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!).

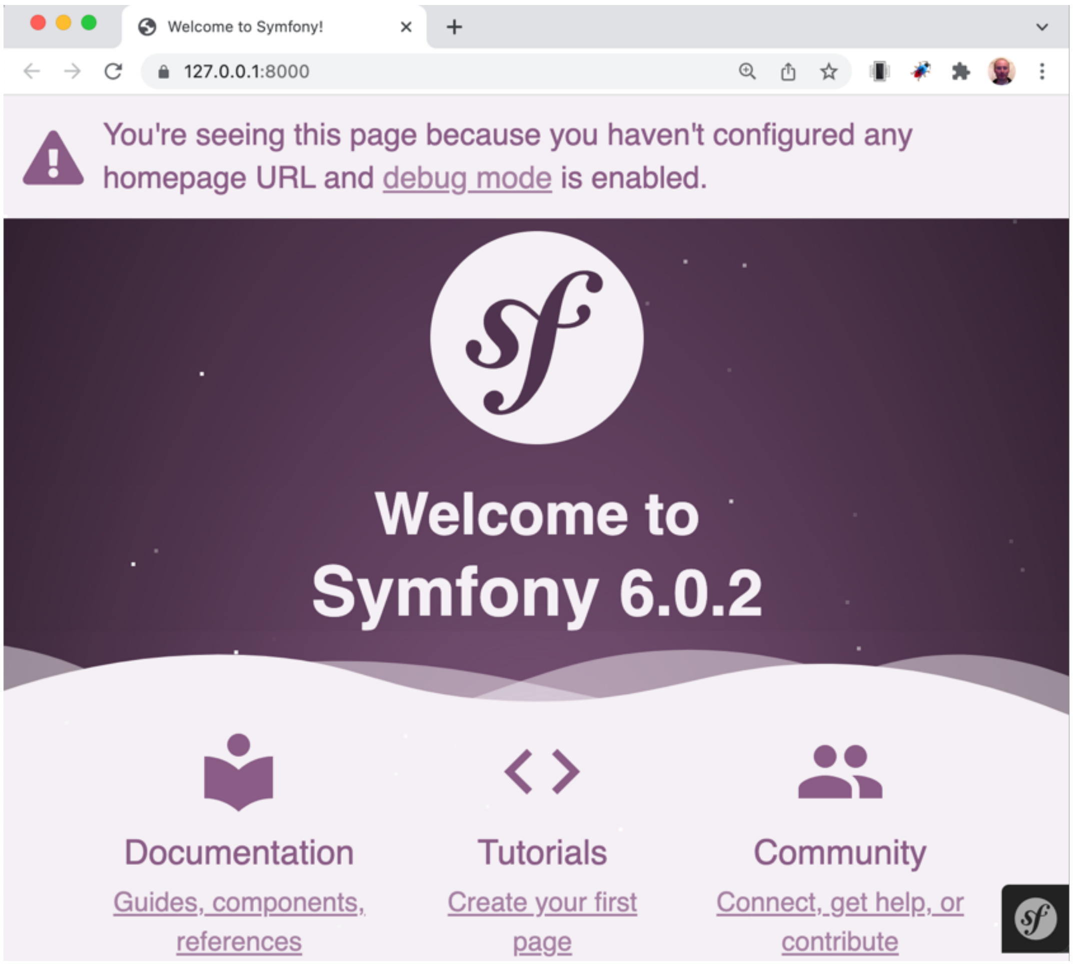

Welcome to Symfony!
127.0.0.1:8000
You're seeing this page because you haven't configured any homepage URL and debug mode is enabled.
Welcome to Symfony 6.0.2
Documentation
Guides, components, references
Tutorials
Create your first page
Community
Connect, get help, or contribute

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:

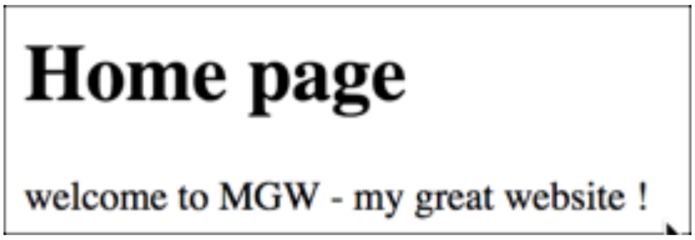
- https://symfonycasts.com/tracks/symfony

(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).



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 '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`).

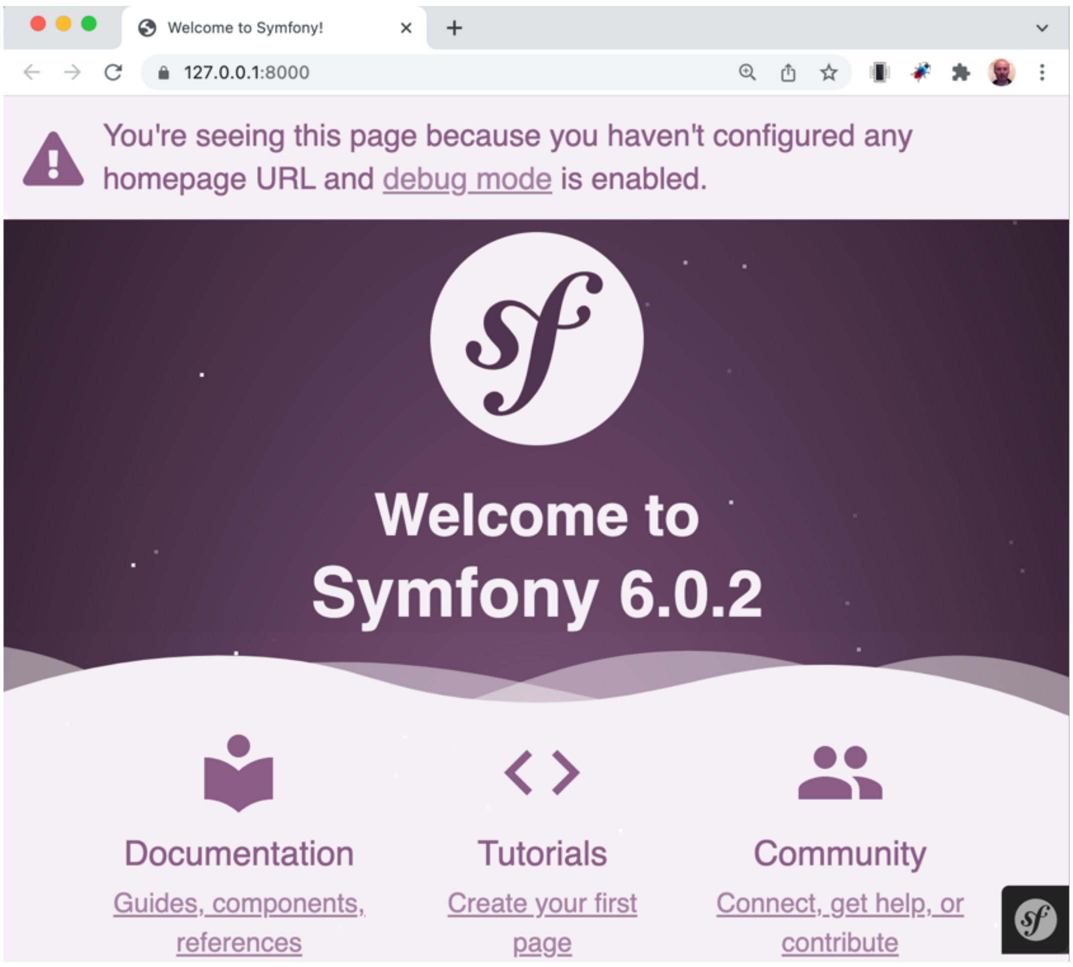

Welcome to Symfony!
127.0.0.1:8000
You're seeing this page because you haven't configured any homepage URL and debug mode is enabled.
Welcome to Symfony 6.0.2
Documentation
Guides, components, references
Tutorials
Create your first page
Community
Connect, get help, or contribute

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`

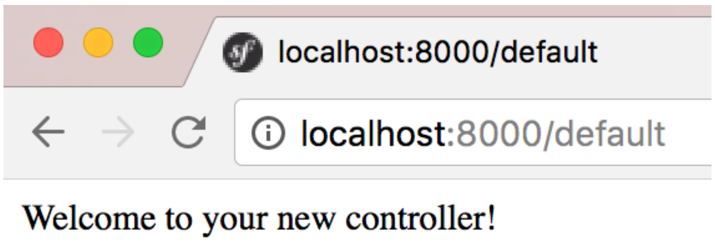
```
    #[Route('/default', name: 'default')]
```

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.



Welcome to your new controller!

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:

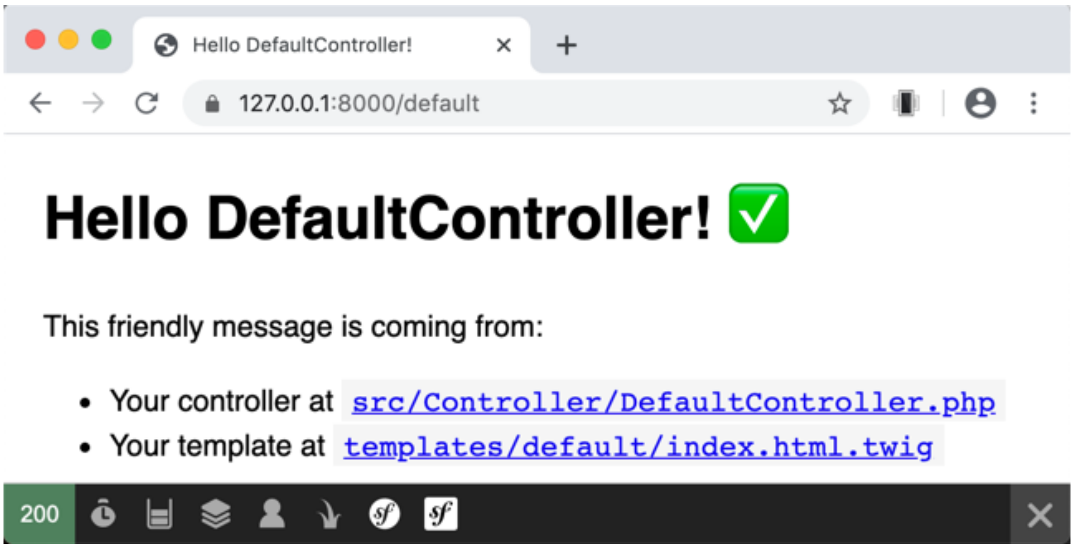
```
public function index()
{
    return $this->json([
        'name' => 'matt',
        'age' => '21 again!',
    ]);
}
```

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.



Hello DefaultController! ✅

This friendly message is coming from:

- Your controller at `src/Controller/DefaultController.php`
- Your template at `templates/default/index.html.twig`

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

3 Twig templating

3.1 Customizing the Twig output (basic02)

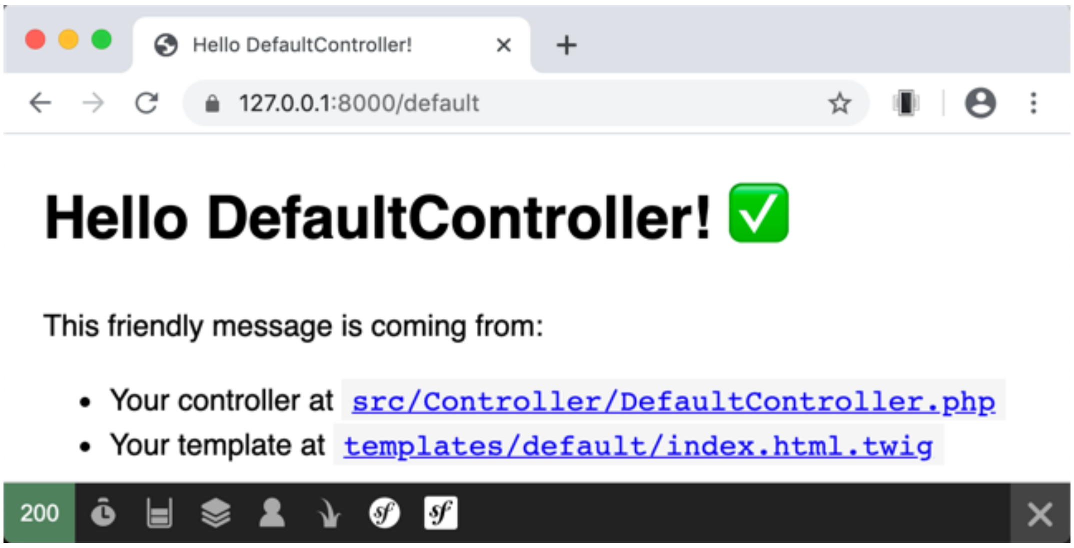
Look at the generated code for the `index()` method of class `DefaultController`:

```
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!`)…



Hello DefaultController! ✅

This friendly message is coming from:

- Your controller at src/Controller/DefaultController.php
- Your template at templates/default/index.html.twig

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`.

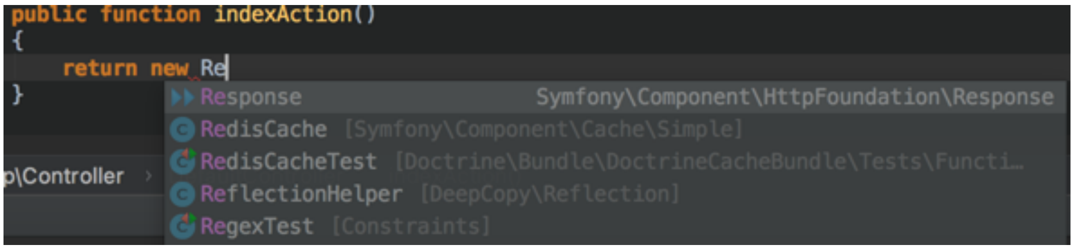
```
class DefaultController extends AbstractController
{
    #[Route('/', name: 'homepage')]
    public function index(): Response
```

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.



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

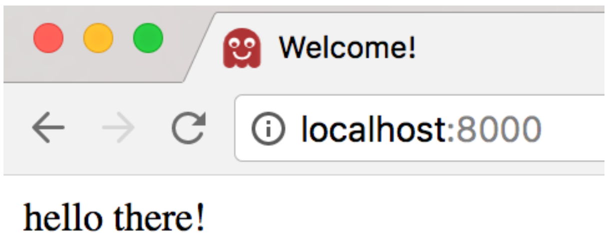
Response Symfony\Component\HttpFoundation\Response
RedisCache [Symfony\Component\Cache\Simple]
RedisCacheTest [Doctrine\Bundle\DoctrineCacheBundle\Tests\Functi…
ReflectionHelper [DeepCopy\Reflection]
RegexTest [Constraints]
p\Controller

Figure 3.2: Screenshot of PHPStorm IDE suggesting namespaces classes.

Here is a complete `DefaultController` class:

```
namespace App\Controller;

use 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.



hello there!

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);
    }
```

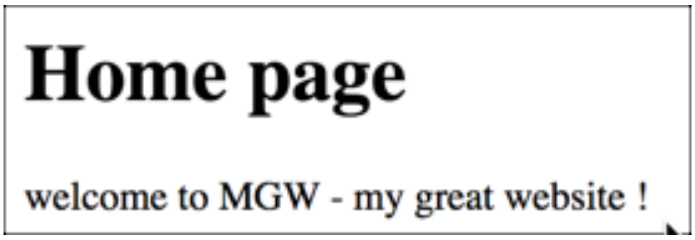
Now let's put our own personal content in that Twig template in `/templates/default/index.html.twig`!

- 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.


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 (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 %}
```

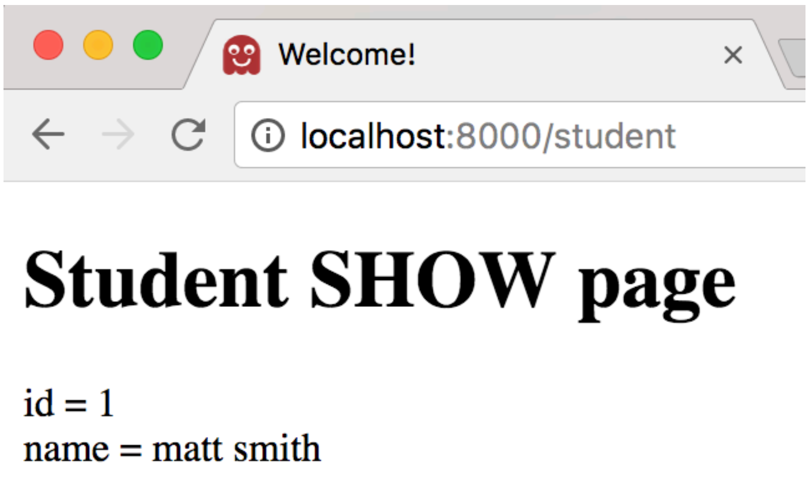
Do the following:

- 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.



Student SHOW page

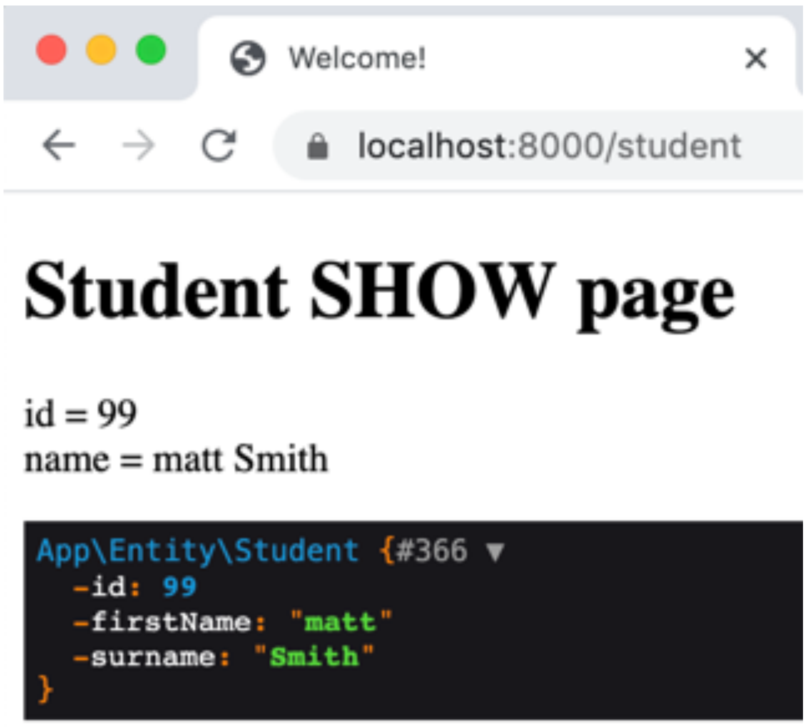
id = 1
name = matt smith

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 %}
```



Student SHOW page

id = 99
name = matt Smith

```
App\Entity\Student {#366 ▼
  -id: 99
  -firstName: "matt"
  -surname: "Smith"
}
```

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
<?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):

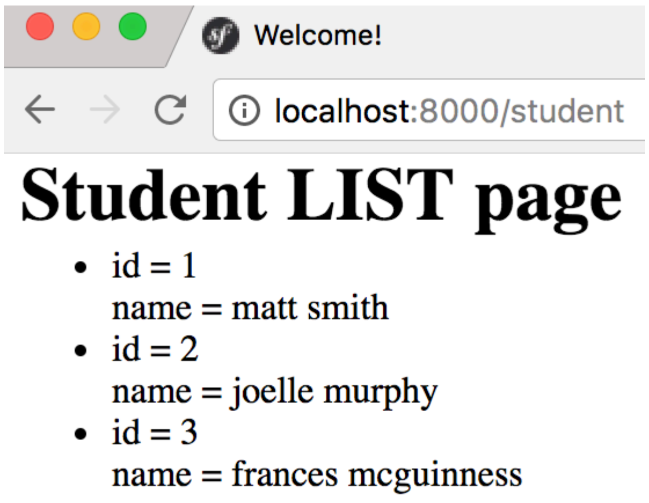
```
{% extends 'base.html.twig' %}

{% 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.



Student LIST page

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

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(int $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(int $id)
{
    if(array_key_exists($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
];
```

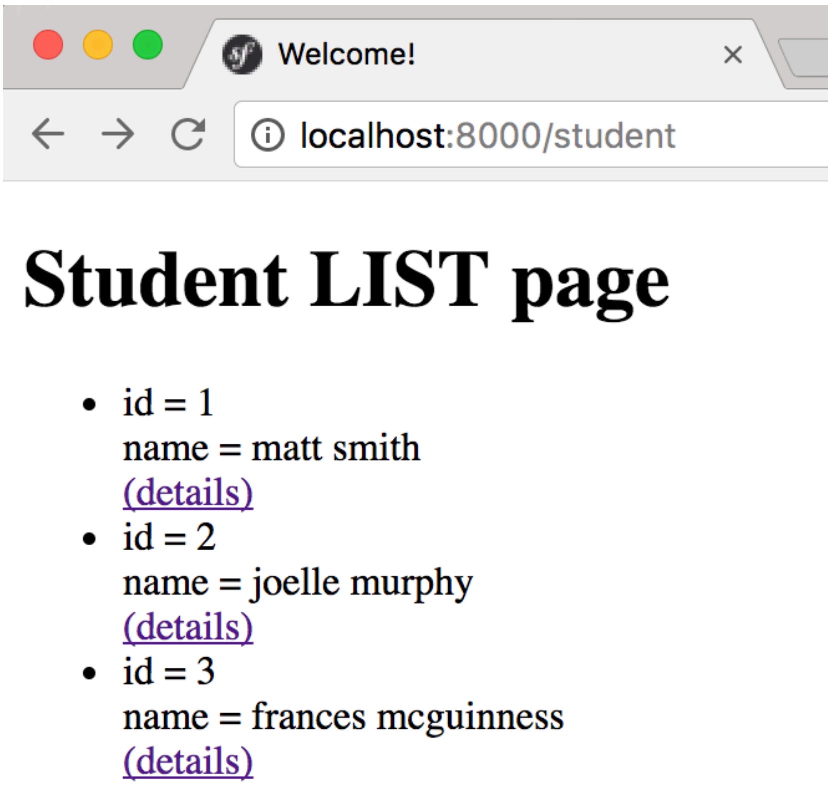
could be represented in Twig as:

```
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.



Student LIST page

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

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(int $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 %}

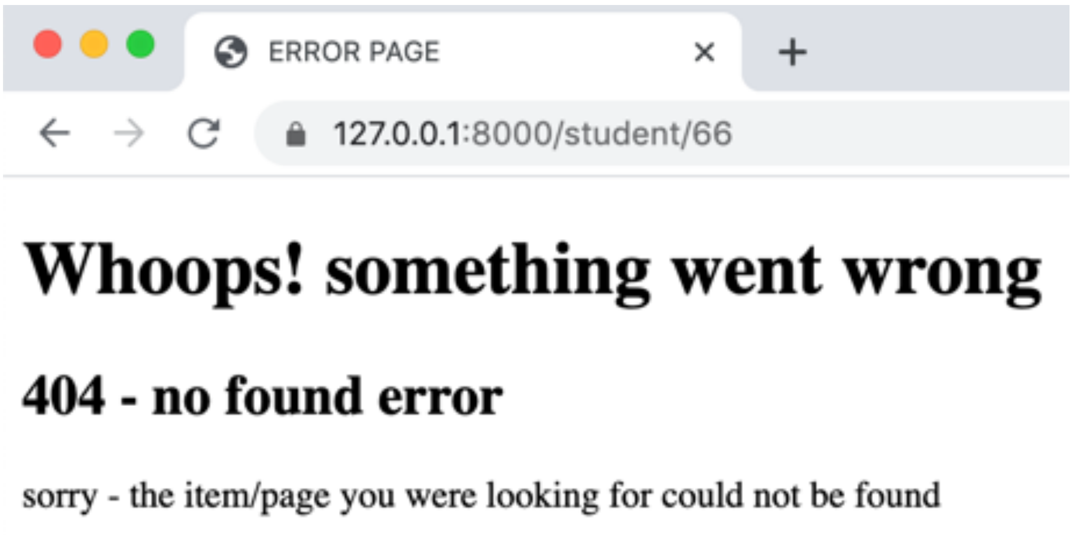
{% block body %}
    <h1>Whoops! something went wrong</h1>

    <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.



Whoops! something went wrong

404 - no found error

sorry - the item/page you were looking for could not be found

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

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 Setting the database connection URL for MySQL

NOTE: This chapter assumes you are starting from the Student basic project from the end of the last chapter…

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` (or whatever **you want to name your database …**)

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:

```
MYSQL_USER=root
MYSQL_PASSWORD=SQLpass
MYSQL_HOST=127.0.0.1
MYSQL_PORT=3306
MYSQL_DATABASE=web3
DATABASE_URL=mysql://${MYSQL_USER}:${MYSQL_PASSWORD}@${MYSQL_HOST}:${MYSQL_PORT}/${MYSQL_DAT
```

NOTE:

- if you use exactly the parameter names above, then you are already using the same values needed to publish your website on Fortrabbit - so it makese sense just to use tese from the word go…

5.3 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`.

5.4 Quick start

5.5 Creat a database

- create the database scema defined in the `.env` file

  `doctrine:database:create'` (do:da:cr‘)

5.6 3 main commands for working with databases

Once you've learnt how to work with Entity classes and Doctrine, these are the 4 commands you need to know (executed from the CLI console `php bin/console ...`):

- create a migration PHP class, containing the SQL to updatee the DB scema to match the entity classes in `/src`

  ‘make:migration’ (`ma:mi`)

- execute a migration (to update schema to match entity classes)

  `doctrine:migrations:migrate` (`do:mi:mi`) (or possibly `doctrine:schema:update --force`)

- load all initial DB data declared in fixture classes

  `doctrine:fixtures;load` (`do:fi:lo`)

5.7 Other useful commands

Validate the DB schema against the entity classes in `/src`

`doctrine:schema:validate`

Run a simple SQL query, to check data in the DB tables:

``` `doctrine:query:sql "select * from modules"` ```

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

5.8 Make your database

We can now use the settings in the `.env` file to connect to the MySQL server and create our database schema:

```
$ php bin/console doctrine:database:create
```

Or the abbreviated version:

```
$ php bin/console do:da:cr
```

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 PHP attributes 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 App\Repository\StudentRepository;
use Doctrine\ORM\Mapping as ORM;
```

```
#[ORM\Entity(repositoryClass: StudentRepository::class)]
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', length: 255)]
private $firstName;

#[ORM\Column(type: 'string', length: 255)]
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 StudentRepository 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 Doctrine\Common\Persistence\ManagerRegistry;
  ```

- make the class extend class `ServiceEntityRepository`

  ```
  class StudentRepository extends ServiceEntityRepository
  ```

- add a constructor method:

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

So the full listing for `StudentRepository` is now:

```
namespace App\Repository;

use App\Entity\Student;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
    public function __construct(ManagerRegistry $registry)

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

6.6 Create a migration (a migration `diff` file)

We now will tell Symfony to create a PHP class to run SQL migration commands required to change the structure of the existing database to match that of our Entity classes. We'll use the abbreviated version of `make:migration` below:

```
$ php bin/console ma:mi

Success!

Next: Review the new migration "src/Migrations/Version20180213082441.php"
Then: Run the migration with php bin/console doctrine:migrations:migrate
See https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html
```

NOTE: This is a shorter way of writing the old `doctrine` command: `php bin/console doctrine:migrations:diff`

A migrations SQL script, similar to the following should have been created in `/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 getDescription(): string
    {
        return '';
    }

    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. We'll use the abbreviated version of `doctrine:mirations:migrate` below:

```
$ php bin/console do:mi:mi

    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.

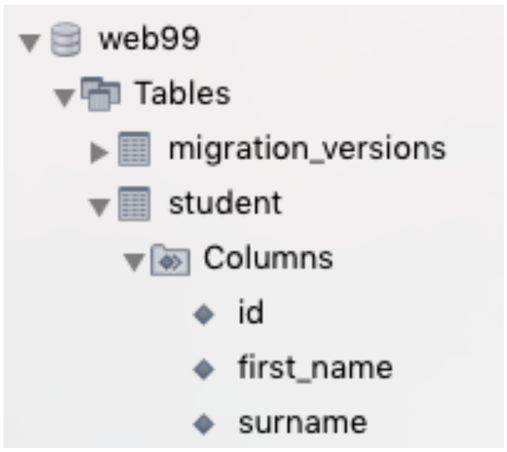


Figure 6.1: Screenshot MySQL Workbench and generated schema and product table.

6.8 Re-validate 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 automatically created 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

   Entity generated! Now let's add some fields!
   You can always add more fields later manually or by re-running this command.

   New property name (press <return> to stop adding fields):
   >

    Success!

   Next: When you're ready, create a migration with make:migration
$
```

In the above `<RETURN>` was pressed to not add any fields automatically. The Maker bundle created 2 classes for us:

- a Student class `src/Entity/Student.php`, containing just a private `id` property and a public `getId()` method
- and a generic StudentRepository class `src/Repository/StudentRepository.php`

That's it - using the `make:entity` CLI tool saves us LOADS of work!

6.11 Use maker to create properties, annotations and accessor methods!

We could automatically create our Student entity and StudentRepository classes from scratch, using the `make` package. It will interactively ask you about fields you wish to create, and add the appropriate annotations and accessor (get/set) methods for you!

If you want to try this, first:

- Delete the entity class: `/src/Entity/Student.php`
- Delete the repository class: `/src/Repository/StudentRepository.php`

Then run the CLI command `make:entity Student`, and at the prompt ask it to create our `firstName` and `surname` text properties (all entities get an auto-incremented `Id` field with us having to ask):

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

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

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):
> firstName

Field type (enter ? to see all types) [string]:
>

Field length [255]:
>

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/Student.php

Add another property? Enter the property name (or press <return> to stop adding fields):
> surname (then keep hitting RETURN for defaults and to complete the entity ...)

 Success!

Next: When you're ready, create a migration with make:migration
```

For each property the Maker bundle wants to know at least 3 things:

- property name (e.g. `firstName` and `surname`)
- property type (default is `string`)
  - for Strings a field length will be asked for (just take the default 255 unless you need more ...)
- whether `NULL` can be stored for property

For `string` properties like `firstName` we just need to enter the property name and hit `<RETURN>` for the defaults (`string`, not nullable). For other types of field you can get a list of types by entering `?` at the prompt:. There are quite a few of them:

```
Field type (enter ? to see all types) [string]:
> ?
```

```
Main types
  * string
  * text
  * boolean
  * integer (or smallint, bigint)
  * float

Relationships / Associations
  * relation (a wizard   will help you build the relation)
  * ManyToOne
  * OneToMany
  * ManyToMany
  * OneToOne

Array/Object Types
  * array (or simple_array)
  * json
  * object
  * binary
  * blob

Date/Time Types
  * datetime (or datetime_immutable)
  * datetimetz (or datetimetz_immutable)
  * date (or date_immutable)
  * time (or time_immutable)
  * dateinterval

Other Types
  * json_array
  * decimal
  * guid
```

6.12 It is BETTER to use the make tool

Personally, I recommend you ALWAYS create your entities with the make CLI tool, since then you know the annotations are correct.

6.13 Final tip - always add a toString() method

When listing values in a table or creating a dropdown HTML form, sometimes Symfony will need to convert an object reference to a string - that's what toString methods are for.

So, rather than trying to do something and getting an error, it is a good habit to ALWAYS write a toString method once you've created an entity class

e.g. add something like the following for the `Student` entity class:

```
public function __toString(): string
{
      returnu $this->firstname . '  ' . $this->surname;
}
```

6.13.1 Tip - IDEs may write a function skeleton for you after you type `__toString` ...

PHPStorm and other IDEs may save you typing by writing a function skeleton for you ...

7

Symfony way of doing database CRUD

7.1 Getting data into the DB

before we can test our DB entity class and repository, we need to get some data into the DB.

Let]'s create a new **route**, in the form: `/students/create/{firstName}/{surname}` that would create a new `Student` row in the DB table containing `{firstName}` and `{surname}`.

We also need to fix our controller, to be able to use the Doctrine DB repository class, rather than our previous D.I.Y. (do-it-yourself) repository class - but more of that later …

7.2 Creating new student records (project `db02`)

Let's add a new route and controller method to our `StudentController` class. This will define the `create()` method that receives parameter `$firstName` and `$surname` extracted from the route `/student/create/{firstName}/{surname}`. This is all done automatically for us, through Symfony seeing the route parameters in the `Route(...)` attribute that immediately precedes the controller method. The 'signature' for our new `create(...)` method names 2 parameters that match those in the `#[Route(...)` annotation comment `create($firstName, $surame)`:

```
#[Route('/student/create/{firstName}/{surname}', name: 'student_create')]
public function create(string $firstName, string $surname)
```

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);
$student->setSurname($surname);
```

Then we see the Doctrine code, to get a reference to an ORM object `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.

What we need is a special object reference `$doctrine` that is a `ManagerRegistry` object, and then we can write the following to create a variable `$em` that is a reference to the ORM `EntityManager`:

```
$em = $doctrine->getManager();
```

One aspect of Symfony that make take some getting used to is how to get access to service objects like the `ManagerRegistry`. We add a typed argument to a controller method signature, and Symfony will **inject** a reference to the desired object. So to get a variable `$doctrine` that is a reference to the Doctrine `ManagerRegistry` we need to do 2 things:

1. Add an appropriate `use` statement at the top of the file
2. Add this as a parameter to the method signature: `ManagerRegistry $doctrine`

So the beginning of our class will have this new `use` statement:

```
<?php

namespace App\Controller;

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

use App\Repository\StudentRepository;
use Doctrine\Persistence\ManagerRegistry;
```

And our method will look like this:

```
#[Route('/student/create/{firstName}/{surname}', name: 'student_create')]
public function create(string $firstName, string $surname, ManagerRegistry $doctrine)
{
    $student = new Student();
    $student->setFirstName($firstName);
    $student->setSurname($surname);
```

Let's now get that reference to the EntityManager:

```
$em = $doctrine->getManager();
```

Now we can use this `$em` EntityManager object to queue the new object for storate in the DB (persist), and then action all queded DB updated (flush):

```
$em->persist($student);
$em->flush();
```

When the `$student` object has been successfully added to the DB, its `id` property will be updated to new auto-generated primary key integer. So we can access the ID of this object via `$student->getId()`.

Finally, let's create a `Response` message to the user telling them the ID of the newly created DB row. For this we need to add a `use` statement, so we can create a `Response` object.

So the code for our create controller method is:

```
<?php

namespace App\Controller;

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

use App\Repository\StudentRepository;
use Doctrine\Persistence\ManagerRegistry;

// we need to add a 'use' statement so we can create a Response object...
use Symfony\Component\HttpFoundation\Response;
...

#[Route('/student/create/{firstName}/{surname}', name: 'student_create')]
public function create(string $firstName, string $surname, ManagerRegistry $doctrine): Response
{
    $student = new Student();
    $student->setFirstName($firstName);
    $student->setSurame($surname);

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

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

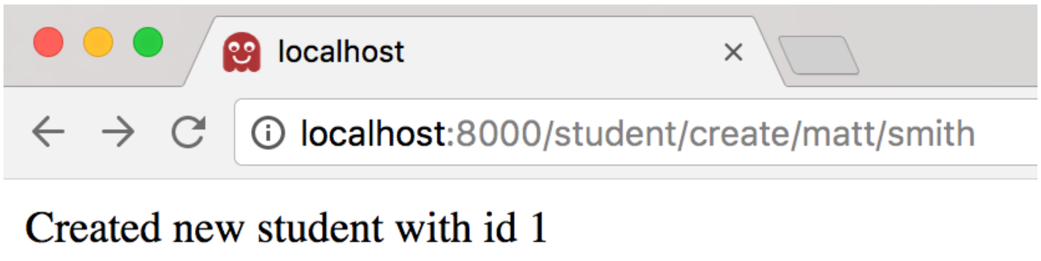
```
        // actually executes the queries (i.e. the INSERT query)
        $em->flush();

        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/student/create/matt/smith
```

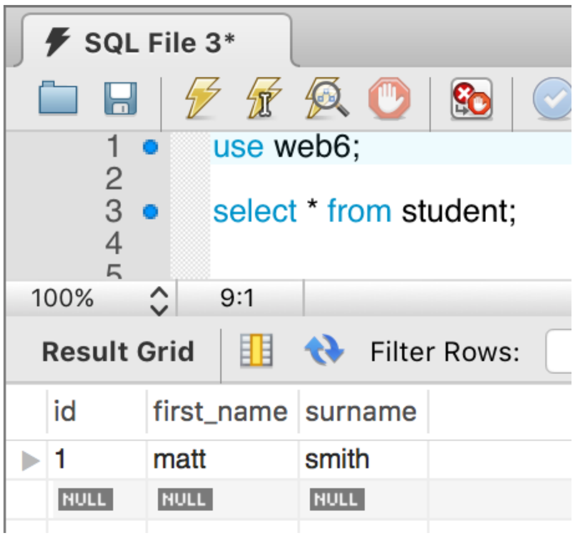
Figure 7.1 shows how a new record `matt smith` is added to the database table via route `/student/create/{firstName}/{surname}`.



Created new student with id 1

Figure 7.1: Creating new student 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.



```
use web6;

select * from student;
```

id	first_name	surname
1	matt	smith
NULL	NULL	NULL

Figure 7.2: Controller created records in PHPMyAdmin.

7.3 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)
```

As usual, we can use the 2-letter shortcut to make writing this SQL query command a bit faster:

```
$ php bin/console do:qu:sql "select * from student"
```

7.4 Updating the `list()` method to use Doctrine

Of course, we already have a route for viewing `Student` objects: `/student/list`. So we just have to update the code for this method to use the generated `StudentRepository` rather than our original D.I.Y. class.

If we have a reference to the ORM ManagerRegistry (`$doctrine`) we can get a reference to the repository class for any of our entity classes as follows:

```
$repositoryObject = $doctrine->getRepository(<EntityClass>::class);
```

so to get a `$studentRepository` object we write:

```
$studentRepository = $doctrine->getRepository(Student::class);
```

Again, we use the Symfony param-converter to **inject** an object reference for us, by simply adding a new parameter to the `list(...)` method signature. So our `list(...)` mehod now looks as follows:

```
#[Route('/student', name: 'student_list')]
public function list(ManagerRegistry $doctrine): Response
{
    $studentRepository = $doctrine->getRepository(Student::class);
    $students = $studentRepository->findAll();

    $template = 'student/list.html.twig';
```

```
        $args = [
            'students' => $students
        ];
        return $this->render($template, $args);
    }
```

7.5 Doctrine Repository "free" methods

Doctrine repositories offer us lots of useful methods, including:

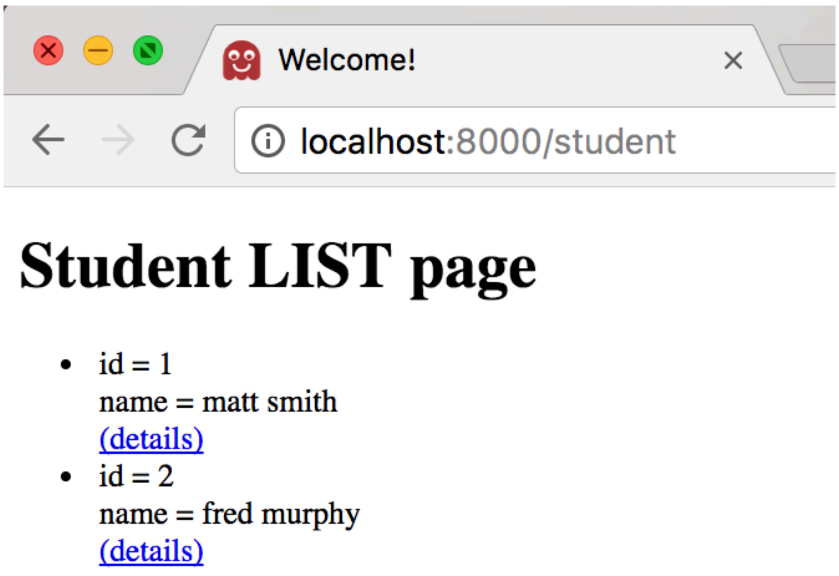
```
// query for a single record by its primary key (usually "id")
$student = $repository->find($id);

// 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');
```

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`.

7.6 Deleting by id

Let's define a delete route `/student/delete/{id}` and a `delete()` 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}', name: 'student_delete')]
public function delete(int $id, ManagerRegistry $doctrine)
{
    $studentRepository = $doctrine->getRepository(Student::class);
    $student = $studentRepository->find($id);

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

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

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

7.7 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}', name: 'student_update')]
    public function update(int $id, string $newFirstName, string $newSurname, ManagerRegistry $d
    {
        $studentRepository = $doctrine->getRepository(Student::class);
        $student = $studentRepository->find($id);

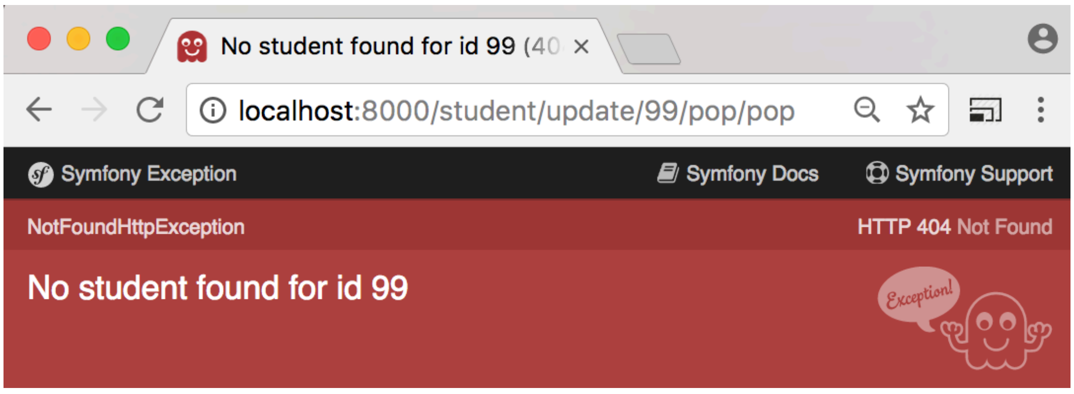
//        $student = $doctrine->getRepository(Student::class)->find($id);

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

        $em = $doctrine->getManager();
        $em->flush();

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

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
Exception!

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 `update()` has been written a little differently:

- we are getting the reference to the repository via the entity manager `$em->getRepository('App:Student')`
- we could also have 'chained' the `find($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($id)`). I.e. we could have written `$student = $doctrine->getRepository(Student::class)->find($id)`. This would be 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 `student_show` for the current object's `id`

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

7.8 Updating our show action

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

```
#[Route('/student/{id}', name: 'student_show')]
public function show(int $id, ManagerRegistry $doctrine): Response
{
    $studentRepository = $doctrine->getRepository(Student::class);
    $student = $studentRepository->find($id);
```

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

```
#[Route('/student/{id}', name: 'student_show')]
public function show(int $id, ManagerRegistry $doctrine): Response
{
    $studentRepository = $doctrine->getRepository(Student::class);
    $student = $studentRepository->find($id);

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

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

[1] read about it at Wikipedia

```
        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
        );
    }
```

7.9 Redirecting to show after create/update

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

```
    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. add an `update(...)` method to be as follows:

```
    /**
     * @Route("/student/update/{id}/{newFirstName}/{newSurname}")
     */
    public function update($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()
        ]);
    }
```

7.10 Given `id` let Doctrine find Product automatically (project `db03`)

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.

So while we list an `{id}` parameter in the attribute preceding the method, in the method signautre itself we have a parmater that is a reference to a complete `Student` object, retrieved from the DB using the provided id value!

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

```
    #[Route('/student/{id}', name: 'student_show')]
    public function show(int $id, ManagerRegistry $doctrine): Response
    {
        $studentRepository = $doctrine->getRepository(Student::class);
        $student = $studentRepository->find($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 show(Student $student): Response
    {
        $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}', name: 'student_delete')]
    public function delete(Student $student, ManagerRegistry $doctrine)
    {
        // store ID so can report it later
        $id = $student->getId();

        // tells Doctrine you want to (eventually) delete the Student (no queries yet)
        $em = $doctrine->getManager();
        $em->remove($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}', name: 'student_update')]
    public function update(Student $student, string $newFirstName, string $newSurname, ManagerRe
    {
        $student->setFirstName($newFirstName);
        $student->setSurname($newSurname);

        $em = $doctrine->getManager();
        $em->flush();

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

NOTE - we will now get ParamConverter errors/exceptions rather than 404 errors if no record matches ID through … so need to deal with those in a different way …

7.11 Creating the CRUD controller automatically from the CLI (project `db04`)

Here is something you might want to look into - automatic generation of controllers and Twig templates (we'll look at this in more detail in a later chapter).

NOTE: If trying out thew CRUD generation below, then make a copy of your current project, and try this out on the copy. Then discard the copy, so you can carry on working on your student project in the next chapter.

To try this out do the following:

1. Delete the `StudentController` class, since we'll be generating one automatically
2. Delete the `templates/student` directory, since we'll be generating those templates automatically
3. Delete the `var` directory, since we'll be generating one automatically
4. Then use the make crud command:

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

You should see the following output in the CLI:

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

     created: src/Controller/StudentController.php
     created: src/Form/Student1Type.php
     created: templates/student/_delete_form.html.twig
```

```
created: templates/student/_form.html.twig
created: templates/student/edit.html.twig
created: templates/student/index.html.twig
created: templates/student/new.html.twig
created: templates/student/show.html.twig

 Success!

Next: Check your new CRUD by going to /student
```

You should find that you have now forms for creating and editing Student records, and controller routes for listing and showing records, and Twig templates to support all of this…

NOTE: As usually, if you get any messages about 'Route not found' or whatever, you need to delete the `/var/cache`,. Or the whole `/var` folder (as long as you aren't using an SQLite file in there instead of MySQL …)

If you look at the code for controller methods like `show(...)` and `delete(...)` you'll find they are very similar to what we wrote by hand previously. For example the `show(...)` method should look something like the following:

```php
#[Route('/{id}', name: 'student_show', methods: ['GET'])]
public function show(Student $student): Response
{
    return $this->render('student/show.html.twig', [
        'student' => $student,
    ]);
}
```

which is just a more succinct way of writing the same as we had before

```php
#[Route('/student/{id}', name: 'student_show')]
public function show(Student $student): Response
{
    $template = 'student/show.html.twig';
    $args = [
        'student' => $student
    ];

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

8 Fixtures - setting up a database state

8.1 Initial values for your project database (project `db05`)

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 Fixtures SAVE YOU TIME!

I cannot stress how **useful** fixtures are when making many changes to a DB structure - as you'll be likely be doing when first developing Symfony projects.

It should be this easy to resolve mismatches between your code and your database schema:

1. delete the DB (or choose a new DB name in your `.env` file - I just add 1 to the DB number - evote01, evote02 etc.)
2. create the DB: `do:da:cr`
3. delete the **contents** of your `migrations` folder (but not the folder itself)

4. Make a new migration: `ma:mi`
5. run your migration: `do:mi:mi`
6. Load your fixtures: `do:fi:lo`

DONE - your DB is now fully in-synch with your entity classes.

If you do NOT have fixtures, you'll now waste time entering lots of test data by hand - every time you have to delete and re-create your DB ....

8.3 Installing and registering the fixtures bundle

8.3.1 Install the bundle

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

```
composer req orm-fixtures
```

You should now see a new directory created `/src/DataFixtures`. Also, there is a sample fixtures class provided `AppFixtures.php`:

```
<?php

namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;

class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        // $product = new Product();
        // $manager->persist($product);

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

Since you'll generally be creating a range of fixture files, named for their content it's a good idea just to delete this file: `/src/DataFixtures/AppFixtures.php`.

8.4 Writing the fixture classes

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,by implementing the `ContainerAwareInterface`.

Let's create a class to create 3 objects for entity `App\Entity\Student`. The class will be declared in file `/src/DataFixtures/StudentFixtures.php`. However, we can generate the skelton for each fixture class using the CLI `make` tool. We also need to add a `use` statement so that our class can make use of the `Entity\Student` class.

The **make** feature will create a skeleton fixture class for us. So let's make class `StudentFixtures`:

```
$ php bin/console make:fixtures StudentFixtures

  created: src/DataFixtures/StudentFixtures.php

   Success!

  Next: Open your new fixtures class and start customizing it.
  Load your fixtures by running: php bin/console doctrine:fixtures:load
  Docs: https://symfony.com/doc/master/bundles/DoctrineFixturesBundle/index.html
```

Since we are going to be creating instance-objects of class `Student` we need to add a `use` statement:

```
...

use App\Entity\Student;

class StudentFixtures extends Fixture
{
```

When we use the CLI command `doctrine:fixtures:load` the `load(...)` method of each fixture object is invoked. So now we need to implement the details of our `load(...)` method for our new class `StudentFixtures`.

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): void
{
    $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();
    }
```

8.5 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
```

or the shorter version: `php bin/console do:fi:lo`

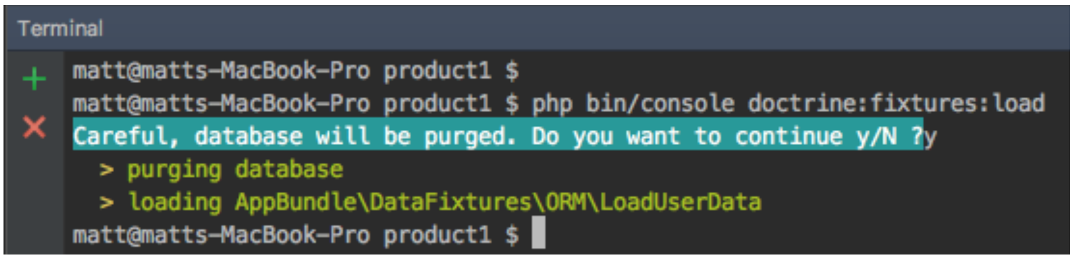
You should then be asked to enter `y` (for YES) if you want to continue:

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

Careful, database "web3" will be purged. Do you want to continue? (yes/no) [no]:

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

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…)

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:
```



```
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.

```
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)
```

NOTE:

If you have loaded fixtures several times, or created other records, then the index of the records may NOT begin at 1.

If you need the id's to start at 1, you can delete DB / delete migrations / create DB / create migration / run migration / load fixtures - for a completely fresh dataabase.

8.6 User Faker to generate plausible test data (project `db06`)

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

NOTE: The original PHP Faker was from `fzaninotto/faker`. But this was a muilti-lingual project, being over 3Mb download. So I've created an English-only fork of that project for student use (< 200k). You can read more in the README on Github.

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 require mattsmithdev/faker-small-english

   Using version ^0.1.0 for mattsmithdev/faker-small-english
   ./composer.json has been updated
   Loading composer repositories with package information
   ...
   Executing script assets:install --symlink --relative public [OK]
   ```

   - you'll now see a `mattsmithdev` folder in `/vendor` containing the Faker classes

2. Add a `uses` statement in our `/src/DataFixtures/LoadStudents.php` class, so that we can make use of the `Faker` class:

   ```
   use Mattsmithdev\FakerSmallEnglish\Factory;
   ```

3. 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:

   ```
   use Mattsmithdev\FakerSmallEnglish\Factory;

   ...

   public function load(ObjectManager $manager): void
   {
       $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();
   }
   ```

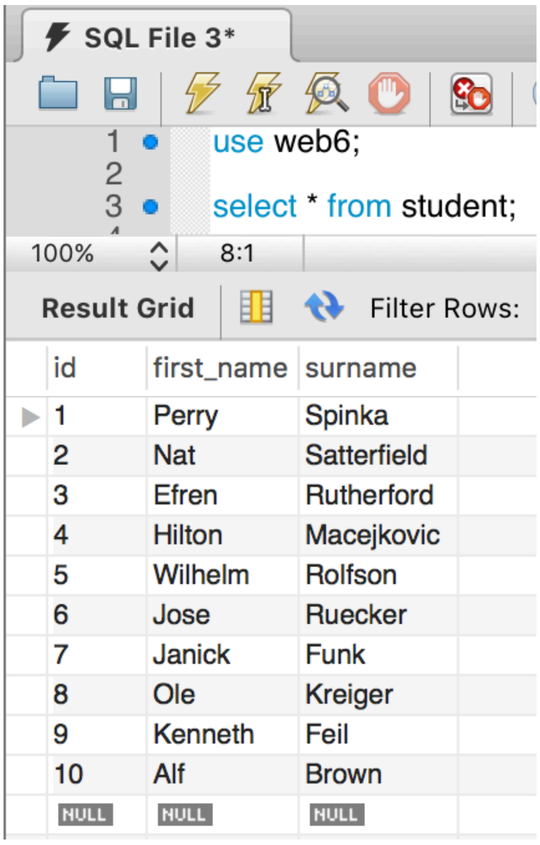
4. use the CLI Doctrine command to run the fixtures creation method:

   ```
   $ php bin/console do:fi:lo
   ```

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

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.

8.6.1 The Faker projects

Learn more about the `Faker` projects:

- Matt's small version of the library (Github)
  - https://github.com/dr-matt-smith/faker-small-english
- Matt's small version of the library (Packagist)
  - https://packagist.org/packages/mattsmithdev/faker-small-english

- the FZaninotto library Matt's project is based on
  - https://github.com/fzaninotto/Faker
- FakerPHP - which has replaced FZaninotto Faker library
  - https://fakerphp.github.io/
  - although it is $>$ 3Mb so still an issue (I'll create a fork using my small Faker library when I have a chance ....)

9

Relating object in different Fixture classes

9.1 Remember this for later

Alhtough, relating entities is covered later in the book, relating fixtures is here, as part of this fixtrues chapter.

So, although you may wish to just read this chapter, but leave its implementation until later, do read through and see how easy it is to create related fixtures for objects of different classes.

9.2 Related objects - option 1 - do it all in one Fixture class

If you need to create fixtures involving related objects of different classes, one solution is to have a single Fixtures class, and create **ALL** your objects in the `load()` method.

However, if you have 100s of objects this makes a pretty long and messy class.

9.3 Related objects - option 2 - store references to fixture objects (project `db07`)

A better solution involves storing a reference to objects created in one fixture class, than can be used to retrieve those objects for use in another fixture class.

Let's create a simple, two-class example of `Student` and `Campus` objects, e.g.:

- Student 1 "Matt Smith" is of Campus "Blanchardstown"
- Student 2 "Sinead Murphy" is of Campus "Tallaght"

Since `Campus` comes alphabetically before `Student`, then let's create our 2 `CAmpus` objects and store references to them, in a new fixtures class `CampusFixtures`

9.3.1 Category entity class

Create a class `Campus` with a single `name` String property.

Use the CLI make tool: `php bin/console ma:en Campus`

9.3.2 CampusFixtures class

Create a class `CampusFixtures` class and create 3 `Campus` objects for "Blanchardstown", "Tallaght" and "City" the usual way.

Use the CLI tools to create your fixtures class: `php bin/console ma:fi CampusFixtures`

```php
<?php

namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;

use App\Entity\Campus;

class CampusFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        $campus1 = new Campus();
        $campus1->setName('Blanchardstown');
        $campus2 = new Campus();
        $campus2->setName('Tallaght');
        $campus3 = new Campus();
        $campus3->setName('City');

        $manager->persist($campus1);
        $manager->persist($campus2);
        $manager->persist($campus3);

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

Now we need to also add 2 named **references** to these `Category` objects. It is these that will allow us to retrieve references to these `Campus` obejcts in our `Student` fixtures class:

```
    public function load(ObjectManager $manager): void
    {
        $campus1 = new Campus();
        $campus1->setName('Blanchardstown');
        $campus2 = new Campus();
        $campus2->setName('Tallaght');
        $campus3 = new Campus();
        $campus3->setName('City');

        $manager->persist($campus1);
        $manager->persist($campus2);
        $manager->persist($campus3);

        $manager->flush();

        // create named references
        $this->addReference('CAMPUS_BLANCH', $campus1);
        $this->addReference('CAMPUS_TALLAGHWT', $campus2);
        $this->addReference('CAMPUS_CITY', $campus3);
    }
```

9.4 updating our `Student` entity class

We can relate entities through properties of type `relation`. Let's add a `campus` property to `Student` objects, relating each `Student` object to one `Campus` object.

Use the make CLI tool to **add** a new property to the `Student` entity class: `bin/console ma:en Student`. NOTE that both to create a new entity class, and to edit an existing entity class we use the same CLI command `make:entity` or `ma:en`.

We only have to answer a few questions: - **name** of new property = `campus` - **data type** of new property = `relation` - **class** the property is relating Student objects to = `Campus` - **relationship type** = `ManyToOne` (many Students related to one Campus) - (once you get to the null-able question you can just keep hitting RETURN to accept all defaults and complete the update of this entity class)

Here is a full summary of the CLI interaction to add this property:

```
php bin/console ma:en Student

Your entity already exists! So let's add some new fields!

New property name (press <return> to stop adding fields):
> campus

Field type (enter ? to see all types) [string]:
> relation

What class should this entity be related to?:
> Campus

What type of relationship is this?
------------ ---------------------------------------------------------------------
 Type         Description
------------ ---------------------------------------------------------------------
 ManyToOne    Each Student relates to (has) one Campus.
              Each Campus can relate to (can have) many Student objects

 OneToMany    Each Student can relate to (can have) many Campus objects.
              Each Campus relates to (has) one Student

 ManyToMany   Each Student can relate to (can have) many Campus objects.
              Each Campus can also relate to (can also have) many Student objects

 OneToOne     Each Student relates to (has) exactly one Campus.
              Each Campus also relates to (has) exactly one Student.
------------ ---------------------------------------------------------------------

Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
> ManyToOne

Is the Student.campus property allowed to be null (nullable)? (yes/no) [yes]:
>
```

If you examine the `Student` entity class you'll now see a new property `campus` as follows:

```
#[ORM\ManyToOne(targetEntity: Campus::class, inversedBy: 'students')]
private $campus;
```

There are also get/set accessor methods for this property.

NOTE: If you look at the `Campus` entity, you'll see that from one of the defaults we accepted, there is now a recipricol array property `students`, so that given a `Campus` object we have an array of all `Student` objects related to it!

9.4.1 Create and run migration

Since we changed our entity clases, we need to create and run a new migration, to sychronise the DB scheme to match these entity classes:

```
$ php bin/console make:mi

  Success!

$ php bin/console do:mi:mi

     WARNING! You are about to execute a migration in database "web4" that could result in schema c
     > y

    [notice] Migrating up to DoctrineMigrations\Version20220111214334
    [notice] finished in 170.8ms, used 20M memory, 1 migrations executed, 4 sql queries
```

9.4.2 `StudentFixtures` class

We can now update our fixtures class for `Student` objects, to relate each new `Student` object to a `Campus` object.

First, we need to add a `use` statement, so that in our `StudentFixtures` class we can refer to `Campus` objects.

```
<?php

namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;

use App\Entity\Student;
use App\Entity\Campus;
```

Next, at the beginning of the `load(...)` method, the first thing we'll do is retrieve the references to the 3 campuses, into suitable named variables:

```
public function load(ObjectManager $manager): void
{
    // create named references
    $campusBlanchardstown = $this->getReference('CAMPUS_BLANCH');
    $campusTallaght = $this->getReference('CAMPUS_TALLAGHT');
    $campusCity = $this->getReference('CAMPUS_CITY');
```

We can then go ahead as before, create the `Student` objects, and set their campuses to these `Campus` object references. So we'll set students 1 and 2 to the Blanchardstown campus, and student 3 to the Tallaght campus:

```
// create our 3 Student objects
$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');

// set the campus for the students
$s1->setCampus($campusBlanchardstown);
$s2->setCampus($campusBlanchardstown);
$s3->setCampus($campusTallaght);
```

9.5 Dependent Fixtures - order of loading is important!

The `StudentFixtures` class is dependent on the `CampusFixtures` class. So we must declare this dependency so that these fixtures are executed in the correct order.

The Doctrine ORM provides a special interface for delcaring fixture dependencies, so we need to add a `use` statement in the `StudentFixtures` class as follows:

```
use Doctrine\Common\DataFixtures\DependentFixtureInterface;
```

We must declare that class `StudentFixtures` implements this interface:

```
class StudentFixtures extends Fixture implements DependentFixtureInterface
```

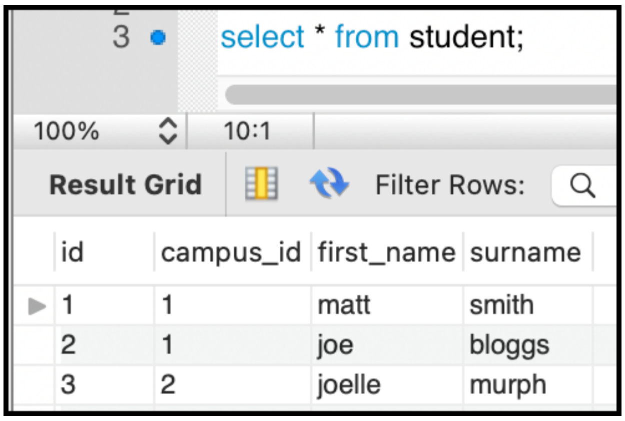
The interface demands that we implement a method `getDependencies()`. We can so do, stating that class `StudentFixtures` is dependent on the `CampusFixtures` class:

```
    public function getDependencies()
    {
        return [
            CampusFixtures::class,
        ];
    }
```

So the complete `StudentFixtures` class looks as follows:

```
    <?php

namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;

use App\Entity\Student;

use Doctrine\Common\DataFixtures\DependentFixtureInterface;


class StudentFixtures extends Fixture implements DependentFixtureInterface
{
    public function load(ObjectManager $manager): void
    {
        // create named references
        $campusBlanchardstown = $this->getReference('CAMPUS_BLANCH');
        $campusTallaght = $this->getReference('CAMPUS_TALLAGHT');
        $campusCity = $this->getReference('CAMPUS_CITY');

        // create our 3 Student objects
        $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');
        // set the campus for the students
        $s1->setCampus($campusBlanchardstown);
        $s2->setCampus($campusBlanchardstown);
        $s3->setCampus($campusTallaght);

        // save these objects to the DB
        $manager->persist($s1);
        $manager->persist($s2);
        $manager->persist($s3);

        $manager->flush();
    }

    public function getDependencies()
    {
        return [
            CampusFixtures::class,
        ];
    }
}
```

See Figure 9.1 to see the `Product` objects listed from the database, with their linked categories.



```
select * from student;
```

id	campus_id	first_name	surname
1	1	matt	smith
2	1	joe	bloggs
3	2	joelle	murph

Figure 9.1: Screenshot of Product fixtures with realted Categories.

Learn more about Symfony fixtures on the Symfony website:

- https://symfony.com/bundles/DoctrineFixturesBundle/current/index.html#loading-the-fixture-files-in-order

10

Foundry Factories for powerful fixture generation

10.1 Foundry and FakerPHP

Foundry is a relatively new Symfony library, allowing us to make **Factories** to easily generate 10s or 100s of test data, exploiting Faker features

Learn more at Symfonycasts:

- Foundry
  - https://symfonycasts.com/screencast/symfony-doctrine/foundry
- FakerPHP
  - https://symfonycasts.com/screencast/symfony-doctrine/foundry-tricks#play

10.2 Adding Foundry to our project (project `db08`)

First, we need to add Foundry to our project:

```
$ composer require zenstruck/foundry
```

You should see some new folders appear in `/vendor` from `zenstruck` and `fakerphp`.

10.3 Generate more students with Faker

First let's see how we generate 10 more students using Faker by itself. Update the `load()` method of class `StudentFixtures` to end with this loop

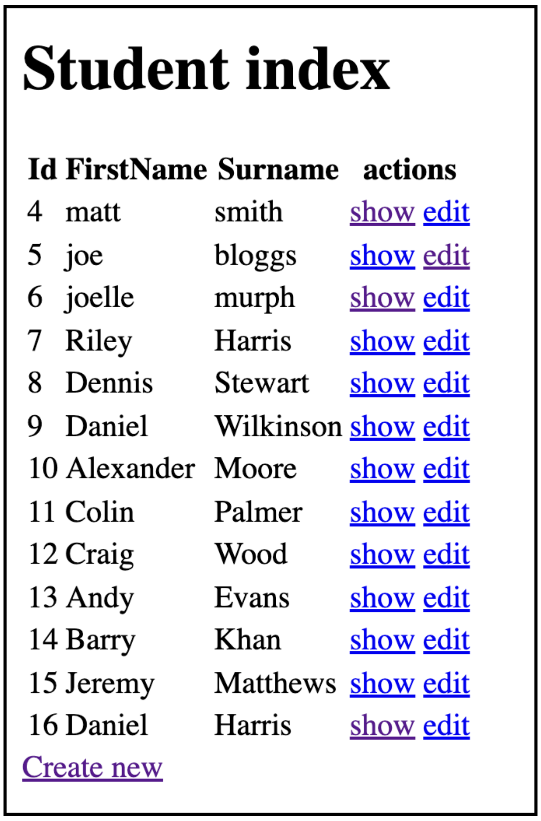
```
namespace App\DataFixtures;

...

// we'll need to refer to the 'Faker' class
use Mattsmithdev\FakerSmallEnglish\Factory;

class StudentFixtures extends Fixture implements DependentFixtureInterface
{
     public function load(ObjectManager $manager): void
    {
        // create named references
        $campusBlanchardstown = $this->getReference('CAMPUS_BLANCH');
        $campusTallaght = $this->getReference('CAMPUS_TALLAGHT');
        $campusCity = $this->getReference('CAMPUS_CITY');

        ... as before ..

        // -- make 10 students with Faker
        $faker = Factory::create();
        $numStudents = 10;
        for ($i=0; $i < $numStudents; $i++) {
            $student = new Student();

            $firstName = $faker->firstNameMale;
            $surname = $faker->lastName;

            $student->setFirstName($firstName);
            $student->setSurname($surname);
            $student->setCampus($campusBlanchardstown);

            $manager->persist($student);
        }

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

So we can see that with Faker, we can loop through and create more `Student` objects to add to the database, with random, plausible data for names.

If we load fixtures with `symfony console do:fi:lo` (or `symfony console doctrine:fixtures:load`) and click the `students` link, we'll see 13 students in total, 3-13 being the ones created by Faker:

```
$ symfony console do:fi:lo

 Careful, database "web5" will be purged. Do you want to continue? (yes/no) [no]:
 > y

   > purging database
   > loading App\DataFixtures\CampusFixtures
   > loading App\DataFixtures\StudentFixtures
```

Figure 10.1 shows a screenshot of a web browser listing the students.



Student index

Id	FirstName	Surname	actions
4	matt	smith	show edit
5	joe	bloggs	show edit
6	joelle	murph	show edit
7	Riley	Harris	show edit
8	Dennis	Stewart	show edit
9	Daniel	Wilkinson	show edit
10	Alexander	Moore	show edit
11	Colin	Palmer	show edit
12	Craig	Wood	show edit
13	Andy	Evans	show edit
14	Barry	Khan	show edit
15	Jeremy	Matthews	show edit
16	Daniel	Harris	show edit

Create new

Figure 10.1: The Faker-generated students seen in web browser.

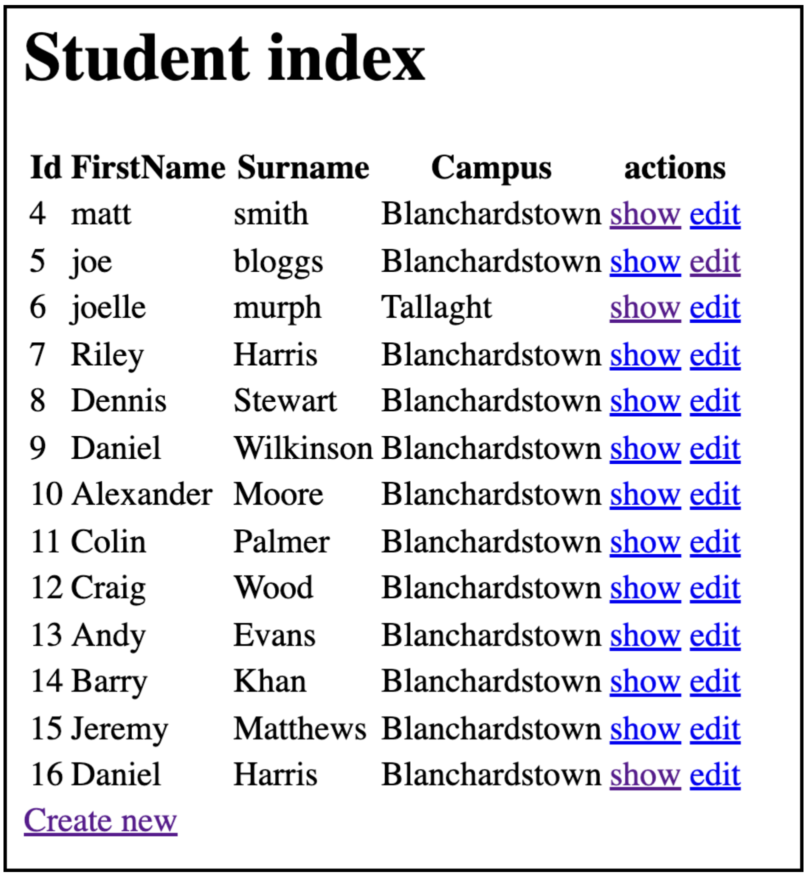
10.4 View campus for students

Let's edit the Twig template listing the students, adding another column to see the campus for each student. Edit the loop in `templates/student/index.html.twig` to look as follows, and

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

{% block title %}Student index{% endblock %}

{% block body %}
  <h1>Student index</h1>

  <table class="table">
      <thead>
          <tr>
              <th>Id</th>
              <th>FirstName</th>
              <th>Surname</th>
              <th>Campus</th> <!-- **** add Campus column heading ****  -->
              <th>actions</th>
          </tr>
      </thead>
      <tbody>
      {% for student in students %}
          <tr>
              <td>{{ student.id }}</td>
              <td>{{ student.firstName }}</td>
              <td>{{ student.surname }}</td>
              <td>{{ student.campus.name }}</td> <!-- ****  output linked campus name ****  --
              <td>
                  <a href="{{ path('student_show', {'id': student.id}) }}">show</a>
                  <a href="{{ path('student_edit', {'id': student.id}) }}">edit</a>
              </td>
          </tr>
```

Figure 10.2 shows a screenshot of a web browser listing the students with the extra campus column.

However, using Foundry we can do things easier, and with more sophistication …



Student index

Id	FirstName	Surname	Campus	actions
4	matt	smith	Blanchardstown	show edit
5	joe	bloggs	Blanchardstown	show edit
6	joelle	murph	Tallaght	show edit
7	Riley	Harris	Blanchardstown	show edit
8	Dennis	Stewart	Blanchardstown	show edit
9	Daniel	Wilkinson	Blanchardstown	show edit
10	Alexander	Moore	Blanchardstown	show edit
11	Colin	Palmer	Blanchardstown	show edit
12	Craig	Wood	Blanchardstown	show edit
13	Andy	Evans	Blanchardstown	show edit
14	Barry	Khan	Blanchardstown	show edit
15	Jeremy	Matthews	Blanchardstown	show edit
16	Daniel	Harris	Blanchardstown	show edit

Create new

Figure 10.2: Students showing campus.

10.5 Creating a Foundry Factory

At the core of Foundry is the **Factory**. So we need to first generate a `Student` Factory. We make a factory with the Symfony console command `make:factory`, then we need to choose which **Entity** class we wish to make the factory for (`Student` in this case):

```
% symfony console make:factory
 // Note: pass --test if you want to generate factories in your tests/ directory
 // Note: pass --all-fields if you want to generate default values for all fields, not only re

 Entity class to create a factory for:
  [0] App\Entity\Campus
  [1] App\Entity\Student
 > 1

 created: src/Factory/StudentFactory.ph
     -- Success!
 Next: Open your new factory and set default values/states.
 Find the documentation at https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.ht
```

You should now see a new class `src/Factory/StudentFactory` in your project. While this lists several methods, the important one is `getDefaults()`, which we can see has been made to use FakerPHP to add text for string properties `firstName` and `surname`.

```
namespace App\Factory;

...

final class StudentFactory extends ModelFactory
{
    ...
    protected function getDefaults(): array
    {
        return [
            'firstName' => self::faker()->text(),
            'surname' => self::faker()->text(),
        ];
    }
```

We can now replace all those lines of code using Faker in the `StudentFixtures` class with just one line (and one `use` statement):

```
namespace App\DataFixtures;
```

```
...

use App\Factory\StudentFactory;

class StudentFixtures extends Fixture implements DependentFixtureInterface
{
     public function load(ObjectManager $manager): void
    {
        // create named references
        $campusBlanchardstown = $this->getReference('CAMPUS_BLANCH');
        $campusTallaght = $this->getReference('CAMPUS_TALLAGHT');
        $campusCity = $this->getReference('CAMPUS_CITY');

        ... as before ..

        // -- make 10 students with Foundry
        StudentFactory::new()->createMany(10);

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

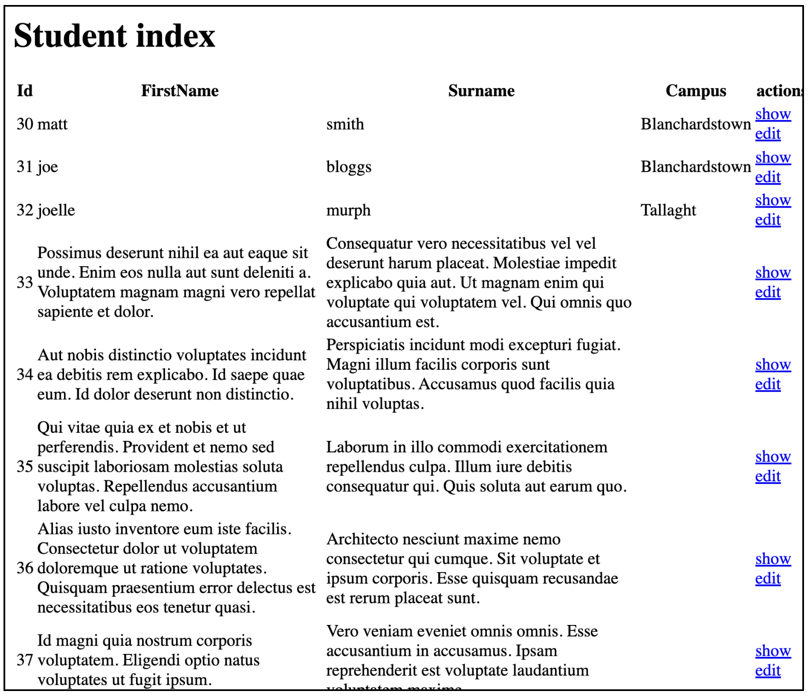
So alredy we can see how using Foundry Factories is making our fixture classes is much simpler. However, if we load fixtures then list students in the browser we get a NULL error (since there is no `Campus` link to these students). First, let's edit our Twig template, to avoid NULL errors - we'll only try to display the `campus.name` string if the `campus` property is not null. So edit `templates/student/index.html.twig` as follows:

```
        {% for student in students %}
            <tr>
                <td>{{ student.id }}</td>
                <td>{{ student.firstName }}</td>
                <td>{{ student.surname }}</td>
                <td> <!-- ****** output linked campus name ---- -->
                    {% if not student.campus is null %}
                        {{ student.campus.name }}
                    {% endif %}
                </td>
```

Now if we list students in the browser (Figure 10.3) we can see that, although there is random text for the names, they are not plausible names.

That's easily fixed - we just need to edit the `getDefaults()` of class `src/Factory/StudentFactory`:

```
namespace App\Factory;
```



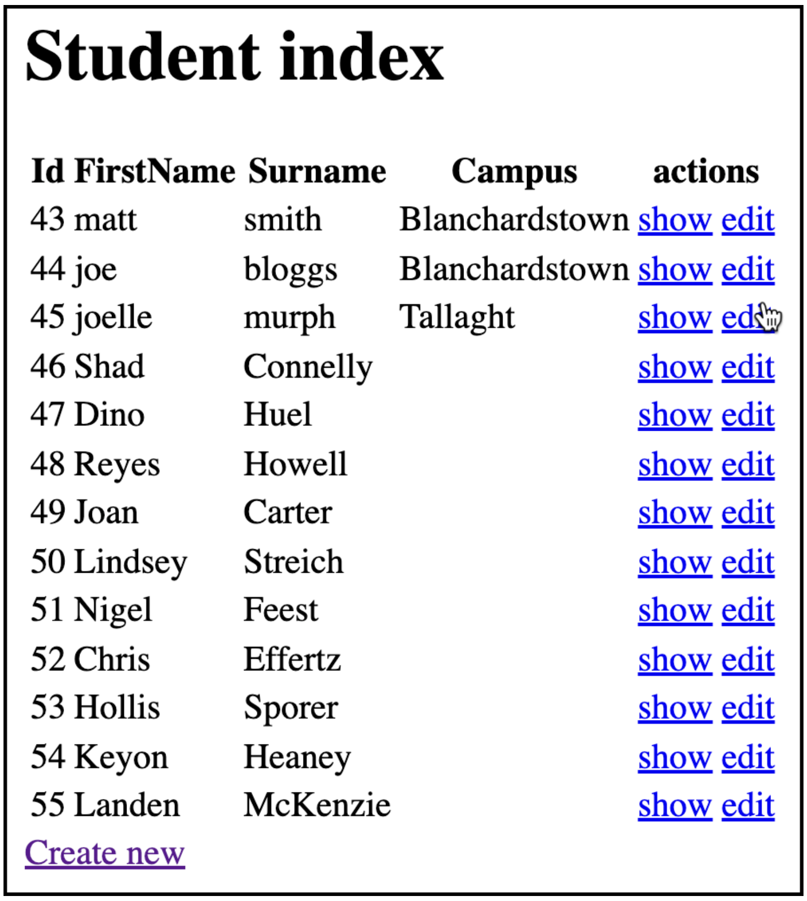
Student index

Id	FirstName	Surname	Campus	action
30	matt	smith	Blanchardstown	show edit
31	joe	bloggs	Blanchardstown	show edit
32	joelle	murph	Tallaght	show edit
33	Possimus deserunt nihil ea aut eaque sit unde. Enim eos nulla aut sunt deleniti a. Voluptatem magnam magni vero repellat sapiente et dolor.	Consequatur vero necessitatibus vel vel deserunt harum placeat. Molestiae impedit explicabo quia aut. Ut magnam enim qui voluptate qui voluptatem vel. Qui omnis quo accusantium est.		show edit
34	Aut nobis distinctio voluptates incidunt ea debitis rem explicabo. Id saepe quae eum. Id dolor deserunt non distinctio.	Perspiciatis incidunt modi excepturi fugiat. Magni illum facilis corporis sunt voluptatibus. Accusamus quod facilis quia nihil voluptas.		show edit
35	Qui vitae quia ex et nobis et ut perferendis. Provident et nemo sed suscipit laboriosam molestias soluta voluptas. Repellendus accusantium labore vel culpa nemo.	Laborum in illo commodi exercitationem repellendus culpa. Illum iure debitis consequatur qui. Quis soluta aut earum quo.		show edit
36	Alias iusto inventore eum iste facilis. Consectetur dolor ut voluptatem doloremque ut ratione voluptates. Quisquam praesentium error delectus est necessitatibus eos tenetur quasi.	Architecto nesciunt maxime nemo consectetur qui cumque. Sit voluptate et ipsum corporis. Esse quisquam recusandae est rerum placeat sunt.		show edit
37	Id magni quia nostrum corporis voluptatem. Eligendi optio natus voluptates ut fugit ipsum.	Vero veniam eveniet omnis omnis. Esse accusantium in accusamus. Ipsam reprehenderit est voluptate laudantium		show edit

Figure 10.3: Students generated by Foundry factory.

```
...

final class StudentFactory extends ModelFactory
{
    ...
    protected function getDefaults(): array
    {
        return [
            'firstName' => self::faker()->text(),
            'surname' => self::faker()->text(),
        ];
    }
```

Now if we re-load fixtures, then list students in the browser (Figure 10.4) we can see that we have believable values for the names again.



Student index

Id	FirstName	Surname	Campus	actions
43	matt	smith	Blanchardstown	show edit
44	joe	bloggs	Blanchardstown	show edit
45	joelle	murph	Tallaght	show edit
46	Shad	Connelly		show edit
47	Dino	Huel		show edit
48	Reyes	Howell		show edit
49	Joan	Carter		show edit
50	Lindsey	Streich		show edit
51	Nigel	Feest		show edit
52	Chris	Effertz		show edit
53	Hollis	Sporer		show edit
54	Keyon	Heaney		show edit
55	Landen	McKenzie		show edit

Create new

Figure 10.4: Students with better names generated by Foundry factory.

10.6 Making Foundry choose one campus

In our `StudentFixtures` class, we have already got a reference to each campus. So we can tell the `StudentFactory` to always set the `campus` property to, say, the Blanchardstown campus. We do this by passing key-value array second argument as follows:

```
StudentFactory::new()->createMany(10,
    ['campus' => $campusBlanchardstown]
);
```

This is fine, if we want **all** of our generated `Student` objects to be linked to a single campus.

10.7 Making Foundry choose one campus

In our `StudentFixtures` class, we have already got a reference to each campus. So we can tell the `StudentFactory` to always set the `campus` property to, say, the Blanchardstown campus. We do this by passing key-value array second argument as follows:

```
StudentFactory::new()->createMany(10,
    ['campus' => $campusBlanchardstown]
);
```

10.8 Getting Foundry to choose randomly from existing Campuses

One way to get Foundry to set the campus for each generated Student to a random campus, is to the use the Faker method `randomElement(<array>)`. The code would look like this:

```
        StudentFactory::new()->createMany(10,
            function() {
                $campusBlanchardstown = $this->getReference('CAMPUS_BLANCH');
                $campusTallaght = $this->getReference('CAMPUS_TALLAGHT');
                $campusCity = $this->getReference('CAMPUS_CITY');

                $campusArray = [$campusBlanchardstown, $campusTallaght, $campusCity];

                $faker = Factory::create();
                $randomCampus = $faker->randomElement($campusArray);
                return ['campus' => $randomCampus];
            }
        );
```

Wow - that's complicated. Part of the reason it's complicated is the need for a function to be used, to ensure each separate Factory-generated student gets a newly created value for campus. But since we have this **anonymous** function, we have to retreive the `Campus` referenced objects,m and create the Faker object, all inside this function that is an argument to the Factory's `createMany(...)` method.

let's find a better way …

10.9 Relating multiple Foundry Factories

Foundry factories work well with each other!. So to greatly simplify our code we need to create a `Campus` factory, even though we are happy to create 3 specific campuses in the `CampusFixtures` class.

Do the following:

1. Generate a `Campus` factory
   - we do this at the command line with console command: `symfony console make:factory`, then choose the `Campus` entity
   - we do **NOT** need to make any changes to this generated factory, since we won't be using it to generate any `Cmpus` objects!
2. Replace the code in the load(…) method of our StudentFixtures class with the following:

```
StudentFactory::new()->createMany(10,
    function() {
        return ['campus' => CampusFactory::random()];
    }
);
```

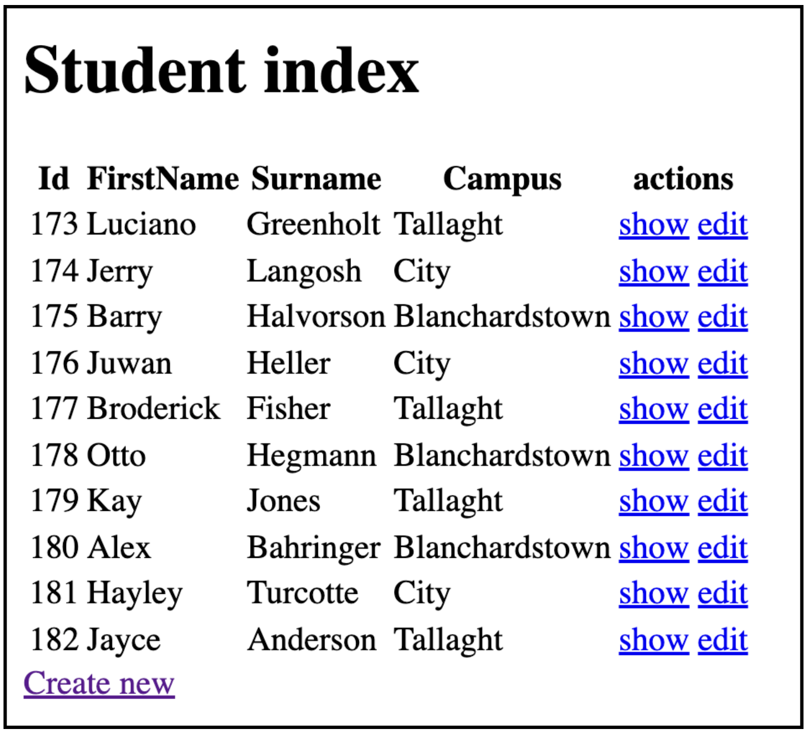
We can see that we still have to pass an anonymous function as the second argument to the `createMany(...)` method. However, we can also see that the code to set the `campus` property of **each** generated `Student` object is just 1 line. This is because we can reference our `CampusFactory` class, whose `random()` method means it will randomly choose one of the `Campus` objects returned by the Doctrine ORM manager.

If we remove the 3 hard-coded `Student` objects from our code, we can reduce our entire `StudentFixtures` class to just a few lines - stating the dependency to generate the `CampusFixtures` first, and then to generate 10 students, each linked to a random `Campus`:

```
namespace App\DataFixtures;

...
```

```
class StudentFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        StudentFactory::new()->createMany(10,
            function() {
                return ['campus' => CampusFactory::random()];
            }
        );

        $manager->flush();
    }

    public function getDependencies()
    {
        return [
            CampusFixtures::class,
        ];
    }
}
```

Now if we re-load fixtures, then list students in the browser (Figure 10.5) we can see that we have nicely generated stuednts, each related randomly to one of our 3 specific campuses.



Student index

Id	FirstName	Surname	Campus	actions
173	Luciano	Greenholt	Tallaght	show edit
174	Jerry	Langosh	City	show edit
175	Barry	Halvorson	Blanchardstown	show edit
176	Juwan	Heller	City	show edit
177	Broderick	Fisher	Tallaght	show edit
178	Otto	Hegmann	Blanchardstown	show edit
179	Kay	Jones	Tallaght	show edit
180	Alex	Bahringer	Blanchardstown	show edit
181	Hayley	Turcotte	City	show edit
182	Jayce	Anderson	Tallaght	show edit

Create new

Figure 10.5: All students generated with random campuses.

10.10 A single fixtures class (project `db09`)

We can actually **completely get rid of** the `CampusFixtures` class altogether!

We can use the `createOne(...)` method of the `CampusFactory` class to create our 3 campuses, then, as above, the `createMany(...)` method of the `StudentFactory` class to create our 10 random students, related randomly to one of the 3 campuses.

So do the following:

1. delete class `CampusFixtures`
2. replace the code for class `StudentFixtures`

```
namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;

use App\Entity\Student;
use App\Entity\Campus;

use App\Factory\StudentFactory;
use App\Factory\CampusFactory;

class StudentFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        CampusFactory::createOne(['name' => 'Blanchardstown']);
        CampusFactory::createOne(['name' => 'Tallaght']);
        CampusFactory::createOne(['name' => 'City']);

        StudentFactory::new()->createMany(10,
            function() {
                return ['campus' => CampusFactory::random()];
            }
        );

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

Since we are not relating fixtures between different classes anymore, we don't need any `getDependencies()` method or use of the `DependentFixtureInterface`.

- perhaps we need a better name for our single, fixtures class - perhaps back to `AppFixtures` - so we could use the default class in future, to make use of our Foundry Factory classes.

So you can begin to see the power of Foundry for creating fixture data in your projects …