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 Introduction

1.1 What is Symfony 3?

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

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

I recommend you install the following:

- PHP 7 (on windows Laragon works pretty well)
- a MySQL database server (on windows Laragon works pretty well)
- a good text editor (I like PHPStorm, but then it's free for educational users…)
- Composer (PHP package manager - on windows Laragon works pretty well)

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

1.3 How to I get started?

Either:

- install the Symfony command line installed, then create a project like this (to create a new project in a directory named `project01`):

```
$ symfony new project01
```

or

- use Composer to create a new blank project for you, like this (to create a new project in a directory named `project01`):

```
$  composer create-project symfony/framework-standard-edition project01
```

Learn about both these methods at the Symfony download-installer page and the Symfony setup page

or

- download one of the projects accompanying this book

1.4 Where are the projects accompanying this book?

There are on Github:

- https://github.com/dr-matt-smith/php-symfony3-book-codes

Download a project (e.g. `git clone URL`), then type `composer update` to download 3rd-party packages into a `/vendor` folder.

1.5 How to I run a Symfony webapp?

1.5.1 From the CLI

If you're not using a database engine like MySQL, then you can use the Symfony console command to 'serve up' your Symfony project from the command line

At the CLI (comamnd line terminal) ensure you are at the base level of your project (i.e. the same directory that has your `composer.json` file), and type the following:

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

1.5.2 Webserver

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

1.5.3 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);
}
```

2

First steps

2.1 It isn't working

If you don't get the default Sfymfony home page, try this:

- copy the contents of `/web/app_dev.php` into `/web/app.php`

WARNING - this is just for now (we'll learn property Symfony configuration later). But this should get you going for now. You should NEVER do this for a project that might actually end up as a public production site!

2.2 All I get is the symfony home page (`project01`)

Figure 2.1 is your basic, default Symfony home page if everything is up and running for a new Symfony project.

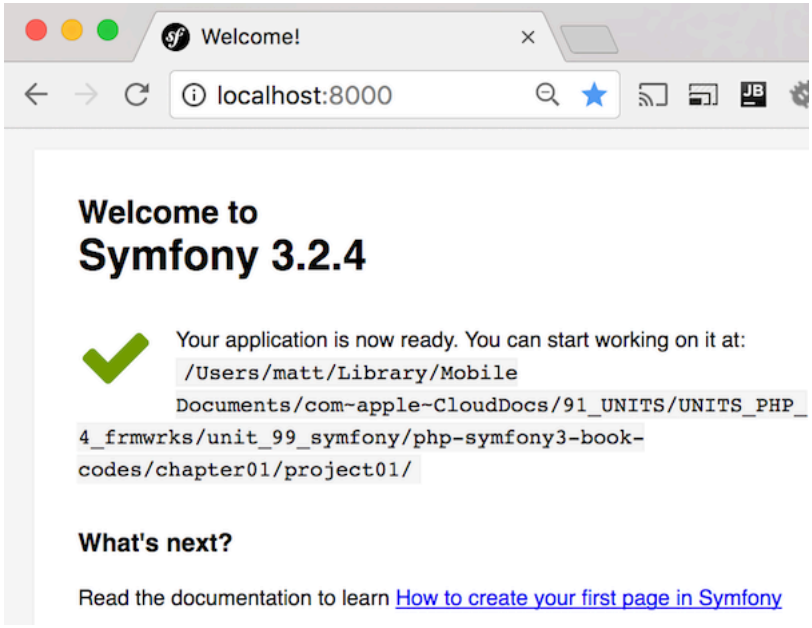


Figure 2.1: New Symfony project home page.

2.3 What we'll make (`project02`)

See Figure 2.2 for a screenshot of the new homepage we'll create this chapter.

There are 3 things Symfony needs to serve up a page (with the Twig templating system):

1. a route
2. a controller class and method
3. a Twig template

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

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

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

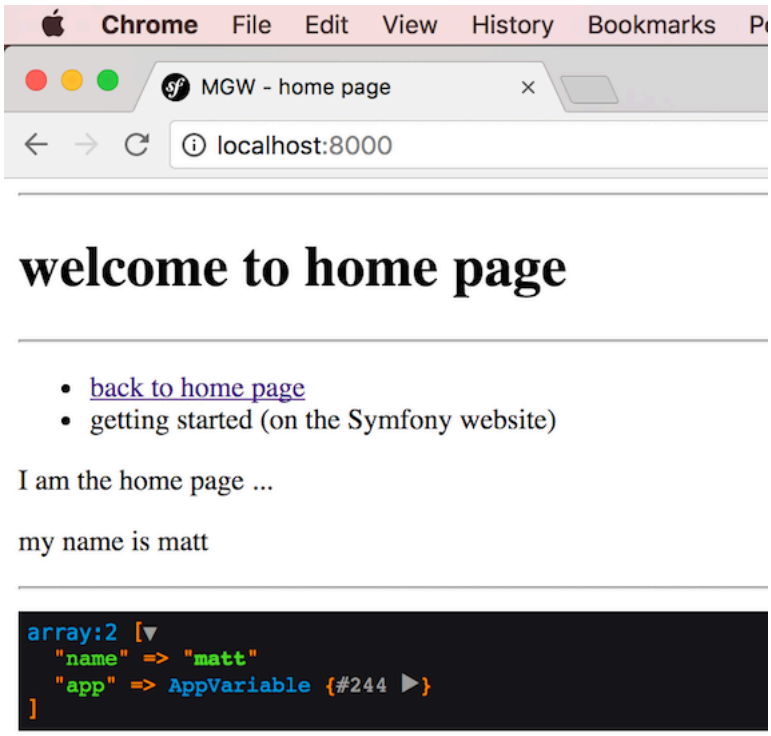


Figure 2.2: New home page.

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

The last (Twig template) can be a single file, and a simpler template that 'extends' a base template (which has all the standard doctype, css, js and core HTML structure in it).

If don't know much about Twig then go off and learn it (you can learn it stand alone, with a simple micro-framework like Silex, and as part of learning Symfony).

2.4 First - get rid of all that default page stuff

We'll stick with the single `AppBundle` that we get provided with a new Symfony project (most logic goes into a 'bundle', we only need one for now).

A new Symfony project places its `DefaultController` at this location:

```
/src/AppBundle/Controller/DefaultController.php
```

Figure 2.3 shows the `DefaultController.php` in this location.

Let's clear out the content of the controller, so there is no code in the body of the `indexAction()` method:

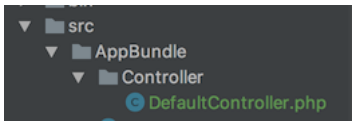


Figure 2.3: Location of Controller classes.

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

NOTES: - leave all the 'uses' statements and the namespace, since they mean any classes we refer to, or annotations we use, all work correctly - leave teh `Route` annotation comment there, since what we are about to write will be what we want to happend for a request for the website home page (i.e. the web root URL of `/` for our webapp) - alse leave the `name="homepage"` part of the annotation route comment, since naming routes is very handy since it makes getting Twig to create links very easy

We want to use the template `index.html.twig`, since they all end in `.html.twig` let's concatenate that on later

```
$templateName = 'index';
```

Twig templates expect to be given an associative array of any special data for the template, so let's illustrate this by passing a parameter `name` with your name (I'm Matt, so that will be my name parameter's value!):

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

There is nothing magic about the array identifier `$argsArray` - it's just a habit I've got into when teaching Twig to my students - so change this (and anything - it's **your** project) to become more confident with working with the different bits of Symfony.

Symfomy's `Controller` class offers a handy method `render()` with accesses the Twig service in the Symfony application, so we can just invoke this method passing the template name (and appending the `.html.twig` string), and the array of arguments:

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

```
public function indexAction(Request $request)
{
    $argsArray =  [
        'name' => 'matt'
    ];

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

Note that this final statement is a `return` statement. Basically any web application received (and interprets the contents of) an HTTP 'request', and builds and sends back an HTTP 'response'. The way Symfony (and most MVC webapps) work is that the controller method invoked for a given route has the responsibility of building and returning a 'response' (or sometimes just the text 'content' of a response, and the MVC application will build an HTTP response around that text content).

2.5 Our 2 Twig templates (`_base.html.twig` and `index.html.twig`)

Twig templates are located in this directory:

```
/app/Resources/views
```

Delete everything in this directory (more of that default homepage stuff that we get with a new Symfony project). We'll create our own Twig templates from scratch in this location next.

Figure 2.4 shows the 2 templates we are about to create in this location.

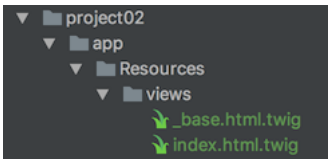


Figure 2.4: Location of Twig templates.

Here is our `_base.html.twig` template for a well-formed HTML 5 page[1]:

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

[1]NOTE - if you want to see the FANTASTICALLY useful Symfony debug toolbar, your pages must render a well-formed HTML document (with doctype, head, body etc.). Using a base Twig template is the simplest way to do this usually.

```
    </head>
    <body>

    <hr>
        {% block body %}
        {% endblock %}

        {% block javascripts %}{% endblock %}
    </body>
</html>
```

There is nothing magic about the array identifier `_base.html.twig` - a habit (I've copied from some project I saw years ago) is to prefix Twig templates if they are a base template (such as this one), or if they are a 'partial' page template (e.g. generating a navbar or side bar). Giving a bunch of files the same preix character means that they'll all be grouped together when listed alphabetically. Another approach is to create a directory (e.g. `/partials`) and put them all in there…

Here is the template for our index page, `index.html.twig`:

```
{% extends '_base.html.twig' %}
{% block pageTitle %}home page{% endblock %}

{% block body %}
    <h1>welcome to home page</h1>
    <ul>
        <li>
            <a href="{{ path('homepage') }}">back to home page</a>
        </li>
        <li>
            <a hrer="http://symfony.com/doc/current/page_creation.html">
            getting started (on the Symfony website)</a>
        </li>
    </ul>

    <p>
        I am the home page ...
    <br>
        my name is {{ name }}
    </p>
    {{ dump() }}
{% endblock %}
```

Some interesting bits in this template:

- the Twig dump command `{{ dump() }}` is very handy, it let's us see a full dump of all the variables Twig has been passed. Both those we explicitly pass like `name`, plus the `app` variable, that let's Twig get access to things like the sessions variables etc.a
- also we see how we can use the route 'name' in Twig to generate an URL for that route. The example in this template is

2.6 See list of all routes

We can use another of Symfony's CLI commands to see a list of all routes - we should see our `homepage` root in that list: `<a href="{{ path('homepage') }}">`. Twig can also pass values for routes that expect parameters such as object IDs etc.

```
php bin/console debug:router
```

We can see there are lots of special routes (many to do with the debugging Symfony profiler). At the end is our homepage route - yah!

Figure 2.5 shows the list of routes we get after entering this statement at the command line.

```
matt@matts-MacBook-Pro project01 (master) $ php bin/console debug:router
 ---------------------------- -------- -------- ------ ----------------------------------
  Name                         Method   Scheme   Host   Path
 ---------------------------- -------- -------- ------ ----------------------------------
  _wdt                         ANY      ANY      ANY    /_wdt/{token}
  _profiler_home               ANY      ANY      ANY    /_profiler/
  _profiler_search             ANY      ANY      ANY    /_profiler/search
  _profiler_search_bar         ANY      ANY      ANY    /_profiler/search_bar
  _profiler_info               ANY      ANY      ANY    /_profiler/info/{about}
  _profiler_phpinfo            ANY      ANY      ANY    /_profiler/phpinfo
  _profiler_search_results     ANY      ANY      ANY    /_profiler/{token}/search/results
  _profiler_open_file          ANY      ANY      ANY    /_profiler/open
  _profiler                    ANY      ANY      ANY    /_profiler/{token}
  _profiler_router             ANY      ANY      ANY    /_profiler/{token}/router
  _profiler_exception          ANY      ANY      ANY    /_profiler/{token}/exception
  _profiler_exception_css      ANY      ANY      ANY    /_profiler/{token}/exception.css
  _twig_error_test             ANY      ANY      ANY    /_error/{code}.{_format}
  homepage                     ANY      ANY      ANY    /
 ---------------------------- -------- -------- ------ ----------------------------------
```

Figure 2.5: List of all routes.

3

Creating our own classes

3.1 What we'll make (`project03`)

See Figure 3.1 for a screenshot of the students list page we'll create this chapter.

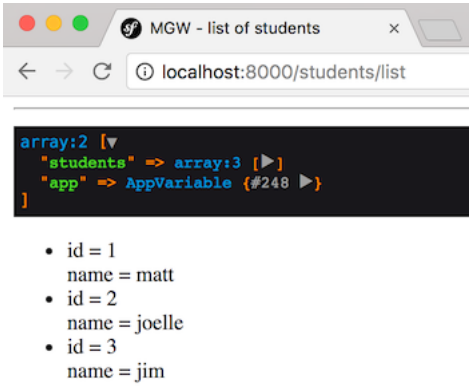


Figure 3.1: Students lists page.

3.2 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 3.2 shows the `DefaultController.php` in this location.

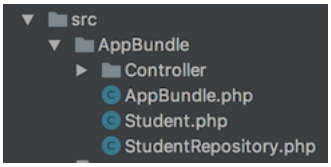


Figure 3.2: Location of Student and StudentRepository classes.

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.
}
```

3.3 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`.

3.4 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 3.3 shows the 2 templates we are about to create in this location.

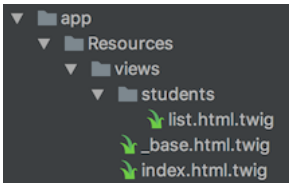


Figure 3.3: Location of Twig template `list.html.twig`.

```
{% 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

4

Doctrine the ORM

4.1 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]

4.2 Quick start

Once you've learnt how to work with Entity classes and Doctrine, these are the 3 commands you need to know:

1. `doctrine:database:create`
2. `doctrine:database:migrate`
3. `doctrine:fixtures:load`

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

4.3 Setting up the database credentials

The simplest way to connect your Symfony application to a MySQL database is by creating/editing the `parameters.yml`

```
# 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
```

This file is located in:

```
/app/config/parameters.yml
```

Note that this file 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. 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 (usualy 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 4.1 shows our new `symfony_book` database created for us.

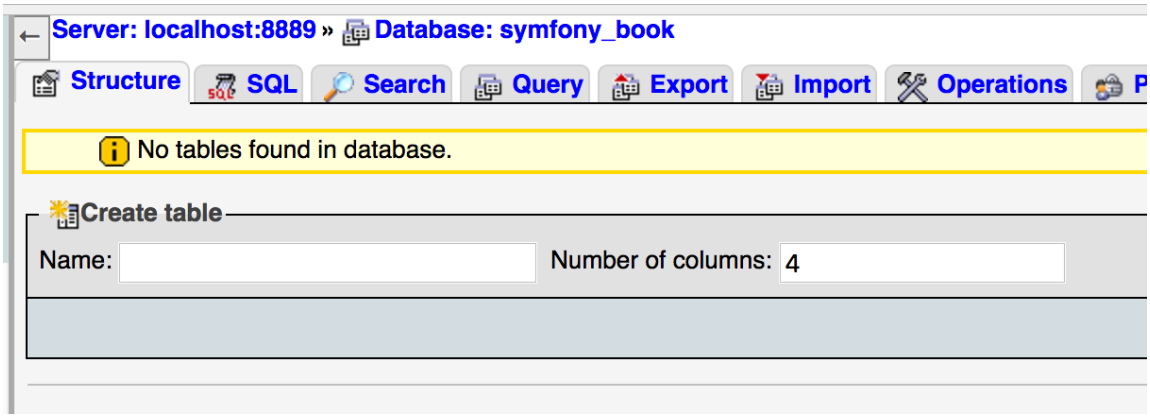


Figure 4.1: CLI created database in PHPMyAdmin.

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 …

5

Working with Entity classes

5.1 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;
}
```

5.2 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
```

5.3 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;
```

5.4 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.
```

5.5 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

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

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

5.6 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 exisited):

```
$ 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 5.1 shows our new `students` table created for us.

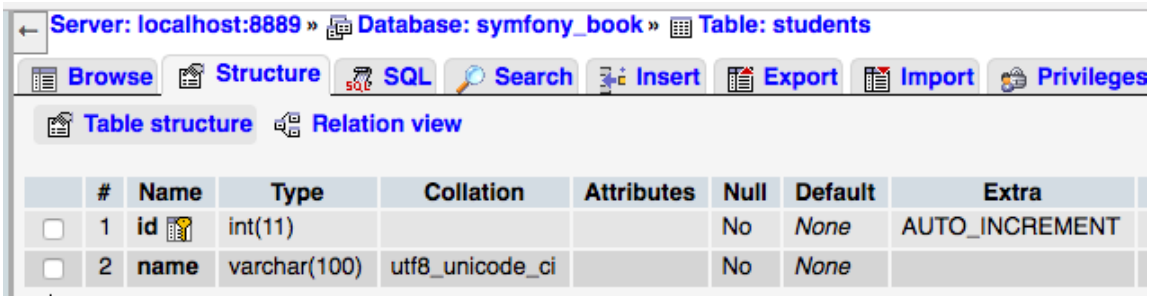


Figure 5.1: CLI created table in PHPMyAdmin.

5.7 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

3_new_student.png

6

Symfony approach to database CRUD

6.1 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 6.1 shows how a new record `freddy` is added to the database table via route `/students/create/{name}`.

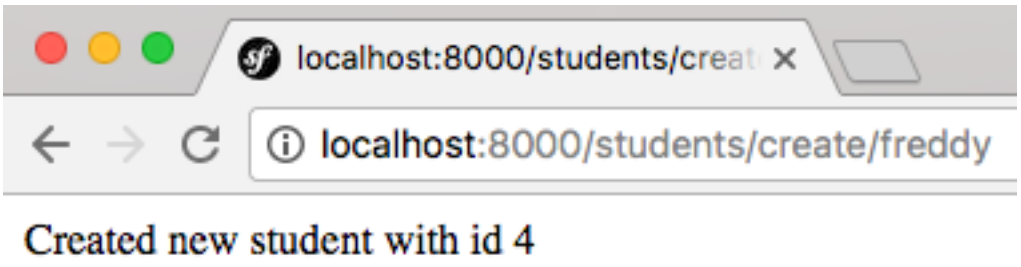


Figure 6.1: Creating new student via route `/students/create/{name}`.

We can see these records in our database. Figure 6.2 shows our new `students` table created for us.

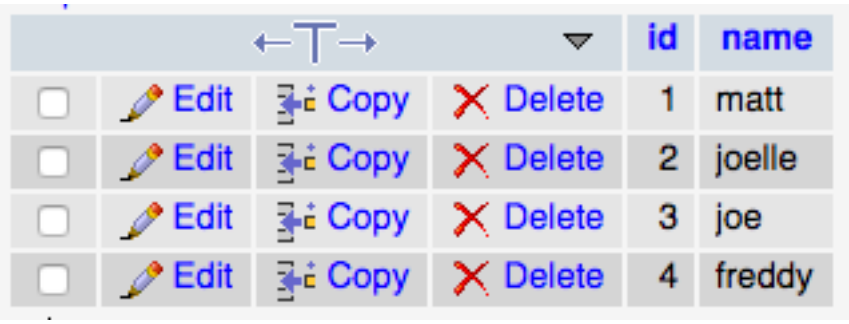


Figure 6.2: Controller created records in PHPMyAdmin.

6.2 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 6.3 shows how a new record `freddy` is added to the database table via route `/students/create/{name}`.

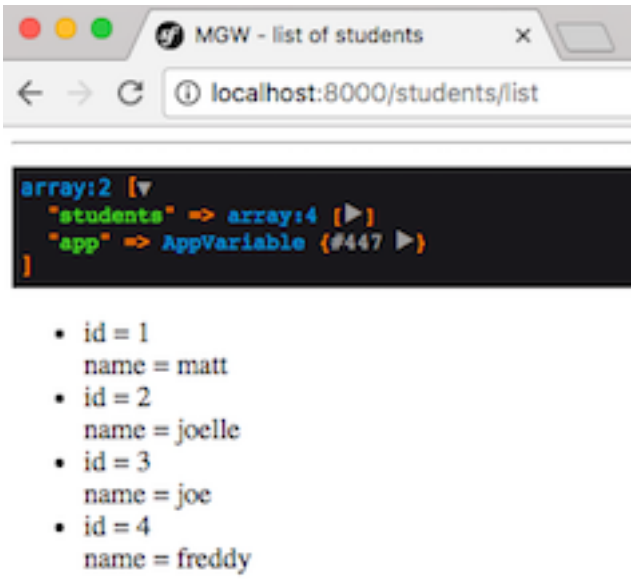


Figure 6.3: Listing all database student records with route `/students/list`.

6.3 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 INSERT query)
```

```
        $em->flush();

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

6.4 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 6.4 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)`). This is an exmaple of using the 'fluent' interface[1] offerede by Doctrine (where methods finish

[1] read about it at Wikipedia

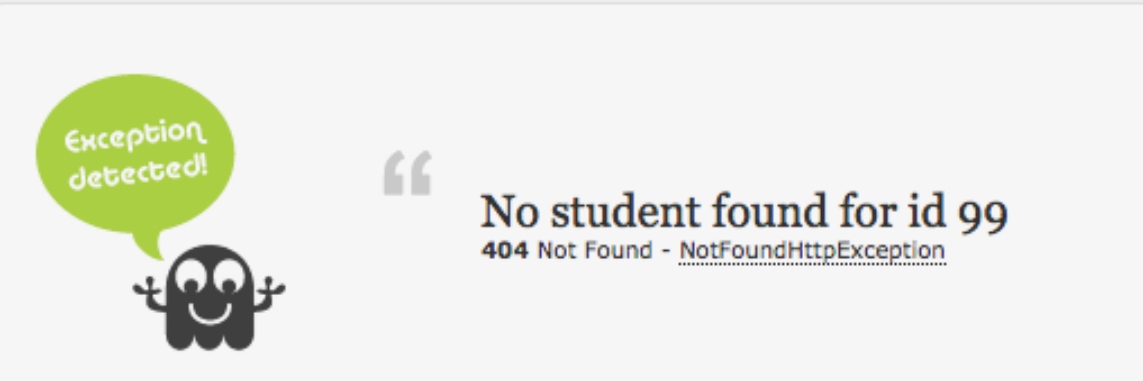


Figure 6.4: Listing all database student records with route `/students/list`.

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 …

6.5 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
```

7

Completing CRUD and linking things together

7.1 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

7.2 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
}
```

7.3 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 ("
```

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 7.1 shows our list of students with the names as links.

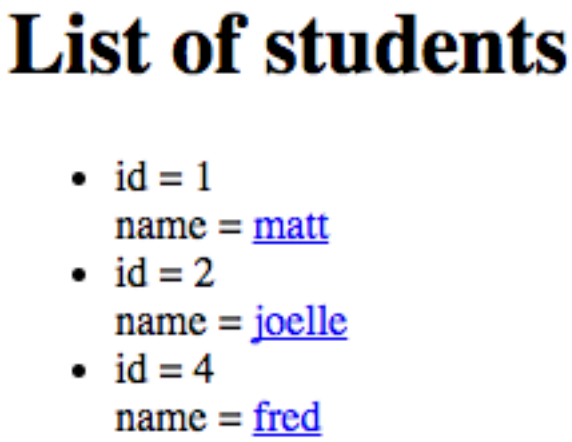


Figure 7.1: List of students with names as link to show pages.

Part III

Froms and form processing

8

DIY forms

8.1 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 8.1.

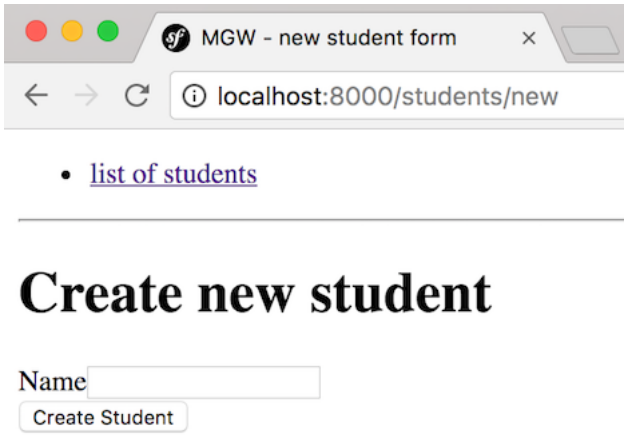


Figure 8.1: Form for a new student

8.2 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 %}
```

8.3 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 %}
```

8.4 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.

8.5 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.

8.6 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 %}
```

8.7 Adding validation in our 'processNewFormAction()z method

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

8.8 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>
```

8.9 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 ...)
```

9

Automatic forms generated from Entities

9.1 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…

9.2 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 9.1 shows a screenshot of the resulting form:

If we look further down (see Figure 9.2) 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.

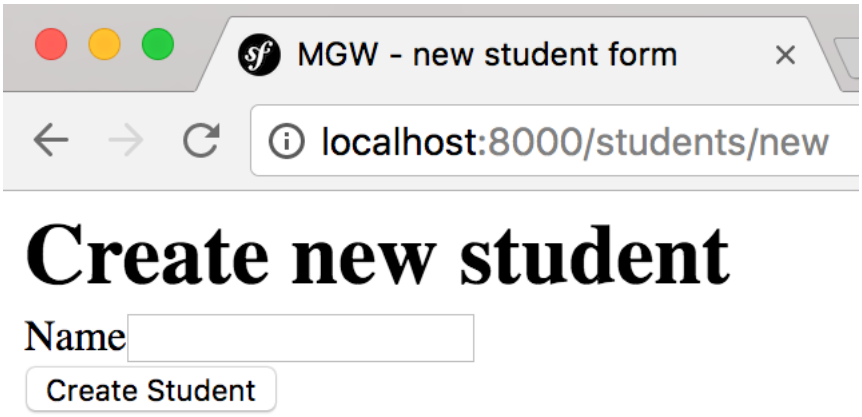


Figure 9.1: Symfony generated new student form (showing footer profiler bar).

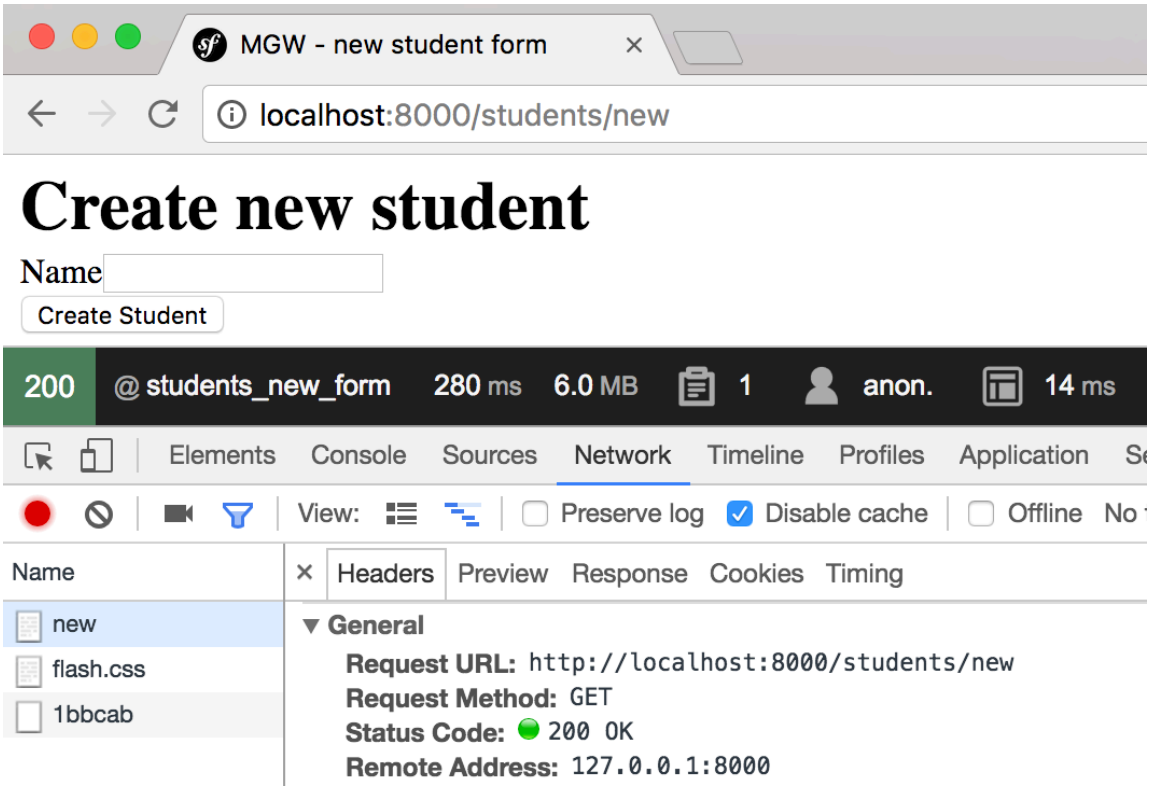


Figure 9.2: Generated student form - showing footer profiler bar.

9.3 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 9.3 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).

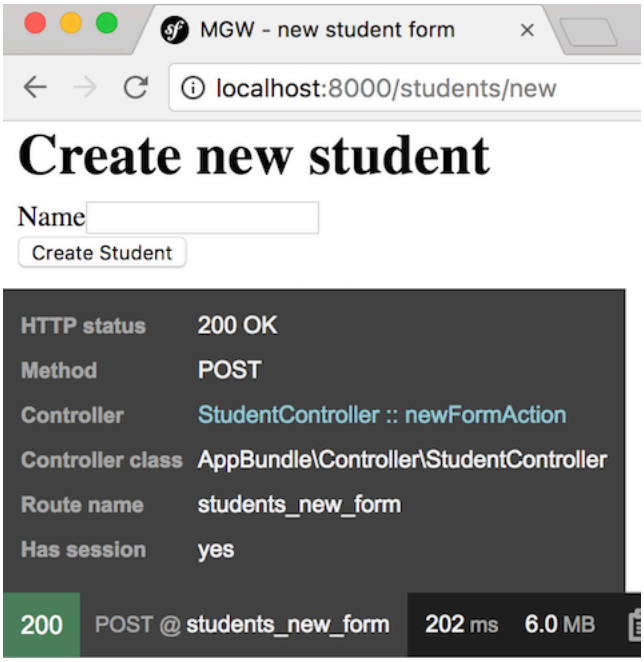


Figure 9.3: Form re-displayed depsite `POST` submission of name `joe-smith`.

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="TJM9iQSmrWWdYLVcbf1J15-
```

```
</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 9.4.

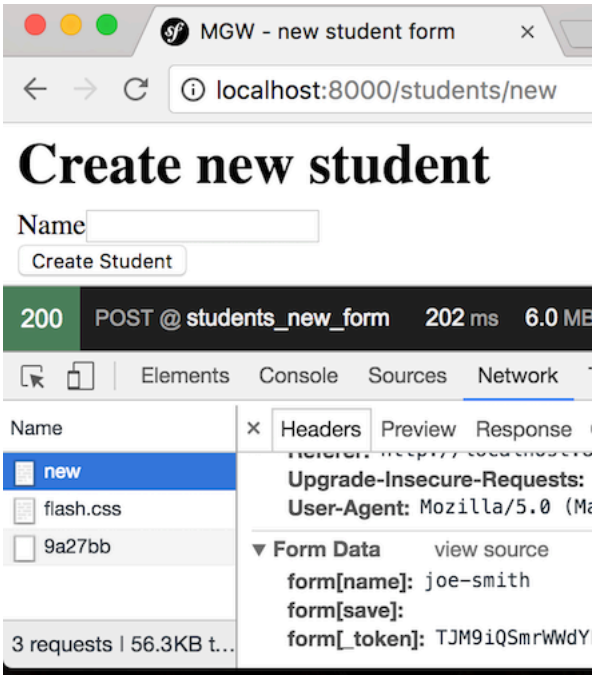


Figure 9.4: Chrome developer tools showing `POST` submitted variable `joe-smith`.

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 9.5 shows us the `POST` variables received.

[1] read more at the Wikipedia postback page

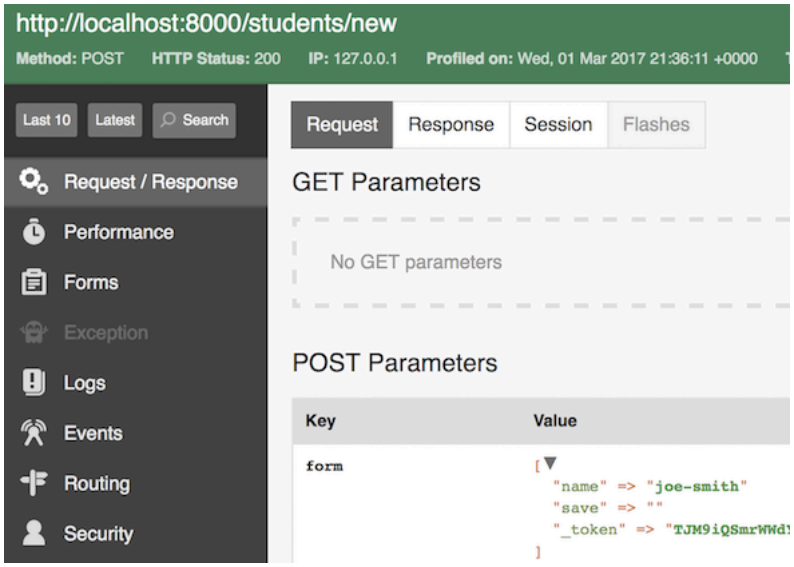


Figure 9.5: Chrome developer tools showing POST submitted variable `joe-smith`.

9.4 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 postback)

 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 9.6.

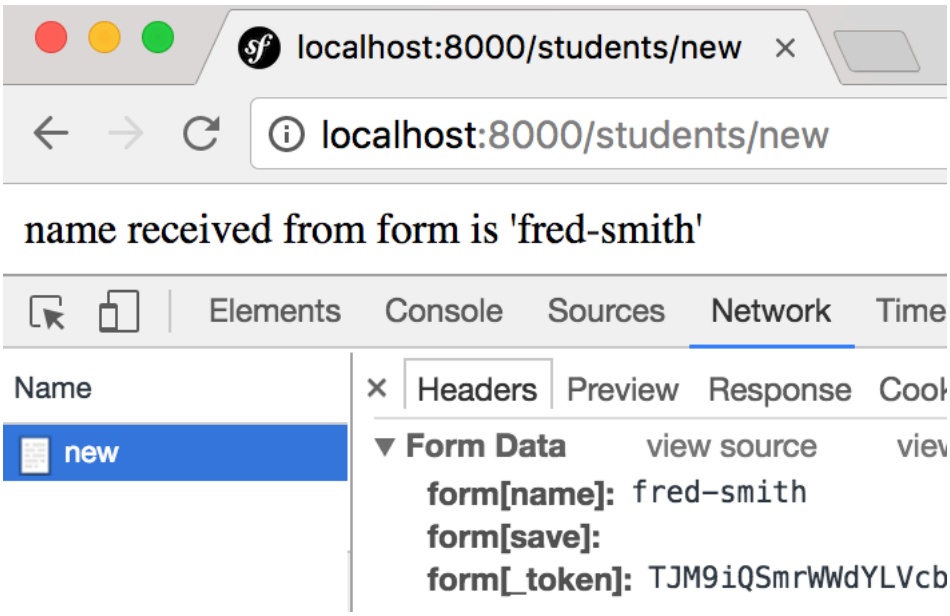


Figure 9.6: Confirmation of postback received name`joe-smith`.

9.5 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');
    }
```

9.6 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);
}
```

10

Customising the display of generated forms

10.1 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.

10.2 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

10.3 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 %}
```

10.4 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>
```

10.5 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 10.1 shows what our CSS styled form looks like to the user.

Figure 10.1: Browser rendering of generated form with CSS.

Learn more at:

- The Symfony form customisation page

10.6 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

Appendices

A

Steps to download code and get website up and running

A.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

A.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`

A.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`

A.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`

A.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/`

B

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
```

C

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 :).

 $
```

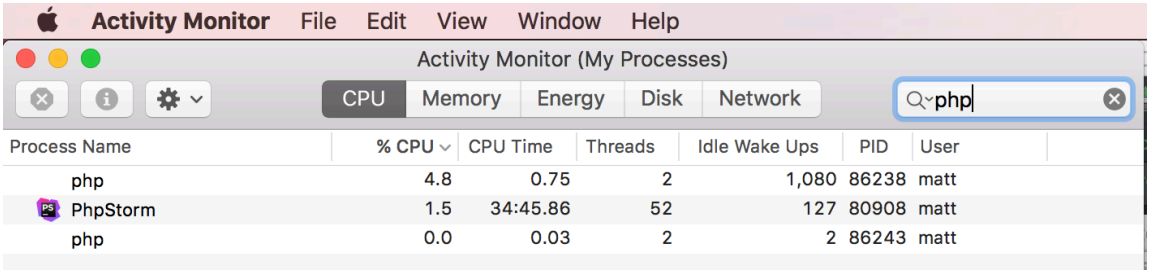
D

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