1

First steps for Web Framework Software Engineering

1.1 Don't confuse different software tools

Please do not confuse the following:

- Git and Github
- PHP and PHPStorm

Here is a short description of each:

- Git: A version control system - can run locally or on networked computer. There are several website that support Git projects, including:
  - Github (perhaps the most well known)
  - Gitlab
  - Bitbucket
  - you can also create and run your own Git web server ...
- Github: A commercial (but free for students!) cloud service for storing and working with projects using the Git version control system
- PHP: A computer programming language, maintained by an international Open Source community and published at `php.net`
- PHPStorm: A great (and free for student!) IDE - Interactive Development Enviroment. I.e. a really clever text editor created just for working with PHP projects. PHPStorm is one of

the professional software tools offered by the **Jetbrains** company.

So in summary, Git and PHP are open source core software. Github and PHPStorm are commercial (but free for students!) tools that support development using Git and PHP.

1.2 Software tools

Ensure you have the following setup:

- PHP 7.1.5 or later
- Composer (up-to-date with `composer self-update`)
- PHPStorm (with educational free account)
- MySQL Workbench
- Git

See Appendix A for checking, and if necessary, installing PHP on your computer. See Appendix B for details about other software needed for working with PHP projects.

1.3 Test software by creating a new Symfony 4 project

Test your software by using PHP and Composer to create a new Symfony 4 project. We'll follow the steps at the Symfony setup web page.

Follow the steps in Appendix C.

1.4 Open up the Symfony 4 project in PHPStorm

Run PHPStorm and open up the Symfony 4 project you have created. Figure 1.1 shows a screenshot of the default Symfony page for a new, empty project.

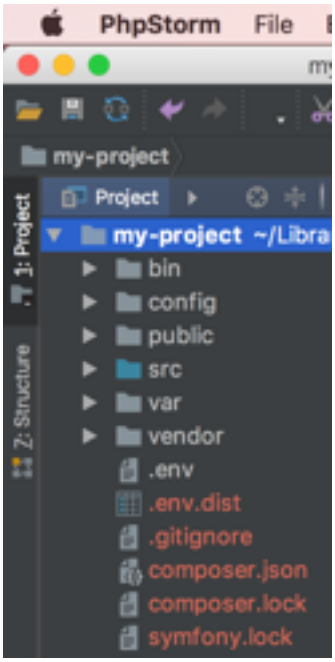


Figure 1.1: PHPStorm with Symfony 4 project open.

1.5 Read/Listen to the free Symfony 4 tutorials

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

So plug in your headphones and watch them, or read the transcripts below the video if you're no headphones.

A good rule is to watch a video or two **before** trying it out yourself.

You'll find the video tutorials at:

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

1.6 Download/Clone the Symfony 4 demo

There is a full demo of a Symfony 4 project, with database access, admin CRUD, login pages etc.

Download a copy and run as described in Appendix D.

1.7 Run the SF Demo tests

Run the Unit and Functional tests in the Symfony Demo - follow the steps described in Appendix E.

1.8 Switch demo from SQLite to MySQL

At present the Symfony demo uses the SQLite driver, working with a databse 'file' in `/var/data`.

Let's change this project to work with a MySQL database schema named `demo`.

Do the following:

1. Run MySQL Workbench
2. Change the **URL** for the projects data in `.env` from:

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

   to

   ```
   DATABASE_URL="mysql://root:pass@127.0.0.1:3306/demo"
   ```

3. Get the Symfony CLI to create the new database schema, type `php bin/console doctrine:database:create`:

   ```
   demo (master) $ php bin/console doctrine:database:create
   Created database `demo` for connection named default


   demo (master) $
   ```

4. Get the Symfony CLI to note any changes that need to happen to the databast to make it match Entites and relationships defined by the project's classes, by typing `php bin/console doctrine:migrations:diff`:

   ```
   demo (master) $ php bin/console doctrine:migrations:diff
  Generated new migration class to "/Users/matt/Library/Mobile Documents/com~apple~CloudDocs/


   demo (master) $
   ```

   A migration file has now been created.

5. Run the migration file, by typing `php bin/console doctrine:migrations:migrate` and then typing `y`:

"'bash demo (master) $ php bin/console doctrine:migrations:migrate

```
Application Migrations
```

```
  WARNING! You are about to execute a database migration that could result in schema changes and data lost. A
   Migrating up to 20180127081633 from 0

     ++ migrating 20180127081633

     -> CREATE TABLE symfony_demo_comment (id INT AUTO_INCREMENT NOT NULL, post_id INT NOT NULL, author_id
     -> CREATE TABLE symfony_demo_post (id INT AUTO_INCREMENT NOT NULL, author_id INT NOT NULL, title VARCH
     -> CREATE TABLE symfony_demo_post_tag (post_id INT NOT NULL, tag_id INT NOT NULL, INDEX IDX_6ABC1CC44
     -> CREATE TABLE symfony_demo_tag (id INT AUTO_INCREMENT NOT NULL, name VARCHAR(255) NOT NULL, UNIQUE I
     -> CREATE TABLE symfony_demo_user (id INT AUTO_INCREMENT NOT NULL, full_name VARCHAR(255) NOT NULL, us
     -> ALTER TABLE symfony_demo_comment ADD CONSTRAINT FK_53AD8F834B89032C FOREIGN KEY (post_id) REFERENC
     -> ALTER TABLE symfony_demo_comment ADD CONSTRAINT FK_53AD8F83F675F31B FOREIGN KEY (author_id) REFERE
     -> ALTER TABLE symfony_demo_post ADD CONSTRAINT FK_58A92E65F675F31B FOREIGN KEY (author_id) REFERENCE
     -> ALTER TABLE symfony_demo_post_tag ADD CONSTRAINT FK_6ABC1CC44B89032C FOREIGN KEY (post_id) REFERE
     -> ALTER TABLE symfony_demo_post_tag ADD CONSTRAINT FK_6ABC1CC4BAD26311 FOREIGN KEY (tag_id) REFERENC

     ++ migrated (0.44s)

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

     ++ finished in 0.44s
     ++ 1 migrations executed
     ++ 10 sql queries

   demo (master) $
```

2

From nothing to controllers and Twig templates

2.1 Basic Symfony 4 recipes (project `basic1`)

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

```
$ composer create-project symfony/skeleton basic1
Installing symfony/skeleton (v4.0.5)
  - Installing symfony/skeleton (v4.0.5): Loading from cache

... etc. ...

$ cd basic1
```

2. Add the Symfony local development server:

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

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

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

Figure 2.1 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`).

1. Add Annotations :

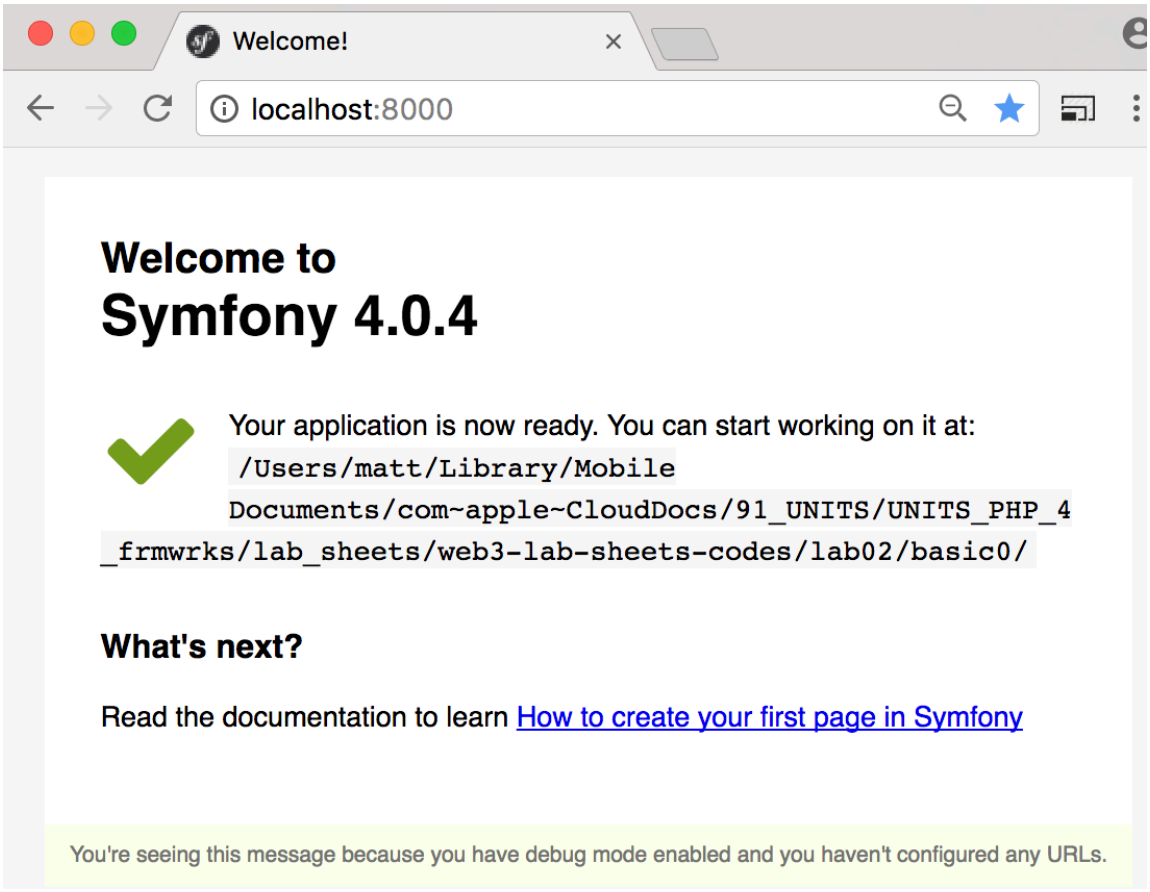


Figure 2.1: Screenshot default Symfony 4 page for web root (when no routes defined).

```
composer req annotations
```

2. Add Twig :

```
composer req twig
```

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

2.2 Development Symfony 4 recipes

1. Add the Symfony Maker recipe (for development setup):

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

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

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

3. Add the Symfony security checker

```
composer req sec-checker
```

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

```
composer req profiler
```

5. Add the Symfony debugging libraries

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

2.3 Install multple libraries in a single `composer` commad

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

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

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

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

2.4 Check your routes

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

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

2.5 Create a controller

Let's create a new homepage (default) controller:

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


    created: src/Controller/DefaultController.php
    Success!
    Next: Open your new controller class and add some pages!
```

Look inside the generated class `/src/Controller/DefaultController.php`.

NOTE: Controller classes are in the namespace `App\Controller`.

Learn more about the Maker bundle:

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

2.6 Run web server to visit new deafault route

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

Oh no! We get a 404 (not found) error!. Figure 2.2 shows a screenshot of error page.

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

The route is:

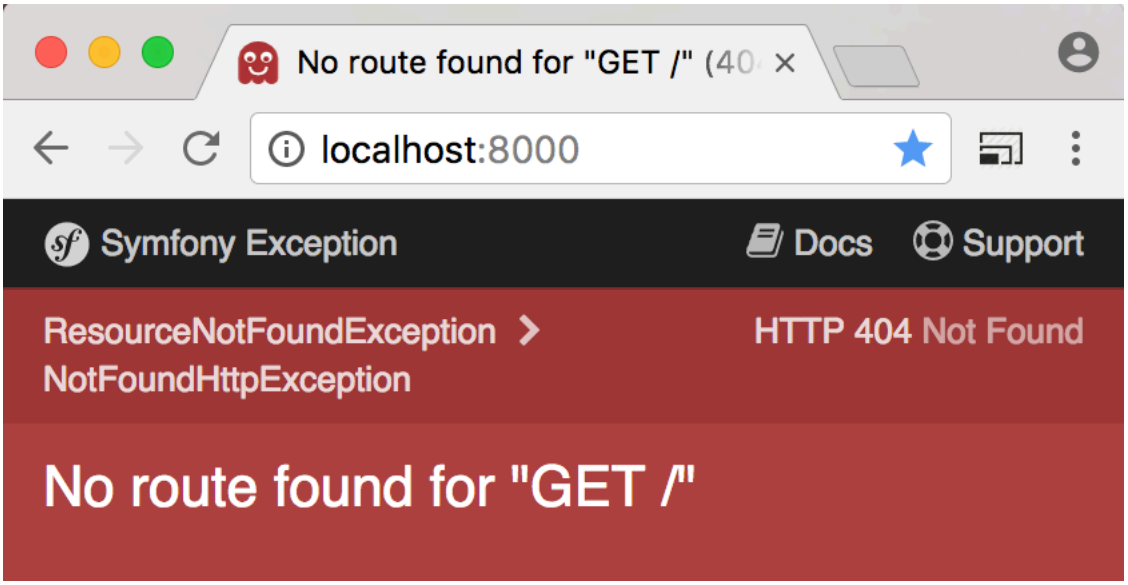


Figure 2.2: Screenshot of 404 error for URL path /.

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

We can see this route in the **annotation** comment preceding controller method `index()` in `src/Controllers/DefaultController.php`

```
class DefaultController extends Controller
{
    /**
     * @Route("/default", name="default")
     */
    public function index()
    {
        ...
```

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

Figure 2.2 shows a screenshot of the message created from our generated default controller method.

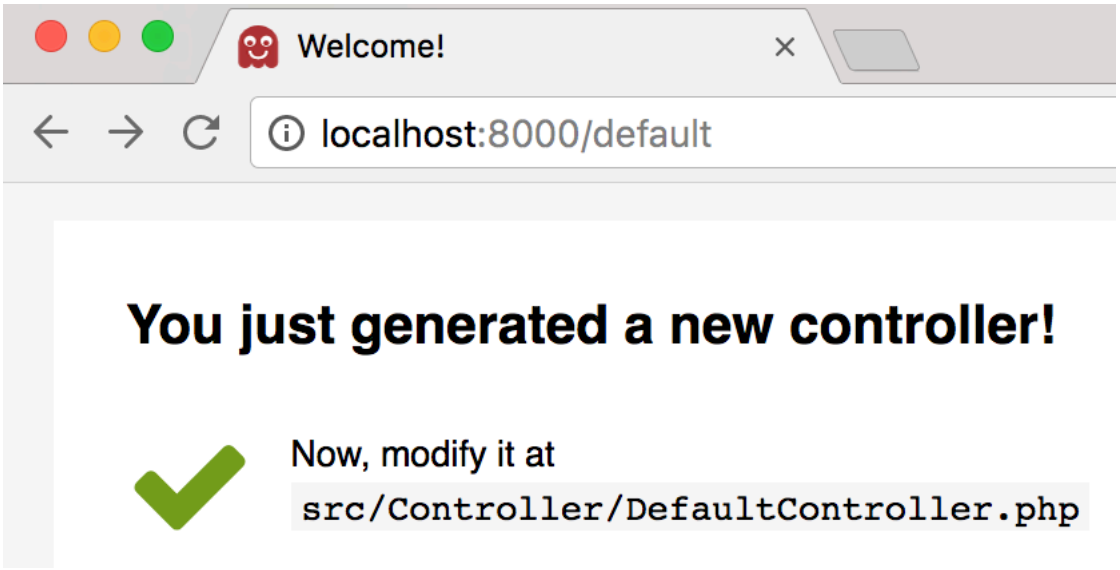


Figure 2.3: Screenshot of generated page for URL path `/default`.

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

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

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

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

Now the route is:

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

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

Luckily, if we are using a PHP-friendly editor like PHPStorm, as we start to type the name of a class, the IDE will popup a suggestion of namespaced classes to choose from. Figure 2.4 shows a

screenshot of PHPStorm offering up a list of suggested classes after we have typed the letters `new Re`. If we accept a suggested class from PHPStorm, then an appropriate `use` statement will be inserted before the class declaration for us.

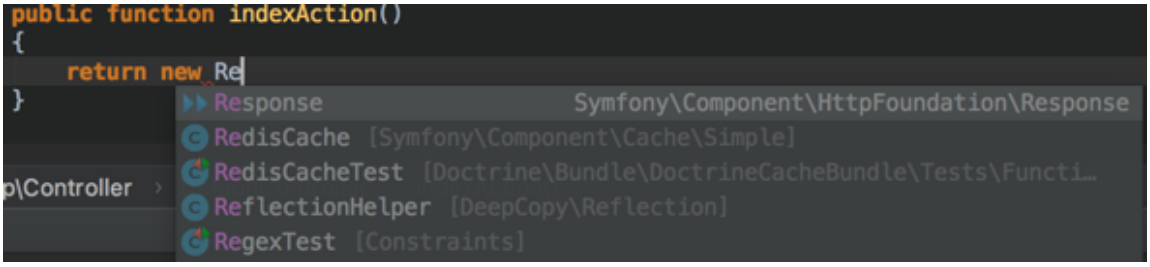


Figure 2.4: Screenshot of PHPStorm IDE suggesting namespaces classes.

Here is a complete `DefaultController` class:

```
namespace App\Controller;

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

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

Figure 2.5 shows a screenshot of the message created from our `Response()` object.

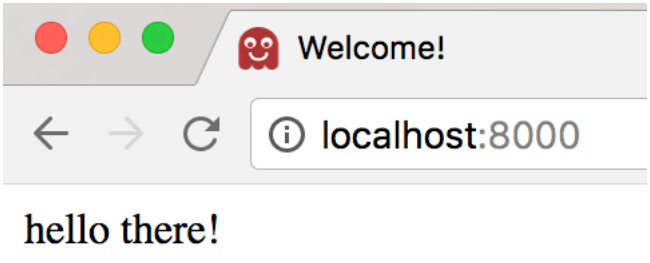


Figure 2.5: Screenshot of page seen for `new Response('hello there!')`.

2.8 Let's create a nice Twig hompage

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

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

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

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

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

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

[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`.

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

Figure 2.6: Screenshot of page from our Twig template.

3

DIY Entities and Repositories

3.1 Goals

Our goals are to:

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

3.2 Let's create an Entity `Student` (project `basic2`)

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

```
<?php
namespace App\Entity;

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

```
    private $surname;
}
```

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

3.3 Create a StudentController class

Generate a StudentController class:

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

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

```
<?php

namespace App\Controller;

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

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

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

```
$ php bin/console cache:clear

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

Let's make this create a student (`1`, `matt`, `smith`) and display it with a Twig template (which we'll write next!). We will also improve the route internal name, changing it to `student_show`, and change the method name to `showAction()`:

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

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

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

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

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

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

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

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

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

Figure 3.1 shows a screenshot our student details web page.

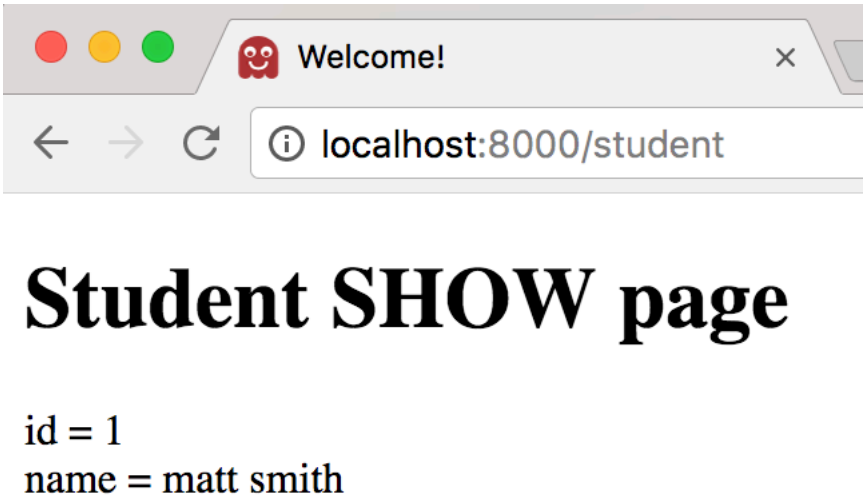


Figure 3.1: Screenshot of student show page.

3.5 Creating an Entity Repository

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

```
<?php
namespace App\Repository;

use App\Entity\Student;

class StudentRepository
{
    private $students = [];

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

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

3.6 The student list controller method

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3.2 shows a screenshot our list of students web page.

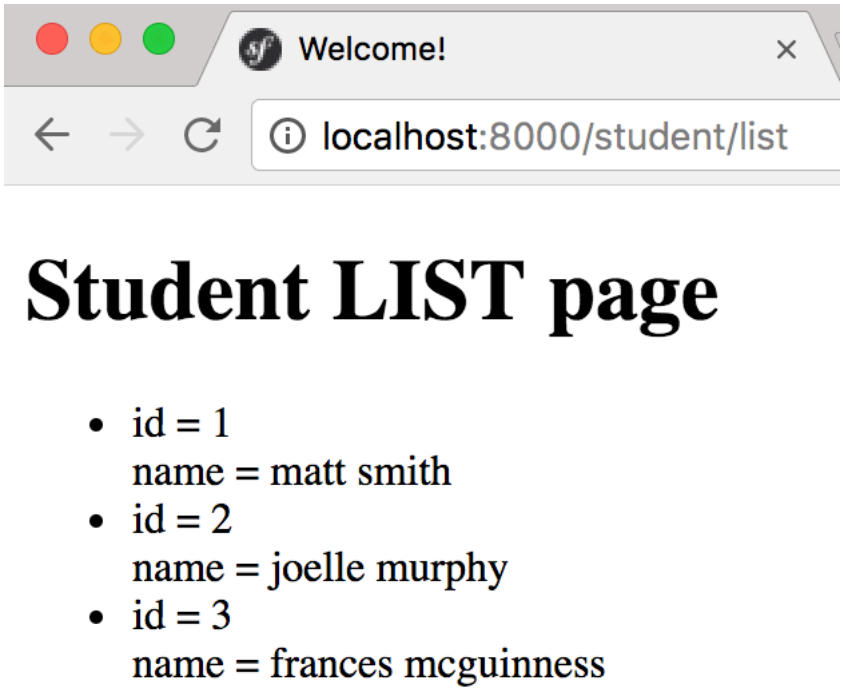


Figure 3.2: Screenshot of student list page.

3.8 Refactor show action to show details of one Student object (project `basic3`)

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

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

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

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

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

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

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

We can check these routes via the console:

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

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

3.9 Make each item in list a link to show

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

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

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

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

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

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

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

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

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

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

could be represented in Twig as:

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

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

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

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

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

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

```
        } else {
            return null;
        }
    }
```

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

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

Figure 3.3 shows a screenshot our list of students web page, with a `(details)` hypertext link to the show page for each individual student obbject.

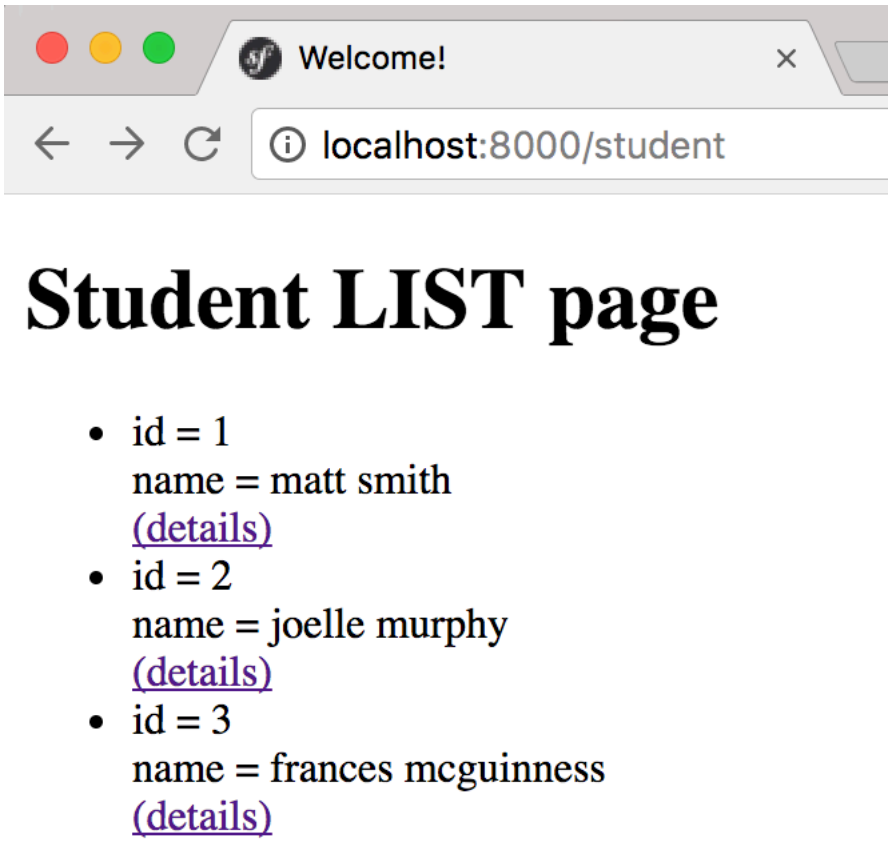


Figure 3.3: Screenshot of student list page, with links to show page for each student object.

3.11 Dealing with not-found issues

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

4

Entities and Databases

4.1 Further reading about Symfony 4 and databases

See the following Symfony documentation page to learn about Symfony 4, Doctine and working with database entity classes:

- https://symfony.com/doc/current/doctrine.html

4.2 Let's create a database entity 'Product' (project `basic4`)

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

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

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

```
 Next: Configuration

  * Modify your DATABASE_URL config in .env

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

Now let's create a basic Product entity (which has a default `id` private property):

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

This has created class `/src/Entity/Product.php` for us, containing:

```
<?php

namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\ProductRepository")
 */
class Product
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    // add your own fields
}
```

Notice the `@ORM` annotation comments - this is getting ready to map to a database table for us...

Let's add a description and price property to this entity:

```
private $description;

private $price;
```

And let's add an annotation comment for each of these properties, so they'll map to a database column for an appropriate data type:

```
    /**
     * @ORM\Column(type="string", length=100)
```

```
     */
    private $description;

    /**
     * @ORM\Column(type="decimal", scale=2, nullable=true)
     */
    private $price;
```

That's enough typing. Generate getters and setters for each property. Although you only need a getter for Id, since the database will be creating an auto-incremented id for each new record.

4.3 Setting the database connection URL for MySQL

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

So change this line in `.env` from:

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

to

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

4.4 Let's create this `web3` database

```
$ php bin/console doctrine:database:create
Created database `web3` for connection named default
```

You should now see the database schema `web3` if you inspect the database contents with your client application.

4.5 Create migrations `diff` file

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

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

Generated new migration class to \
"/.../basic4/src/Migrations/Version20180201223133.php" from schema differences.
```

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

```
namespace DoctrineMigrations;

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

/**
 * Auto-generated Migration: Please modify to your needs!
 */
class Version20180201223133 extends AbstractMigration
{
    public function up(Schema $schema)
    {
        ...
        $this->addSql('CREATE TABLE product ...
```

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

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

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

    Application Migrations


WARNING! You are about to execute a database migration that could result in \
schema changes and data lost. Are you sure you wish to continue? (y/n)y
Migrating up to 20180201223133 from 0

  ++ migrating 20180201223133

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

  ++ migrated (0.14s)

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

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

You can see the results of creating the database schema and creating table(s) to match your ORM entities using a database client such as MySQL Workbench. Figure 5.1 shows a screenshot of MySQL Workbench showing new database `web3` and its `product` table to match our `Product` entity class.

Figure 4.1: Screenshot MySQL Workbench and generated schema and product table.

4.7 Creating a controller class and method to add records to the database

Create a new controller class `ProductController`:

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

 created: src/Controller/ProductController.php

  Success!

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

Replace the default `index()` method with a create method to create a new product given a description and price via the URL:

Here's the code for this `createAction(...)` method

```
/**
 * @Route("/product/create/{description}/{price}", name="product_create")
```

```
     */
    public function createAction($description, $price)
    {
        // create new product object
        $product = new Product();
        $product->setDescription($description);
        $product->setPrice($price);

        // persist (save/store) this object's contents to the database
        $em = $this->getDoctrine()->getManager();
        $em->persist($product);
        $em->flush();

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

We can see that creating a new `Product` object is straightforward, given `$description` and `$price` from the URL-encoded GET name=value pairs:

```
$product = new Product();
$product->setDescription($description);
$product->setPrice($price);
```

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

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

4.8 Run server and create products through URLs

Run the web server, and visit 2 pages, to create 2 products (hammer price 9.99, and nail price 0.10):

```
http://localhost:8000/product/create/hammer/9.99
http://localhost:8000/product/create/nail/0.10
```

Figure 5.2 shows screenshots of visiting these two URLs, and the message confirming new product items were created. Note how the `id` property was retrieved from the objects, having been auto-incremented by the database management system.

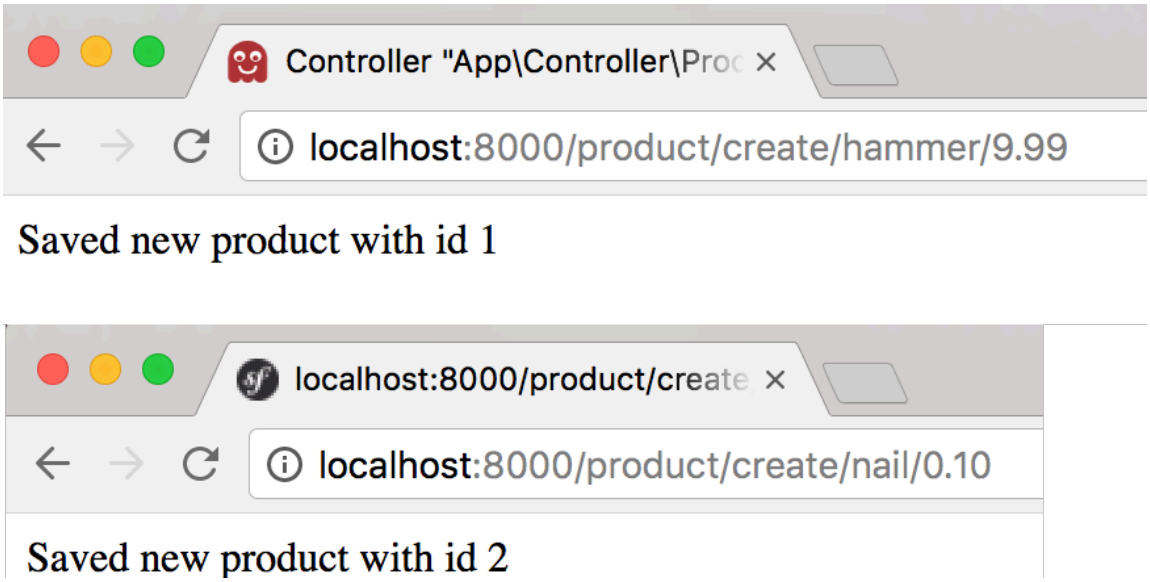


Figure 4.2: Screenshot of URLs to generated new products.

4.9 Query database with SQL from CLI server

The `doctine: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 product'

/.../lab02/basic4/vendor/doctrine/common/lib/Doctrine/Common/Util/Debug.php:71:
array (size=2)
  0 =>
    array (size=3)
      'id' => string '1' (length=1)
      'description' => string 'hammer' (length=6)
      'price' => string '9.99' (length=4)
  1 =>
    array (size=3)
      'id' => string '2' (length=1)
      'description' => string 'nail' (length=4)
      'price' => string '0.10' (length=4)
```

We can, of course, also run such a query from our database client. Figure 5.3 shows screenshots of visiting these two URLs, and the message confirming new product items were created. Note how the `id` property was retrieved from the objects, having been auto-incremented by the database management system.

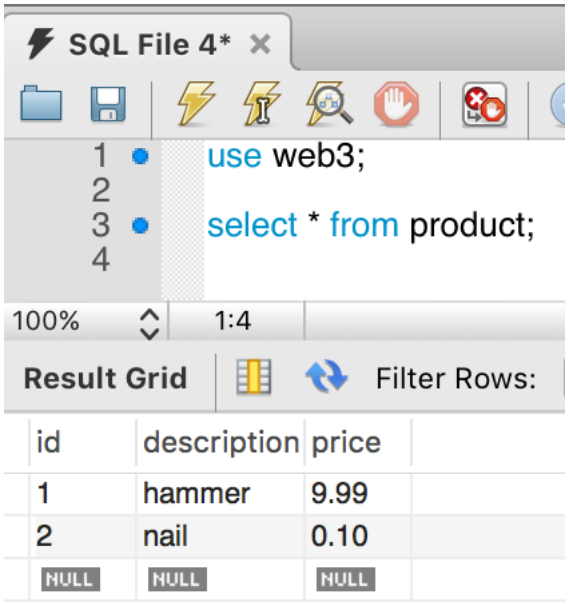


Figure 4.3: Screenshot of MySQL Workbench selecting all products.

4.10 List and Show actions for Products

When we generated the `Product` entity class, Symfony also created `ProductRepository` class.

In a controller can we can get a refererence to the `ProductRepository` class by writing:

```
$productRepository = $this->getDoctrine()->getRepository(Product::class);
```

So we can now add `showAction($id)` and `listAction()` methods to our `ProductController` class as follows:

```
/**
 * @Route("/product/{id}", name="product_show")
 */
public function showAction($id)
{
    $productRepository = $this->getDoctrine()->getRepository(Product::class);
    $product = $productRepository->find($id);

    // we are assuming $product is not NULL....

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

```
}

/**
 * @Route("/product", name="product_list")
 */
public function listAction()
{
    $productRepository = $this->getDoctrine()->getRepository(Product::class);
    $products = $productRepository->findAll();

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

As you can see, apart from how we get a reference to the doctrine repository class, these methods are very similar to our DIY student-and-repository controller methods.

4.11 List and Show Twig templates

Likewise, we can write very similar Twig templates (in directory `/templates/product`) for the show and list actions.

The Twig for the `/templates/product/show.html.twig` template is:

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

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

    <p>
        id = {{ product.id }}
        <br>
        description = {{ product.description }}
        <br>
        price = &euro; {{ product.price }}
    </p>
{% endblock %}
```

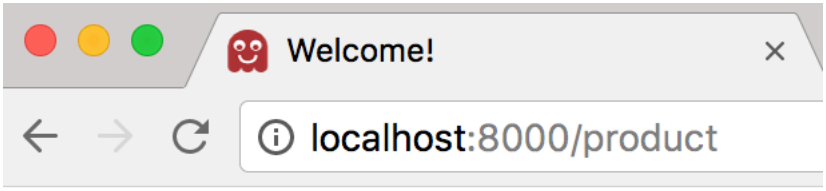
And for the `list.html.twig` template:

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

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

    <ul>
        {% for product in products %}
            <li>
                id = {{ product.id }}
                <br>
                description = {{ product.description }}
                <br>
                price = &euro; {{ product.price }}
                <br>
           <a href="{{ url('product_show', {id : product.id} ) }}">(details)</a>
            </li>
        {% endfor %}
    </ul>
{% endblock %}
```

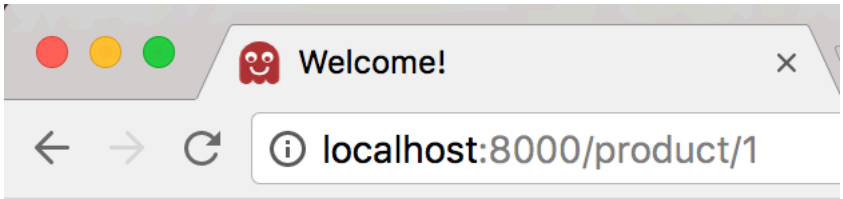
Figures 5.4 and 5.5 shows screenshots of the web pages for the product list and show routes.



Product LIST page

- id = 1
  description = hammer
  price = € 9.99
  (details)
- id = 2
  description = nail
  price = € 0.10
  (details)

Figure 4.4: Screenshot of listing all products.



Product SHOW page

id = 1
description = hammer
price = € 9.99

Figure 4.5: Screenshot of showing one product.

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

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

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

```
/**
 * @Route("/product/{id}", name="product_show")
 */
public function showAction($id)
{
    $productRepository = $this->getDoctrine()->getRepository(Product::class);
    $product= $productRepository->find($id);

    // we are assuming $product is not NULL....

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

to just:

```
public function showAction(Product $product)
{
    $template = 'product/show.html.twig';
    $args = [
        'product' => $product
    ];
    return $this->render($template, $args);
}
```

The para-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 para-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

4.13 Re-direct to Show route after creating/updating a record

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

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

So instead we can tell Symfony to redirect to the `product_show` route for

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

5

Entities and Databases

5.1 Further reading about Symfony 4 and databases

See the following Symfony documentation page to learn about Symfony 4, Doctine and working with database entity classes:

- https://symfony.com/doc/current/doctrine.html

5.2 Let's create a database entity 'Product' (project `basic4`)

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

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

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

```
 Next: Configuration

  * Modify your DATABASE_URL config in .env

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

Now let's create a basic Product entity (which has a default `id` private property):

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

This has created class `/src/Entity/Product.php` for us, containing:

```
<?php

namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\ProductRepository")
 */
class Product
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    // add your own fields
}
```

Notice the `@ORM` annotation comments - this is getting ready to map to a database table for us...

Let's add a description and price property to this entity:

```
private $description;

private $price;
```

And let's add an annotation comment for each of these properties, so they'll map to a database column for an appropriate data type:

```
    /**
     * @ORM\Column(type="string", length=100)
```

```
     */
    private $description;

    /**
     * @ORM\Column(type="decimal", scale=2, nullable=true)
     */
    private $price;
```

That's enough typing. Generate getters and setters for each property. Although you only need a getter for Id, since the database will be creating an auto-incremented id for each new record.

5.3 Setting the database connection URL for MySQL

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

So change this line in `.env` from:

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

to

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

5.4 Let's create this `web3` database

```
$ php bin/console doctrine:database:create
Created database `web3` for connection named default
```

You should now see the database schema `web3` if you inspect the database contents with your client application.

5.5 Create migrations `diff` file

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

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

Generated new migration class to \
"/.../basic4/src/Migrations/Version20180201223133.php" from schema differences.
```

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

```
namespace DoctrineMigrations;

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

/**
 * Auto-generated Migration: Please modify to your needs!
 */
class Version20180201223133 extends AbstractMigration
{
    public function up(Schema $schema)
    {
        ...
        $this->addSql('CREATE TABLE product ...
```

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

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

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

    Application Migrations


WARNING! You are about to execute a database migration that could result in \
schema changes and data lost. Are you sure you wish to continue? (y/n)y
Migrating up to 20180201223133 from 0

  ++ migrating 20180201223133

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

  ++ migrated (0.14s)

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

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

You can see the results of creating the database schema and creating table(s) to match your ORM entities using a database client such as MySQL Workbench. Figure 5.1 shows a screenshot of MySQL Workbench showing new database `web3` and its `product` table to match our `Product` entity class.

Figure 5.1: Screenshot MySQL Workbench and generated schema and product table.

5.7 Creating a controller class and method to add records to the database

Create a new controller class `ProductController`:

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

 created: src/Controller/ProductController.php

  Success!

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

Replace the default `index()` method with a create method to create a new product given a description and price via the URL:

Here's the code for this `createAction(...)` method

```
/**
 * @Route("/product/create/{description}/{price}", name="product_create")
```

```
     */
    public function createAction($description, $price)
    {
        // create new product object
        $product = new Product();
        $product->setDescription($description);
        $product->setPrice($price);

        // persist (save/store) this object's contents to the database
        $em = $this->getDoctrine()->getManager();
        $em->persist($product);
        $em->flush();

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

We can see that creating a new `Product` object is straightforward, given `$description` and `$price` from the URL-encoded GET name=value pairs:

```
$product = new Product();
$product->setDescription($description);
$product->setPrice($price);
```

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

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

5.8 Run server and create products through URLs

Run the web server, and visit 2 pages, to create 2 products (hammer price 9.99, and nail price 0.10):

```
http://localhost:8000/product/create/hammer/9.99
http://localhost:8000/product/create/nail/0.10
```

Figure 5.2 shows screenshots of visiting these two URLs, and the message confirming new product items were created. Note how the `id` property was retrieved from the objects, having been auto-incremented by the database management system.

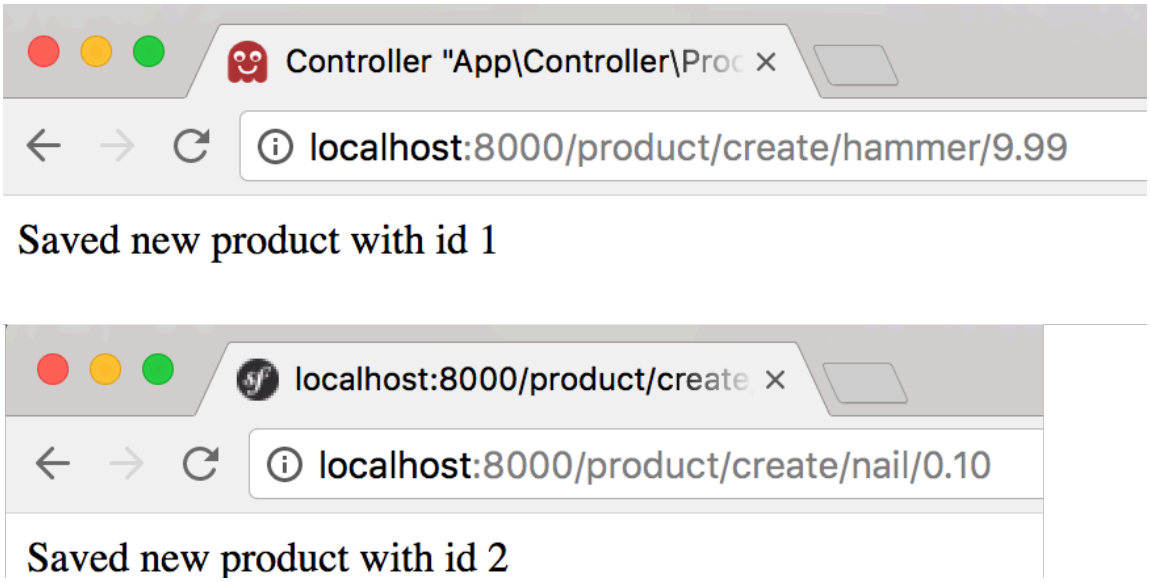


Figure 5.2: Screenshot of URLs to generated new products.

5.9 Query database with SQL from CLI server

The `doctine: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 product'

/.../lab02/basic4/vendor/doctrine/common/lib/Doctrine/Common/Util/Debug.php:71:
array (size=2)
  0 =>
    array (size=3)
      'id' => string '1' (length=1)
      'description' => string 'hammer' (length=6)
      'price' => string '9.99' (length=4)
  1 =>
    array (size=3)
      'id' => string '2' (length=1)
      'description' => string 'nail' (length=4)
      'price' => string '0.10' (length=4)
```

We can, of course, also run such a query from our database client. Figure 5.3 shows screenshots of visiting these two URLs, and the message confirming new product items were created. Note how the `id` property was retrieved from the objects, having been auto-incremented by the database management system.

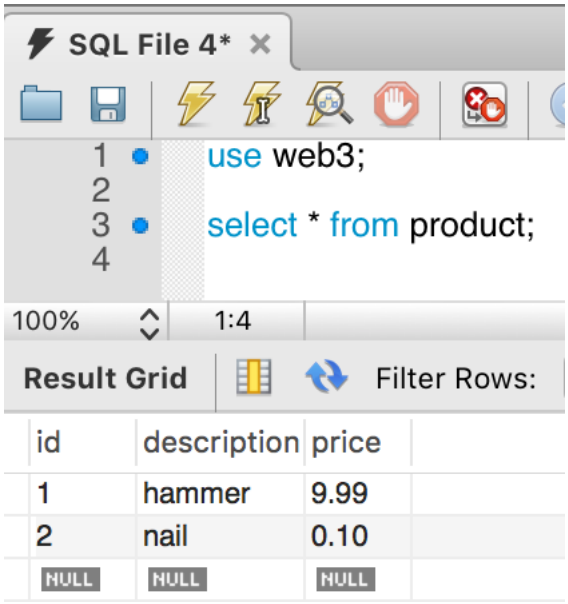


Figure 5.3: Screenshot of MySQL Workbench selecting all products.

5.10 List and Show actions for Products

When we generated the `Product` entity class, Symfony also created `ProductRepository` class.

In a controller can we can get a refererence to the `ProductRepository` class by writing:

```
$productRepository = $this->getDoctrine()->getRepository(Product::class);
```

So we can now add `showAction($id)` and `listAction()` methods to our `ProductController` class as follows:

```
/**
 * @Route("/product/{id}", name="product_show")
 */
public function showAction($id)
{
    $productRepository = $this->getDoctrine()->getRepository(Product::class);
    $product = $productRepository->find($id);

    // we are assuming $product is not NULL....

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

```
}

/**
 * @Route("/product", name="product_list")
 */
public function listAction()
{
    $productRepository = $this->getDoctrine()->getRepository(Product::class);
    $products = $productRepository->findAll();

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

As you can see, apart from how we get a reference to the doctrine repository class, these methods are very similar to our DIY student-and-repository controller methods.

5.11 List and Show Twig templates

Likewise, we can write very similar Twig templates (in directory `/templates/product`) for the show and list actions.

The Twig for the `/templates/product/show.html.twig` template is:

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

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

    <p>
        id = {{ product.id }}
        <br>
        description = {{ product.description }}
        <br>
        price = &euro; {{ product.price }}
    </p>
{% endblock %}
```

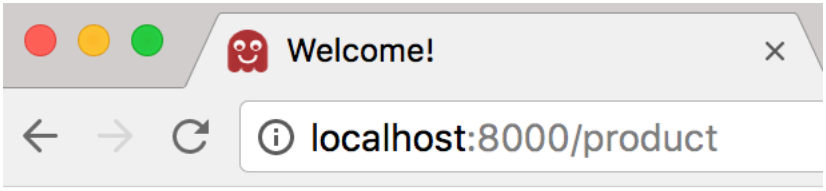
And for the `list.html.twig` template:

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

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

    <ul>
        {% for product in products %}
            <li>
                id = {{ product.id }}
                <br>
                description = {{ product.description }}
                <br>
                price = &euro; {{ product.price }}
                <br>
           <a href="{{ url('product_show', {id : product.id} ) }}">(details)</a>
            </li>
        {% endfor %}
    </ul>
{% endblock %}
```

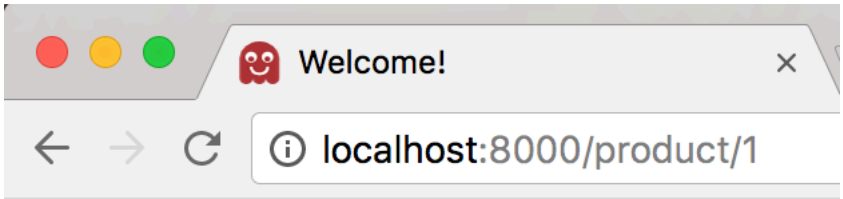
Figures 5.4 and 5.5 shows screenshots of the web pages for the product list and show routes.



Product LIST page

- id = 1
  description = hammer
  price = € 9.99
  (details)
- id = 2
  description = nail
  price = € 0.10
  (details)

Figure 5.4: Screenshot of listing all products.



Product SHOW page

id = 1
description = hammer
price = € 9.99

Figure 5.5: Screenshot of showing one product.

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

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

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

```
/**
 * @Route("/product/{id}", name="product_show")
 */
public function showAction($id)
{
    $productRepository = $this->getDoctrine()->getRepository(Product::class);
    $product= $productRepository->find($id);

    // we are assuming $product is not NULL....

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

to just:

```
public function showAction(Product $product)
{
    $template = 'product/show.html.twig';
    $args = [
        'product' => $product
    ];
    return $this->render($template, $args);
}
```

The para-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 para-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

5.13 Re-direct to Show route after creating/updating a record

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

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

So instead we can tell Symfony to redirect to the `product_show` route for

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

A

PHP Windows setup

A.1 Check if you have PHP installed and working

You need PHP version 7.1.3 or later.

Check your PHP version at the command line with:

```
> php -v
PHP 7.1.5 (cli) (built: May  9 2017 19:49:10)
Copyright (c) 1997-2017 The PHP Group
Zend Engine v3.1.0, Copyright (c) 1998-2017 Zend Technologies
```

If your version is older than 7.1.5, or you get an error about command not understood, then complete the steps below.

A.1.1 Download the latest version of PHP

Get the latest (7.2.1 at the time of writing) PHP Windows ZIP from:

- php.net click the **Windows Downloads** link

Figure A.1 shows a screenshot of the `php.net` general and Windows downloads page. The `ZIP` file to download (containing `php.exe` … don't download the source code version unless you want to build the file from code …):

Do the following:

- unzip the PHP folder into: `C:\php`

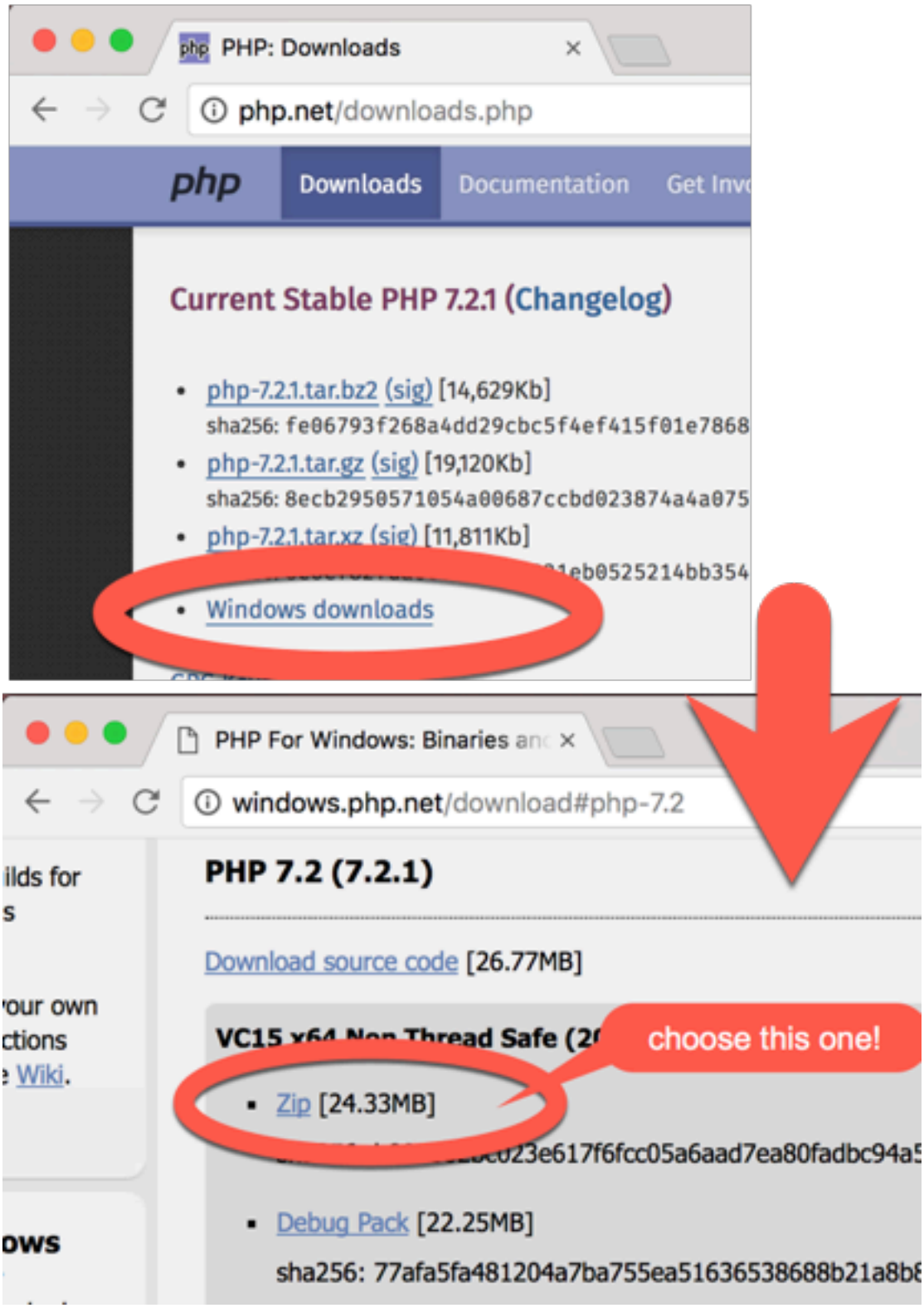


Figure A.1: PHP.net / Windows ZIP download pages.

- so you should now have a file `php.exe` inside `C:\php`, along with lots of other files
- make a copy the file `C:\php\php.ini-development`, naming the copy `C:\php\php.ini`
- open a new terminal CLI window (so new settings are loaded) and run `php --ini` to confirm the location of the `php.ini` file that you've just created. Note the following for a Mac - for Windows it should (hopefully) tell you it found the ini file in `c:\php\php.ini`:

```
$ php --ini
Configuration File (php.ini) Path: /Applications/MAMP/bin/php/php7.1.8/conf
Loaded Configuration File:         /Applications/MAMP/bin/php/php7.1.8/conf/php.ini
Scan for additional .ini files in: (none)
Additional .ini files parsed:      (none)
```

A.2 Add the path to `php.exe` to your System environment variables

Whenever you type a command at the CLI (Command Line Interface) Windows searches through all the directories in its `path` environment variable. In order to use PHP at the CLI we need to add `c:\php` to the `path` environment variable so the `php.exe` executable can be found.

Via the System Properties editor, open your Windows Evironment Variables editor. The **system** environment variablesa re in the lower half of the Environment Variables editor. If there is already a system variable named `Path`, then select it and click the **Edit** button. If none exists, then click the **New** button, naming the new variable **path**. Add a new value to the **path** variable with the value `c:\php`. Then click all the **Okay** buttons needed to close all these windows.

Now open a windows **Cmd** window and try the `php -v` - hopefully you'll see confirmation that your system now has PHP installed and in the **path** for CLI commands.

Figure A.2 shows a screenshot of the Windows system and environment variables editor.

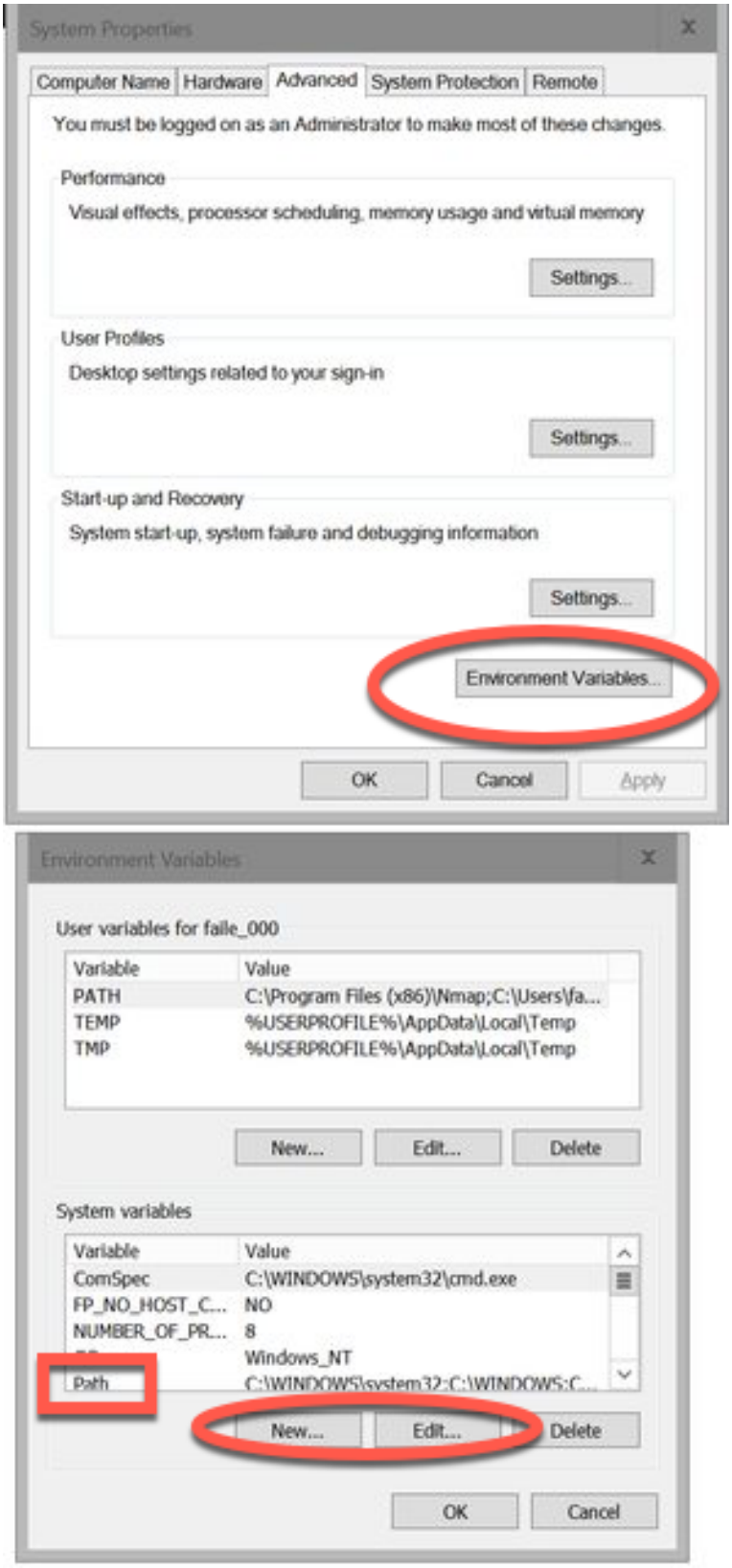


Figure A.2: The Windows Environment Variables editor.

A.3 PHP Info & SQL driver test

For database work we need to enable the PDO[1] options for MySQL and SQLite (see later database exercises for how to do this)

Although PHP may have been installed, and its SQL drivers too, they may have not been enabled. For this module we'll be using the SQLite and MySQL drivers for PHP – to talk to databases. The function `phpinfo()` is very useful since it displays many of the settings of the PHP installation on your computer / website.

1. In the current (or a temporary) direcotry, create file `info.php` containing just the following 2 lines of code:

```
<?php
print phpinfo();
```

2. At the CLI run the built-in PHP web server to serve this page, and visit: localhost:8000/info.php in your web browser

```
php -S localhost:8000
```

In the PDO section of the web page (`CTL-F` and search for `pdo` ...) we are looking for **mysql** and **sqlite**. If you see these then great!

Figure A.3 shows a screenshot the Windows system and environment variables editor.

But, if you see "no value" under the PDO drivers section, then we'll need to edit file `c:\php\php.ini`:

1. In a text editor open file `c:\php\php.ini` and locate the "Dynamic Extensions" section in this file (e.g. use the editor Search feature - or you could just search for `pdo`)

2. Now remove the semi-colon `;` comment character at the beginning of the lines for the SQLite and MySQL DLLs to enable them as shown here:

```
;;;;;;;;;;;;;;;;;;;;;
; Dynamic Extensions ;
;;;;;;;;;;;;;;;;;;;;;

.. other lines here ...
extension=php_pdo_mysql.dll <<<<<<<<<< here is the PDO MYSQL driver line
;extension=php_pdo_oci.dll
;extension=php_pdo_odbc.dll
```

[1] PDO = PHP Database Objects, the modern library for managing PHP program communications with databases. Avoid using old libries like `mysql` (security issues) and even `mysqli` (just for MySQL). PDO offers an object-oriented, standardized way to communicate with many different database systems. So a project could change the databse management system (e.g. from Oracle to MySQL to SQLite), and only the database connetion optins need to change - all other PDO code will work with the new database system!

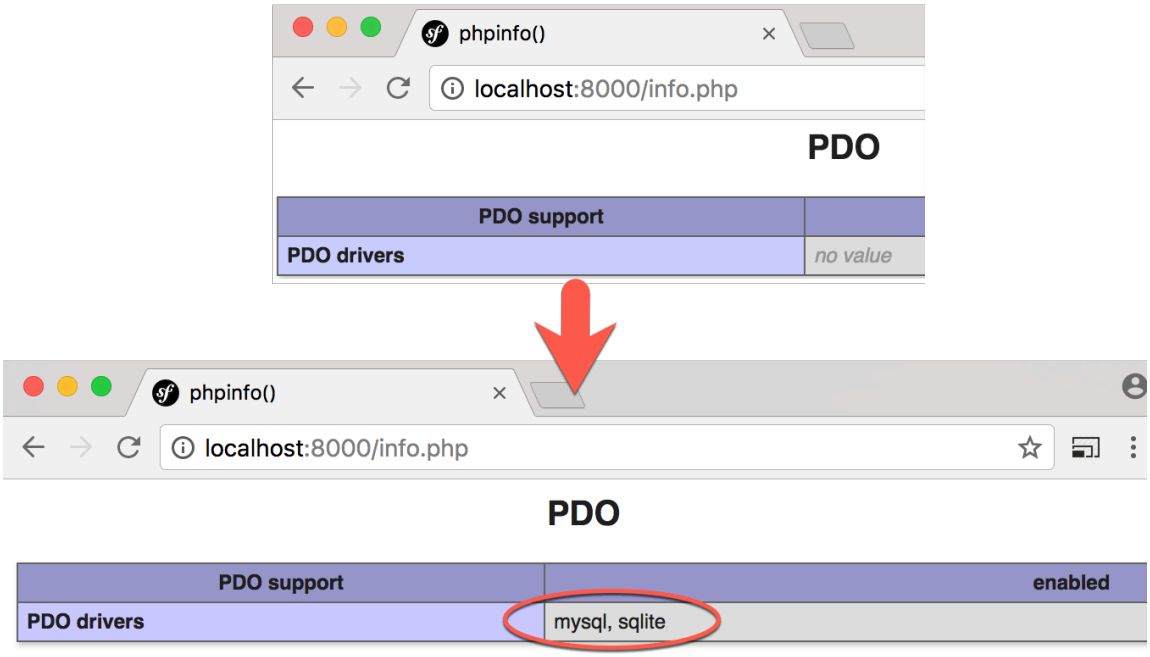


Figure A.3: The PDO section of the `phpinfo()` information page.

```
;extension=php_pdo_pgsql.dll
extension=php_pdo_sqlite.dll <<<<<<<  here is the PDO SQLITE driver line
```

3. Save the file. Close your Command Prompt, and re-open it (to ensure new settings are used).
   - Run the webserver again and visit: `localhost:8000/info.php` to check the PDO drivers.

NOTE: Knowing how to view `phpinfo()` is very handy when checking server features.

B

Get/Update your software tools

NOTE: All the following are already available on the ITB college computers. All you may need to do is:

1. ensure that Composer is up to date by running:

   ```
   composer self-update
   ```

2. enable the PDO options for MySQL and SQLite (see Appendix A for how to do this by editing ther `c:\php\php.ini` file …)

B.1 Composer

The Composer tool is actually a **PHAR** (PHP Archive) - i.e. a PHP application packaged into a single file. So ensure you have PHP installed and in your environment **path** before attempting to install or use Composer.

Ensure you have (or install) an up-to-date version of the Composer PHP package manager.

```
composer self-update
```

B.1.1 Windows Composer install

Get the latest version of Composer from

- getcomposer.org

- run the provided **Composer-Setup.exe** installer (just accept all the default options - do NOT tick the developer mode)
  - https://getcomposer.org/doc/00-intro.md#installation-windows

B.2 PHPStorm editor

Ensure you have your free education Jetbrains licence from:

- Students form: https://www.jetbrains.com/shop/eform/students (ensure you use your ITB student email address)

Downdload and install PHPStorm from:

- https://www.jetbrains.com/phpstorm/download/

To save lots of typing, try to install the following useful PHPStorm plugins:

- Twig
- Symfony
- Annotations

B.3 MySQL Workbench

While you can work with SQLite and other database management systems, many ITB modules use MySQLWorkbench for database work, and it's fine, so that's what we'll use (and, of course, it is already installed on the ITB windows computers …)

Download and install MySQL Workbench from:

- https://dev.mysql.com/downloads/workbench/

B.4 Git

Git is a fantastic (and free!) DVCS - Distributed Version Control System. It has free installers for Windows, Mac, Linus etc.

Check is Git in installed on your computer by typing `git` at the CLI terminal:

```
> git
usage: git [--version] [--help] [-C <path>] [-c name=value]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

```
These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

...

collaborate (see also: git help workflows)
   fetch      Download objects and refs from another repository
   pull       Fetch from and integrate with another repository or a local branch
   push       Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.

>
```

If you don't see a list of **Git** commands like the above, then you need to install Git on your computer.

B.5 Git Windows installation

Visit this page to run the Windows Git installer.

- https://git-scm.com/downloads

NOTE: Do **not** use a GUI-git client. Do all your Git work at the command line. It's the best way to learn, and it means you can work with Git on other computers, for remote terminal sessions (e.g. to work on remote web servers) and so on.

C

A new, empty Symfony 4 web application

C.1 Create a new symfony project

Create and `cd` into a folder for these labs. Then create a new Symfony 4, empty project by typing:

```
> mkdir lab01
lab01> composer create-project symfony/skeleton my-project
```

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

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

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

```
        Quit the server with CTRL-C.
        Run composer require server for a better web server.
lab01>
```

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

Let's do what we are told:

1. Change to the project directory `my-project`:

   ```
   lab01> cd my-project
   lab01/my-project>
   ```

2. Execute command to run the PHP built-in webserver, serving files from directory `public`[1]:

   ```
   lab01/my-project> php -S localhost:8000 -t public
   ```

3. Browse to the http://localhost:8000/ URL.

   Figure C.1 shows a screenshot of the default Symfony page for a new, empty project.

[1] `localhost` is the same as `127.0.0.1` so use whichever you prefer.

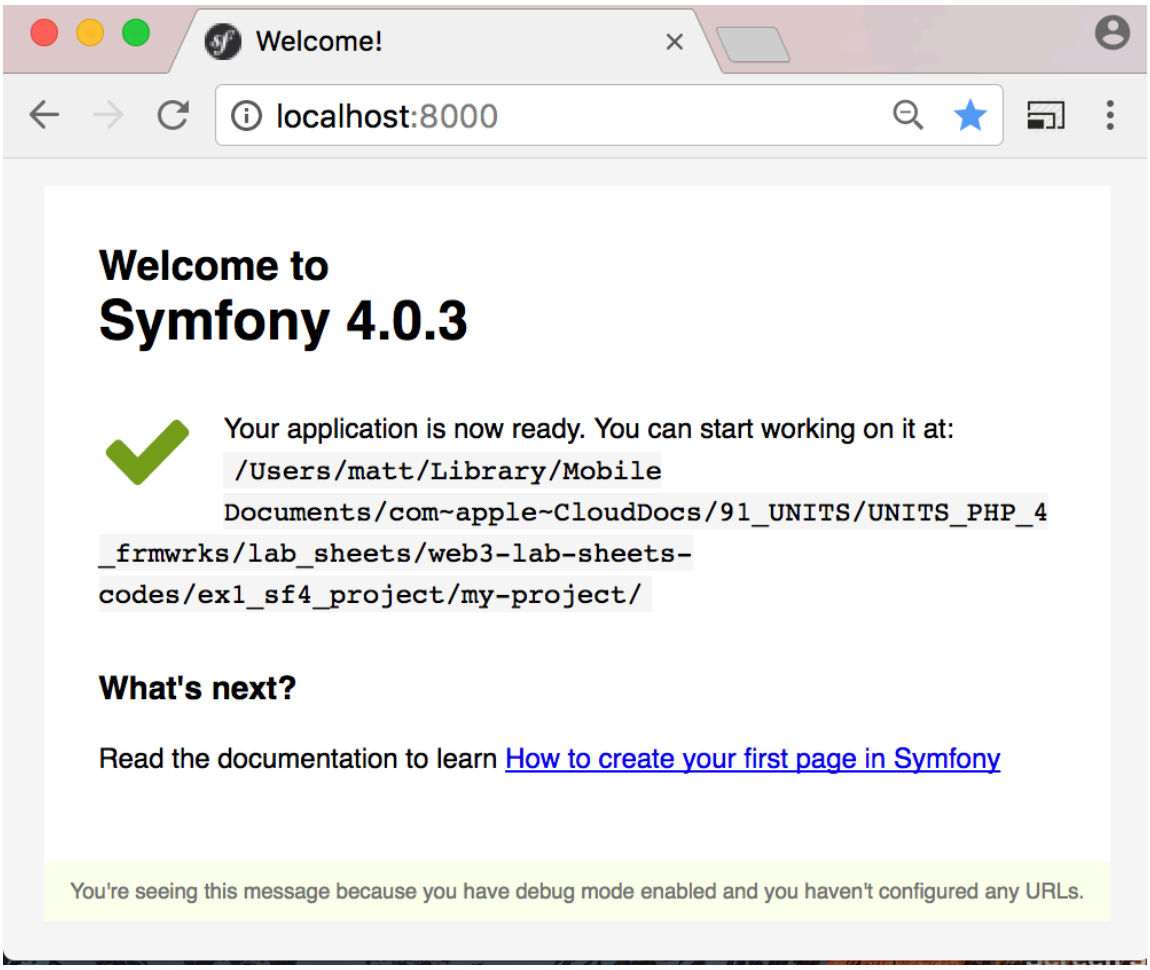


Figure C.1: Default Symfony 4 web page.

D

The fully-featured Symfony 4 demo

D.1 Visit Github repo for full Symfony demo

Visit the project code repository on Github at: https://github.com/symfony/demo

D.2 Git steps for download (clone)

If you have Git setup on your computer (it is on the college computers) then do the following:

- copy the clone URL into the clipboard
- open a CLI (Command Line Interface) window
- navigate (using `cd`) to the location you wish to clone[1]
- use `git clone <url>` to make a copy of the project on your computer

```
lab01 $ git clone https://github.com/symfony/demo.git
Cloning into 'demo'...
remote: Counting objects: 7165, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 7165 (delta 4), reused 8 (delta 2), pack-reused 7150
Receiving objects: 100% (7165/7165), 6.79 MiB | 1.41 MiB/s, done.
Resolving deltas: 100% (4178/4178), done.
```

[1] For a throw-away exercise like this I just create a directry named `temp` (with `mkdir temp`) and `cd` into that…

```
lab01 $
```

D.3 Non-git download

If you don't have Git on your computer, just download and **unzip** the project to your computer (and make a note to get **Git** installed a.s.a.p.!)

D.4 Install dependencies

Install any required 3rd party components by typing `cd`ing into folder `demo` and typing CLI command `composer install`. A **lot** of dependencies will be downloaded and installed!

```
lab01/demo $ composer install
Loading composer repositories with package information
Installing dependencies (including require-dev) from lock file
Package operations: 89 installs, 0 updates, 0 removals
  - Installing ocramius/package-versions (1.2.0): Loading from cache
  - Installing symfony/flex (v1.0.65): Loading from cache
  ...
  - Installing symfony/phpunit-bridge (v4.0.3): Loading from cache
  - Installing symfony/web-profiler-bundle (v4.0.3): Loading from cache
  - Installing symfony/web-server-bundle (v4.0.3): Loading from cache
Generating autoload files
ocramius/package-versions:  Generating version class...
ocramius/package-versions: ...done generating version class
```

D.5 Run the demo

Run the demo with `php bin\console server:run`

(Windows) You may just need to type `bin\console  server:run` since I think there is a `.bat` file in `\bin`:

```
lab01/demo$ php bin/console server:run

 [OK] Server listening on http://127.0.0.1:8000

// Quit the server with CONTROL-C.
```

```
PHP 7.1.8 Development Server started at Tue Jan 23 08:19:05 2018
Listening on http://127.0.0.1:8000
Document root is /Users/matt/Library/Mobile Documents/com~apple~CloudDocs/91_UNITS/UNITS_PHP_4_frmw
Press Ctrl-C to quit.
```

D.6 View demo in browser

Open a browswer to `localhost:8000` and play around with the website. Figure D.1 shows a screenshot of the default Symfony page for a new, empty project.

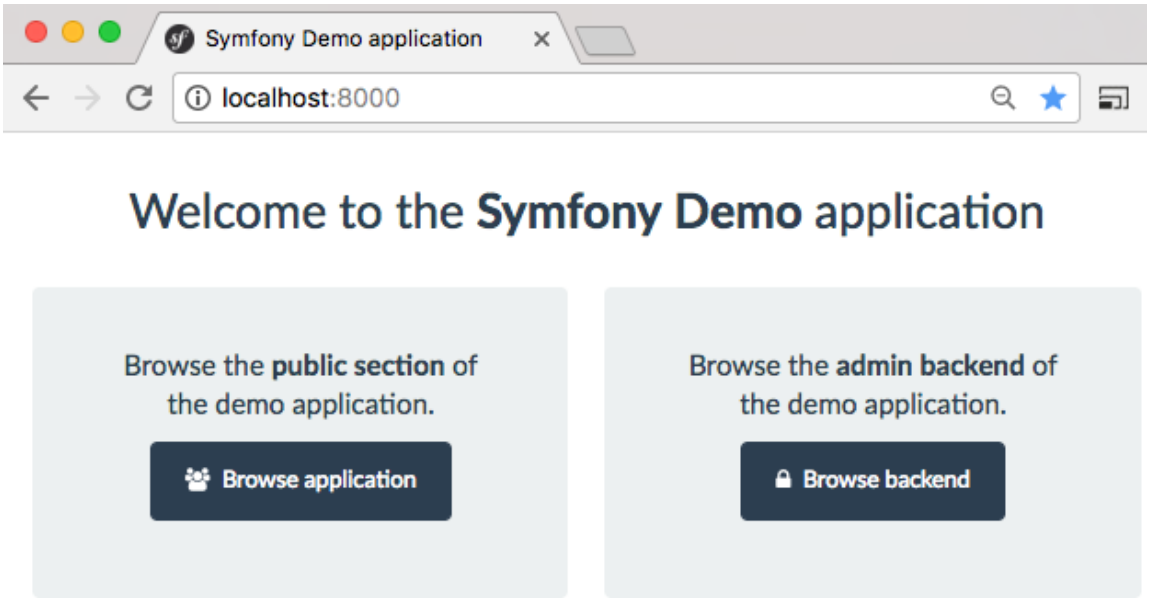


Figure D.1: Default Symfony 4 demo project

D.7 Explore the code in PHPStorm

Open the code for the project in PHPStorm, and look especially in the `/controllers` and `/templates` directories, to work out what is going on

E

Running the tests in the SF4 demo

E.1 The project comes with configuration for `simple-phpuit`. Run this once to download the depedencies:

```
lab01/demo $ vendor/bin/simple-phpunit
   ./composer.json has been updated
   Loading composer repositories with package information
   Updating dependencies
   Package operations: 19 installs, 0 updates, 0 removals
     - Installing sebastian/recursion-context (2.0.0): Loading from cache
     ...
     - Installing symfony/phpunit-bridge (5.7.99): Symlinking from /Users/matt/lab01/demo/vendor/sy
   Writing lock file
   Generating optimized autoload files

lab01/demo $
```

E.2 Run the tests

Run the tests, by typing `vendor\bin\simple-phpunit`[1]

[1] The backlash-forward slash thing is annoying. In a nutshell, for file paths for Windows machines, use backslashes, for everything else use forward slashes. So it's all forward slashes with Linux/Mac machines :-)

```
lab01/demo $ vendor/bin/simple-phpunit
PHPUnit 5.7.26 by Sebastian Bergmann and contributors.

Testing Project Test Suite
.............................................                    49 / 49 (100%)

Time: 27.65 seconds, Memory: 42.00MB

OK (49 tests, 88 assertions)
matt@matts-MacBook-Pro demo (master) $
```

E.3 Explore directory `/tests`

Look in the `/tests` directory to see how those tests work. For example Figure E.1 shows a screenshot of the admin new post test in PHPStorm.

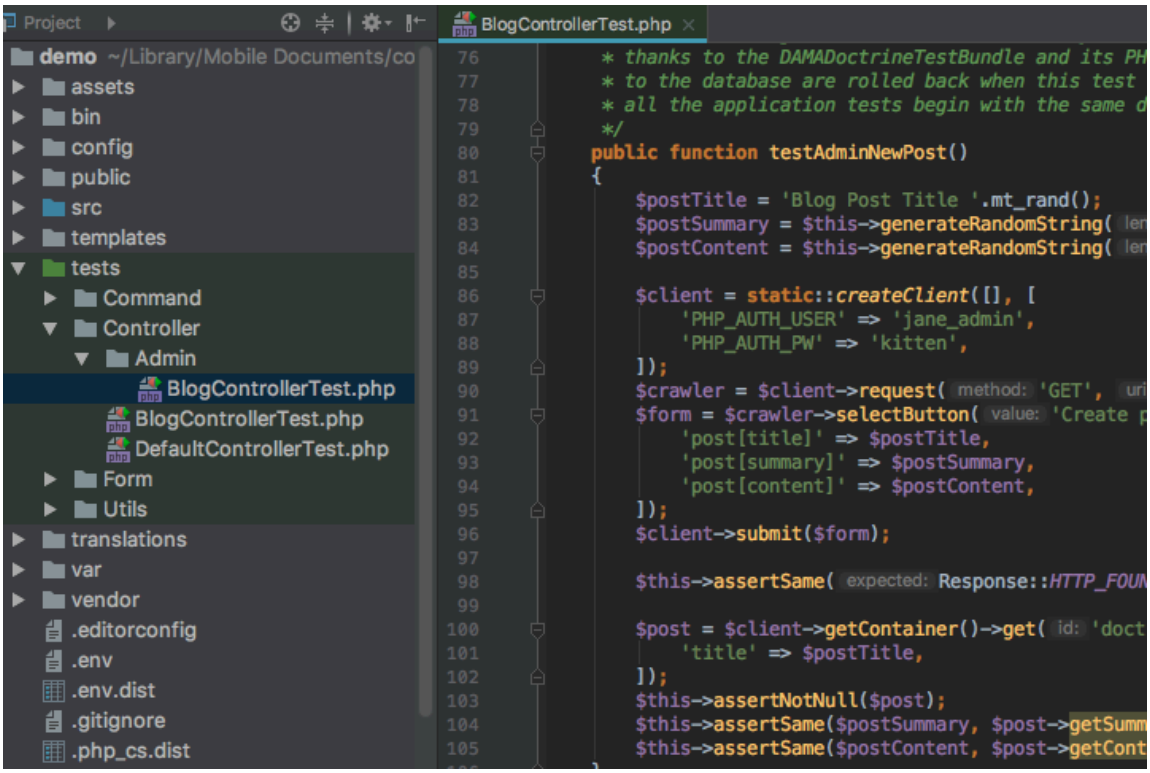


Figure E.1: The admin new post test in PHPStorm

E.4 Learn more

Learn more about PHPUnit testing and Symfony by visiting:

- https://symfony.com/doc/current/testing.html