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

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

Acknowledgements

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

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

Table of Contents

Part I

Introduction to Symfony

1

Introduction

1.1 What is Symfony 5?

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 8.1 (download/install from php.net)
  - (it will work with 8.0, but 8.1 allows enumerations which is handy …)
- a good text editor (I like PHPStorm, but then it's free for educational users…)
- Composer (PHP package manager - a PHP program)
- the Symfony command-line tool
  - https://symfony.com/download

The following are also a good idea: - a MySQL database server - e.g. MySQLWorkbench Community is free and cross-platform - Git - see GitforWindows

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

or Symfony's (not free) SymfonyCloud PHP-as-a-Service (PaaS):

- https://symfony.com/cloud/

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 D.

1.3 Check sysstem requiremnets

Once you've installed the Symfony command-line tool, check your system setup with the `symfony check:requirements` command:

```
$ symfony check:requirements

Symfony Requirements Checker
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

> PHP is using the following php.ini file:
/usr/local/etc/php/8.1/php.ini

> Checking Symfony requirements:

.............................


 [OK]
 Your system is ready to run Symfony projects
```

1.4 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 empty web application project, named `project01` (or whatever you wish) by typing:

```
$ symfony new --full project01
```

NOTE: If for some reason you don't have the Symfony command line tool installed, you can also create a project using Composer:

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

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

```
$ symfony new --full project01
* Creating a new Symfony project with Composer
  (running /usr/local/bin/composer create-project symfony/website-skeleton /Users/matt/project01)

* Setting up the project under Git version control
  (running git init /Users/matt/project01)

 [OK] Your project is now ready in /Users/matt/project01
```

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

1.5 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-symfony-6-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 install` to download any required 3rd-party packages into a `/vendor` folder
    - NOTE: `composer install` installs the **same** component versions as defined in the `composer.lock` file. `composer update` will attempt to install the **most up-to-date stable** versions of the components in the `composer.json` file.
4. Then run your web server (see below) and explore via a web browser

1.6 How to I run a Symfony webapp?

1.6.1 From the CLI

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

```
$ symfony serve
```

NOTE: This is short for `symfony server:start`

If you don't have the Symfony command line tool installed you could also use the PHP built-in web server:

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

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

See Figure 1.1 for a screenshot of the default Symfony 5 home page (with a message saying you've not configured a home page!).

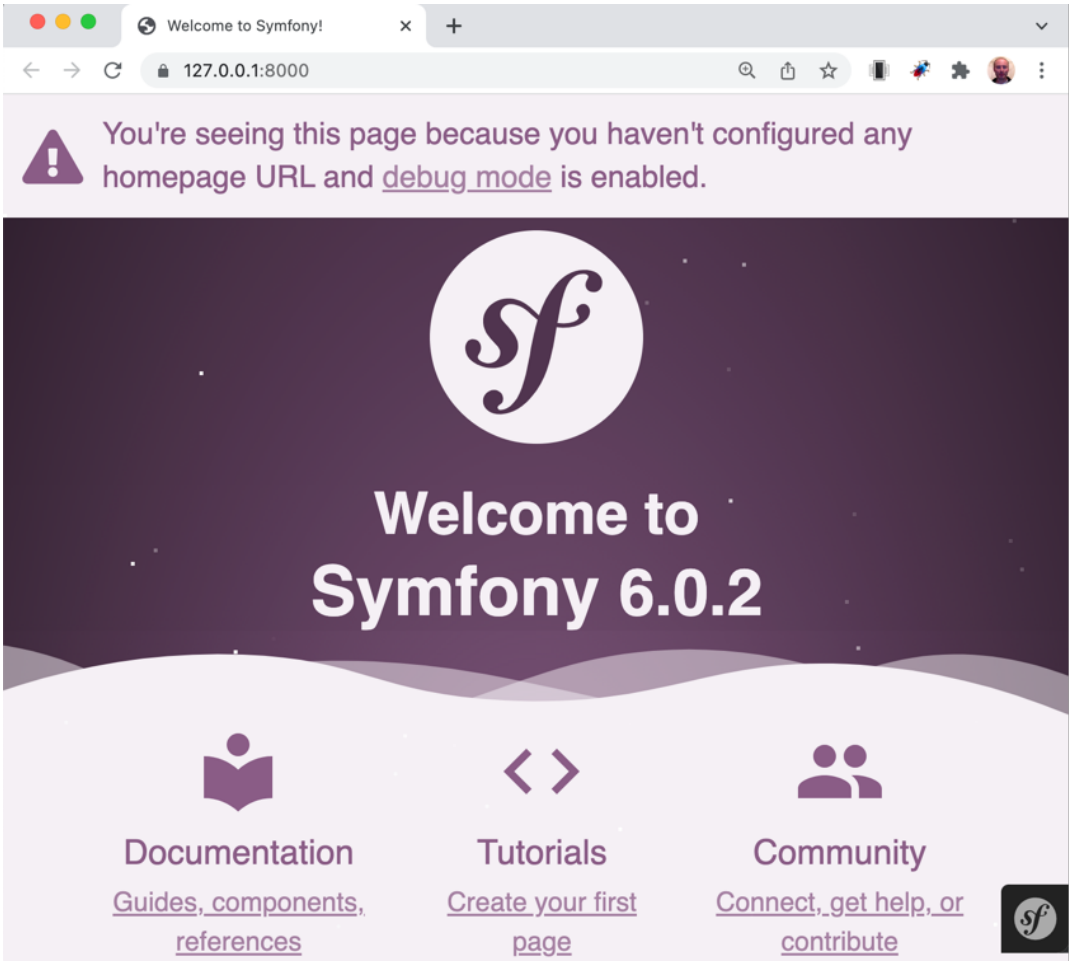

Figure 1.1: Screenshot default Symfony emopty new project home page.

1.6.2 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.

1.7 It isn't working! (Problem Solving)

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

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

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

1.9 Any free videos about SF to get me going?

Yes! Those nice people at Symfonycasts have released a bunch of free videos all about Symfony (and OO PHP in general).

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://symfonycasts.com/tracks/symfony

(ask Matt to ask his contacts in Symfonycasts to try to get his students a month's free access … if your Github Education Pack free acess has expired …)

2

First steps

2.1 What we'll make (`basic01`)

See Figure 2.1 for a screenshot of the new homepage we'll create in our first project (after some setup steps).

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 'attributes' , which declare the route in a line beginning `#` 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:

- a attribute Route comment for URL pattern `/` (i.e. website route)
  - `#[Route('/', name: 'homepage')]`
  - the Symfony "router" system attempts 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 project (and then `cd` into it):

```
$ symfony new --full project01

* Creating a new Symfony project with Composer
... etc. ...

[OK] Your project is now ready in /Users/matt/Documents/Books/php-symfony-6-book/codes/

$ cd basic01
```

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

```
$ symfony serve
   Tailing Web Server log file (/Users/matt/.symfony/log/ec56398112e31dba20d3fec928509d0cec5c3764.l
    Tailing PHP-FPM log file (/Users/matt/.symfony/log/ec56398112e31dba20d3fec928509d0cec5c3764/53f
    WARNING read /Users/matt/.symfony/var/ec56398112e31dba20d3fec928509d0cec5c3764: is a directory

    [OK] Web server listening
      The Web server is using PHP FPM 8.1.1
      https://127.0.0.1:8000

    [Web Server ] Jan  2 18:53:06 |INFO   | PHP    listening path="/usr/local/Cellar/php/8.1.1/sbin
    [PHP-FPM    ] Jan  2 18:53:06 |NOTICE | FPM    ready to handle connections

   // 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`).

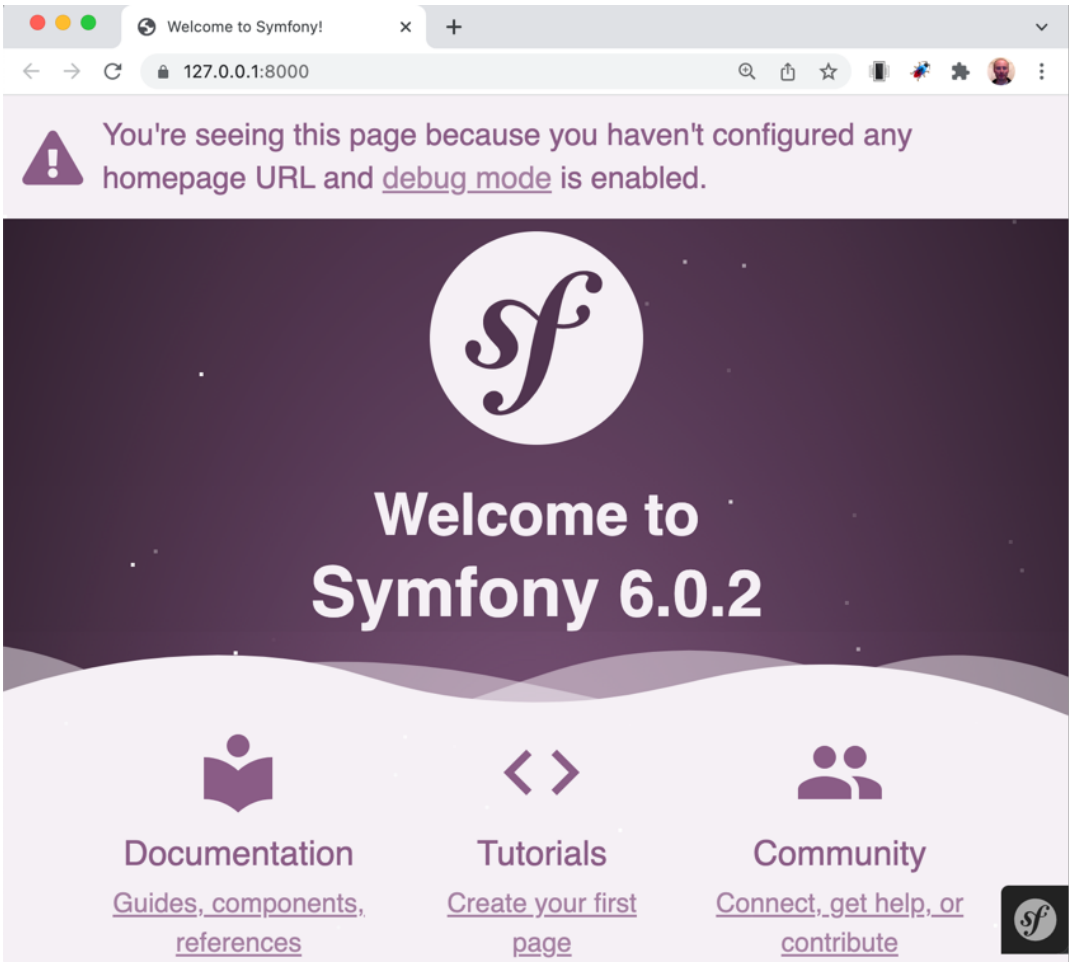

Figure 2.2: Screenshot default Symfony 4 page for web root (when no routes defined).

2.3 List the routes

There should not be any (non-debug) routes yet. All routes starting with an underscore _ symbol are debugging routes used by the verye useful Symfony profiler - this creates the information footer at the bottom of our pages when we are developing Symfony applications.

but let's check at the console by typing `php bin/console debug:router`:

```
$ php bin/console debug:router
 ------------------------- -------- -------- ------ ----------------------------------
  Name                      Method   Scheme   Host   Path
 ------------------------- -------- -------- ------ ----------------------------------
  _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
  _preview_error            ANY      ANY      ANY    /_error/{code}.{_format}
 ------------------------- -------- -------- ------ ----------------------------------
```

NOTE:

- you can usually shorten the Symfony CLI commands to 1 of 2 letters, e.g. `debug:router` could be written `de:ro` …

The only routes we can see all start with an underscore (e.g. `_preview_error`), so no application routes have been declared yet …

2.4 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. Tell Symfony to create a new homepage (default) controller. A since a class will be created

starting with the controller name, ensure your controlle rname starts with a CAPITAL letter, e.g. `Default` not `default`:

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

     created: src/Controller/DefaultController.php
     created: templates/default/index.html.twig

      Success!

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

Symfony controller classes are stored in directory `/src/Controller`. We can see that a new controller class has been created named `DefaultController.php` in folder `/src/Controller`.

A second file was also created, a view template file `templates/default/index.html.twig`,

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

```
<?php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

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

This default controller uses a **Twig** template to return an HTML page:

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

{% block title %}Hello DefaultController!{% endblock %}

{% block body %}
```

```
<style>
    .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font: 18px/1.5 sans-serif; }
    .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
</style>

<div class="example-wrapper">
    <h1>Hello {{ controller_name }}! TICK</h1>

    This friendly message is coming from:
    <ul>
        <li>Your controller at <code><a href="{{ '/Users/matt/Documents/Books/php-symfony-6-book/code
        <li>Your template at <code><a href="{{ '/Users/matt/Documents/Books/php-symfony-6-book/codes/
    </ul>
</div>
{% endblock %}
```

Let's 'make this our own' by changing the contents of the Response returned to a simple text response. Do the following:

- comment-out the body of the `index()` method
- at the top of the class add a `use` statement, so we can make use of the Symfony HTTFoundation class `Response`

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

- write a new body for the `index()` method to output a simple text message response:

  ```
  return new Response('Welcome to your new controller!');
  ```

So the listing of your `DefaultController` should look as follows:

```
<?php
    namespace App\Controller;

    use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
    use Symfony\Component\HttpFoundation\Response;
    use Symfony\Component\Routing\Annotation\Route;

    class DefaultController extends AbstractController
    {
        #[Route('/default', name: 'default')]
        public function index(): Response
        {
            return new Response('Welcome to your new controller!');
    //        return $this->render('default/index.html.twig', [
```

```
    //            'controller_name' => 'DefaultController',
    //        ]);
        }
    }
```

2.5 Run web server to visit new default route

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

But we see that default Symfony welcome page, not our custom response text message!

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
--------------------------- -------- -------- ------ ------------------------------------
 _...(all those debug routes starting with _ )
 default                     ANY      ANY      ANY    /default
```

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

```
    #[Route('/default', name: 'default')]
```

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

NOTE:

- if you still don't see our custom welcome page, then try first clearing the 'cache' to see the result of code changes we have just made. To speed things up Syfony uses a cache (memory) of recent Responses - but if you've made code chance the cached pages and routes may be out of date…
- to clear the cache using a Symfony CLI comment type `php app/console cache:clear`
- to clear the cache by deleting the files themselves, DELETE the `/var` folder
  - you can safely delete this folder at any time (unless you are using SQLite and storing your DB files there…)

Figure 2.3 shows a screenshot of the message created from our generated default controller method.

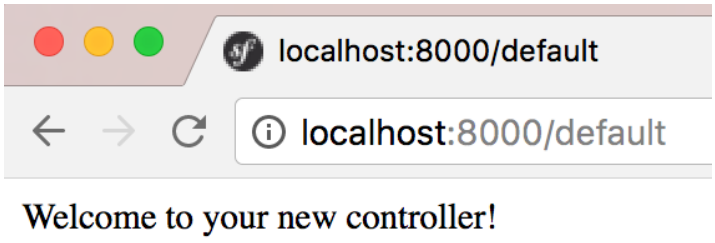

Figure 2.3: Screenshot of generated page for URL path `/default`.

2.6 Other types of Response content

We could also have asked our Controller function to return JSON rather than text. We can create JSON either using Twig, or with the inherited `->json(...)` method. For example, try replacing the body of your `index()` method with the following:

```
public function index()
{
    return $this->json([
        'name' => 'matt',
        'age' => '21 again!',
    ]);
}
```

2.7 The default Twig page

If we return our `index()` method back to what was first automatically generated for us, we can see an HTML page in our browser that is output from the Twig template:

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

Figure 2.4 shows a screenshot of the Twig HTML page that was automatically generated.

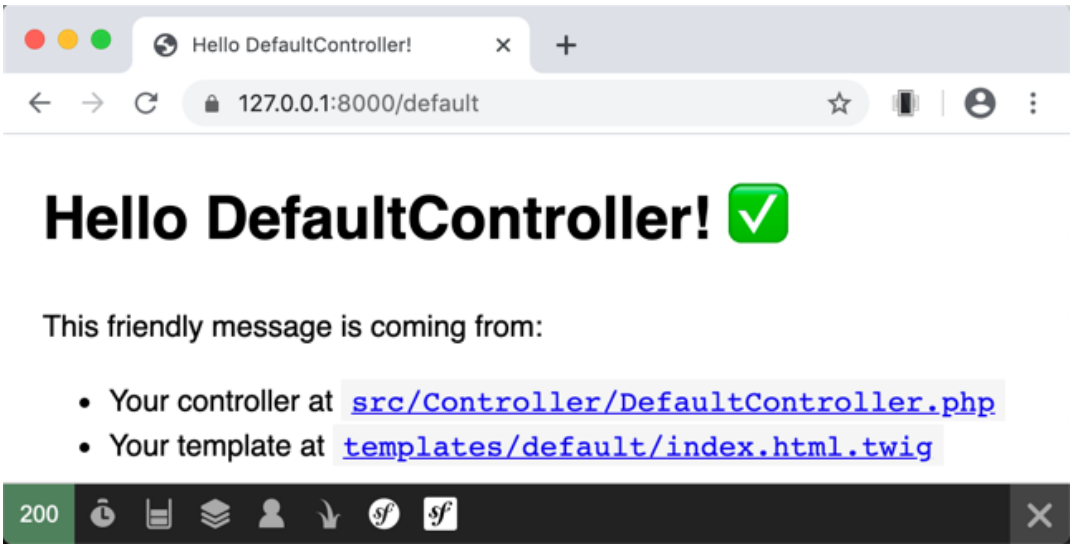

Figure 2.4: Screenshot of generated Twig page for URL path `/default`.

3

Twig templating

3.1 Customizing the Twig output (`basic02`)

Look at the generated code for the `index()` method of class `DefaultController`:

```
namespace App\Controller;

use Symfony\...

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

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.

NOTE: The actual look of the default generated Twig content may be a little different (e.g. 19 Feb 2019 it now says `Hello DefaultController!`)...

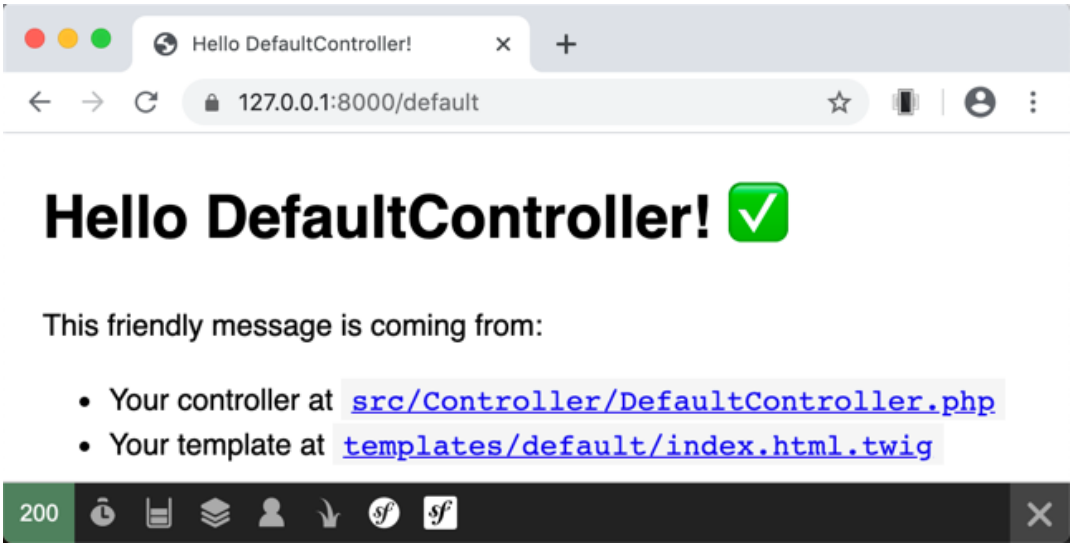

Figure 3.1: Screenshot of generated page for URL path `/default`.

3.2 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`.

```
class DefaultController extends AbstractController
{
    #[Route('/', name: 'homepage')]
    public function index(): Response
```

Now the route is as follows (from typing `php bin/console de:ro`):

```
 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.

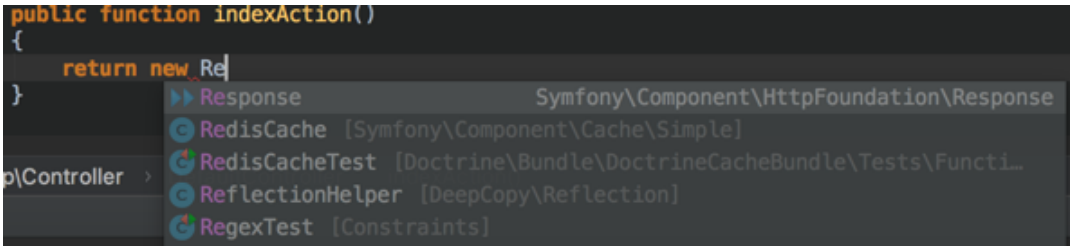

Figure 3.2: Screenshot of PHPStorm IDE suggesting namespaces classes.

Here is a complete `DefaultController` class:

```
namespace App\Controller;

use Symfony\Component\Routing\Annotation\Route;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends AbstractController
{
    #[Route('/', name: 'homepage')]
    public function index(): Response
    {
        return new Response('Hello there!');
    }
}
```

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

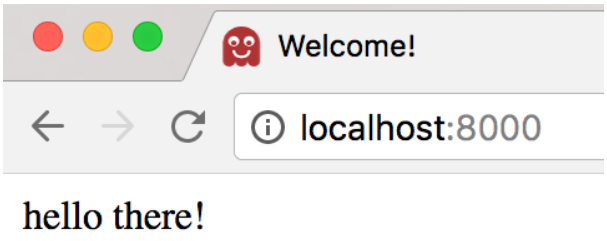

Figure 3.3: Screenshot of page seen for `new Response('hello there!')`.

3.3 Clearing the cache

Sometimes, when we've added a new route, we still get an error saying the route was not found, or showing us out-of-date content. This can be a problem of the Symfony **cache**.

So clearing the cache is a good way to resolve this problem (you may get in the habit of clearing the cache each time you add/change any routes).

You can clear the cache in 2 ways:

1. Simply delete directory `/var/cache`
2. Use the CLI command to clear the cache:

```
$ 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.

$
```

3.4 Let's create a nice Twig home page

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[1], 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()
```

[1]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`.

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

Now let's put our own personal content in that Twig template in `/templates/default/index.html.twig`!

- Replace the contents of file `index.html.twig` with the following:

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

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

    <p>
        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.

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 (by hand - not using the **make** tool)
- 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 `src/Entity/Student.php` class:

```
<?php
namespace App\Entity;

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

```
    private string $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.

So you should end up with accessor method for each propety such as:

```
/**
 * @return int
 */
public function getId(): int
{
    return $this->id;
}

/**
 * @param int $id
 */
public function setId(int $id): void
{
    $this->id = $id;
}

... etc... for the other propeties ...
```

4.3 Create a StudentController class

Generate a StudentController class:

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

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

```
<?php

namespace App\Controller;

use Symfony\...

class StudentController extends AbstractController
{
    #[Route('/student', name: 'student')]
```

```
        public function index(): Response
        {
            return ... default code here ...
        }
    }
```

NOTE:

- as well as creating the class `/src/Controller/StudentController.php`, a folder and Twig template has also been created for you in `/templates/student/index.html.twig`

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 (you can shorten to `ca:cl`)

```
$ 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 new controller index method 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 `show()`. So your class (with its `use` statements, especially for `App\Entity\Student`) looks as follows now:

```
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

use App\Entity\Student;

class StudentController extends AbstractController
{
    #[Route('/student', name: 'student')]
    public function index(): Response
    {
        $student = new Student();
        $student->setId(99);
        $student->setFirstName('matt');
        $student->setSurname('Smith');
```

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

NOTE:: Ensure your code has the appropriate `use` statement added for the `App\Entity\Student` class (since it's not in the same namespace as the controller, we have to add a `use` statement) - a nice IDE like PHPStorm will add this for you…

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

In folder `/templates/student` create a new Twig template `show.html.twig` containing the following:

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

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

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

Do the following:

- Run the web server

  ```
  symnfony serve
  ```

- Visit `/student`
  - you should see our student details displayed as a nice HTML page.

Figure 4.1 shows a screenshot our student details web page.

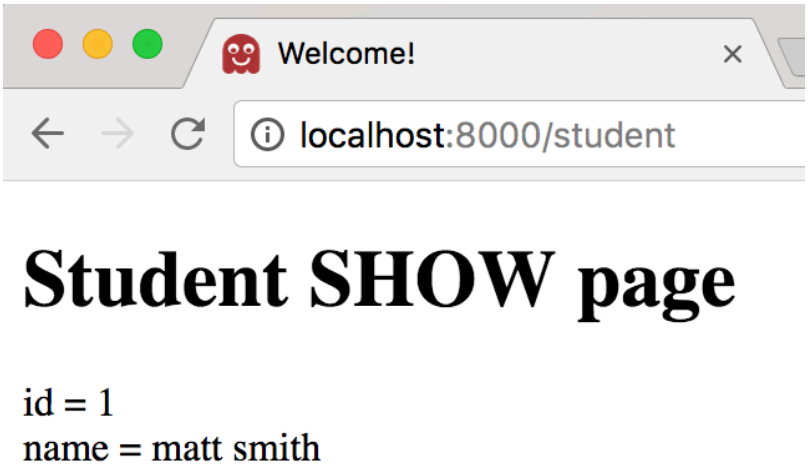

Figure 4.1: Screenshot of student show page.

4.5 Twig debug `dump(...)` function

A very useful feature of Twig is its `dump(...)` function. This outputs to the web page a syntax colored dump of the variable its passed. It's similar to the PHP `var_dump(...)` function. Figure 4.2 shows a screenshot of adding the following to our `show.html.twig` template:

```
{% block body %}
    <h1>Student SHOW page</h1>
    <p>
        id = {{ student.id }}
        <br>
        name = {{ student.firstName }} {{ student.surname }}
    </p>

    {{ dump (student) }}
{% endblock %}
```

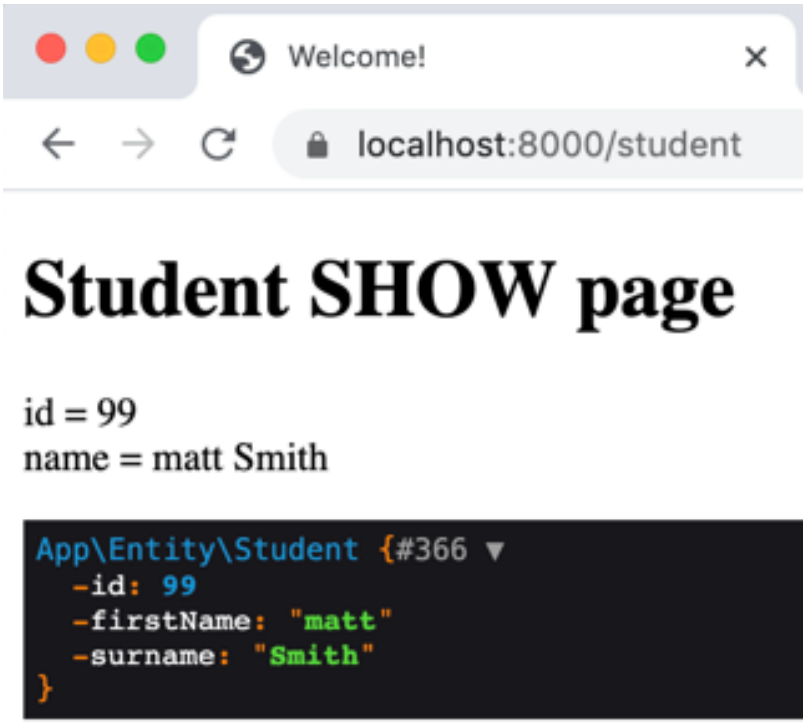

Figure 4.2: Screenshot of student show page.

4.6 Creating an Entity Repository (`basic04`)

We will now move on to work with an **array** of `Student` objects, which we'll make easy to work with by creating a `Repository` class. Let's create the `StudentRepository` class to work with collections of Student objects. Create PHP class file `StudentRepository.php` in directory `/src/Repository`:

```
<?php
namespace App\Repository;

use App\Entity\Student;

class StudentRepository
{
    private $students = [];

    public function __construct()
    {
        $id = 1;
        $s1 = new Student();
        $s1->setId(1);
        $s1->setFirstName('matt');
        $s1->setSurname('smith');
        $this->students[$id] = $s1;

        $id = 2;
        $s2 = new Student();
        $s2->setId(2);
        $s2->setFirstName('joelle');
        $s2->setSurname('murphy');
        $this->students[$id] = $s2;

        $id = 3;
        $s3 = new Student();
        $s3->setId(3);
        $s3->setFirstName('frances');
        $s3->setSurname('mcguinness');
        $this->students[$id] = $s3;
    }

    public function findAll()
    {
```

```
            return $this->students;
        }
    }
```

4.7 The student list controller method

Let's replace the contents of our `index()` method in the `StudentController` class, with one that will retrieve the array of student records from an instance of `StudentRepository`, and pass that array to our Twig template. The Twig template will loop through and display each one.

Replace the existing method `index()` of controller class `StudentController` with the following:

```
...

use App\Repository\StudentRepository;

use App\Entity\Student;

class StudentController extends AbstractController
{
    #[Route('/student', name: 'student')]
    public function index(): Response
    {
        $studentRepository = new StudentRepository();
        $students = $studentRepository->findAll();

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

So our routes remain the same, with the URL pattern `/student` being routed to our `StudentController->index()` method:

```
$ php bin/console debug:router

 ------------------------- -------- -------- ------ ----------------------------------
  Name                      Method   Scheme   Host   Path
 ------------------------- -------- -------- ------ ----------------------------------
```

```
_... (lots of other debug profiler routes)
homepage                    ANY      ANY      ANY    /
student                     ANY      ANY      ANY    /student
```

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

In directory `/templates/student` create Twig template `list.html.twig` with the following (you may wish to duplicate the `show` template and edit it to match this):

```
{% 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`, and you should see a list of all student details displayed as a nice HTML page.

Figure 4.3 shows a screenshot our list of students web page.

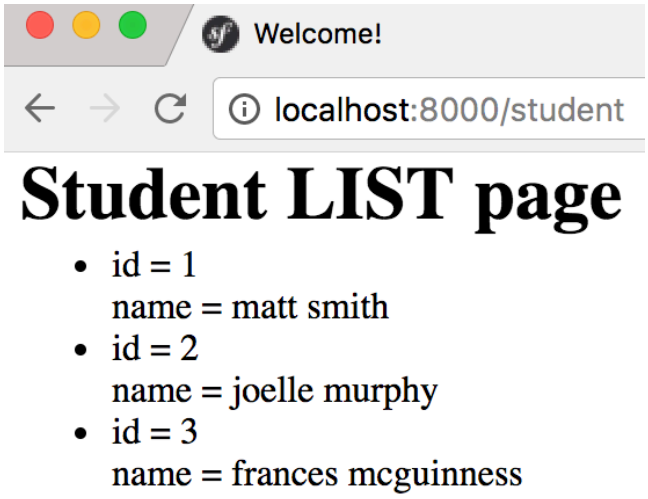

Figure 4.3: Screenshot of student list page.

4.9 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 write a new `StudentController` method `show()` to do this. 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, coincidentally, 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 ...

NOTE: We'll give this **show** route the internal name `student_show` - these internal route names are used when we create links between pages in our Twig templates, and so it's important to name them meaninfully and consistently to make later coding straightforward.

```
#[Route('/student/{id}', name: 'student_show')]
public function show($id): Response
{
    $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 list(): Response
{
    ... as before
}
```

We can check these routes via the console:

- `/student/{id}` will invoke our `show($id)` method
- `/student` will invoke our `list()` method

```
------------------------- -------- -------- ------ ----------------------------------
 Name                      Method   Scheme   Host   Path
------------------------- -------- -------- ------ ----------------------------------
 _... (lots of other debug profiler routes)
 homepage                  ANY      ANY      ANY    /
 student_list              ANY      ANY      ANY    /student
 student_show              ANY      ANY      ANY    /student/{id}
```

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 adding/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.

4.10 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 ...

4.11 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/index.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="{{ path('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 `path(...)` in the form:

```
path('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:

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

Figure 4.4 shows a screenshot our list of students web page, with a `(details)` hypertext link to the show page for each individual student object.

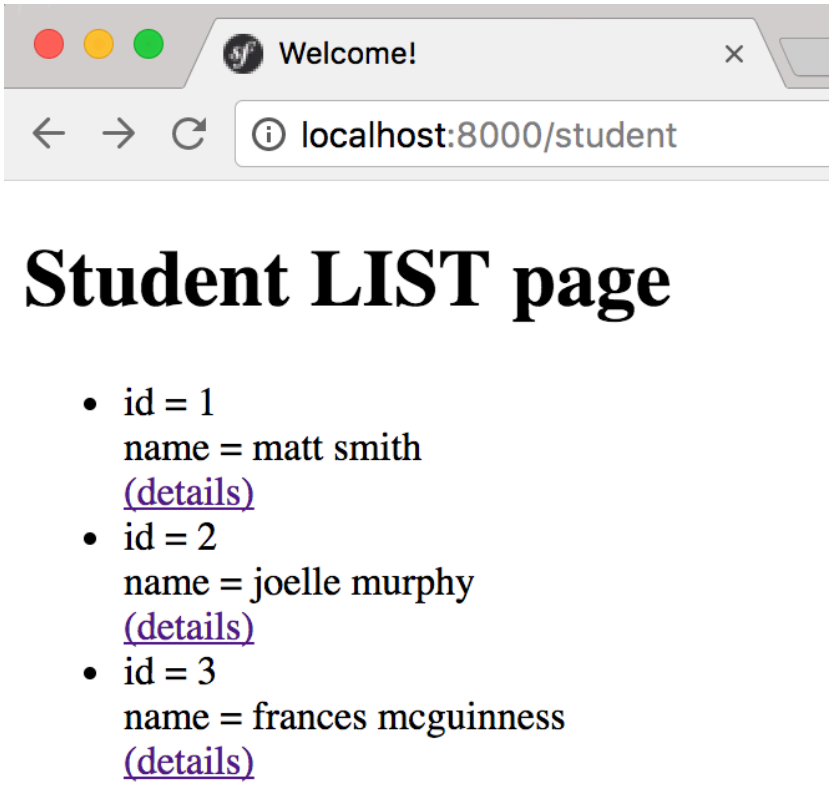

Figure 4.4: Screenshot of student list page, with links to show page for each student object.

4.12 Dealing with not-found issues (project `basic06`)

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 show($id): Response
{
    $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 title %}ERROR PAGE{% endblock %}

{% 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 %}
```

NOTE: We have overridden the `title` Twig block, so that the page title is `ERROR PAGE...`

Figure 4.5 shows a screenshot of our custom 404 error template for when no such student can be found for the given ID.

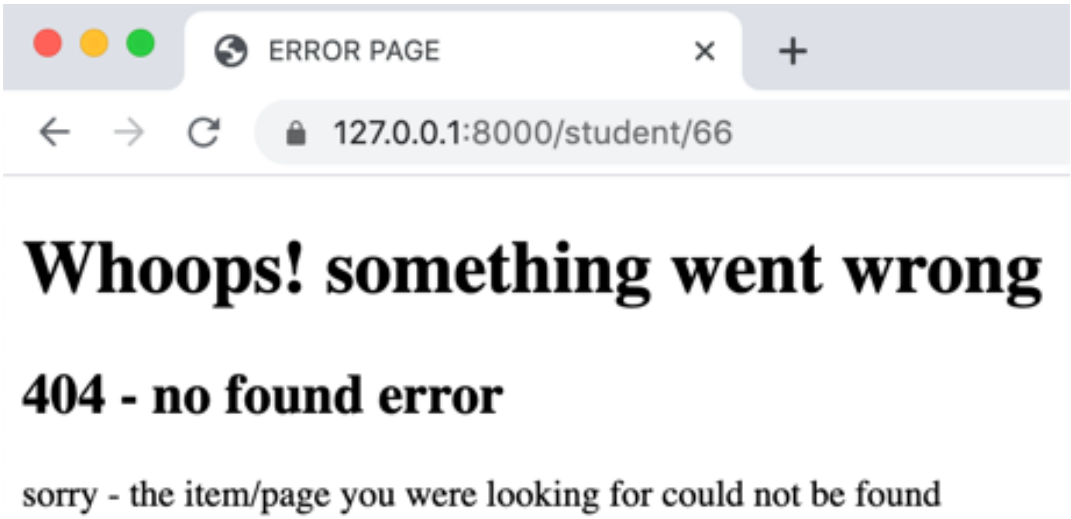

Figure 4.5: Error page for non-existant student ID = 66.