86_64.dll
2. Move the downloaded file to C:\php\ext
3. Edit `C:\php\php.ini` and add the line
   ```
   zend_extension = C:\php\ext\php_xdebug-2.6.0-7.1-vc14-x86_64.dll
   ```
4. Restart the webserver

Figure P.1: Screenshot xDebug wizard output.

P.4 PHP Function `phpinfo()`

The `phpinfo()` output is a summary (as an HTML page) of your PHP setup. Figure P.2 shows a screenshot of a browser showing a PHP info page.



PHP Version 7.1.5

System	Windows NT E034-50014003 10.0 build 16299 (Windows 10) AMD6
Build Date	May 9 2017 19:45:14
Compiler	MSVC14 (Visual C++ 2015)

Figure P.2: Screenshot phpinfo in browser.

To use the 'wizard' you need to generate, copy and then paste the text output of `print phpinfo()` into the web page form.

To get the output from `phpinfo()` you can do one of these:

- at the CLI type

  `` `php -r 'print phpinfo();' > info.html ``

  and then get the contents of file `info.html`

- create a temporary directory, containing PHP file `index.php` that contains

  `php      <?php      print phpinfo();`

  run your webserver and visit the directory. Then copy and paste the contents of your browser window

- in Symfony you could create a temporary controller method that outputs a Reponse containing the outut of `phpinfo()`, e.g.

```
/**
 * @Route("/info")
 */
public function infoAction()
{
    return new Response( phpinfo() );
}
```

P.5 More information

For more information follow the steps at:

- xDebug Windows wizard
- xDebug project home page

List of References