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?

If you're not using a database engeine 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 oproject (i.e. the same directory that has your `composer.json` file), and type the following:

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

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.

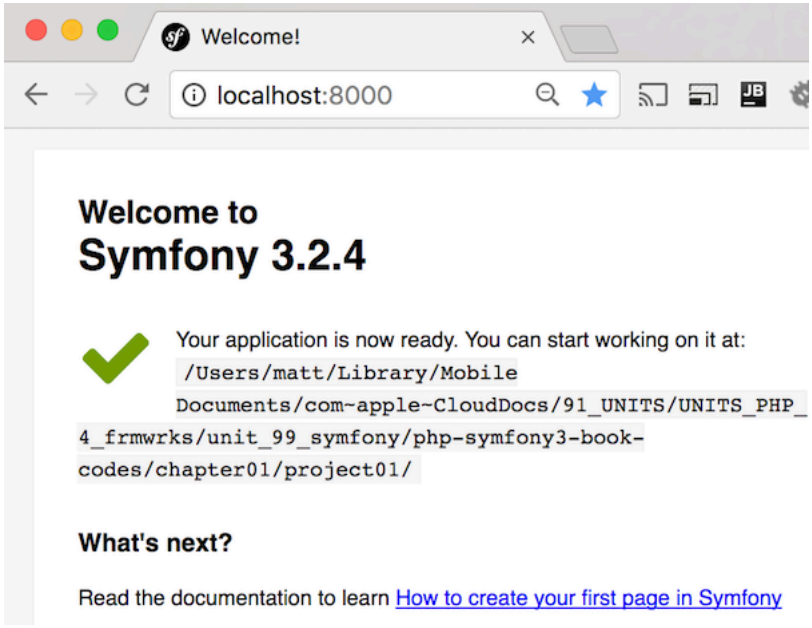


Figure 2.1: New Symfony project home page.

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
    ];
```

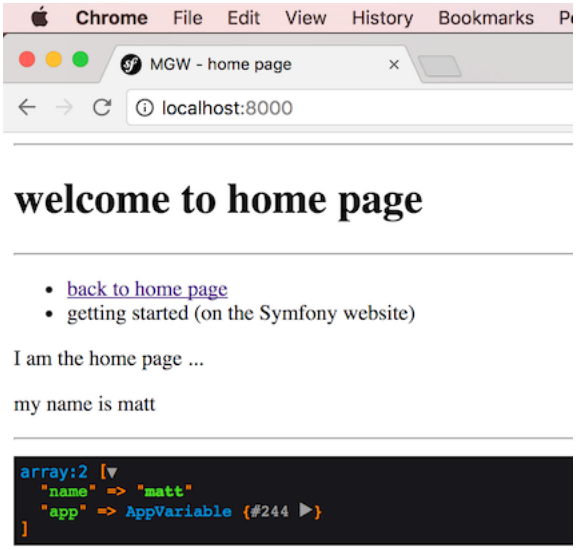


Figure 2.2: New home page.

```
        $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.

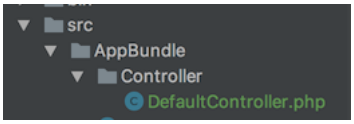


Figure 2.3: Location of Controller classes.

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

method:

```
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
```

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

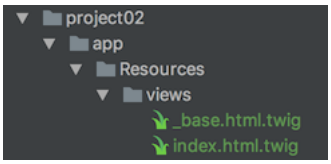


Figure 2.4: Location of Twig templates.

Here is our `_base.html.twig` template:

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

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:

```
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.

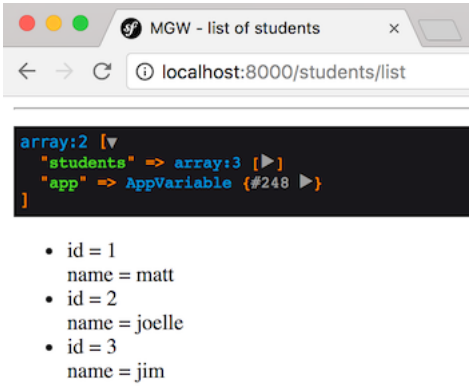


Figure 3.1: Students lists page.

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.

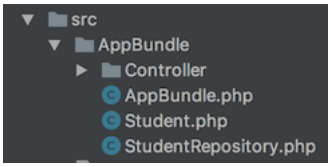


Figure 3.2: Location of Student and StudentRepository classes.

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.

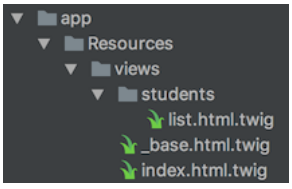


Figure 3.3: Location of Twig template `list.html.twig`.

```
{% 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.

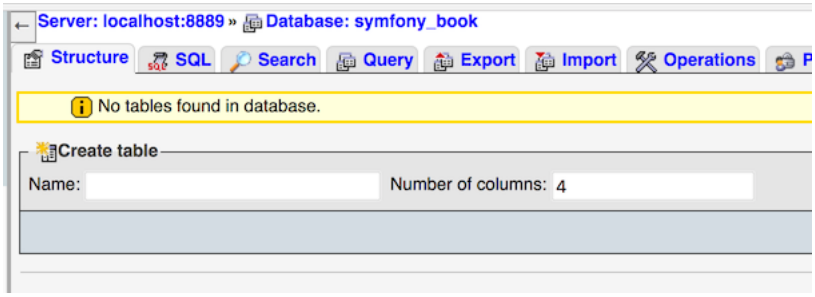


Figure 4.1: CLI created database in PHPMyAdmin.

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 …

A

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
```

A

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 :).

 $
```

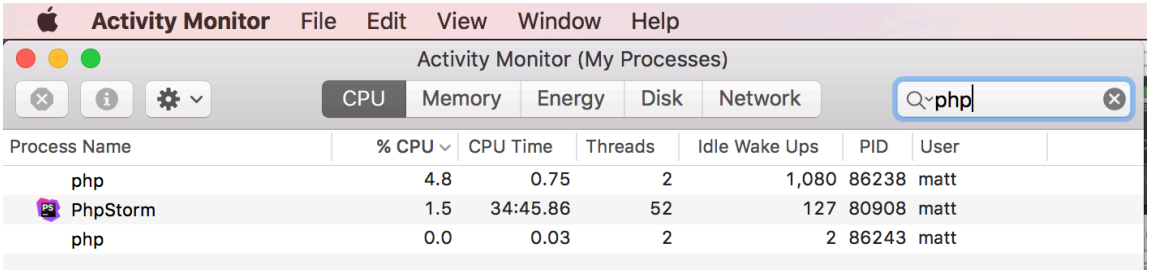
A

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