AN INTRODUCTION TO SYMFONY 3
(for people that already know OO-PHP and some MVC stuff)

by
Dr. Matt Smith
mattsmithdev.com
gorynggе.com
https://github.com/dr-matt-smith

2017

Acknowledgements

Thanks to ...

Table of Contents

Part I

Introduction to Symfony

1

Project Introduction

1.1 Project Objectives

The main goal of this project was to create a simple multi-player based party game in the Unity video game engine, akin to Mario Party for example. The game has to have entertainment value. Have simple rules, be easy to follow and must be capable of being completed, the game must not be too difficult. The main goals for the game are as follows - There are a minimum of three players. - One player controls an antagonist type character represented by a series of fire balls. These fireballs are lined up horizontally and vertically. - The fireballs lined up vertically can only move from left to right, and the the fireballs lined up horizontally can only move up and down. - Two or more players control the protagonist characters. It is their job to each try and survive the level for 60 seconds. - It is the player controlling the antagonist's role to move the fireballs around in an attempt to kill the protagonist players. - While it is the protagonists job to avoid these fireballs while also attempting to hinder the other player(s). - There are pickups which can be collected at certain intervals throughout the level. - One pickup is a shield which provides the player with invincibility for a short period of time from the fireballs. - One pickup is a ice blast, which when picked up adds to an inventory and is used as a projectile to freeze the other player in place. - There are two game over screens. One which displays when all protagonist are killed, and another which displays when a protagonist(s) survive for 60 seconds.

See Figure 1.1 for a screenshot.

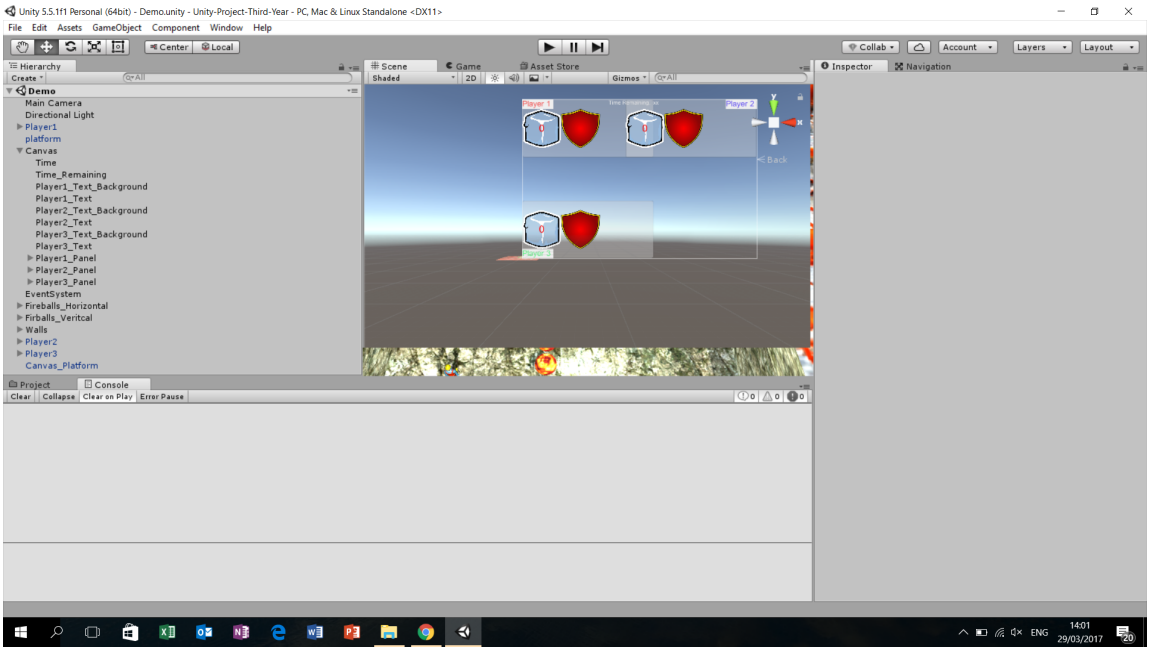


Figure 1.1: A screenshot.

1.2 Overview of Project

For the third year project I decided to develop a game within the Unity engine that would have multi-player elements to it. When the game was completed it would be tested within the engine using Xbox controllers and the keyboard on the PC itself.

The game consists of a minimum of 3 players, one player controlling the antagonist represented by a series of fireballs, and at least two players controlling the protagonists consisting of the "Gravity Guys". The two players try so survive the level, while also hindering the other players with pickups which can be collected throughout the level.

The project was originally supposed to be a group project, it eventually became an individual project. The project started out as a group deciding what would be the best way to undergo research in how to develop this game? What type of a game would it be? How could this game be developed into an easy to play, and most of all, a fun multi-player experience?

Research was started by looking at the Unity engine itself and previous games which have been developed in the engine and the capabilities shown in these games and if it would be possible to implement these features into the game I was developing. More research was conducted into multi-player player games in general, such as what makes them fun to play and how can you and your friends enjoy the experience as much as possible?

Also looked at was the issues that some of these games had, and how exactly could they be improved upon in the game that was being developed for this project. The final game would then hopefully

be completed using the information gathered from the conducted research. A demo of the project was displayed in December to show how the basics of how the game would work, with a full working version being finished by the end of April.

1.3 Overview of Report

Chapter 2 will contain the research which was conducted into the development of this game. Chapter 3 will contain an analysis of how the game works and the features of the game. While in chapter 4 we will look at the code used to make all of this feasible.

1.4 Research Conducted

1.5 Introduction

In this chapter I will present all the relevant research relating to the project that has been conducted. This includes research of the Unity engine, games developed in the engine and the history of multi-player video games and why are they so popular.

1.6 Unity and Video Game Engines

Almost every one on planet earth has probably come into contact with video games at some point in their life, whether it through playing them, watching someone else play them or even seeing a movie previously based off of a video game. Yes, it is impossible to avoid video games on modern society. But how exactly are video games made? How does what you interact with on screen from traversing through space, to saving your princess from castle to castle, to even playing a virtual sport without ever having to leave your house made possible? It's simple really, these are all made possible through the software frameworks which are video game engines. These engines are used to design the video games in which you play on your consoles, your PCs and even your mobile devices. These frameworks provide for you, the tools to make your video game such as, the physics of your game, the animations, the scripting and the AI. Popular video game engines include the Frostbite engine, the Unreal engine, and of course the Unity engine.

The Unity Engine was first announced in 2005 originally only to be used on Mac OS due to its popularity it has since been extended to 27 platforms such as Desktop, Games Consoles, Virtual Reality and even Smart Tv's. The popularity of the platform increased even more with the launch of Unity 5 in 2015 with more than 5 billion games developed in Unity downloaded in the third quarter of 2016 alone, with the technology being popular among developers more than any other

third party game development software, with developers including the likes of Ubisoft, Square Enix, Warner Bros, Obsidian and even Coca Cola.

Many popular games in today's gaming culture have been developed in Unity including the likes of, Slender: The Eight Pages, a popular survival horror game in which a player must collect eight pages scattered throughout a map all while avoiding someone who is following them. 7 Days to Die, another survival horror game set in an open world where the player must craft and survive in the wilderness while fighting against zombies. Cities Skylines, a simulator in which the user builds entire cities and the rules in which the city operates, and recently the extremely popular mobile game Pokemon GO was developed in Unity, a game in which you walk around with your phone and then encounter Pokemon and use the camera in your phone to capture them.

All of this shows that there are many different features in which can be implemented into games made using thus engine and just how creative a developer can be when developing a game which is one the goals of my project was to try make the game as creative as possible.

1.7 Multi-Player Video Games

2

Creating our own classes

2.1 What we'll make (`project03`)

See Figure 2.1 for a screenshot of the students list page we'll create this chapter.

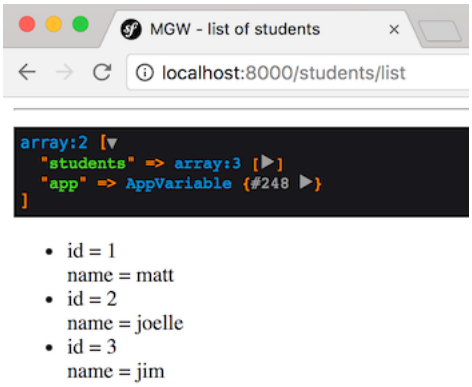


Figure 2.1: Students lists page.

2.2 A collection of `Student` records

Although we'll be moving on to use a MySQL database soon for persistent data storage, let's start off with a simple DIY (Do-It-Yourself) situation of an entity class (`Student`) and a class to work with collections of those entities (`StudentRepository`).

We can then pass an array of `Student` records to a Twig template and loop through to display them one-by-one.

Here is our `Student.php` class:

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

    public function __construct($id, $name){
        $this->id = $id;
        $this->name = $name;
    }

    public function getId()
    {
        return $this->id;
    }

    public function getName()
    {
        return $this->name;
    }
}
```

So each student has simply an 'id' and a 'name', with public getters for each and a constructor.

Here is our `StudentRepository` class:

```
class StudentRepository
{
    private $students = [];

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

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

So our repository has a constructor which hard-codes 3 `Student` records and adds them to its array. There is also the public method `getAll()` that returns the array.

The simplest location for our own classes at this point in time, is in the onl 'bundle' we have, the `AppBundle`. So we can declare our PHP class files in directry `/src/AppBundle`. Figure 2.2 shows the `DefaultController.php` in this location.

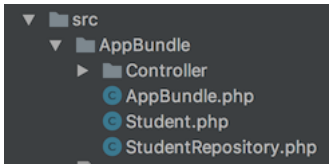


Figure 2.2: Location of Student and StudentRepository classes.

Following the way Symfonhy projects use the PSR-4 namespacing system, we will namespace the class with exactly the same name as the directory they are located in.

```
namespace AppBundle;

class Student
{
    ... etc.
}
```

2.3 Using `StudentRepository` in a controller

Since we now have created our namespaced classes we can use them in a controller. Let's create a new controller to work with requests relating to `Student` objects. We'll name this `StudentController` and locate it in `/src/AppBundle/Controller` (next to our existing `DefaultController`).

Here is the listing for `StudentController.php` (note we need to add a `use` statement so that we can refer to class `StudentRepository`):

```
use AppBundle\StudentRepository;

class StudentController extends Controller
{
    /**
```

```
     * @Route("/students/list")
     */
    public function listAction(Request $request)
    {
        $studentRepository = new StudentRepository();
        $students = $studentRepository->getAll();

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

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

We can see from the above that we have declared a controller method `listAction` in our `StudentController`. We can also see that this controller action will be invoked when the webapp receives a HTTP request with the route pattern `/students/list`.

The logic exectuted by the method is to get the array of `Student` records from an instance of `StudentRepository`, and then to pass this array to be rendered by the Twig template `students/list.html.twig`.

2.4 Creating the Twig template to loop to display all students

We will now create the Twig template `list.html.twig', in locaton`/app/Resources/views/students‘.

Figure 2.3 shows the 2 templates we are about to create in this location.

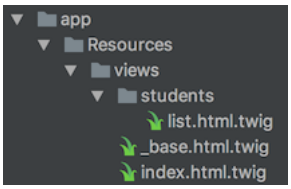


Figure 2.3: Location of Twig template `list.html.twig`.

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

{% block pageTitle %}list of students{% endblock %}
```

```
{% block body %}

    {{ dump() }}

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

{% endblock %}
```

Part II

Symfony and Databases

3

Doctrine the ORM

3.1 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)[http://symfony.com/doc/current/doctrine.html]

3.2 Quick start

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

1. `doctrine:database:create`
2. `doctrine:database:migrate` (or possibly `doctrine:schema:update --force`)
3. `doctrine:fixtures:load`

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

3.3 Setting up your project to work with MySQL or SQLite

You need to decide which database system you'll use for your project, and then configure the project with details of the database driver, host/path etc. See the following Appendices to learn about these issues, and to find specific instructions for both MySQL and SQLite (both are very easy to setup for Symfony):

- Appendix D describing the parameter and config files in `/app/config`
- Appdenix E describing how to set up a project for MySQL
- Appdendix F describing how to set up a project for SQLite

  NOTE this appendix also includes a link to the SQLite website page helping you decide whether SQLite is suitable

If you aer working on a small project / small website, often you'll find SQLite easier to setup and faster to work with (since you don' need to run any database server etc.). So it's worth a few minutes thinking before choosing which database system to work with before you go ahead and configure your project.

4

Working with Entity classes

4.1 A `Student` entity class

Doctrine expects to find entity classes in a directory named `Entity`, so let's create one and move our `Student` class there. We can also delete class `StudentRepository` since Doctrine will create repository classes automatically for our entities (which we can edit if we need to later to add project-specific methods).

Do the following:

1. create directory `/src/AppBundle/Entity`
2. move class `Student` to this new directory
3. delete class `StudentRepository`

We also need to add to the namespace inside class `Student`, changing it to `AppBundle\Entity`. We also need to remove all methods, since Doctrine with create getter and setters etc. automatically. So edit class `Student` to look as follows, i.e. just listing the properties 'id' and 'name':

```
namespace AppBundle\Entity;

class Student
{
    private $id;
    private $name;
}
```

4.2 Using annotation comments to declare DB mappings

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

Our first comment is for the class, stating that it is an ORM entity and mapping it to database table `students`:

```
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="students")
 */
class Student
```

4.3 Declaring types for fields

We now use annotations to declare the types (and if appropriate, lengths) of each field. Also for the 'id' we need to tell it to `AUTO_INCREMENT` this special field.

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

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

4.4 Valdiate 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 Doctring to create the actual database table, 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] FAIL - The database schema is not in sync with the current mapping file.
```

4.5 Generating getters and setters

We can tell Doctrine to complete the creation of the entity class with the `generate:entities` command:

```
php bin/console doctrine:generate:entities AppBundle/Entity/Student
```

We can also add our **own** logic to the entity class, for any special getters etc.

You can tell Doctrine to generate all entities for a given 'bundle' (but ?? it may overwrite any edits you've made to entites[1])

```
$ php bin/console doctrine:generate:entities AppBundle
```

So we now have getters and setters (no setter for ID since we don't change the AUTO INCREMENTED db ID value) added to our class `Student`:

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

/**
 * Set name
 *
 * @param string $name
 *
 * @return Student
```

[1] NOTE TO SELF - CHECK THIS WHEN YOU HAVE A CHANCE

```
     */
    public function setName($name)
    {
        $this->name = $name;
        return $this;
    }

    /**
     * Get name
     *
     * @return string
     */
    public function getName()
    {
        return $this->name;
    }
```

4.6 Creating tables in the database

Now our entity `Student` is completed, we can tell Doctrine to create a corresponding table in the database (or ALTER the table in the database if one previously existed):

```
$ php bin/console doctrine:schema:update --force
```

if all goes well you'll see a couple of confirmation messages after entering the command above:

```
Updating database schema...
Database schema updated successfully! "1" query was executed
$
```

You should now see a new table in the database in your DB manager. Figure 4.1 shows our new `students` table created for us.

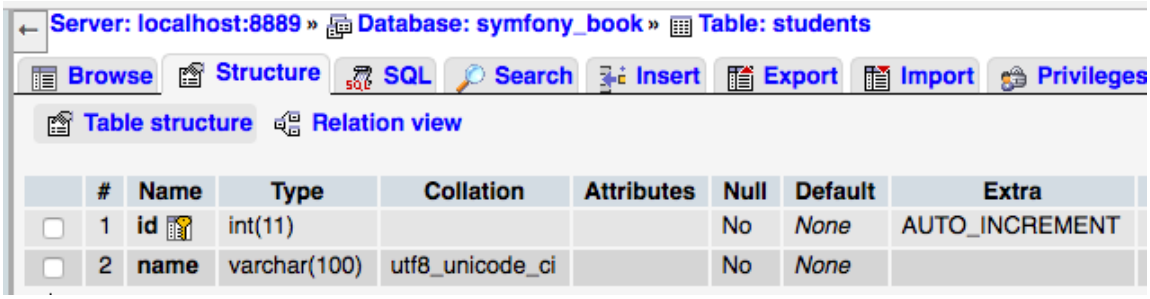


Figure 4.1: CLI created table in PHPMyAdmin.

4.7 Generating entities from an existing database

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

- Symfony docs on inferring entites from existing db tables

5

Symfony approach to database CRUD

5.1 Creating new student records

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

```
/**
 * @Route("/students/create/{name}")
 */
public function createAction($name)
{
    $student = new Student();
    $student->setName($name);

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

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

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

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

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

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

Figure 5.1 shows how a new record `freddy` is added to the database table via route `/students/create/{name}`.

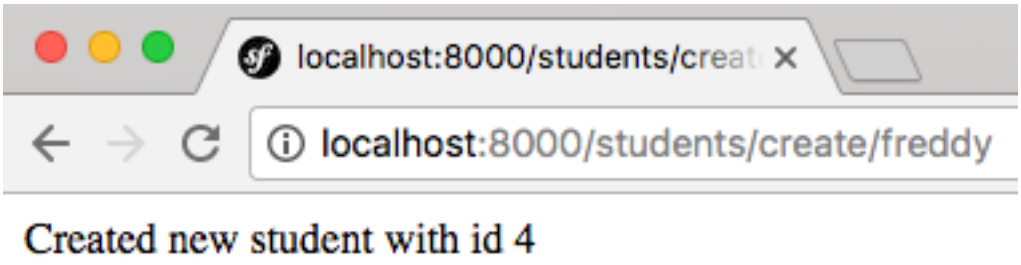


Figure 5.1: Creating new student via route `/students/create/{name}`.

We can see these records in our database. Figure 5.2 shows our new `students` table created for us.

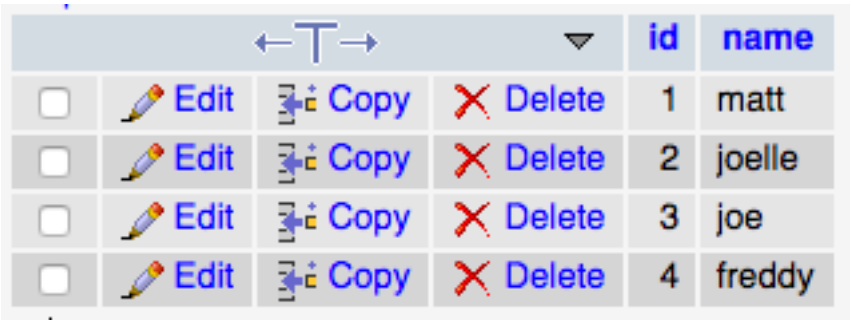


Figure 5.2: Controller created records in PHPMyAdmin.

5.2 Updating the listAction() to use Doctrine

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

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

Doctrine repositories offer us lots of useful methods, including:

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

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

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

// dynamic method names to find a group of products based on a column value
$products = $repository->findByPrice(19.99);
```

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

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

Our listAction() method now looks as follows:

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

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

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

Figure 5.3 shows how a new record `freddy` is added to the database table via route `/students/create/{name}`.

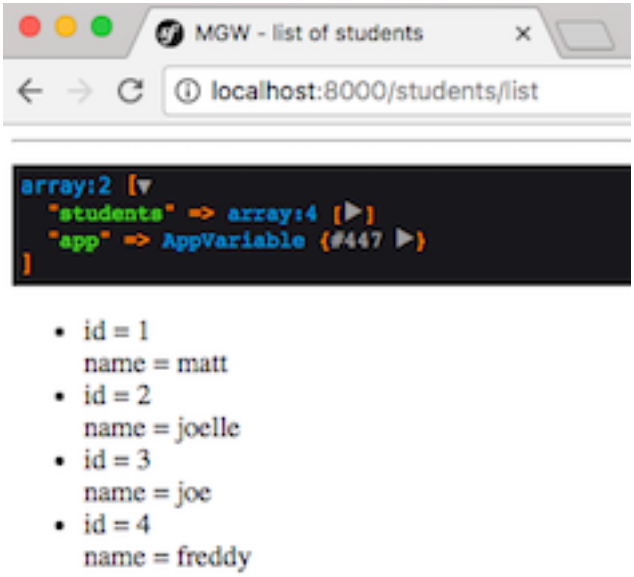


Figure 5.3: Listing all database student records with route `/students/list`.

5.3 Deleting by id

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

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

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

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

    // actually executes the queries (i.e. the DELETE query)
```

```
        $em->flush();

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

5.4 Updating given id and new name

We can do something similar to update. In this case we need 2 parameters: the id and the new name. 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("/students/update/{id}/{newName}")
     */
    public function updateAction($id, $newName)
    {
        $em = $this->getDoctrine()->getManager();
        $student = $em->getRepository('AppBundle:Student')->find($id);

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

        $student->setName($newName);
        $em->flush();

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

Until we write an error handler we'll get Symfony style exception pages, such as shown in Figure 5.4 when trying to update a non-existant student with id=99.

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

- we are getting the reference to the repository via the entity manager `$em->getRepository('AppBundle:Student')`
- we are 'chaining' the `find($id)` method call onto the end of the code to get a reference to the repository (rather than storing the repostory object reference and then invoking `find($id)`).

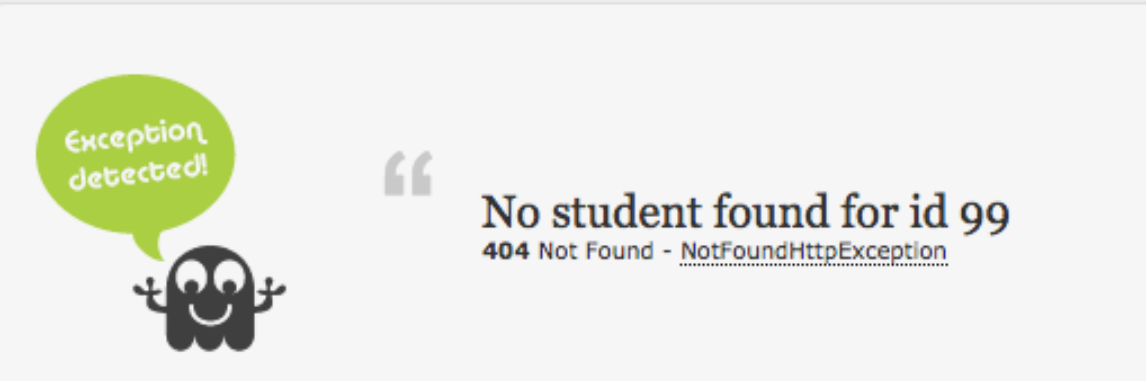


Figure 5.4: Listing all database student records with route `/students/list`.

This is an example of using the 'fluent' interface[1] offered by Doctrine (where methods finish by returning an reference to their object, so that a sequence of method calls can be written in a single statement.

- rather than returning a `Response` containing a message, this controller method redirect the webapp to the route named `homepage`

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

5.5 Creating the CRUD controller automatically from the CLI

Here is something you might want to look into ...

```
$ php app/console generate:doctrine:crud --entity=AppBundle:Student --format=annotation --wi
```

[1] read about it at Wikipedia

6

Completing CRUD and linking things together

6.1 Show one record (given id)

Let's add a final method to read (the 'R' in CRUD!) and show a single record to the user.

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

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

    $templateName = 'students/show';
```

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

We have named the route `students_show`. In fact we should go back and name **all* the routes we've just created controller methods for.

Our show method does the following:

- attempts to find a record for the given id (we get since we've an id in the route pattern, and a correspondingly named parameter for our method)
- throws an exception if no record could be found for that id
- creates a Twig argument array containing a single item congtaining our student record
- returns the Response created by rendering the `students/show.html.twig` template

6.2 Our template

We now need to creat the `students/show.html.twig` template. This will be created in `app/Resources/views/students`:

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

{% block pageTitle %}show one student{% endblock %}

{% block body %}

<h1>Show one student</h1>

<p>
id = {{ student.id }}

<p>
name = {{ student.name }}

<hr>
<a href="{{ path('students_list') }}">list of students</a>

{% endblock %}
```

This templates does the following:

- extends the base template and defines a page title
- shows a level 1 heading, and paragraphs for the id and name
- offers a link back to the list of students (using the route name `students_list`)

So we'd better ensure the `listAction()` controller method names its path with this identifier:

```
/**
 * @Route("/students/list", name="students_list")
 */
public function listAction(Request $request)
{
    ... etc
}
```

6.3 Making each name in the list be a link to its show page

Let's update our list template so that each name is itself a link to the show page (giving the id of each record).

A first attempt could be like this:

```
<a href="{{ path('students_show') }}/{{ student.id }}">
{{ student.name }}
</a>
```

But we get a Symfony error when we attempt to display this list page, complaining:

```
An exception has been thrown during the rendering of a template
("Some mandatory parameters are missing ("id") to generate a URL for route "students_show".").
```

Symfony can't see that we're trying to add on the id after the show route. So we need to pass the id parameter inside the Twig `path()` function as follows:

```
<a href="{{ path('students_show', {id:student.id}) }}">
    {{ student.name }}
</a>
```

There are lots of round and curly brackets all over the place, but try to remember that `path()` is a Twig function, taking the route name as the first parameter and the id (from student.id) as the second parameter.

Figure 6.1 shows our list of students with the names as links.

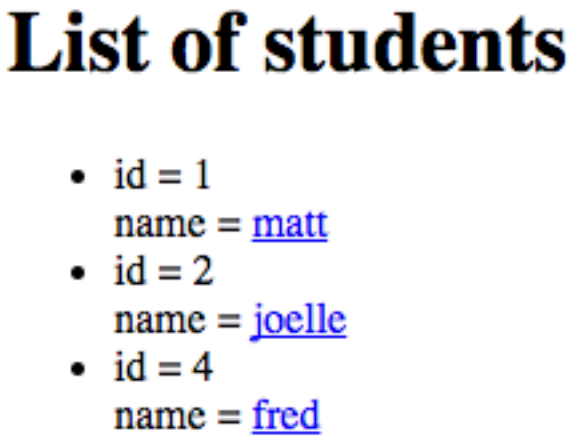


Figure 6.1: List of students with names as link to show pages.

Part III

Froms and form processing

7

DIY forms

7.1 Adding a form for new Student creation (`project05`)

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

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

The form will look as show in Figure 7.1.

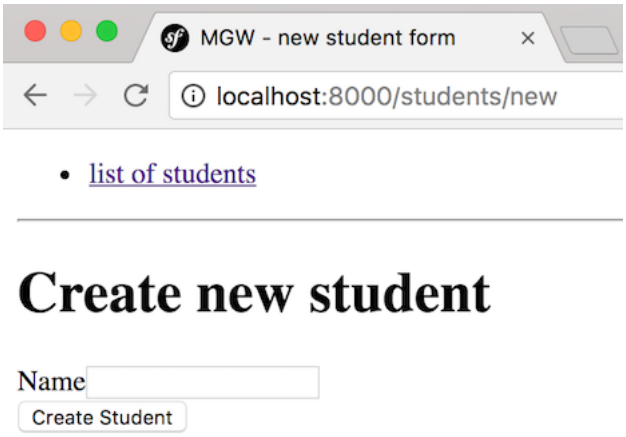


Figure 7.1: Form for a new student

7.2 Twig new student form

Here is our new student form '/app/views/students/new.html.twig':

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

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

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

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

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

```
/**
 * @Route("/students/new", name="students_new_form")
 */
public function newFormAction(Request $request)
```

```
{
    $argsArray = [
    ];

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

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

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

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

7.4 Controller method to process POST form data

We can access POST submitted data using the following expression:

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

So we can extract and store in `$name` the POST `name` parameter by writing the following:

```
$name = $request->request->get('name');
```

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

```
    /**
     * @Route("/students/processNewForm", name="students_process_new_form")
     */
    public function processNewFormAction(Request $request)
    {
        // extract 'name' parameter from POST data
        $name = $request->request->get('name');

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

Note that we then invokve our existing `createAction()` method, passing on the extracted `$name` string.

7.5 Validating form data, and displaying temporary 'flash' messages in Twig (project06)

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

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

7.6 Three kinds of flash message: notice, warning and error (project06)

Typically we create 3 differnt kinds of flash notice:

- notice
- warning
- error

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

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

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

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

7.7 Adding validation in our 'processNewFormAction()z method

So let's add some validation logic to our processing of the new student form data:

7.8 Adding flash display (with CSS) to our Twig template

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

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

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

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

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

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

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

7.9 Adding validation logic to our form processing controller method

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

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

```
        if(empty($name)){
            $this->addFlash(
                'error',
                'student name cannot be an empty string'
            );

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

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

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

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

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

{% block body %}

    <h1>Create new student</h1>

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

    (... show HTML form as before ...)
```

8

Automatic forms generated from Entities

8.1 Using the Symfony form generator (`project07`)

Given an object of an Entity class, Symfony can analyse its property names and types, and generate a form (with a little help).

So in a controller we can create a `$form` object, and pass this as a Twig variable to the template `form`. Twig offers 3 special functions for rendering (displaying) forms, these are:

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

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

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

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

The 'magic' happens in the controller method…

8.2 Updating StudentController->newFormAction()

Let's refactor `newFormAction()` to use Symfony's FormBuilder to create the form for us, based on an instance of class `Student`:

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

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

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

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

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

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

We can see that the method does the following:

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

Figure 8.1 shows a screenshot of the resulting form:

If we look further down (see Figure 8.2) we can see that the Symfony debug profiler bar footer (and the Chrome HTTP request information) shows that we are looking at an HTTP GET request to `localhost:8000/students/new` that received a `200 OK` HTTP response code.

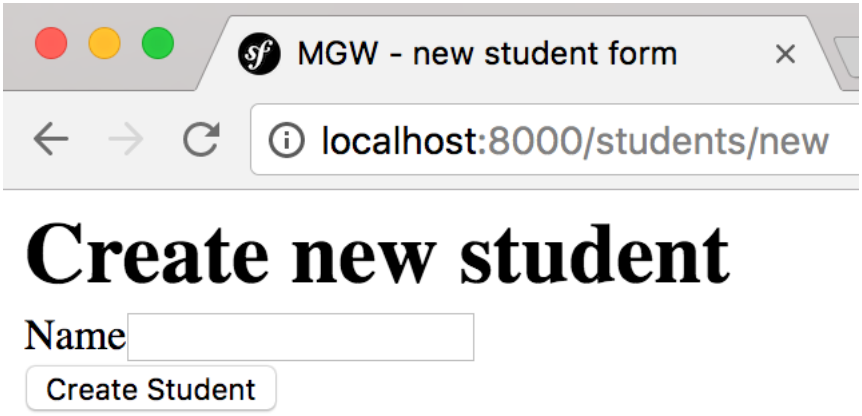


Figure 8.1: Symfony generated new student form (showing footer profiler bar).

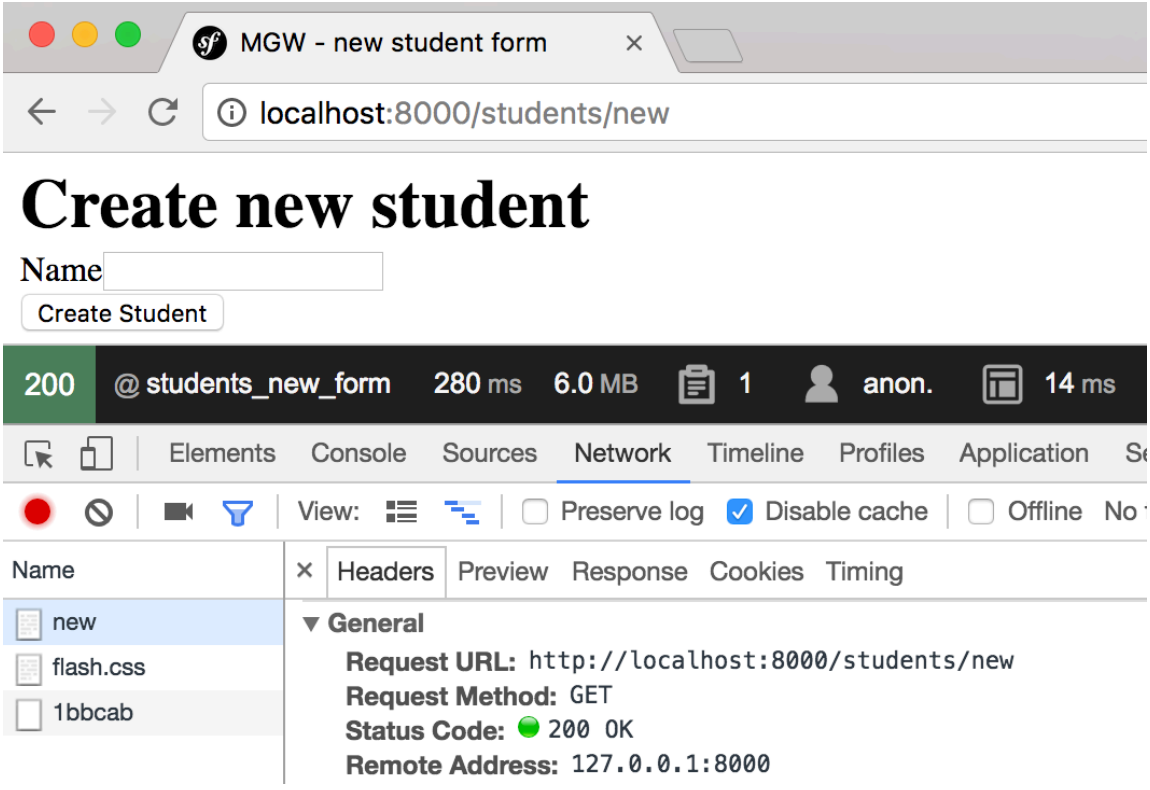


Figure 8.2: Generated student form - showing footer profiler bar.

8.3 Entering data and submitting the form

We find, however, that we haven't done enough if we actually enter a name (e.g. `joe-smith`) and submit the form via the submit button. Figure 8.3 shows that we just see a new empty form again! What we expect when we click a form submit button is for the entered values to be submitted to the server as an HTTP POST method. This is what has happned, **but** this request has been sent to the same URL as we used to display the form, i.e. `localhost:8000/students/new`. At present, our controller method does not distinguish between `GET` and `POST` methods, so simply responds by rendering the form again base on, another, new empty `Student` object. The Symfony footer profile bar shows us that it was a `POST` HTTP method request by writing `POST@students_new_form` (the name of the matched route, as defined in the controller annotation comment).

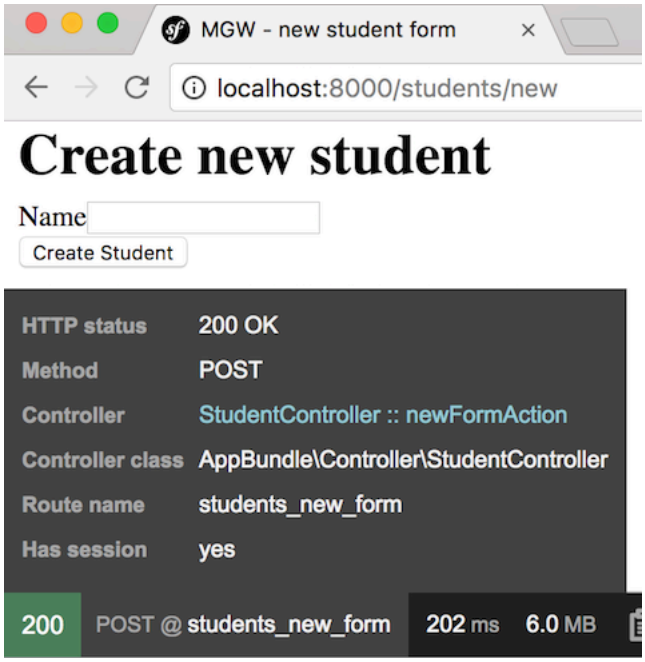


Figure 8.3: Form re-displayed depsite `POST` submission of name `joe-smith`.

We can see **why** the form submits to the same request URL as was used to display the form, if we look at the generated HTML (Chrome right-click `View Page Source`):

```
<h1>Create new student</h1>

<form name="form" method="post">
<div id="form"><div><label for="form_name" class="required">Name</label>
<input type="text" id="form_name" name="form[name]" required="required" /></div>
<div>
<button type="submit" id="form_save" name="form[save]">Create Student</button></div>
<input type="hidden" id="form__token" name="form[_token]" value="TJM9iQSmrWWdYLVcbflJ15-
```

```
</form>
```

Becase there is no `action` attribute in the `<form>` tag, then browsers automatically submit back to the same URL. This is known in web development as a **postback** and is very common[1].

If we use the Chrome developer tools again, after submitting name `joe-smith` we can see that the name has been sent in the body of the `POST` request to our webapp, as `form[name]`. We can see these details in Figure 8.4.

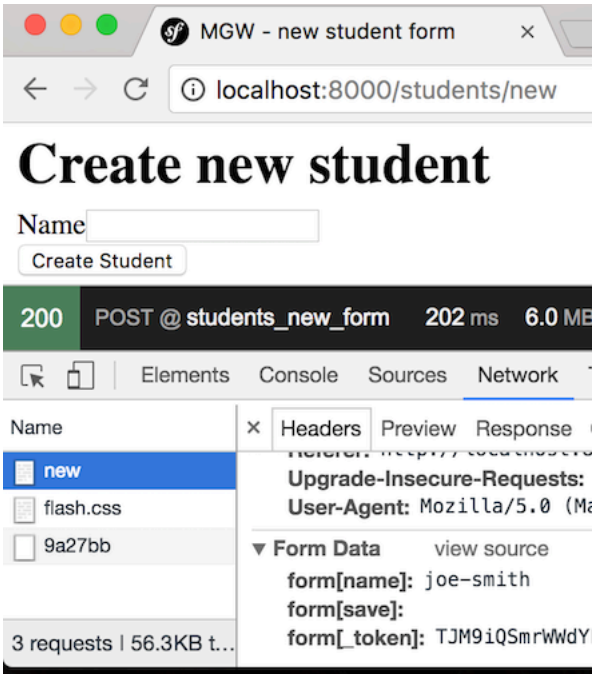


Figure 8.4: Chrome developer tools showing `POST` submitted variable `joe-smith`.

We can also delve further into the details of the request and our Symfony applications handing of the request by clicking on the Symfony debug toolbar, and, for example, clicking the `Request` navigation link on the left. Figure 8.5 shows us the `POST` variables received.

[1] read more at the Wikipedia postback page

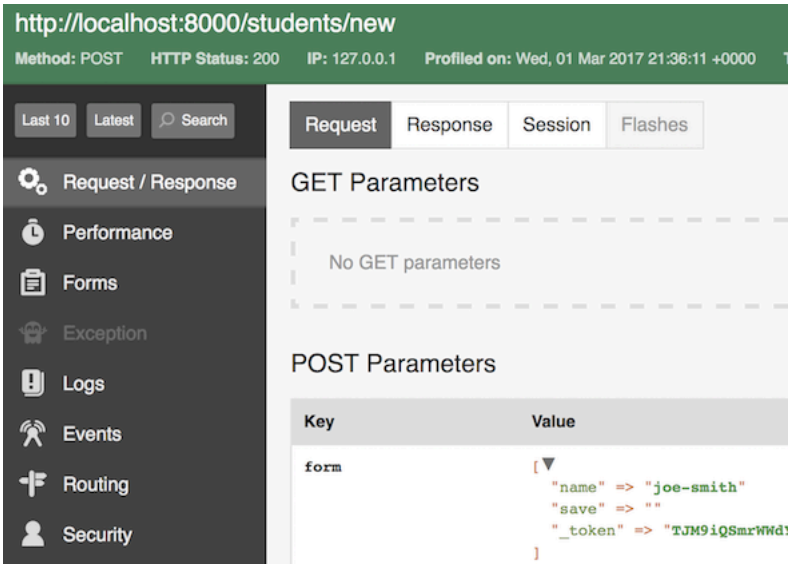


Figure 8.5: Chrome developer tools showing POST submitted variable `joe-smith`.

8.4 Detecting and processing postback form submission (and validation) (`project08`)

Since the form is posted back to the same URL as to display the form, then the same controller will be invoked. So we need to add some conditional logic in our controller to decide what to do. This logic will look like this:

```
prepare the form
tell the form to handle the request (i.e. get data from the Request into the form if its a p

IF form has been submitted (POST method) AND values submitted are all valid THEN
    process the form data appropriately
    return an approprate Response (or redirect appropriately)

OTHERWISE
    return a Response that renders the form
```

First let's do something really simply, if we detect the form has been submitted, let's just `var_dump()` the name received in the request and `die()`.

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

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

        /// ---- start processing POST submission of form
        $form->handleRequest($request);

        if($form->isSubmitted()){
            $student = $form->getData();
            $name = $student->getName();

            print "name received from form is '$name'";
            die();
        }

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

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

So as we can see above, after creating the form, we tell the form to examine the HTTP request to determine if it was a postback (i.e. POST method), and if so, to extract data from the request and store that data in the `Student` object inside the form:

```
$form->handleRequest($request);
```

Next, we can now test (with form method `isSubmitted()`) whether this was a POST request, and if so, we'll extract the `Student` object into `$student`, then get the name from this object, into `$name`, then print out the name and `die()`:

```
if($form->isSubmitted()){
    $student = $form->getData();
    $name = $student->getName();

    print "name received from form is '$name'";
    die();
}
```

However, if the form was not a postback submission (i.e. `isSubmitted()`), then we continue to create our Twig argument array and render the template to show the form.

The output we get, when submitting the name `joe-smith` with the above is shown in Figure 8.6.

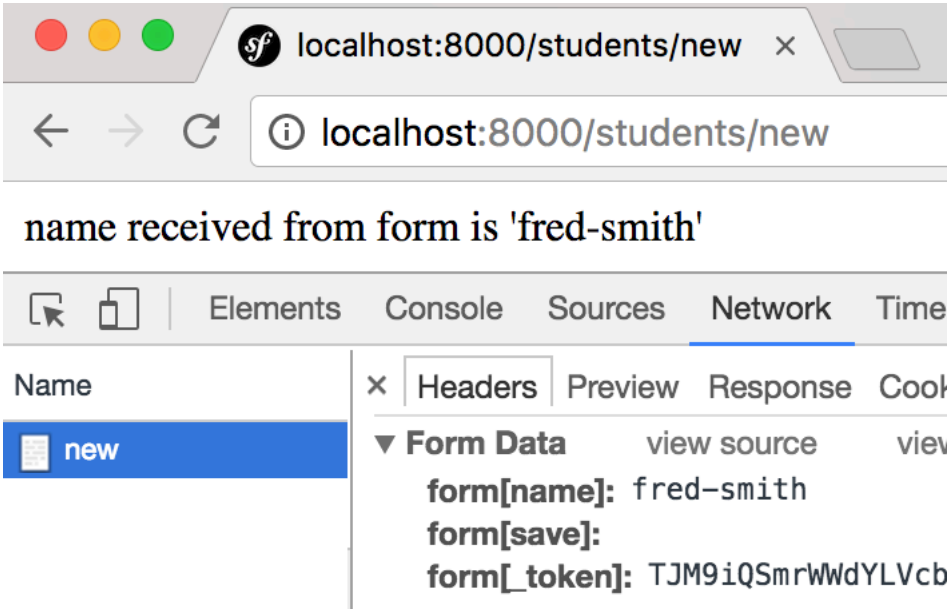


Figure 8.6: Confirmation of postback received name`joe-smith`.

8.5 Invoking the `createAction(...)` method when valid form data submitted

Let's write code to submit the extracted name property of the `Student` object in the form, to our existing `createAction(...)` method. So our conditional block, for the condition that if the form has been subnmitted **and** its data is valid will be:

```
if ($form->isSubmitted() && $form->isValid()) {
    $student = $form->getData();
    $name = $student->getName();
    return $this->createAction($name);
}
```

Here is a reminder of our `createAction($name)` method. Note that the final statement has been to redirect to the list of students route, after successful creation (and database persistance) of a new student object:

```
public function createAction($name)
{
    $student = new Student();
    $student->setName($name);
```

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

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

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

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

8.6 Final improvements (`project09`)

The final changes we might make include:

- to **remove** the route annotation for method `createAction(...)` - so it can only be invoked through our postback new student form route
- refactor method `createAction(...)` to receive a `Student` object - simplyfying the code in each method

So the refactored listing for method `createAction(...)` is:

```
    /**
     * @param Student $student
     *
     * @return \Symfony\Component\HttpFoundation\RedirectResponse
     */
    public function createAction(Student $student)
    {
        // entity manager
        $em = $this->getDoctrine()->getManager();

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

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

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

And our refactored method `newFormAction()` is:

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

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


     /// ---- start processing POST submission of form
     $form->handleRequest($request);

     if ($form->isSubmitted() && $form->isValid()) {
         $student = $form->getData();
         return $this->createAction($student);
     }

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

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

8.7 Video tutorials about Symfony forms

Here are some video resources on this topic:

- Code Review form validation with `@Assert`

9

Customising the display of generated forms

9.1 Understanding the 3 parts of a form (`project10`)

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

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

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

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

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

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

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

9.2 Using a Twig form-theme template

Symfony provides several useful Twig templates for common form layouts.

These include:

- wrapping each form field in a `<div>`
  - form_div_layout.html.twig
- put form inside a table, and each field inside a table row `<tr>` element
  - form_table_layout.html.twig
- Boostrap CSS framework div's and CSS classes
  - bootstrap_3_layout.html.twig

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

```
twig:
    debug:            "%kernel.debug%"
    strict_variables: "%kernel.debug%"
    form_themes:
      - 'form_div_layout.html.twig'
```

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

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

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

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

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

So we could display our new student form this way:

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

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

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

9.4 Customising display of parts of each form field

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

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

For example:

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

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

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

So we could display our new student form this way:

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

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

        {{ form_label(form.name) }}

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

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

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

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

```
<div>
    <div class="errors">

    </div>

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

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

9.5 Adding some CSS style to the form

We could, of course add some CSS so style labels nicely. We can add a `stylesheets` block to our Twig template:

```
{% block stylesheets %}
<style>
    label {
        display: inline-block;
        float: left;
        width: 10rem;
        padding-right: 0.5rem;
        font-weight:bold;
        color: blue;
        text-align: right;
    }

    .form-field {
        padding-bottom: 1rem;
    }
</style>
{% endblock %}
```

We can edit our `body` block to add the CSS class `form-field` to the `<div>` containing our `name` form field elements:

```
{% block body %}
```

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

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

            {{ form_label(form.name) }}

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

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

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

Note - by displaying errors for field `name` before the label, we ensure the label will always 'float' left of the text input box from the form widget.

Figure 9.1 shows what our CSS styled form looks like to the user.

Figure 9.1: Browser rendering of generated form with CSS.

Learn more at:

- The Symfony form customisation page

9.6 Specifying a form's method and action

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

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

Learn more at:

- Introduction to the Form component

Part IV

Symfony code generation

10

Generating entities from the CLI

10.1 Generating an 'elective' module entity from the CLI

Continuing our student/college example project, let's consider the case where students can select several subject elective 'modules', and store them in a 'basket' of electives. We'll learn about sessions for the shopping basket functionality in the next part, so for now let's create the `Elective` entity and use some CRUD to enter some records in the database.

We are going to use Doctrine's interactive CLI command to create class `AppBundle/Entity.php` for us. Entities have an integer `id` AUTO-INCREMENT primary key by default, so we just need to ask Doctrine to add string fields for `moduleCode` and `moduleTitle`, and an integer number of academic `credits` field - ensure your Webserver is running before working with Doctrine …

php bin/console generate:doctrine:entity –entity=AppBundle:Elective

First let's tell Doctrine that we want to create a new entity `Elective` in our `AppBundle`:

```
$ php bin/console generate:doctrine:entity --entity=AppBundle:Elective
```

Doctrine then tells us what is doing:

```
Welcome to the Doctrine2 entity generator


This command helps you generate Doctrine2 entities.
```

Then Doctrine tells us we need an entity 'shortcut name', but it also offers us one in square brackets, which we can accept by pressing `<Return>`:

```
First, you need to give the entity name you want to generate.
You must use the shortcut notation like AcmeBlogBundle:Post.

The Entity shortcut name [AppBundle:Elective]:
```

Next Doctrine asks us how we will declare the mapping information between this entity and the database table, again it offers us a default (annotation) in square brackets, we we accept by pressing `<Return>`:

```
Determine the format to use for the mapping information.

Configuration format (yml, xml, php, or annotation) [annotation]:
```

Finally Doctrine asks us to start describing each field we want.

```
Instead of starting with a blank entity, you can add some fields now.
Note that the primary key will be added automatically (named id).

Available types: array, simple_array, json_array, object,
boolean, integer, smallint, bigint, string, text, datetime, datetimetz,
date, time, decimal, float, binary, blob, guid.
```

Each field needs:

- field name
- field type
- field length (if string, not needed for some fields, like integer)
- Is nullable
- Unique

In most cases all we need to do is name the field, and either accept the default `string` data type (or correct it to integer or decimal), and then accept the defailts for the remaining field properties.

So let's create a string field `moduleCode`. Since `string` is the default, all we need to type is the field name and then press `<Return>` to accept the remaining defaults:

```
New field name (press <return> to stop adding fields): moduleCode
Field type [string]:
Field length [255]:
Is nullable [false]:
Unique [false]:
```

Let's do the same for string field `moduleCode`:

```
New field name (press <return> to stop adding fields): moduleTitle
Field type [string]:
Field length [255]:
```

```
Is nullable [false]:
Unique [false]:
```

Now we'll declare `integer` field `credits`. Don't worry, you don't have to type out the whole word `integer` - the CLI command will spot what you're typing after a couple of characters and you can accept it by pressing, you've guessed it, `<Returnu>`:

```
New field name (press <return> to stop adding fields): credits
Field type [string]: integer
Is nullable [false]:
Unique [false]:
```

When we've declared all the fields we wish to at this time, we just press `<Return>` when asked for the next field;s name:

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

Doctrine then goes off to create our Entity class, with all its getters and setters. and prints our a confirmation message of success, and telling us it created both an Entity class `Entity/Elective.php` and an associated Repository class `Repository/ electiveRepository.php` for Bundle `AppBundle`:

```
Entity generation


  created ./src/AppBundle/Entity/Elective.php
> Generating entity class src/AppBundle/Entity/Elective.php: OK!
> Generating repository class src/AppBundle/Repository/ElectiveRepository.php: OK!


Everything is OK! Now get to work :).
```

See Appendix H for another example of interactive CLI entity generation with the Doctrine command line tool.

10.2 Creating tables in the database

Now our entity `Elective` is completed, we can tell Doctrine to create a corresponding table in the database (or ALTER the table in the database if one previously existed):

```
$ php bin/console doctrine:schema:update --force
```

11

CRUD controller and templates generation

11.1 Symfony's CRUD generator

Symfony offers a very powerful CRUD generator command:

```
php bin/console generate:doctrine:crud --entity=AppBundle:Elective --format=annotation
--with-write --no-interaction
```

With the single command above Symfony will generate a CRUD controller (`ElectiveController`) and also create a directory containing Twig templates (`app/Resources/views/elective/index.html.twig` etc.).

11.2 The generated CRUD controller

Let's first look at the namespaces and class declaration line:

```
<?php

namespace AppBundle\Controller;

use AppBundle\Entity\Elective;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
```

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

/**
 * Elective controller.
 *
 * @Route("elective")
 */
class ElectiveController extends Controller
{
```

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

If we look in directory `app/Resources/views/elective/` we'll see the following generated templates:

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

Note that all these generated templates extend Twig class `base.html.twig`. If we want to continue using the identifier `_base.html.twig`, then we'll need to edit each of these generated templates to correct the extended Twig class name.

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

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

```
    /**
     * Lists all elective entities.
     *
     * @Route("/", name="elective_index")
     * @Method("GET")
     */
    public function indexAction()
    {
        $em = $this->getDoctrine()->getManager();

        $electives = $em->getRepository('AppBundle:Elective')->findAll();
```

```
            return $this->render('elective/index.html.twig', array(
                'electives' => $electives,
            ));
        }
```

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

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

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

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

```
{% for elective in electives %}
    <tr>
        <td><a href="{{ path('elective_show', { 'id': elective.id }) }}">
            {{ elective.id }}</a>
        </td>
        <td>{{ elective.moduleCode }}</td>
        <td>{{ elective.moduleTitle }}</td>
        <td>{{ elective.credits }}</td>
        <td>
            <ul>
                <li>
                    <a href="{{ path('elective_show', { 'id': elective.id }) }}">show</a>
                </li>
                <li>
                    <a href="{{ path('elective_edit', { 'id': elective.id }) }}">edit</a>
                </li>
            </ul>
        </td>
    </tr>
{% endfor %}
```

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

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

Remember in `/_base.html.twig` we have defined a block for style sheets:

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

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

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

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

Figure 11.1 shows a screenshot of how our list of electives looks,rendered by the `elective/index.html.twig` template.

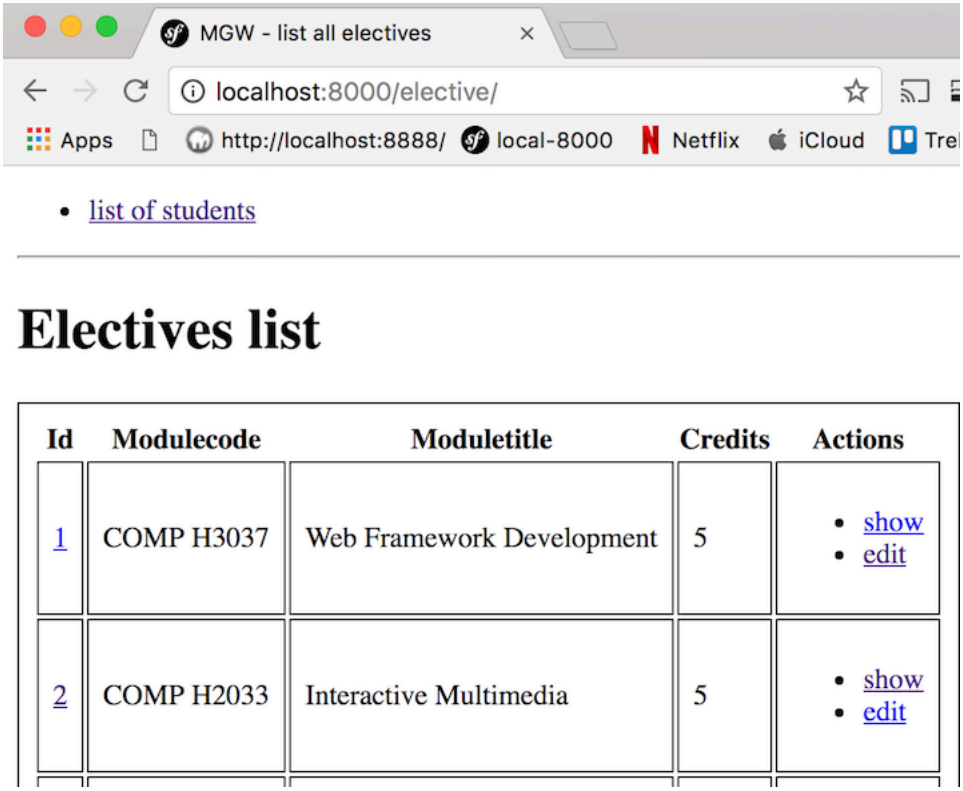


Figure 11.1: List of electives in HTML table.

11.4 The generated `newAction()` method

The method and Twig template for a new `Elective` work just as you might expect. A n empty form will be displayed and upon valida submission the user will be redirected to the `show` action form for the newly created entity.

```
/**
 * Creates a new elective entity.
 *
 * @Route("/new", name="elective_new")
 * @Method({"GET", "POST"})
 */
public function newAction(Request $request)
{
    $elective = new Elective();
    $form = $this->createForm('AppBundle\Form\ElectiveType', $elective);
    $form->handleRequest($request);

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

        return $this->redirectToRoute('elective_show', array('id' => $elective->getId()))
    }

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

11.5 The generated `showAction()` method

Initially, the generated 'show' method looks fine as jst as we might write:

```
/**
 * Finds and displays a elective entity.
 *
 * @Route("/{id}", name="elective_show")
```

```
     * @Method("GET")
     */
    public function showAction(Elective $elective)
    {
        $deleteForm = $this->createDeleteForm($elective);

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

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

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

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

-

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

The 'edit' and 'delete' generated methods are as you might expect. The show method creates a form, and also include code to process valid submission of the edited entity. Note that it redirects to itself upon successful save of edits.

```
    /**
     * Displays a form to edit an existing elective entity.
     *
     * @Route("/{id}/edit", name="elective_edit")
     * @Method({"GET", "POST"})
     */
    public function editAction(Request $request, Elective $elective)
    {
        $deleteForm = $this->createDeleteForm($elective);
        $editForm = $this->createForm('AppBundle\Form\ElectiveType', $elective);
```

```
        $editForm->handleRequest($request);

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

            return $this->redirectToRoute('elective_edit', array('id' => $elective->getId()));
        }

        return $this->render('elective/edit.html.twig', array(
            'elective' => $elective,
            'edit_form' => $editForm->createView(),
            'delete_form' => $deleteForm->createView(),
        ));
    }
```

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

```
    /**
     * Deletes a elective entity.
     *
     * @Route("/{id}", name="elective_delete")
     * @Method("DELETE")
     */
    public function deleteAction(Request $request, Elective $elective)
    {
        $form = $this->createDeleteForm($elective);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            $em = $this->getDoctrine()->getManager();
            $em->remove($elective);
            $em->flush($elective);
        }

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

11.7 The generated method `createDeleteForm()`

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

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

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

```
<form name="form" method="post" action="/elective/3">
    <input type="hidden" name="_method" value="DELETE" />
    <input type="submit" value="Delete">
    <input type="hidden" id="form__token" name="form[_token]" value="YayBB5j6Yjiyps-c6MJRxn8vHBjO-l
</form>
```

Part V

Sessions

12

Introduction to Symfony sessions

12.1 Remembering foreground/background colours in the session (`project12`)

Let's start out Symfony sessions learning with the ability to store (and remember) foreground and background colours[1]. First let's add some HTML in our `index.html.twig` page to display the value of our 2 stored values.

We will assume we have 2 Twig variables:

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

```
colours = [
    'foreground' => 'blue',
    'background' => 'pink'
]
```

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

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

```
<p>
    using default colours = {{ default_colours }}
```

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

```
</p>
<ul>
    {% for property, colour in colours %}
        <li>
        {{ property }} = {{ colour }}
        </li>
    {% endfor %}
</ul>
```

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

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

12.2 Twig default values (in case nothing in the session)

Let's write some Twig code to attempt to read the `colours` array from the SESSION, but failing that, then setting default values into Twig variable `colours`.

First we assume we'll get a value from the session (so we set `default_colours` to `no`), and we attempt to read the session variable array `colours` and store it in Twig variable `colours`. To read a value from the Symfony `app` variable's `session` property we write a Twig expressison in the form `app.session.get('<attribute_key>')`:

```
{% set default_colours = 'no' %}

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

Now we test whether or not `colours` is NULL (i.e. we could not read anything in the session for the given key). We test if a variable is `null` with Twig expression `if <variable> is null`:

```
{% if colours is null %}
    {% set default_colours = 'yes' %}

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

As we can see, if `colours` was NULL then we set `default_colors` to `yes`, and we use Twig's JSON-like format for setting key-value pairs in an array.

12.3 Working with sessions in Symfony Controller methods

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

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

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

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

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

```
session_start();
session_destroy();
```

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

Note: You may wish to read about **how to start a session in Symfony**[2].

12.4 Symfony's 2 session 'bags'

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

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

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

```
$session->set('<key>', <value>);
```

Here's how we store our colours array in the Symfony application session from our controllers:

```
// create colours array
$colours = [
    'foreground' => 'blue',
    'background' => 'pink'
];

// store colours in session 'colours'
```

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

```
$session = new Session();
$session->set('colours', $colours);
```

We can clear everything in a session by writing:

```
$session = new Session();
$session->remove('electives');

$session->clear();
```

12.5 Storing values in the session in a controller action

We'll add code to store colours in the session to our `DefaultController->indexAction()` method (i.e. the website home page controller):

```
public function indexAction(Request $request)
{
    // create colours array
    $colours = [
        'foreground' => 'blue',
        'background' => 'pink'
    ];

    // store colours in session 'colours'
    $session = new Session();
    $session->set('colours', $colours);

    $argsArray =  [
        'name' => 'matt'
    ];

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

Figure 12.1 shows the output of the colours from the session array when visiting the website home-page.

Learn more at about Symfony sessions at:

- Symfony and sessions

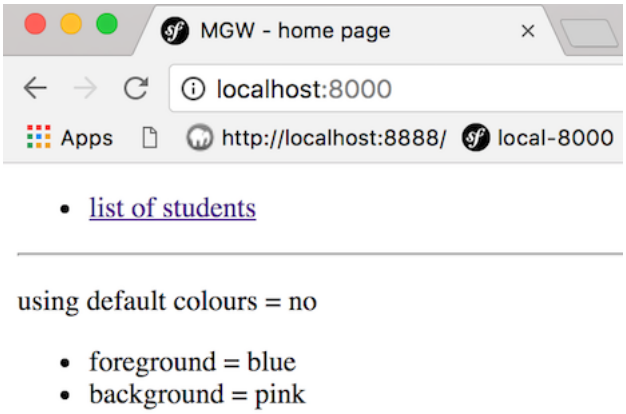


Figure 12.1: Homepage showing colours from session array.

12.6 Getting the colours into the HTML head `<style>` element (`project13`)

Since we have an array of colours, let's finish this task logically by moving our code into `_base.html.twig` and creating some CSS to actually set the foreground and background colours using these values.

So we remove the Twig code from template `index.html.twig` and paste it, slighly edited, into `_base.html.twig` as follows.

Add the following **before** we start the HTML doctype etc.

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

{# default = blue #}
{% if colours is null %}
    {% set colours = {
        'foreground': 'black',
        'background': 'while'
       }
    %}
{% endif %}
```

So now we know we have our Twig variable `colours` assigned values (either from the session, or from the defaults. Now we can update the `<head>` of our HTML to include a new `body {}` CSS rule, that pastes in the values of our Twig array `colours['foreground']` and `colours['background']`:

```
<!DOCTYPE html>
<html>
<head>
```

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

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

            body {
                color: {{ colours['foreground'] }};
                background-color: {{ colours['background'] }};
            }
        </style>
    </head>
```

Figure 12.2 shows our text and background colours applied to the CSS of the website homepage.

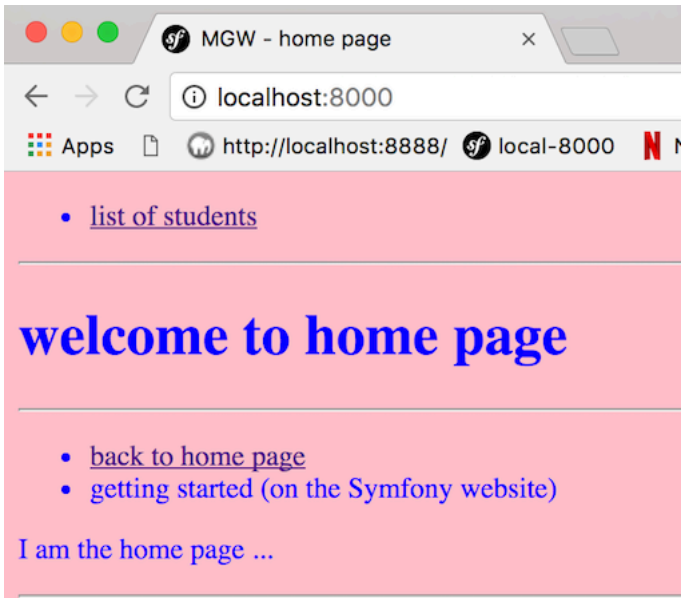


Figure 12.2: Homepage with session colours applied via CSS.

12.7 Testing whether an attribute is present in the current session

Before we work with a session attribute in PHP, we may wish to test whether it is present. We can test for the existance of an attribute in the session bag as follows:

```
if($session->has('<key>')){
     //do something
 }
```

12.8 Removing an item from the session attribute bag

To remove an item from the session attribute bag write the following:

```
$session->remove('<key>');
```

12.9 Clearing all items in the session attribute bag

To remove all items from the session attribute bag write the following:

```
$session->clear();
```

13

Working with a session 'basket' of electives

13.1 Shopping cart session attribute bag example (`project14`)

When you're leaning sessions, you need to build a 'shopping cart'! Let's imagine our students can select several subject elective 'modules', and store them in a 'basket' of electives.

We've created an `Elective` entity, and its CRUD controller and templates. So now let's add the 'shopping basket' functionality to add elective modules into a session basket.

We will have an `basket` item in the session, containing an array of `Elective` objects adding the the basket. This array will be indexed by the `id` property of each Elective (so we won't add the same module twice to the array), and items are easy to remove by unsetting.

13.2 Debugging sessions in Twig

As well as the Symfony profiler, there is also the powerful Twig functiond `dump()`. This can be used to interrogate values in the session.

You can either dump **every** variable that Twig can see, with `dump()`. This will list arguments passed to Twig by the controller, plus the `app` variable, containing sesison data and other applicaiton object properties. Or you can be more specific, and dump just a particular object or variable. For example we'll be building an attribute stack session array named `basket`, and the contents of this array can be dumped in Twig with the following statement:

```
{{ dump(app.session.get('basket')) }}
```

Figure 13.1 shows our `basket[]` array in the session `Attribute Bag`, navigating through the Twig `dump()` output as follows:

```
app> requestStack> requests[0]> session> storage> bags> attributes> basket[]
```

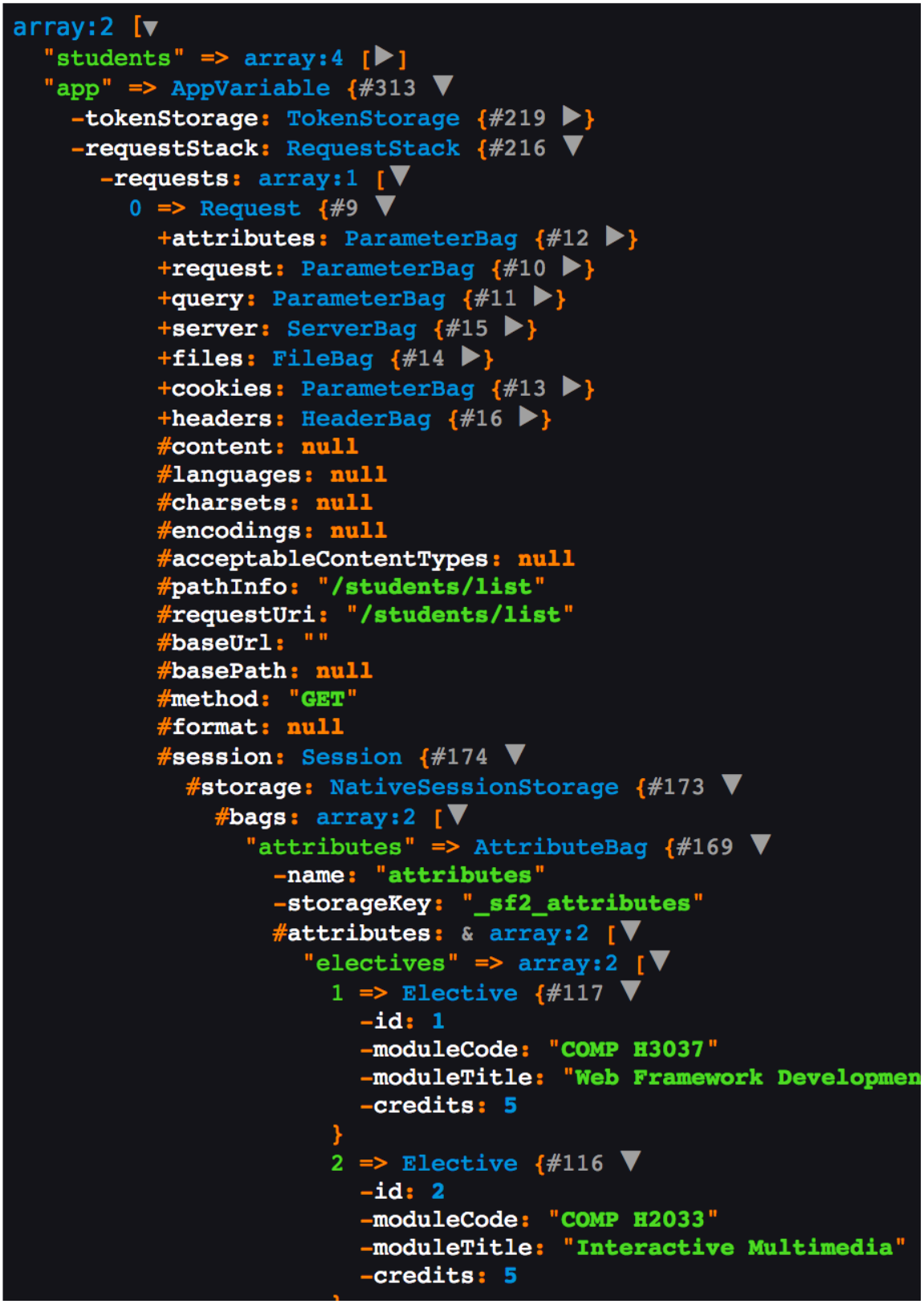


Figure 13.1: Twig dump of session attribute bag.

13.3 Basket index route, to list contents of electives basket

We'll write our code in a new controller class `ElectiveBasketCobtroller.php` in directory `/src/AppBundle/Controller/`. Note that we have added the `@Route` prefix `/basket/` to all controller actions in this class by writing a `@Route` annotation comment for the class declaration:

```
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use AppBundle\Entity\Elective;
use Symfony\Component\HttpFoundation\Session\Session;

/**
 * Elective controller.
 *
 * @Route("/basket")
 */
class ElectiveBasketController extends Controller
{
```

Our electives basket controller index action is very simple, since all the work extacting values from the session will be done by our Twig template. So our index action simply returns the Twig rendering of template `basket/index.html.twig`:

```
    /**
     * @Route("/", name="electives_basket_index")
     */
    public function indexAction()
    {
        // no need to put electives array in Twig argument array - Twig can get data direct from
        $argsArray = [
        ];

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

13.4 Controller method - `clearAction()`

Let's write another simple method next - a method to remove any `basket` attribute from the session. We can achieve this with the statement `$session->remove('basket')`:

```
/**
 * @Route("/clear", name="electives_basket_clear")
 */
public function clearAction()
{
    $session = new Session();
    $session->remove('basket');

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

Note that we are redirecting to route `electives_basket_index`.

13.5 Adding an Elective object to the basket

The logic to add an object into our session `basket` array requires a little work. First we need to get a PHP array `$electives`, that is either what is currently in the session, or a new empty array if no such array was found in the session:

```
/**
 * @Route("/add/{id}", name="electives_basket_add")
 */
public function addToElectiveCart(Elective $elective)
{
    // default - new empty array
    $electives = [];

    // if no 'electives' array in the session, add an empty array
    $session = new Session();
    if($session->has('basket')){
        $electives = $session->get('basket');
    }
```

Note above, that we are relying on the 'magic' of the Symfony param-converter here, so that the integer 'id' received in the request is converted into its corresponding Elective object for us.

Next we get the 'id' of the Elective object, and see whether it can be found already in array `$electives`. If if is not already in the array, then we add it to the array (with the 'id' as key), and store the updated array in the session under the attribute bag key `basket`:

```
    // get ID of elective
    $id = $elective->getId();
```

```
        // only try to add to array if not already in the array
        if(!array_key_exists($id, $electives)){
            // append $elective to our list
            $electives[$id] = $elective;

            // store updated array back into the session
            $session->set('basket', $electives);
        }
```

Finally (whether we changed the session `basket` or not), we redirect to the basket index route:

```
    return $this->redirectToRoute('electives_basket_index');
```

13.6 The delete action method

The delete action method is very similar to the add action method. In this case we never need the whole Elective object, so we can keep the integer `id` as the parameter for the method.

We start (as for add) by ensuring we have a PHP variable array `$electives`, whether or not one was found in the session.

```
    /**
     * @Route("/delete/{id}", name="electives_basket_delete")
     */
    public function deleteAction(int $id)
    {
        // default - new empty array
        $electives = [];

        // if no 'electives' array in the session, add an empty array
        $session = new Session();
        if($session->has('basket')){
            $electives = $session->get('basket');
        }
```

Next we see whether an item in this array can be found with the key `$id`. If it can, we remove it with `unset` and store the updated array in the session attribute bag with key `basket`.

```
        // only try to remove if it's in the array
        if(array_key_exists($id, $electives)){
            // remove entry with $id
            unset($electives[$id]);
```

```
            if(sizeof($electives) < 1){
                return $this->redirectToRoute('electives_basket_clear');
            }

            // store updated array back into the session
            $session->set('basket', $electives);
        }
```

Finally (whether we changed the session `basket` or not), we redirect to the basket index route:

```
return $this->redirectToRoute('electives_basket_index');
```

13.7 The Twig template for the basket index action

The work extacting the array of electives in the basket and displaying them is the task of template `index.html.twig` in `/app/Resources/views/basket`.

First, we attempt to retrieve item `basket` from the session, and also Twig `dump()` this session attribute:

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

{{ dump(app.session.get('basket')) }}
```

Next we have a Twig `if` statement, displaying an empty basket message if `basket_electives` is null (i.e.

```
{% if basket_electives is null %}
    <p>
        you have no electives in your basket
    </p>
```

The we have an `else` statement (for when we did retrieve an array), that loops through creating an unordered HTML list of the basket items:

```
{% else %}
    <ul>
        {% for elective in basket_electives %}
            <li>
                <hr>
                id = {{ elective.id }}
                <br>
                module code = {{ elective.moduleCode }}
                <br>
```

```
                module title = {{ elective.moduleTitle }}
                <br>
                credits = {{ elective.credits }}
                <br>
                <a href="{{ path('electives_basket_delete', { 'id': elective.id }) }}">(remo
            </li>
        {% endfor %}
    </ul>
{% endif %}
```

Note that a link to the `delete` action is offered at the end of each list item.

Finally, a paragraph is offered, containing a list to clear all items from the basket:

```
<p>
    <a href="{{ path('electives_basket_clear') }}">CLEAR all items in basket</a>
</p>
```

Figure 13.2 shows a screenshot of the basket index page, listing each item in the session array.

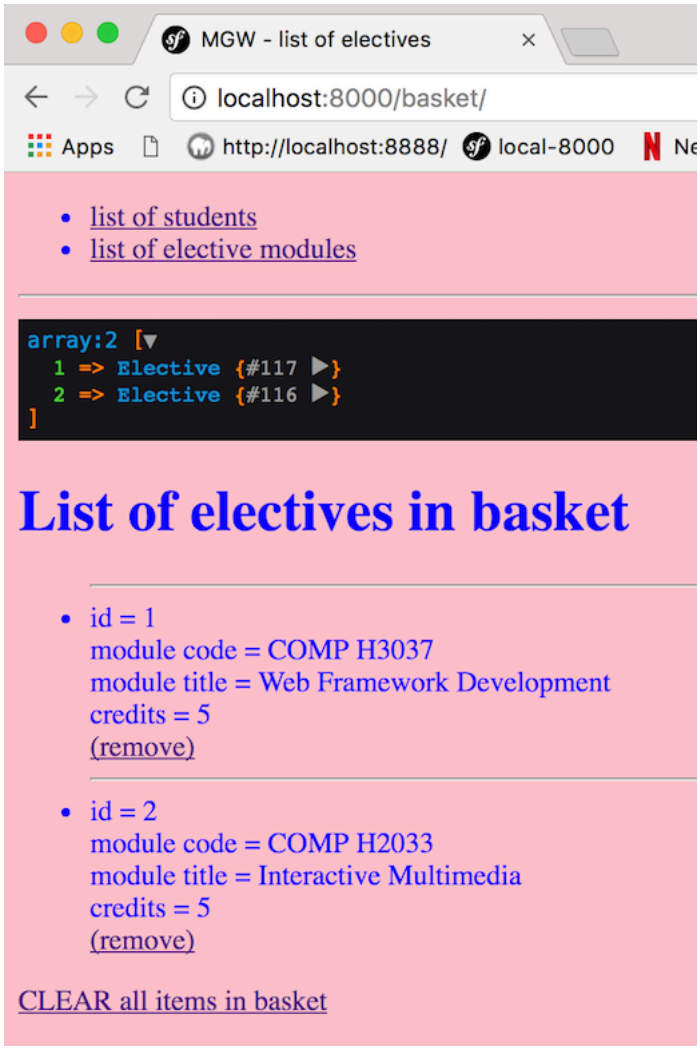


Figure 13.2: Shopping basket of elective modules.

13.8 Adding the 'add to basket' link in the list of electives

To link everything together, we can now add a link to 'add to basket' in our electives index template. So when we see a list of electives we can add one to the basket, and then be redirected to see the updated basket of elective modules. We see below an extra list item for path `electives_basket_add` in template `index.html.twig` in directory `/app/Resources/views/elective/`:

```
{% for elective in electives %}
    <tr>
        <td><a href="{{ path('elective_show', { 'id': elective.id }) }}">{{ elective.id }}</
        <td>{{ elective.moduleCode }}</td>
        <td>{{ elective.moduleTitle }}</td>
        <td>{{ elective.credits }}</td>
        <td>
            <ul>
                <li>
                    <a href="{{ path('elective_show', { 'id': elective.id }) }}">show</a>
                </li>
                <li>
                    <a href="{{ path('elective_edit', { 'id': elective.id }) }}">edit</a>
                </li>
                <li>
                    <a href="{{ path('electives_basket_add', { 'id': elective.id }) }}">add
                </li>
            </ul>
        </td>
    </tr>
{% endfor %}
```

Figure 13.3 shows a screenshot of the list of elective modules page, each with an 'add to basket' link.

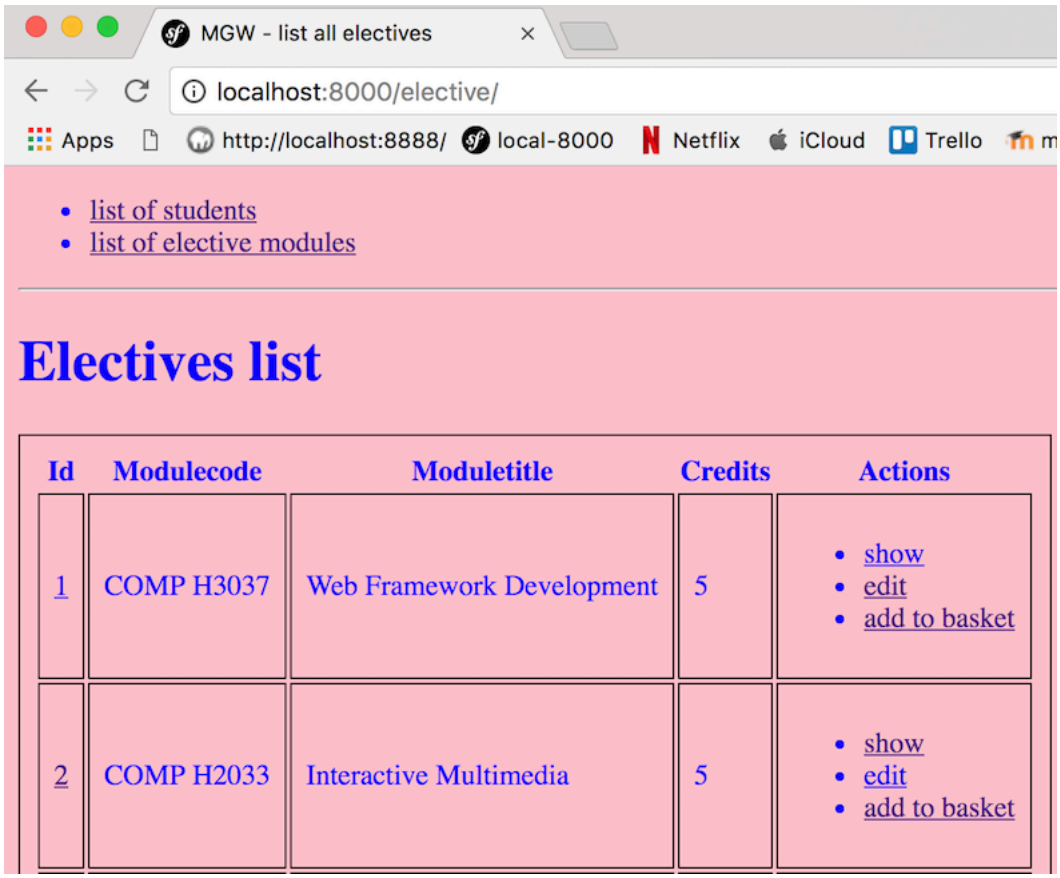


Figure 13.3: List of electives with 'add to basket' link.

Part VI

Security and Authentication

14

Simple authentication (logins!) with Symfony sessions

14.1 Create a `User` entity (`project15`)

Let's use the CLI to generate a `User` entity for us. We'll use the `--no-interaction` option and specify 2 string fields (each with length 255) for `username` and `password`:

```
php bin/console generate:doctrine:entity --no-interaction --entity=AppBundle:User
--fields="username:string(255) password:string(255)"
```

For now we won't worry about hashing the password - we'll learn how to do that later.

14.2 Create Database table for our entity

Now let's use the CLI to update our Database schema and create a table corresponding to our new entity:

```
$ php bin/console doctrine:schema:update --force
```

14.3 Create `User` CRUD from CLI

Now let's create a CRUD controller for users:

```
php bin/console generate:doctrine:crud --entity=AppBundle:User --format=annotation
--with-write --no-interaction
```

We now have a new controller class `UserController`, and also new view templates[1]:

```
/app/Resources/views/user/edit.html.twig
/app/Resources/views/user/index.html.twig
/app/Resources/views/user/new.html.twig
/app/Resources/views/user/show.html.twig
```

14.4 New routes (from annotations of controller methods)

Let's look at the new routes added by our generated CRUD controller. We can do this two ways:

- from the CLI command `php bin/console debug:router`
- selecting 'Routes' from the Symfony profiler page

Figure 14.1 shows a screenshot the Symfony of the profiler page listing all routes (hint - enter an invalid route and it will list them all, e.g. `/user99`).

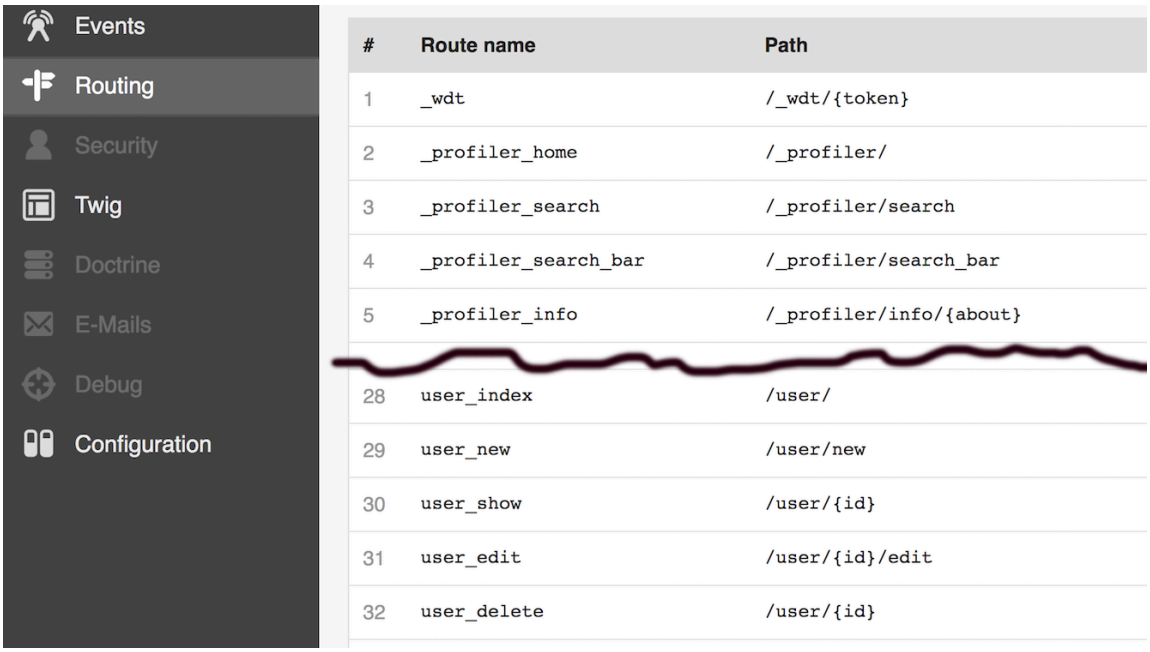


Figure 14.1: List of CRUD-generated user routes.

We can see that these automatically generated routes are very 'succinct' (using as few words as possible). The sequence is important, also the HTTP methods (or simulated methods like `DELETE`).

[1] If you use `_base.html.twig` you'll have to edit the `extends` statement for each of these templates, since `base.html.twig` is assumed and automatically coded.

14.5 WARNING - watch our for 'verbs' being interpreted as entity 'id's ...

Imagine we write a new method, `loginAction()` at the **end** of our `UserController`, with the route annotation `/user/login`. When requested with the HTTP `GET` method, the show route `/user/{id}` will match before it gets down to the `/user/login` route pattern. The Symfony param-converted will then attempt to retreive a `User` record from the database with an 'id' value of `login`, and will fail. This will result in the param-converted throwing a 404-not found exception.

Figure 14.2 shows a screenshot of the exception thrown.

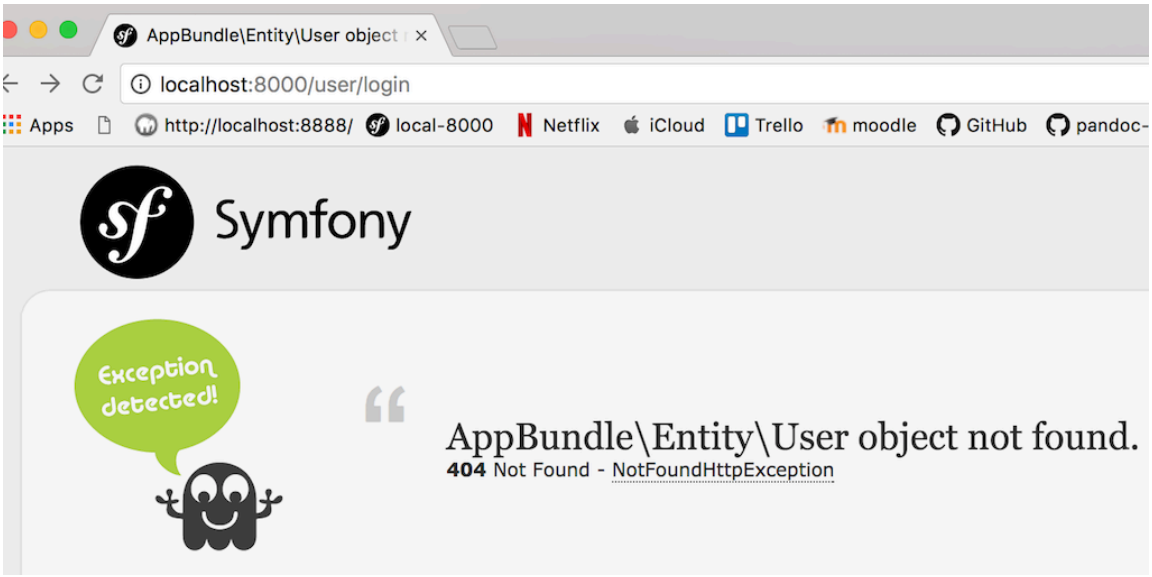


Figure 14.2: 404 not found exception for `/user/login`.

If we look in the profiler it will tell us which route it matched with. Figure 14.3 shows the profiler telling us it matched route `/user/{id}`.

25	students_show	/students/show/{id}	Path does not match
26	students_process_new_form	/students/processNewForm	Path does not match
27	students_new_form	/students/new	Path does not match
28	user_index	/user/	Path does not match
29	user_new	/user/new	Path does not match
30	user_show	/user/{id}	Route matches!

Figure 14.3: Profiler showing route matches `/user/{id}`.

We can solve this problem in several different ways. Let's solve it by creating a separate

`LoginController` class, with routes `/login` and `/logout`. Since these routes will not be prefixed by `/user` neither word 'login' or 'logout' will be interpreted as an 'id' for a user. Other solutions include:

- locating the `loginAction()` method, and its associated route, earlier in the `UserController` than the `showAction()`. Although the less we have to rely on the **sequence** of methods in a class, the less chance we'll encounter issues like this.
- adding a 'verb' for each action's route. So the `showAction()` method will have route `/user/show/{id}` and the `deleteAction()` method will have route `/user/delete/{id}`, and so on. This is why the `editAction()` route ends with `/edit` (although putting the verb after the parameter seems odd to me …).

14.6 Create a 'login' Twig template (`project16`)

Before we create the `LoginController` PHP class let's first create the Twig template to display the login form.

Let's just copy the `/user/new.html.twig` template (in directory `/app/Resources/views`) created with our CRUD - since a login (just as with new user) needs a form asking for 'username' and 'password'. We'll copy that to `/login.html.twig` (in the root views directory of `/app/Resources/views`). We'll change the level 1 heading and button label to `Login`, and remove the `Back to the list` link: , and change the name of the submit button to 'l

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

{% block body %}
    <h1>Login</h1>
    {{ form_start(form) }}
        {{ form_widget(form) }}
        <input type="submit" value="Login" />
    {{ form_end(form) }}
{% endblock %}
```

As we can see above, this Twig template is now basically a level 1 HTML heading `Login`, the start and end form tags (via Twig functions `form_start` and `form_end`), and then the form widget (input fields and labels etc.), plus a regular `Login` submit button.

Since we can anticipate that we may wish to display flash login error messages to the user, we'll add a `<div>` with CSS class `flash-error` (pink background and some padding) after the level 1 heading:

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

```
{% block body %}
    <h1>Login</h1>

    {% if app.session.flashBag.has('error') %}
        <div class="flash-error">
            {% for msg in app.session.flashBag.get('error') %}
                {{ msg }}
            {% endfor %}
        </div>
    {% endif %}

    {{ form_start(form) }}
        {{ form_widget(form) }}
        <input type="submit" value="Login" />
    {{ form_end(form) }}
{% endblock %}
```

14.7 A loginAction() in a new SecurityController

Now we'll create a new controller class to handle login/logout/authentication etc. In directory `/src/AppBundle/Controllers` create new class `SecurityController`. We can base method `loginAction()` for route `/login` on a copy of method `UserController->newAction()`.

We need to do the following:

- change the route annotation comment to `@Route("/login", name="login")`
- change method name to `loginAction()`
- for now just delete all the statements inside the `if` statement for a succsfully submitted form (so after submission of the form, we just see the form again - note the form is 'sticky' since the `$user` object is rem
- the name of the Twig template is simply `login`

```
/**
 * login form
 *
 * @Route("/login", name="login")
 * @Method({"GET", "POST"})
 */
public function loginAction(Request $request)
{
    $user = new User();
    $form = $this->createForm('AppBundle\Form\UserType', $user);
```

```
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
        }

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

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

14.8 Problem - the Symfony User form renders password as visible plain text

While we saved a little time and energy re-using the new User form for our login form, we can see from the screenshot in Figure 14.4 that the password field is rendered in HTML as visible plain text.

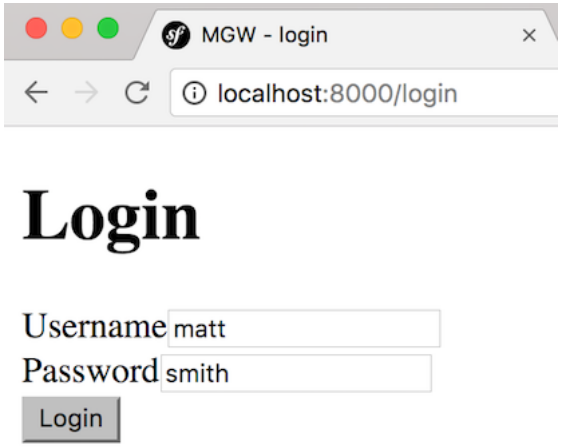


Figure 14.4: Login form with visible plain text password HTML form field.

This is because the default `UserType` form, that was created as part of the CRUD generation, saw that `password` was a text field in the Entity `User`, so by default generates a plain text HTML input field.

** Huh?? the `UserType` form?? **

Yes, part of the CRUD generation also involves creating a class for each entity's Form. So in `/src/AppBundle/Form` the `UserType` form class that was created. If we look carefuly at the code we copied from `UserController->newAction()` we see that to create the form from a `User` object we are Symfony to use class `AppBundle\Form\UserType`:

```
$form = $this->createForm('AppBundle\Form\UserType', $user);
```

We can change this by specifying that we want any forms displaying the `User` passowrd field to be rendered using the `PasswordType` Symfony form type. We just have to add this in to the `UserType` form class that was created in `/src/AppBundle/Form/UserType.php`:

```
class UserType extends AbstractType
{
    /**
     * {@inheritdoc}
     */
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('username')->add('password');
    }
```

We need to add `PasswordType::class` to the part where the 'password field is added to the form:

```
class UserType extends AbstractType
{
    /**
     * {@inheritdoc}
     */
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('username')->add('password', PasswordType::class);
    }
```

We also need to add the corresponding `use` statement so that this class knows about the `PasswordType` class we are using:

```
use Symfony\Component\Form\Extension\Core\Type\PasswordType;
```

Figure 14.5 shows a wildcarded password HTML form field now.

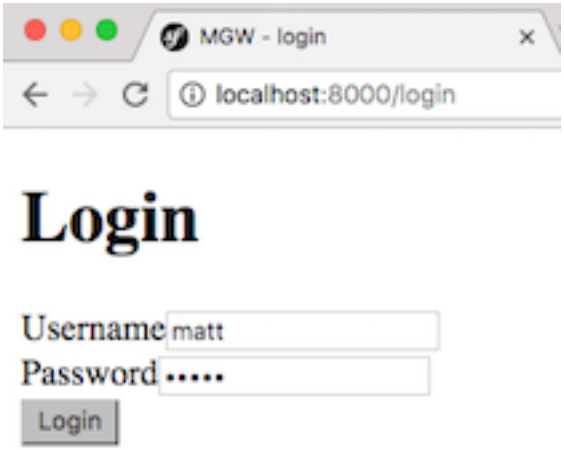


Figure 14.5: Login form with obscured wildcard password HTML form field.

14.9 Handling login form submission

We can now begin our work on handling the POST submission of login details. Let us abstract away the task of authentication to a method (we'll write in a minute) `authenticate()`. So we can now write the content of our `if(submitted and valid)` statement block to do the following:

- IF successful authentication for contents of `$user`
- THEN store `$user` in the session and redirect to a secure admin home page
- ELSE
  - add an error to the flash bag
  - clear the password field (login forms should not have 'sticky' passwords) and recreate the form with this updated user object
  - then fall through to display the form again

Here is this login implemented in our `loginAction()` method:

```
if ($form->isSubmitted() && $form->isValid()) {
    if($this->canAuthenticate($user)) {
        // store user in session
        $session->set('user', $user);

        // redirect to ADMIN home page
        return $this->redirectToRoute('admin_index');
    } else {
        $this->addFlash(
            'error',
```

```
                'bad username or password, please try again'
            );

            // create new form with user that has no password - password should not be 'sticky'
            $user->setPassword('');
            $form = $this->createForm('AppBundle\Form\UserType', $user);

            // fall through to login form at end of this method
        }
    }
```

14.10 An Admin home page (to test authentication)

Let's add the admin controller, with an action for an admin homepage (the route named `admin_index` which we redirect to after a valid login). Figure 14.6 shows this admin home page. At present we can visit this page with no login authentication with request URL `/admin/`.

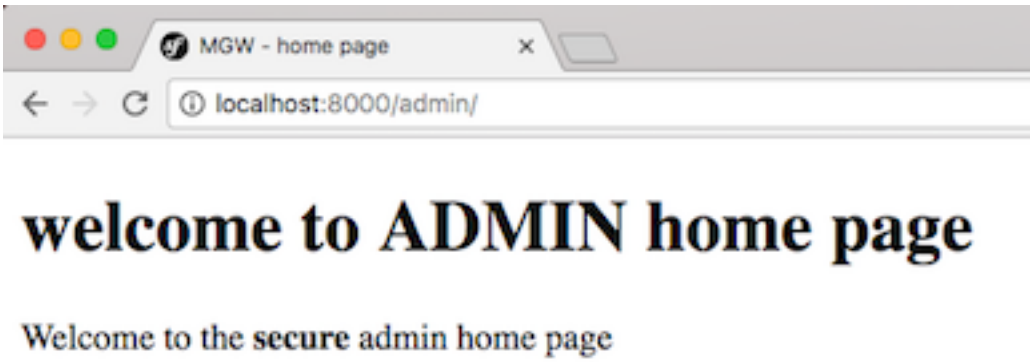


Figure 14.6: Unsecured admin home page.

```
/**
 * Class AdminController
 * @package AppBundle\Controller
 *
 * @Route("/admin")
 */
class AdminController extends Controller
{
    /**
     * @Route("/", name="admin_index")
     */
```

```
    public function indexAction(Request $request)
    {
        $templateName = '/admin/index';
        return $this->render($templateName . '.html.twig', []);
    }
}
```

NOTE: Why have route prefix for a class when there is only one route? Well, having a route prefix means Symfony resolves `/admin` with no trailing slash as `/admin/` with no complaining! Figure 14.7 shows how a trailing forward slash is automatically added to a request to `/admin`.

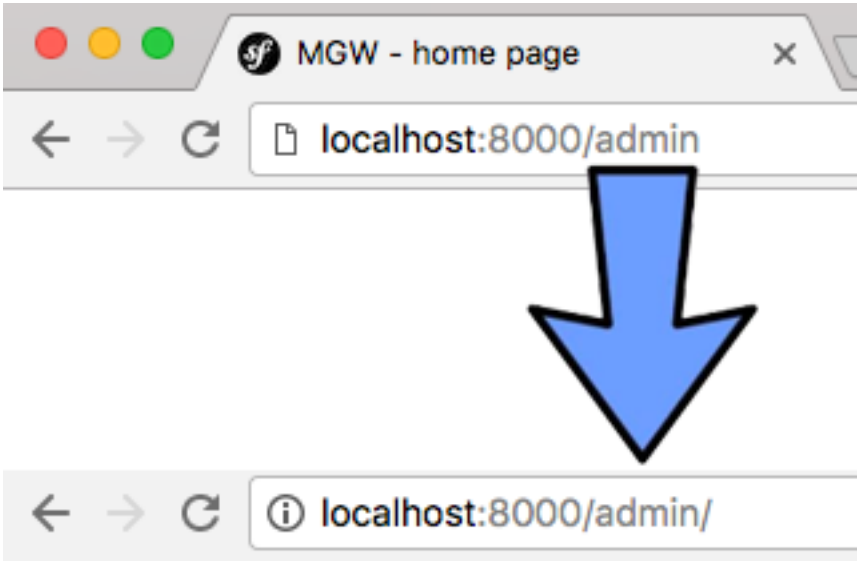


Figure 14.7: Symfony adding trailing slash to admin home page request.

Let's add a Twig template `app/Resources/views/admin/index.html.twig` for a simple admin home page:

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

{% block pageTitle %}home page{% endblock %}

{% block body %}
    <h1>welcome to ADMIN home page</h1>

    <p>
        Welcome to the <strong>secure</strong> admin home page
    </p>
{% endblock %}
```

14.11 Authenticating against hard-coded credentials and storing User object in the session

We can now complete our session-based security, by implementing our authenticate() method in the SecurityController, and storing a User object in the session after successful login. Here is the code for that mehtod (where we hardcode valid username and password 'admin'):

```
/**
 * @param User $user
 * @return bool
 *
 * return whether or not contents of $user is a valid username/password combination
 */
public function canAuthenticate(User $user)
{
    $username = $user->getUsername();
    $password = $user->getPassword();

    return ('admin' == $username) && ('admin' == $password);
}
```

We can now add SESSION logic to our `AdminController->indexAction()` method, testing for a `user` token in the SESSION before allowing display of the admin home page. We need to:

- get a reference to the current session
- test whether there is a token user in the current session (if yes, we can go ahead and render the admin home page)
- if no user token in the session, then we'll add a flash error to the session Flash bag, and redirecet to the login page

NOTE Due to the way redirects work in Symfony 3, flash messages live for 2 requests during a redirect, so we need to clear the flash bag before adding the message, otherwise we'll see the message twice … a bit odd but this approach seems to work …

```
public function indexAction(Request $request)
{
    $session = new Session();

    if ($session->has('user')){
        $templateName = '/admin/index';
        return $this->render($templateName . '.html.twig', []);
    }
```

```
        // if get here, not logged in,
        // empty flash bag and create flash login first message then redirect
        $session->getFlashBag()->clear(); // avoids seeing message twice ...
        $this->addFlash(
            'error',
            'please login before accessing admin'
        );

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

Figure 14.8 shows automatic redirection to the login page, when user attempts to view admin home page when not logged in.

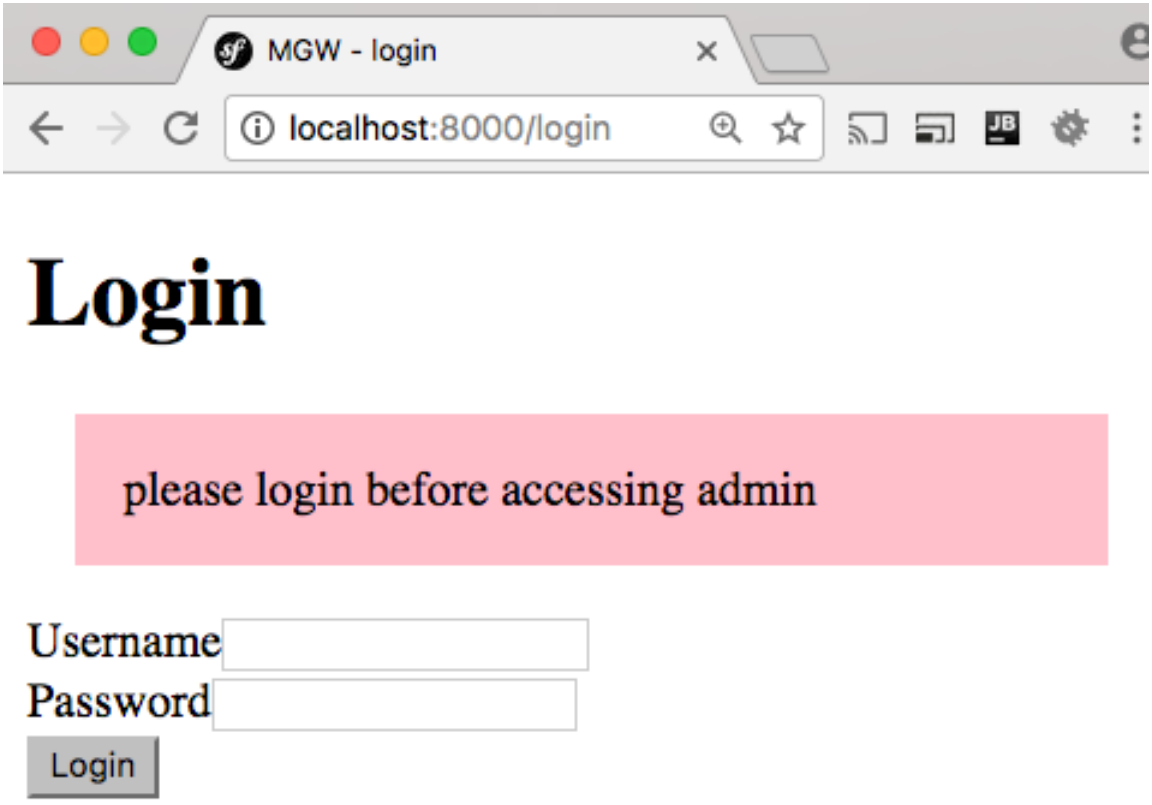


Figure 14.8: User redirected to login page after requesting `/admin`, with flash error (when not logged-in).

If we login with the credentials username=‘`admin`’ and password=‘`admin`’, we get to see the admin home page, and we can see, from the Symfony profiler, that a user object is stored in the session (See Figure 14.9).

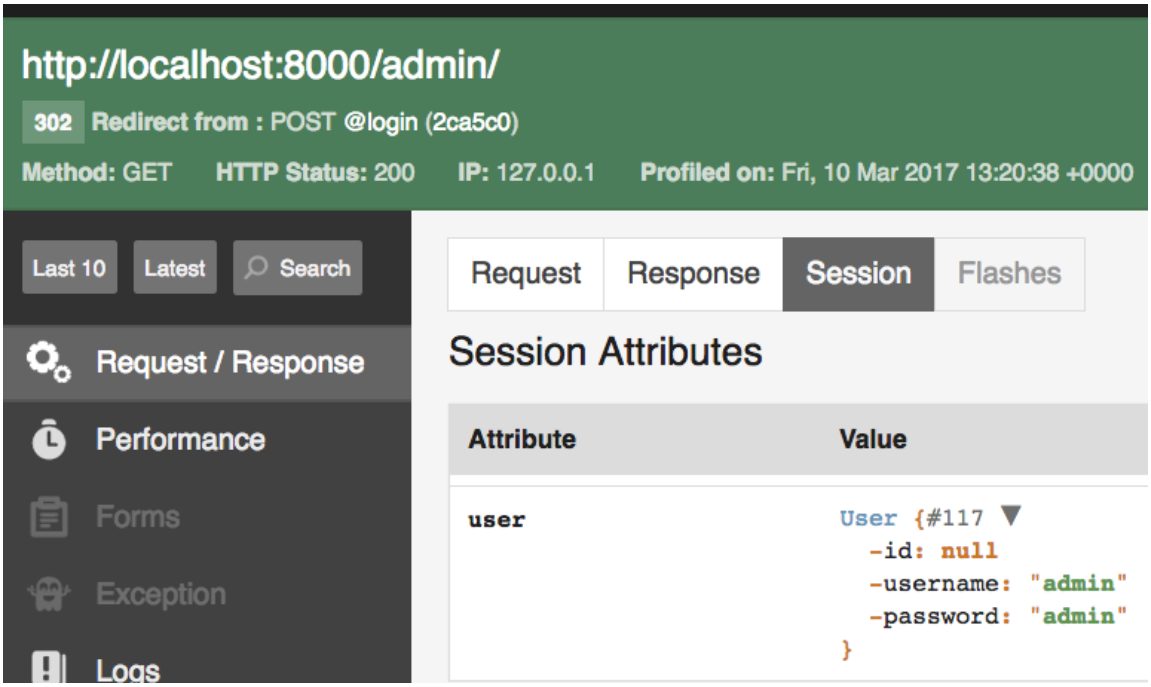


Figure 14.9: User token in session, after requesting admin home (when logged in).

14.12 Informing user if logged in

If the user is accessing the admin pages, let's inform them of the user they are logged-in as, and offer them a logout link. We can add some CSS for a page header in `/web/css/header.css` to show a grey shaded header with some padding and right aligned text (and add an import statement for this stylesheet to `_base.html.twig`):

```
header {
    text-align: right;
    padding: 0.5rem;
    border-bottom: 0.1rem solid black;
    background-color: darkgray;
}
```

We can also add some Twig logic to our `_base.html.twig` template to display (on every page) login detaila, and login/logout link as appropriate:

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

{% if user is null %}
    <p>
        you are not logged in: <a href="{{ path('login') }}">login</a>
    </p>
```

```
{% else %}
    <header>
        You are logged in as: {{ user.username }}
        <br>
        <a href="{{ path('logout') }}">logout</a>
    </header>
{% endif %}
```

Figure 14.10 shows automatic redirection to the login page, when user attempts to view admin home page when not logged in.

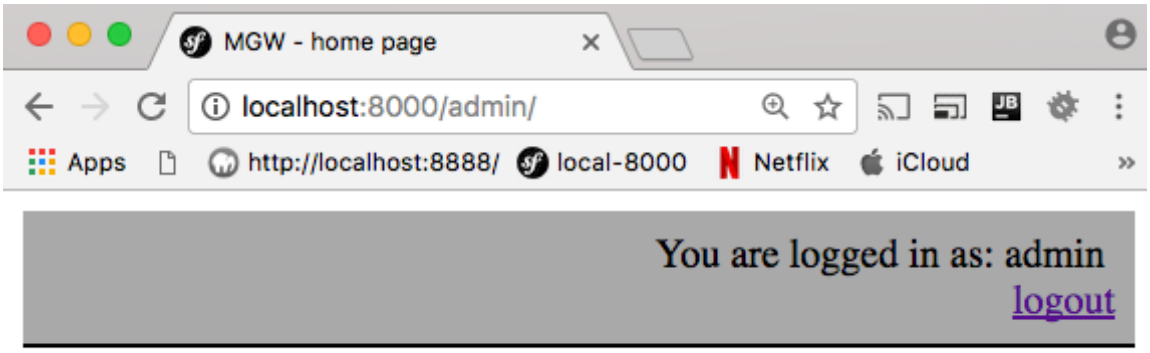


Figure 14.10: Page header with CSS, username and logout link.

14.13 Working with different user roles

Often we need to identify **which** kind of user has logged in. This can be done by extending our User entity to have a 'role' property. Either make this an integer (foreign key to a `Role` Entity), or just have text values. Symfony's own security system follows the PHP constant naming convention of upper case, underscore separated names for roles, such as:

- `ROLE_USER`
- `ROLE_ADMIN`
- `ROLE_MODERATOR`
- etc.

So I suggest you follow this. The steps you'd need to take would include:

1. update the `User` entity to have a string 'role' property
2. regenerate the getters and setters
3. regenerate the CRUD (and Form)
4. update the form, so that passwords are rendered as password fields
5. edit your secure page controller methods to check for user roles (e.g. admin home page may require `ROLE_ADMIN` in the user object in the session)

14.14 Moving on ... the Symfony security system

Rather than this D.I.Y. (Do-It-Yourself) approach to security with sessions, it may be wise to move forward and learn about Symfony's powerful security component:

- The Symfony Security system

15

Introduction to Symfony security features

15.1 Create a new `blog` project (`project17`)

Create a brand new project named `blog` (or whatever you want). See Appendix B for a quick list of actions to create a new Symfony project.

15.2 Adding an unsecured admin home page

Now let's create an `AdminController` in `/src/AppBuundle/Controller`, with an index route for route `/admin`. At some later point you add a nice Twig template to create the page, but for now we'll settle for a one-line hardcoded `Response` of `<html><body>Admin page!</body></html>`:

```
namespace AppBundle\Controller;

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

/**
 * @Route("/admin")
 */
class AdminController extends Controller
```

```
{
    /**
     * @Route("/", name="admin_index")
     */
    public function adminAction()
    {
        return new Response('<html><body>Admin page!</body></html>');
    }
}
```

As we can see in Figure 15.1, at present this is unsecured and we can access it in our browser via URL.

```
http://localhost:8000/app_dev.php/admin
```

We can see the route is unsecured by looking at the user information from the Symfony debug bar when visiting the default home page.

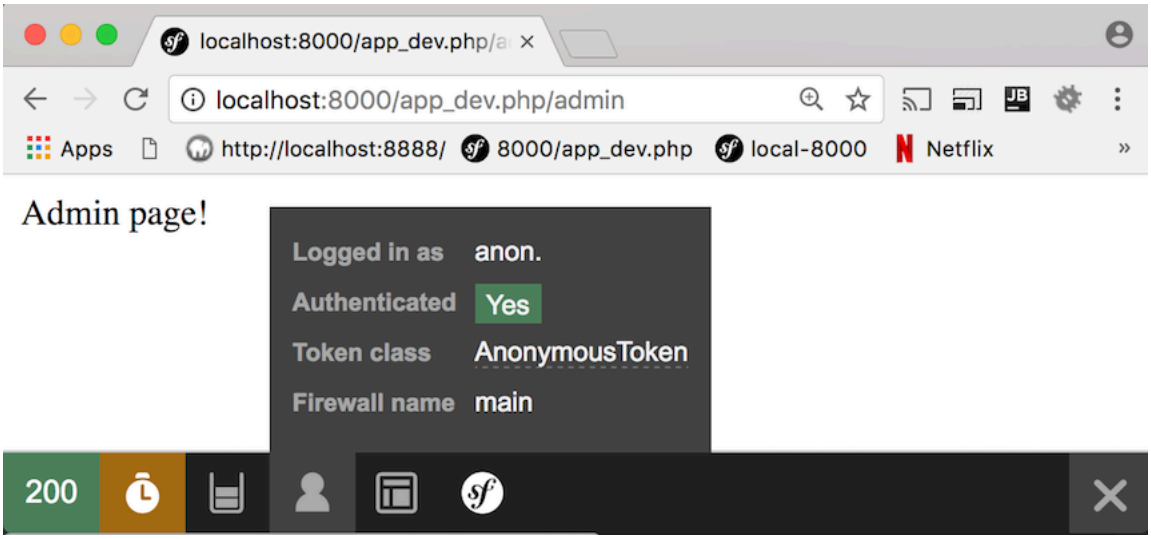


Figure 15.1: Admin home page, with anonymous user access (access not secured).

15.3 Security a route with annotation comments

We are going to add a security annotation comment to declare that the admin index route is only permitted for users with `ROLE_ADMIN`. We need to add a `use` statement, so that the annotation comments for `@Security` are parsed correctly:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
```

We can now add an annotation comment in the same comment DOCBLOCK as the route annotation

(immediately before the controller method), requiring users to have `ROLE_ADMIN` to be permitted to access this route:

```
<?php
namespace AppBundle\Controller;

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

/**
 * @Route("/admin")
 */
class AdminController extends Controller
{
    /**
     * @Route("/admin")
     * @Security("has_role('ROLE_ADMIN')")
     */
    public function adminAction()
    {
        return new Response('<html><body>Admin page!</body></html>');
    }
}
```

Now if we try to access `http://localhost:8000/app_dev.php/admin` we'll see (as in Figure 15.2) an error stating that `full authentication is required to access this resource` - which is impressive since we have not yet defined any users or authentication methods for users!

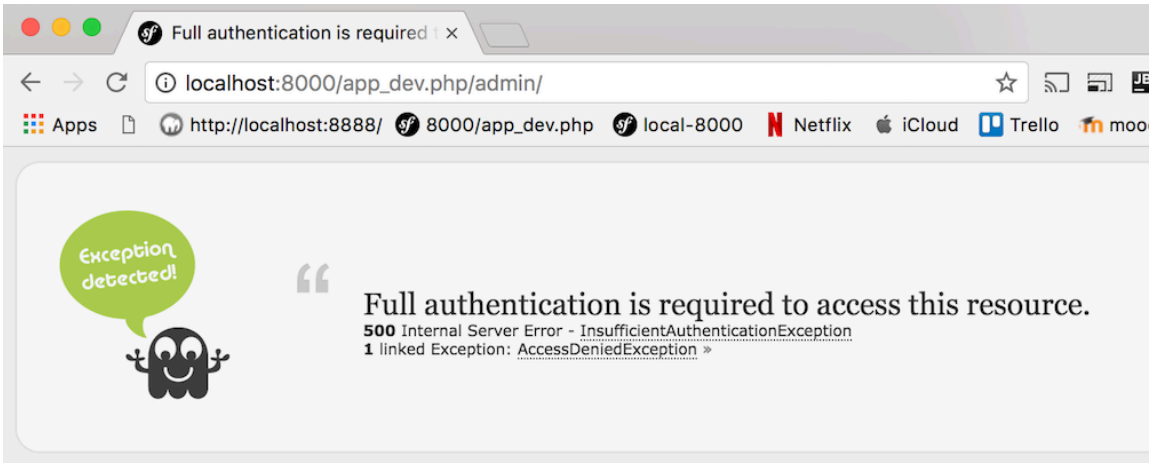


Figure 15.2: Not authenticated access denied for `/admin`.

15.4 Read some of the Symfony security documents

There are several key Symfony reference pages to read when starting with security. These include:

- Introduction to security
- How to build a traditional login form
- Using CSRF protection

15.5 Core features about Symfony security

There are several related features and files that need to be understood when using the Symnfony security system. These include:

- **firewalls**
- **providers** and **encoders**
- **route protection**
- user **roles**

Core to Symfony security are the **firewalls** defined in `app/config/security.yml`. Symfony firewalls declare how route patterns are protected (or not) by the security system. Here is its default contents (less comments - lines starting with hash `#` character):

```
security:
    providers:
        in_memory:
            memory: ~
```

```
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        main:
            anonymous: ~
```

Symfony considers **every** request to have been authenticated, so if no login action has taken place then the request is considered to have been authenticated to be **anonymous** user `anon`. We can see in this `anon` user in Figure 15.3 this looking at the user information from the Symfony debug bar when visiting the default home page.

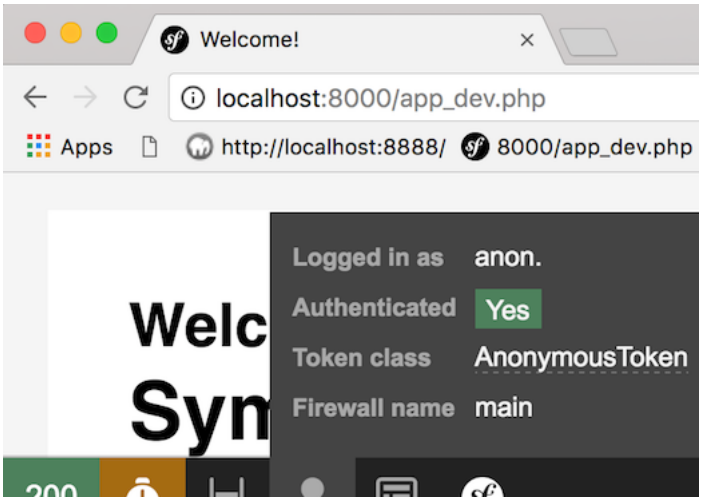


Figure 15.3: Symfony profiler showing anonymous user authentication.

A Symfony **provider** is where the security system can access a set of defined users of the web application. The default is simply `in_memory` - although usually larger applications have users in a database or from a separate API. We see that the `main` firewall simply states that users are permitted (at present) any request route pattern, and anonymous authenticated users (i.e. ones who have not logged in) are permitted.

NOTE In some Symfony documentation you'll see `default` instead of `main` for the default firewall. Both seem to work the same way (i.e. as the default firewall settings). So choose one and stick with it. Since my most recent new Symfony project called this `main` in the `security.yml` file I'll stick with that one for now …

The `dev` firewall allows Symfony development tools (like the profiler) to work without any authentication required. Leave it in `security.yml` and just ignore the `dev` firewall from this point onwards.

15.6 Using default browser basic HTTP authentication

Let's tell Symfony to use the web browser's built-in HTTP login form to handle username/password input for us (we'll add a custom login form later). We do this by adding a line[1] at the end of our `security.yml` file, stating that authentication will be via the `http_basic` method:

```
security:
    providers:
        in_memory:
            memory: ~

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        main:
            anonymous: ~
            http_basic: ~
```

If we try to access `http://localhost:8000/app_dev.php/admin` again we'll see the browser default username/password login form, as shown in Figure 15.4.

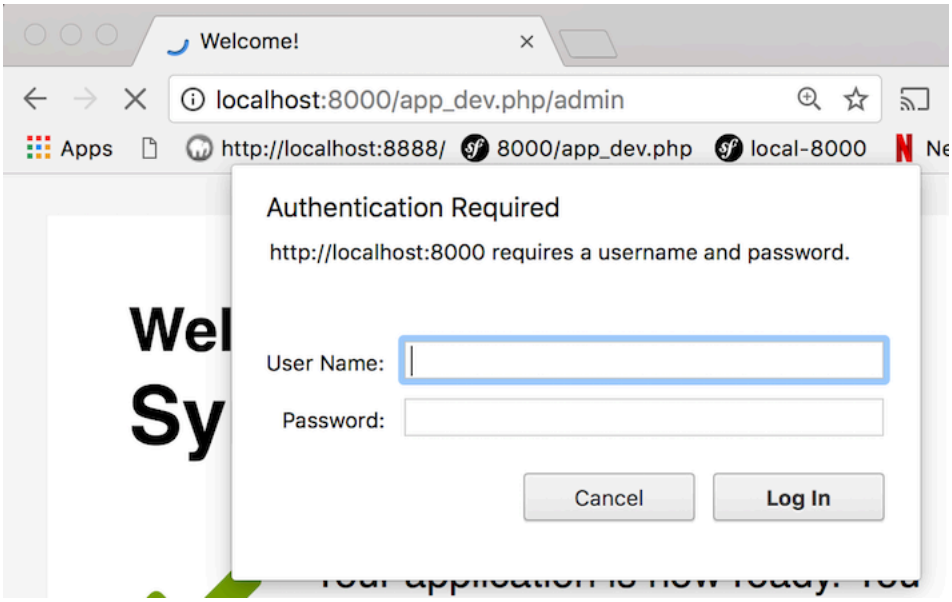


Figure 15.4: Admin home page, requiring HTTP basic browser login.

[1] In fact we can simply uncomment the provided line - just remove the has # symbol

15.7 Defining some users and their roles

Let's hard-code some users. Symfony looks at the **user providers** for where to find users and their credentials. We can hard code some users in the `memory` provider in `security.yml`. Let's define the following users:

- `user` (password `user`) with `ROLE_USER`
- `admin` (password `admin`) with `ROLE_ADMIN`
- `matt` (password `smith`) with `ROLE_ADMIN`

We add these users in the `memory` section of the `providers` section of `security.yml`. Note we also must also define the `encoder` that user's passwords are encoded with. For now we'll just use un-encoded `plaintext`. So we add an `encoders` section to `security.yml` too[2].

```
security:
    encoders:
        Symfony\Component\Security\Core\User\User: plaintext

    providers:
        in_memory:
            memory:
                users:
                    user:
                        password: user
                        roles: 'ROLE_USER'
                    admin:
                        password: admin
                        roles: 'ROLE_ADMIN'
                    matt:
                        password: smith
                        roles: 'ROLE_ADMIN'

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        default:
            anonymous: ~
            http_basic: ~
```

Figure 15.5 shows successful access to the admin home page after a login of `uername=admin` and

[2]If you don't declare an encoder you'll get a `No encoder has been configured` Exception error message.

`password=admin`. Figure 15.6 shows us in the Symfony profiler that the user `admin` has the security token `USER_ADMIN`.

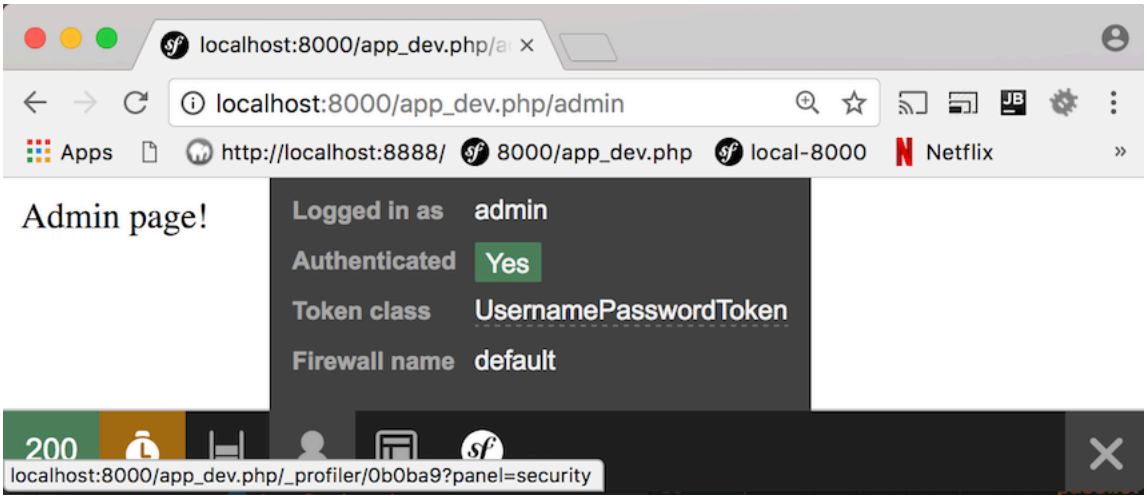


Figure 15.5: Admin home page, with profilering showing 'admin' login.

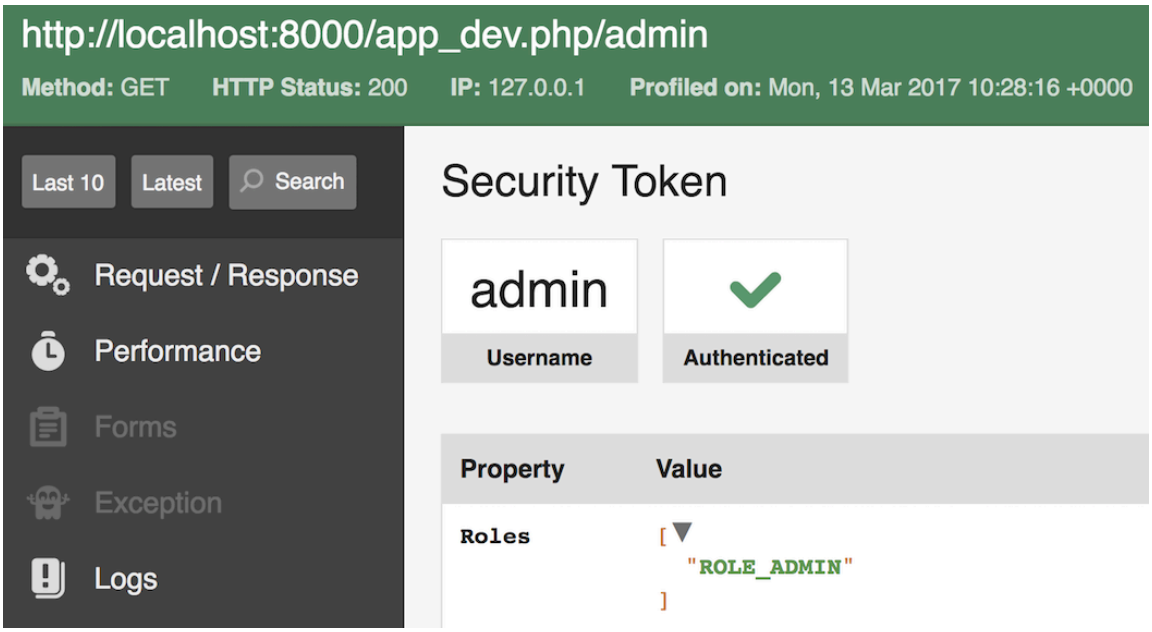


Figure 15.6: Symfony profiler showing `ROLE_ADMIN` token.

15.8 Security a route - method 2 - `security.yml` access control

The Symfony security examples offer a second method of securing routes, by adding an `access_contol` section to `security.yml`. Just as with the annotation comment, we declare that route `/admin` requires the user to have `ROLE_ADMIN`:

```
access_control:
    - { path: ^/admin, roles: ROLE_ADMIN }
```

So having added this to the default `security.yml` we would now have the following:

```
security:

    providers:
        in_memory:
            memory: ~

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        main:
```

```
            anonymous: ~

    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
```

And we'd have to add the `http_basic` authentication, and users and encoders etc. just as previously.

Important NOTE The `access_control` section of `security.yml` is **NOT** part of the `firewalls` section. Indentation is very important in YAML files, so ensure the `access_control` section start at the same level of indentation as `firewalls`. Otherwise you'll get the rather unhelpful `Access level "0" unknown` error message :-)

15.9 Hard to logout with `http_basic`

Apart from clearing the recent browser history, it seems that basic HTTP authentication (via the browser's built-in login page) doesn't prover any simple way to logout:

- Symfony logout section

So next we'll add a custom (Twig) login form, then we'll add a logout route to our application…

16

Custom login page and a logout route

16.1 Custom login form controller (`project18`)

The Symfony documentation tell's us how to create a custom login form, with CSRF protection, so let's do that.

- How to build a Traditional Login form
- CSRF protection in the Login form (NOTE the default settings work fines for this - we just need to make sure any Twig templates we write display the appropriate hidden CSRF form fields…)

First we need to replace out `http_basic` login authentication with our own, custom login form. We do this by replacing `http_basic: ~` in our `main` firewall with the a `form_login` entry.

NOTE Below I have commented-out the `http_basic` entry, to make it clear where we are replacing its entry:

```
security:
    encoders:
        Symfony\Component\Security\Core\User\User: plaintext

    providers:
        in_memory:
            memory:
                users:
```

```
                user:
                    password: user
                    roles: 'ROLE_USER'
                admin:
                    password: admin
                    roles: 'ROLE_ADMIN'
                matt:
                    password: smith
                    roles: 'ROLE_ADMIN'

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        main:
            anonymous: ~
#            http_basic: ~
            form_login:
                login_path: login
                check_path: login
```

The above declares that the route for an authentication login form is named `login` (we'll add a controller naming that route next). We are defining 2 important properties for the security system[1]:

- `login_path` - this is the route users will be redirected to if they attempt to access a resource but are do not have the authentication permitted to do so
- `check_path` - this is the route which the login form will submit a POST request to

You can read more about these paths, and other customisable features of the Symfony login system in the Symfony documentation:

- Symfony login and `check_path` reference

Let's create a new `SecurityController` in `src/AppBundle/Controller/` which declares the login route, and also tells our application to render a Twig custom login form template.

```
namespace AppBundle\Controller;

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

[1] Note - we could also specifric CSRF token settings here, but the Symfony security defaults all work fine Symfony default security settings

```
class SecurityController extends Controller
{
    /**
     * @Route("/login", name="login")
     */
    public function loginAction(Request $request)
    {
        // logic to show login form goes here
    }
}
```

NOTE We have broken-down the final steps of naming the Twig template and building the Twig argument array (simplying the one-liner code from the Symfony documentation):

```
public function loginAction(Request $request)
{
    $authenticationUtils = $this->get('security.authentication_utils');

    // get the login error if there is one
    $error = $authenticationUtils->getLastAuthenticationError();

    // last username entered by the user
    $lastUsername = $authenticationUtils->getLastUsername();

    // Twig stuff
    $templateName = 'security/login';
    $argsArray = [
        'last_username' => $lastUsername,
        'error'         => $error,
    ];

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

Looking at the above we can note the following:

- the first statement get a reference to the Symfony security utilities service `$authenticationUtils`
- the next 2 statements get any stored error `$error`, and the previous username `$last_username` (for repeated login attempts)
- finally we have our Twig statements, declaring that the login template is in views directory `security`, and building and then passing to Twig an arguments array containing the error and last username

We also can see that there is no logic in this method to **process** the submission of the form. The Symfony security system will process login form submission by looking through all its **providers** to see if it can match with a username/password pair, and acting accordingly.

16.2 Creating the login form Twig template

Let's write our Twig template for the login form (copied from the Symfony documentation pages). A heading 1 and some paragraph tags have been added, also the special form hidden element for CSRF protection.

Figure 16.1 shows the login form we'll create.

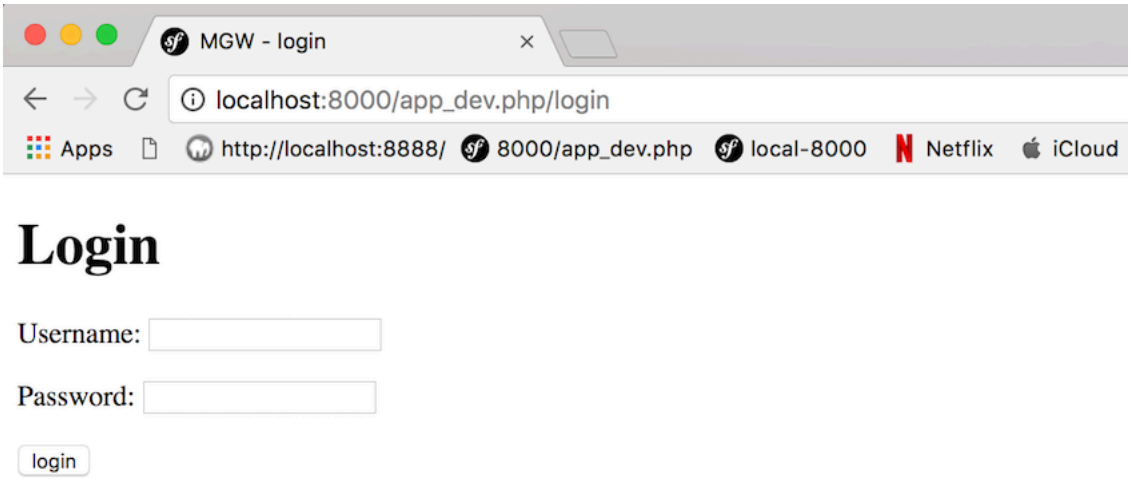


Figure 16.1: Customer login form.

Here is `app/Resources/views/security/login.html.twig`:

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

{% block pageTitle %} login page {% endblock %}

{% block body %}
    <h1>Login</h1>

    {% if error %}
        <div>{{ error.messageKey|trans(error.messageData, 'security') }}</div>
    {% endif %}

    <form action="{{ path('login') }}" method="post">
```

```
        <input type="hidden" name="_csrf_token"
               value="{{ csrf_token('authenticate') }}"
        >

        <p>
        <label for="username">Username:</label>
        <input type="text" id="username" name="_username" value="{{ last_username }}" />

        <p>
        <label for="password">Password:</label>
        <input type="password" id="password" name="_password" />

        <p>
        <button type="submit">login</button>
    </form>

{% endblock %}
```

Above we can see the following in our Login Twig template:

- a level 1 heading
- display of any Twig `error` variable received
- the HTML `<form>` open tag, which we see submits via HTTP `POST` method to the route named `login`
- a hidden form field with the `_csrf_token` to protect against forged request attacks one (CSRF tokens help protect web applications against cross-site scripting request forgery attacks and forged login attacks[2]).
- the `username` label and text input field (and value of the `last_username` if any)
- the `password` label and password input field
- the submit button named `login`

16.3 Adding a `/logout` route

We can define a route to logout very easily in Symfony, with no need for any controller method. In `app/config/routing.yml` we add our login route, and its redirect to the website home page `/`. We add our 2 lines to the end of this existing configuration file, since the default contents of this file tell Symfony to look for route annotation comments in our controllers:

[2]More about forged login attacks on Wikipedia

```
app:
    resource: '@AppBundle/Controller/'
    type: annotation

logout:
    path: /logout
```

We also need to define the logout route as part of our security firewall. So in `security.yml` we add the following to the default firewall:

```
logout:
    path:   /logout
    target: /
```

So our full `security.yml` now looks as follows:

```
security:
    encoders:
        Symfony\Component\Security\Core\User\User: plaintext

    providers:
        in_memory:
            memory:
                users:
                    user:
                        password: user
                        roles: 'ROLE_USER'
                    admin:
                        password: admin
                        roles: 'ROLE_ADMIN'
                    matt:
                        password: smith
                        roles: 'ROLE_ADMIN'


    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        default:
            anonymous: ~
            form_login:
```

```
            login_path: login
            check_path: login

        logout:
            path:   /logout
            target: /
```

Figure 16.2 shows that we can see the logout route is available from the Symfony profile toolbar. We can, of course, also enter the route directly in the browser address bar, e.g. via URL:

```
http://localhost:8000/app_dev.php/logout
```

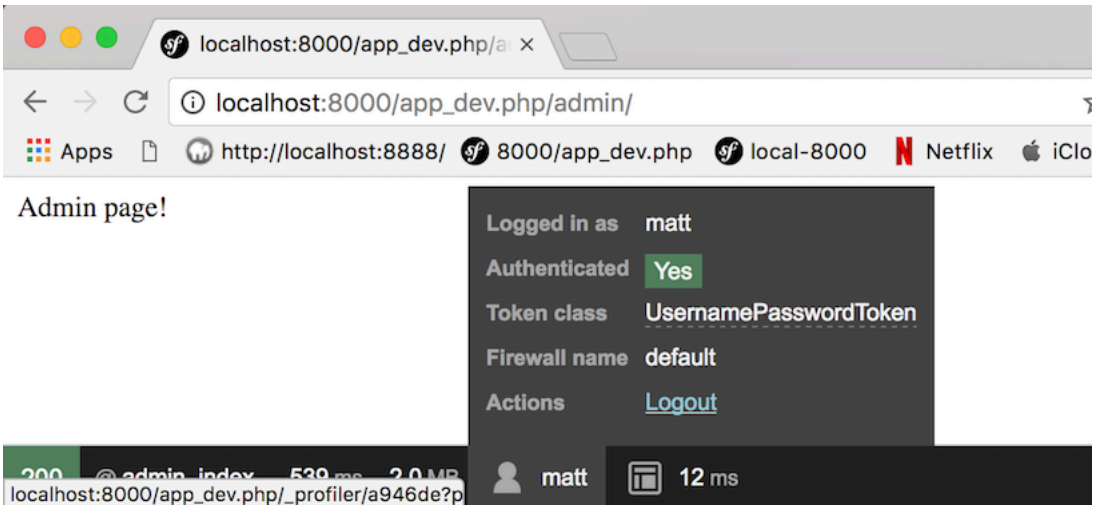


Figure 16.2: Symfony profiler user logout action.

In either case we'll logout any currently logged-in user, and return the anonymously authenticated user `anon` with no defined authentication roles.

17

Encoding the user passwords

17.1 Encoding the user passwords (`project19`)

It is **not** good practice to store user passwords as plain text, so let's change the encoder and store hashed passwords instead. The Symfony introduction to security documentation page tells us how to encode user passwords:

- (Encoding user passwords)[http://symfony.com/doc/current/security.html#c-encoding-the-user-s-password]

First, in `security.yml` we need to change the encoder from `plaintext` to BCrypt as follows:

```
encoders:
    Symfony\Component\Security\Core\User\User:
        algorithm: bcrypt
        cost: 12
```

Now we need to replace the plaintext passwords for our 3 users (`user`, `admin` and `matt`) with their BCrypted passwords. We can do this by using the Symfony command line tool that will tell us the encoded string for a given password (using the encoder specified in `security.yml`). So we'll need to run this 3 times (and copy-paste the encoded password into our `security.yml` YAML file each time:

```
$ php bin/console security:encode-password
```

Figure 17.1 shows the interactive password encoding session for password `user`:

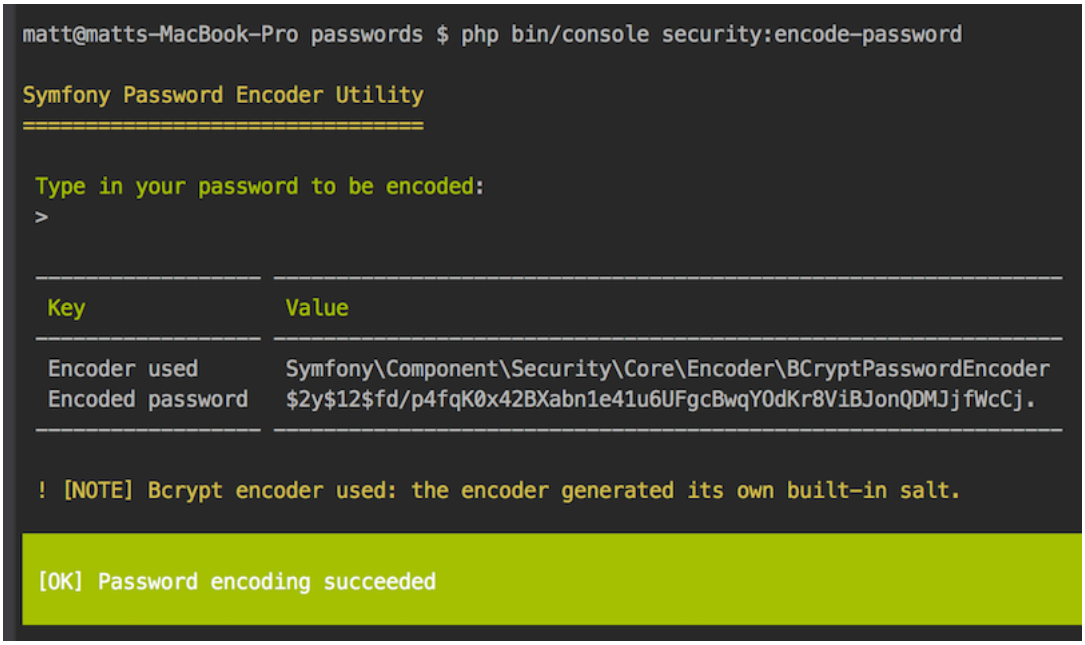


Figure 17.1: CLI password encoding (for password `user`).

So the full listing for our `security.yml` configuration, stating the encoder and the hashed passwords looks like this:

```
security:
    encoders:
        Symfony\Component\Security\Core\User\User:
            algorithm: bcrypt
            cost: 12

    providers:
        in_memory:
            memory:
                users:
                    user:
                        password: $2y$12$pUaaC6cwub1NkwNvSm/FnuR3rli8YgjIg1Di68hqX4J1TnGpLc2
                        roles: 'ROLE_USER'
                    admin:
                        password: $2y$12$ROCN/MhD6UORsr0xsrHT/.RETqtgm8nQmdbOsC2o4w4RyHrUhXc
                        roles: 'ROLE_ADMIN'
                    matt:
                        password: $2y$12$4UWrrc1pkskcCMDpcj4XzeLVsn5Tlk4zkQJAyrSaoDnOnY1wgHU
                        roles: 'ROLE_ADMIN'
```

```
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        default:
            anonymous: ~
            form_login:
                login_path: login
                check_path: login

            logout:
                path:   /logout
                target: /
```

17.2 Those nice people at KnpUniversity...

If you want to go further and really learn Symfony security, with your own User entity and database storage etc. then a great place to start would be the KnpUniversity video tutorial at:

- KnpUniversity - Symfony Security: Beautiful Authentication, Powerful Authorization

Part VII

Entity associations (one-to-many relationships etc.)

18

Doctrine associations (entity relationships)

18.1 Some useful reference sources

Any non-trivial project involving databases involves one-to-many and many-to-many relationships. the Doctrine ORM system makes it very easy to declare, and manipulate datasets with foreign-key relationships.

Some useful information sources on this topuc include:

- How to Work with Doctrine Relations
- Forms EntityType Field

18.2 Simple example: Users and their county (`project22`)

Each User lives in a county (e.g. Matt Smith lives in County Kildare (in Ireland!). So if we have a reference to a User object instance, then we want to easily be able to follow the foreign key link to the details of the county in which that User lives.

18.3 Create the County Entity

NOTE First setup your project to either use MySQL or SQLite (see Appendices E and F).

First let's generate a simple `County` Entity - it will have an automatically assigned integer `id`, and a text `name` property:

```
php bin/console generate:doctrine:entity --no-interaction
--entity=AppBundle:County --fields="name:string(255)"
```

You should now have a basic `County` Entity class in `/src/AppBundle/Entity/`.

18.4 Create basic `User` entity

Now let's create a `User` entity, with `username`, `password` and a `county`. Since we are using an ORM we can specify that the `county` property of each user should be a reference to an object instance of class `AppBundle\Entity\County`:

```
php bin/console generate:doctrine:entity --no-interaction
--entity=AppBundle:User --fields="username:string(255) password:string(255) county:AppBundle
```

18.5 Update Entity `User` to declare many-to-one association

Change this entry for the `county` field in Entity `User` **from** this:

```
/**
 * @var \AppBundle\Entity\County
 *
 */
private $county;
```

to the following (declaring the man to one relationship and creating a foreign key field 'county_id' to store the id for the relationship)

```
/**
 * @var \AppBundle\Entity\County
 *
 * @ORM\ManyToOne(targetEntity="County")
 * @ORM\JoinColumn(name="county_id", referencedColumnName="id")
 */
private $county;
```

18.6 Complete generation of Entities

We can now make Symfony generate getters and setters and complete the entity creation:

```
php bin/console doctrine:generate:entities AppBundle
```

18.7 Update the database schema

We now tell Symfony/Doctrine to update the database scheme to match our Entities:

```
php bin/console doctrine:schema:update --force
```

18.8 CRUD and views generation

We can now generate the CRUD controllers, `Type` form classes, and Twig views for CRUD actions for both our Entities User and County:

```
php bin/console generate:doctrine:crud --entity=AppBundle:User --format=annotation
--with-write --no-interaction

php bin/console generate:doctrine:crud --entity=AppBundle:County --format=annotation
--with-write --no-interaction
```

18.9 MILESTONE 1 - we can now list users and work with counties

At this point, we can now list users, and work with counties (CRUD), as illustrated in Figure 18.1.

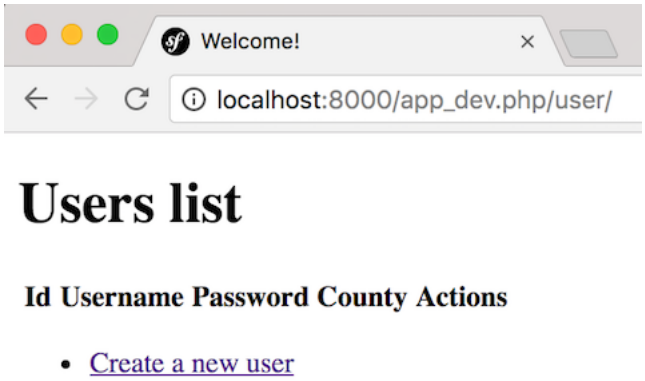


Figure 18.1: Users list (index action) working.

However, were we to try to create or edit a user, we'd get an error, since the default Form Type for `User` doesn't generate a drop-down meun based on the text `name` values for `County` entities.

18.10 Editing the UserType form for county names

We need to make the `User` form generate a choice list from the different `County` entities in our database.

First we need toadd a 'use' statement for `/src/AppBundle/Form/UserType.php`:

```
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
```

Then next we need to replace this one line:

```
$builder->add('username')->add('password')->add('county')        ;
```

with these 2 lines:

```
$builder->add('username')->add('password');

$builder->add('county', EntityType::class, [
    'class' => 'AppBundle:County',

    // use the User.username property as the visible option string
    'choice_label' => 'name',
]);
```

Now our form should work, proving a drop-down choice menu when we edit or create a `User` record. We can see this in Figure 18.2.

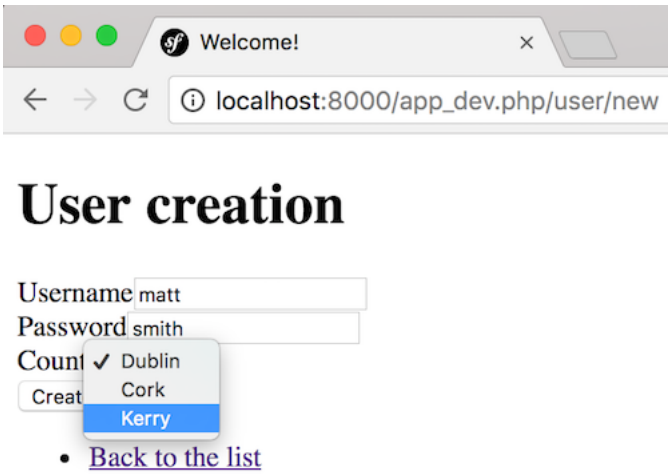


Figure 18.2: New User form, with list of county names.

18.11 Add county names to Twig templates

Finally, we can add a `County` name details to our User **index** and **show** Twig templates.

1. Let's add County name in the list of users for the index USer action. We edit `app/Resources/views/user/index.html.twig` and add a `<th>` entry and for each User a `<td>` entry. We use dot `.` notation to show an object reference being followed, so we can literatally write `user.county.name` to the get the `name` property, of the `county` object that is referred to for the current `user` in the loop (see Figure 18.3):

```
<th>County</th>
...

<td>{{ user.county.name }}</td>
```

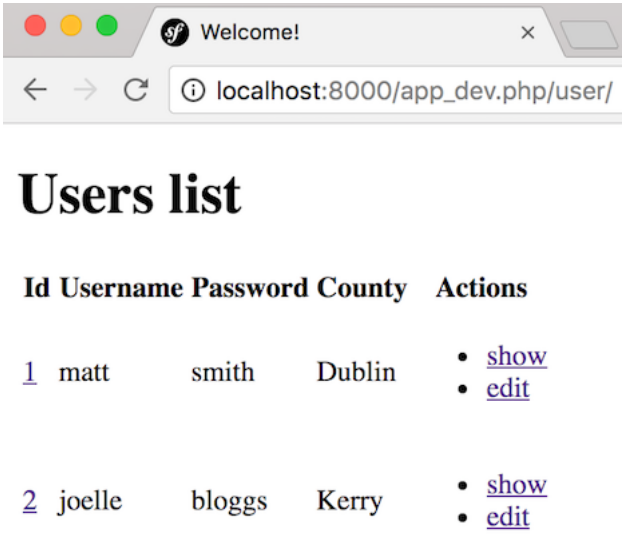


Figure 18.3: List of users, incuding their county.

We do something similar for the **show** User action, adding another table row for the `County` (see Figure 18.4):

```
<tr>
    <th>County</th>
    <td>{{ user.county.name }}</td>
</tr>
```

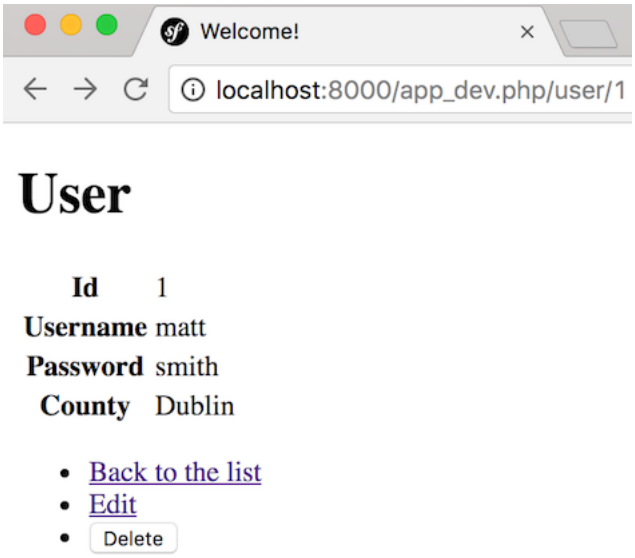


Figure 18.4: Show one User, including their county.

Part VIII

Appendices

A

Solving problems with Symfony

A.1 No home page loading

Ensure web server is running (either from console, or a webserver application with web root of the project's `/web` directory).

Point your web browser to the `app_dev.php` front controller script, e.g.:

```
http://localhost:8000/app_dev.php
```

A.2 "Route not Found" error after adding new controller method

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

- delete directory `/var/cache`

Symfony caches (stores) routing data and also rendered pages from Twig, to speed up reponse 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.

A.3 Issues with timezone

Try adding the following construction to `/app/AppKernel.php` to solve timeszone problems:

```
public function __construct($environment, $debug)
{
    date_default_timezone_set( 'Europe/Dublin' );
    parent::__construct($environment, $debug);
}
```

B

Quick setup for new 'blog' project

B.1 Create a new project, e.g. 'blog'

Use the Symfony command line installer (if working for you) to create a new project named 'blog' (or whatever you want!)

```
$ symfony new blog
```

Or use Composer:

```
$ composer create-project symfony/framework-standard-edition blog
```

Read more at:

- Symfony create project reference

B.2 Set up your localhost browser shortcut to for `app_dev.php`

Set your web browser shortcut to the `app_dev.php`, i.e.:

```
http://localhost:8000/app_dev.php
```

B.3 Add `run` shortcut to your Composer.json scripts

Make life easier - add a "run" Composer.json script shortcut to run web server from command line:

```
"scripts": {
    "run":"php bin/console server:run",
    ...
```

B.4 Change directories and run the app

Change to new project directory and run the app

```
/~user/$ cd blog
/~user/blog/$ composer run
```

Now visit: `http://localhost:8000/app_dev.php` in your browser to see the welcome page

B.5 Remove default content

If you want a **completely blank** Symfony project to work with, then delete the following:

```
/src/AppBundle/Controller/DefaultController.php
/app/Resources/views/default/
/app/Resources/views/base.html.twig
```

Now you have no controllers or Twig templates, and can start from a clean slate…

C

Steps to download code and get website up and running

C.1 First get the source code

First you need to get the source code for your Symfony website onto the computer you want to use

C.1.1 Getting code from a zip archive

Do the following:

- get the archive onto the desired computer and extract the contents
- if there is no `/vendor` folder then run CLI command `composer update`

C.1.2 Getting code from a Git respository

Do the following:

- on the computer to run the server `cd` to the web directory
- clone the repository with CLI command `git clone <REPO-URL>`
- populate the `/vendor` directory by running CLI command `composer update`

C.2 Once you have the source code (with vendor) do the following

- update `/app/config/parameters.yml` with your DB user credentials and name and host of the Database to be used
- start running your MySQL database server (assuming your project uses MySQL)
- create the database with CLI command `php bin/console doctrine:database:create`
- create the tables with CLI command `php bin/console doctrine:schema:update --force`

C.3 Run the webserver

Either run your own webserver (pointing web root to `/web`, or

- run the webserver with CLI command `php bin/console server:run`
- visit the website at `http://localhost:8000/`

D

About `parameters.yml` and `config.yml`

D.1 Project (and deployment) specific settings in (`/app/config/parameters.y`

Usually the project-specific settings are declared in this file:

```
/app/config/parameters.yml
```

These parameters are referred to in the more generic `/app/config/config.yml`.

For example the host of a MySQL database for the project woudl be defined by the following variable in `parameters.yml`:

```
parameters:
    database_host: 127.0.0.1
```

Note that this file (`parameters.yml`) is include in the `.gitignore`, so it is **not** archived in your Git folder. Usually we need different parameter settings for different deployments, so while on your local, development machine you'll have certain settings, you'll need different settings for your public production 'live' website. Plus you don't want to accidently publically expose your database credentials on a open source Github page :-)

If there isn't already a `parameters.yml` file, then you can copy the `parameters.yml.dist` file end edit it as appropriate.

D.2 More general project configuration (/app/config/config.yml)

The file /app/config/config.yml is actually the one used by Symfony when it looks up project settings. So the config.yml file uses references to the variables declared in the /app/config/parameters.yml file. For example the following lines in config.yml make a reference to the variable database_path that is declared in parameters.yml:

```
doctrine:
    dbal:
        driver:   pdo_mysql
        host:     "%database_host%"
```

For many projects we need to make **no changes** to the contents of config.yml. Although, since Symfony is setup with defaults for a MySQL database, if we are using SQLIte, for example then we do need to change the configuration settings, as well as declaring appropriate variables in parameters.yml. This is discussed in Appendix , describing how to set up a Symfony project to work with SQLite.

E

Setting up for MySQL Database

E.1 Declaring the parameters for the database (`/app/config/parameters.yml`)

Usually the project-specific settings are declared in this file:

```
/app/config/parameters.yml
```

These parameters are referred to in the more generic `/app/config/config.yml` - which for MySQL projects we don't need to touch.

The simplest way to connect your Symfony application to a MySQL database is by setting the following variables in `parameters.yml` (located in (`/app/config/`):

```
# This file is auto-generated during the composer install
parameters:
    database_host: 127.0.0.1
    database_port: null
    database_name: symfony_book
    database_user: root
    database_password: null
```

Note, you can learn move about `parameters.yml` and `config.yml` in Appendix D.

You can replace `127.0.0.1` with `localhost` if you wish. If your code cannot connect to the database check the 'port' that your MySQL server is running at (usually 3306 but may be different, for example my Mac MAMP server uses 8889 for MySQL for some reason). So my parameters look

like this:

```
parameters:
    database_host:     127.0.0.1
    database_port:     8889
    database_name:     symfony_book
    database_user:     symfony
    database_password: pass
```

We can now use the Symfony CLI to **generate** the new database for us. You've guessed it, we type:

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

You should now see a new database in your DB manager. Figure E.1 shows our new `symfony_book` database created for us.

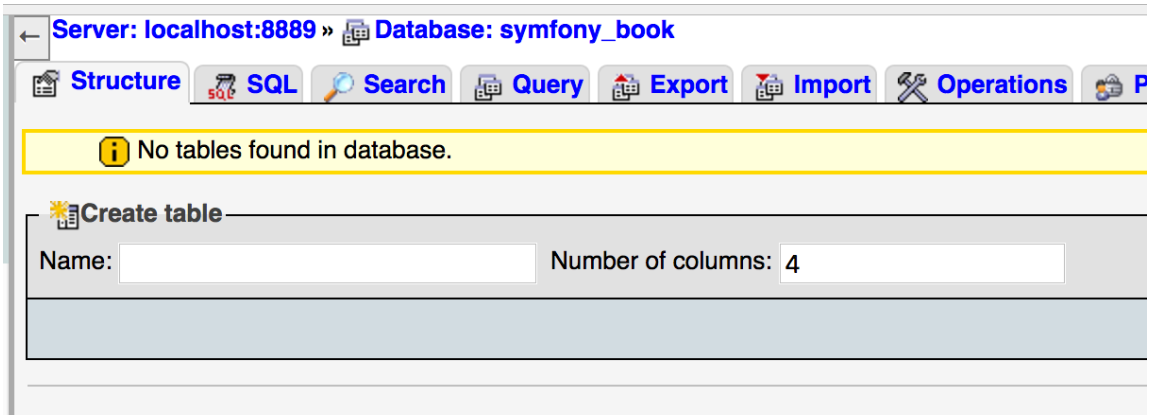


Figure E.1: CLI created database in PHPMyAdmin.

NOTE Ensure your database server is running before trying the above, or you'll get an error like this:

```
[PDOException] SQLSTATE[HY000] [2002] Connection refused
```

now we have a database it's time to start creating tables and populating it with records …

F

Setting up for SQLIte Database

F.1 SQLite suitable for most small-medium websites

For small/medium projects, and learning frameworks like Symfony, it's often simplest to just use a file-based SQLite database.

Learn more about SQLite at the project's website, and their discussion of when SQLite is a good choices, and when a networked DBMS like MySQL is more apppropritate:

- SQLite website
- Appropriate Uses For SQLite

F.2 Create directory where SQLite database will be stored

Setting one up with Symfony is **very** easy. These steps assume you are gong to use an SQLite database file named `data.sqlite` located in directory `/var/data`.

Our first step to configuring a Symfony project to work with SQLite is to ensure the directory exists where the SQLIte file is to be created. The usual location for Symfony projects is `/var/data`. So create directory `data` in `/var` if it doesn't already exist in your project.

F.3 Declaring the parameters for the database (`/app/config/parameters`

In `/app/parameters.yml` replace the default `database_host/name/user/password` parameters with a single parameter `database_path` as follows:

```yaml
    parameters:
        database_path: ../var/data/data.sqlite
        mailer_transport: smtp
        mailer_host: 127.0.0.1
        etc.
```

F.4 Setting project configuraetion to work with the SQLite database driver and path (`/app/config/config.yml`)

In `/app/config.yml` change the `doctrine` settings **from** these MySQL defaults:

```yaml
    # Doctrine Configuration
    doctrine:
        dbal:
            driver:   pdo_mysql
            host:     "%database_host%"
            port:     "%database_port%"
            dbname:   "%database_name%"
            user:     "%database_user%"
            password: "%database_password%"
            charset:  UTF8
```

to these SQLite settings:

```yaml
    # Doctrine Configuration
    doctrine:
        dbal:
            driver:   pdo_sqlite
            path:     "%kernel.root_dir%/%database_path%"
```

That's it! You can now tell Symfony to create your database with CLI command:

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

You'll now have an SQLite database file at `/var/data/data.sqlite`. You can even use the PHP-Storm to open and read the DB for you. See Figures F.1 and F.2.

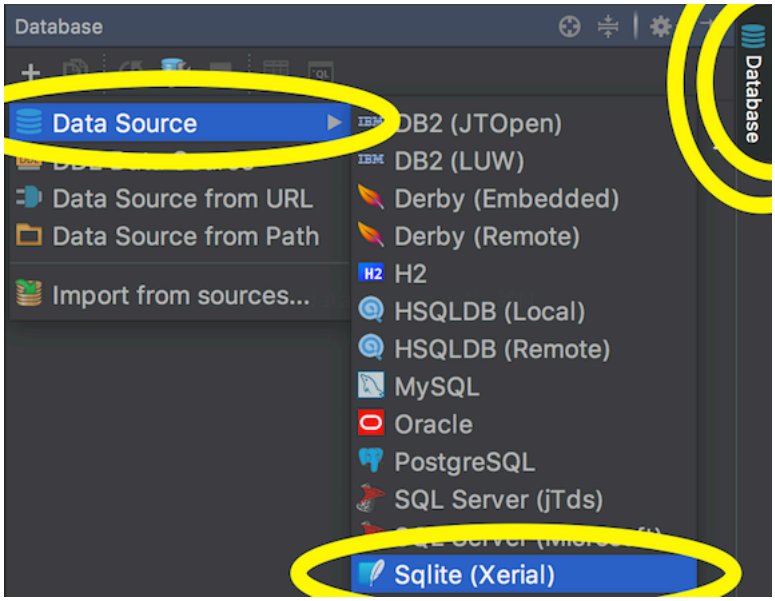


Figure F.1: Open SQLite view in PHPMyAdmin.

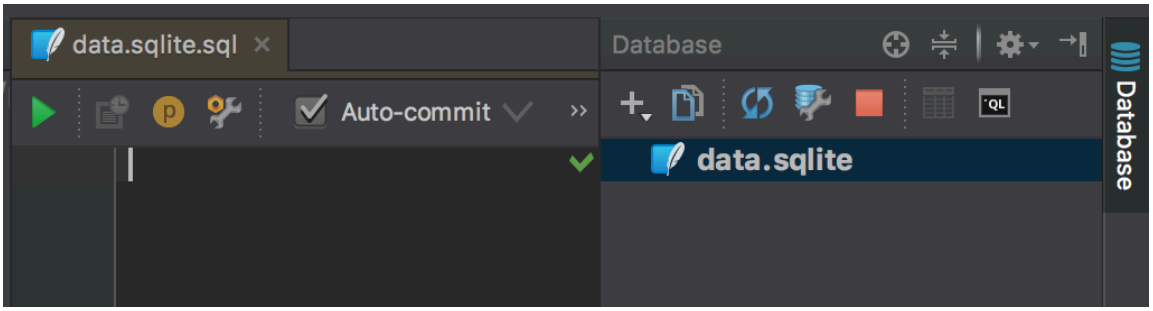


Figure F.2: Viewing `/var/data.sqlite` in PHPStorm.

Avoiding issues of SQL reserved words in entity and property names

Watch out for issues when your Entity name is the same as SQL keywords.

Examples to **avoid** for your Entity names include:

- user
- group
- integer
- number
- text
- date

If you have to use certain names for Entities or their properties then you need to 'escape' them for Doctrine.

- Doctrine identifier escaping

You can 'validate' your entity-db mappings with the CLI validation command:

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

H

Transcript of interactive entity generation

The following is a transcript of an interactive session in the terminal CLI to create an `Item` entity class (and related `ItemRepository` class) with thse properties:

- title (string)
- price (float)

You start this interactive entity generation dialogue with the following console command:

```
php bin/console doctrine:generate:entity
```

Here is the full transcript (note all entites are automatically given an 'id' property):

```
$ php bin/console doctrine:generate:entity

  Welcome to the Doctrine2 entity generator

This command helps you generate Doctrine2 entities.

First, you need to give the entity name you want to generate.
You must use the shortcut notation like AcmeBlogBundle:Post.

The Entity shortcut name: AppBundle:Product/Item

Determine the format to use for the mapping information.
```

```
Configuration format (yml, xml, php, or annotation) [annotation]:

Instead of starting with a blank entity, you can add some fields now.
Note that the primary key will be added automatically (named id).

Available types: array, simple_array, json_array, object,
boolean, integer, smallint, bigint, string, text, datetime, datetimetz,
date, time, decimal, float, binary, blob, guid.

New field name (press <return> to stop adding fields): description
Field type [string]:
Field length [255]:
Is nullable [false]:
Unique [false]:

New field name (press <return> to stop adding fields): price
Field type [string]: float
Is nullable [false]:
Unique [false]:

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

  Entity generation

  created ./src/AppBundle/Entity/Product/
  created ./src/AppBundle/Entity/Product/Item.php
> Generating entity class src/AppBundle/Entity/Product/Item.php: OK!
> Generating repository class src/AppBundle/Repository/Product/ItemRepository.php: OK!

  Everything is OK! Now get to work :).

$
```

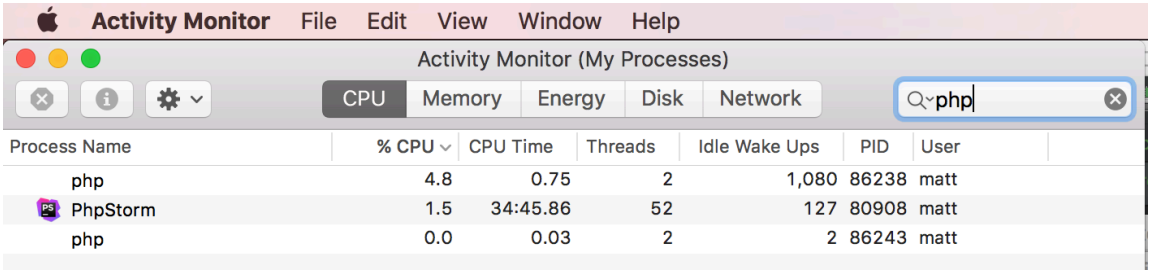
I

Killing ‘php’ processes in OS X

Do the following:

- run the **Activity Monitor**
- search for Process Names that are `php`
- double click them and choose `Quit` to kill them

voila!



List of References