

AN INTRODUCTION TO SYMFONY 6
(for people that already know OO-PHP and some MVC stuff)

by
Matt Smith, Ph.D.
<https://github.com/dr-matt-smith>

Acknowledgements

Thanks to Ryan Weaver, for suggesting I update things to Symfony 6 in 2022 :-)

Thanks to the PHP and Symfony open-source international communities.

Table of Contents

Acknowledgements	i
I Introduction to Symfony	1
1 Introduction	3
1.1 What is Symfony 5?	3
1.2 What to I need on my computer to get started?	3
1.3 Check sysstem requiremnets	4
1.4 How to I get started with a new Symfony project	4
1.5 Where are the projects accompanying this book?	5
1.6 How to I run a Symfony webapp?	5
1.6.1 From the CLI	5
1.6.2 From a Webserver application (like Apache or XAMPP)	6
1.7 It isn't working! (Problem Solving)	7
1.8 Can I see a demo project with lots of Symfony features?	7
1.9 Any free videos about SF to get me going?	7
2 First steps	9
2.1 What we'll make (basic01)	9
2.2 Create a new Symfony project	10
2.3 List the routes	13
2.4 Create a controller	13
2.5 Run web server to visit new default route	16
2.6 Other types of Response content	17
2.7 The default Twig page	17
3 Twig templating	19
3.1 Customizing the Twig output (basic02)	19
3.2 Specific URL path and internal name for our default route method	20
3.3 Clearing the cache	22
3.4 Let's create a nice Twig home page	22

4	Creating our own classes	25
4.1	Goals	25
4.2	Let's create an Entity <code>Student</code> (<code>basic03</code>)	25
4.3	Create a <code>StudentController</code> class	26
4.4	The show student template <code>/templates/student/show.html.twig</code>	28
4.5	Twig debug <code>dump(...)</code> function	29
4.6	Creating an Entity Repository (<code>basic04</code>)	31
4.7	The student list controller method	32
4.8	The list student template <code>/templates/student/list.html.twig</code>	33
4.9	Refactor show action to show details of one <code>Student</code> object (project <code>basic05</code>)	34
4.10	Adding a <code>find(\$id)</code> method to the student repository	36
4.11	Make each item in list a link to show	36
4.12	Dealing with not-found issues (project <code>basic06</code>)	39
II	Symfony and Databases	41
5	Doctrine the ORM	43
5.1	What is an ORM?	43
5.2	Setting the database connection URL for MySQL	44
5.3	Setting the database connection URL for SQLite	44
5.4	Quick start	45
5.5	Make your database	45
6	Working with Entity classes	47
6.1	A <code>Student</code> DB-entity class (project <code>db01</code>)	47
6.2	Using annotation comments to declare DB mappings	47
6.3	Declaring types for fields	48
6.4	Validate our annotations	48
6.5	The <code>StudentRepository</code> class (<code>/src/Repository/StudentRepository</code>)	49
6.6	Create a migration (a migration <code>diff</code> file)	50
6.7	Run the migration to make the database structure match the entity class declarations	50
6.8	Re-validate our annotations	52
6.9	Generating entities from an existing database	52
6.10	Note - use maker to save time (project <code>db02</code>)	52
6.11	Use maker to create properties, annotations and accessor methods!	53
7	Symfony approach to database CRUD	57
7.1	Creating new student records (project <code>db02</code>)	57
7.2	Query database with SQL from CLI server	60
7.3	Updating the <code>listAction()</code> to use Doctrine	60
7.4	Deleting by id	61

TABLE OF CONTENTS

7.5	Updating given id and new name	62
7.6	Updating our show action	63
7.7	Redirecting to show after create/update	64
7.8	Given id let Doctrine find Product automatically (project db03)	65
7.9	Creating the CRUD controller automatically from the CLI (project db04)	68
8	Fixtures - setting up a database state	71
8.1	Initial values for your project database (project db05)	71
8.2	Installing and registering the fixtures bundle	71
8.2.1	Install the bundle	71
8.3	Writing the fixture classes	72
8.4	Loading the fixtures	73
8.5	User Faker to generate plausible test data (project db06)	75
8.6	Foundry and FakerPHP	77
9	Relating object in different Fixture classes	79
9.1	Related objects - option 1 - do it all in one Fixture class	79
9.2	Related objects - option 2 - store references to fixture objects	79
9.3	Category class	80
9.4	CategoryFixtures class	80
9.5	Product class	81
9.6	ProductFixtures class	81
10	Alternative approach using the DependentFixturesInterface	85
III	Forms and form processing	87
11	DIY forms	89
11.1	Preparation	89
11.2	Adding a form for new Student creation (project form01)	89
11.3	Refactor our create(...) method	90
11.4	Twig new student form	91
11.5	Controller method (and annotation) to display new student form	91
11.6	Controller method to process POST form data	92
11.7	Validating form data, and displaying temporary 'flash' messages in Twig	93
11.8	Three kinds of flash message: notice, warning and error	93
11.9	Adding flash display (with CSS) to our Twig template (project form02)	94
11.10	Adding validation logic to our form processing controller method	95
11.11	Postback logic (project form03)	96
11.12	Extra notes	99
12	Automatic forms generated from Entities	101

12.1 Using the Symfony form generator (project form04)	101
12.2 The Symfony form generator via Twig	101
12.3 Updating StudentController->new()	102
12.4 Postback - form submits to same URL	105
12.5 Using form classes (project form05)	106
12.6 Video tutorials about Symfony forms	109
13 Customising the display of generated forms	111
13.1 First let's Bootstrap this project (project form06)	111
13.2 Configure Twig to use the Bootstrap theme	111
13.3 Add the Bootstrap CSS import into our base Twig template	112
13.4 Add the Bootstrap JavaScript import into our base Twig template.	112
13.5 Run site and see some Bootstrap styling	113
13.6 Adding elements for navigation and page content	115
13.7 Add Bootstrap navigation bar	116
13.8 Styling list of links in navbar	118
13.9 Adding the hamburger-menu and collapsible links	119
14 Customizing display of Symfony forms	121
14.1 Understanding the 3 parts of a form (project form07)	121
14.2 Using a Twig form-theme template	122
14.3 DIY (Do-It-Yourself) form display customisations	122
14.4 Customising display of parts of each form field	123
14.5 Specifying a form's method and action	124
IV Custom Repository Queries with forms	127
15 Custom database queries	129
15.1 Search for exact property value (project query01)	129
15.2 Fixtures	129
15.3 Add new route and controller method for category search	131
15.4 Aside: How to the free 'helper' Doctrine methods work?	132
15.5 Testing our search by category	135
16 Custom database queries	139
16.1 Search Form for exact property value (project query02)	139
16.2 The form in Twig template	141
16.3 Controller method to process form submission	142
16.4 Getting rid of the URL search route	143
17 Wildcard vs. exact match queries	145
17.1 Search Form for partial description match (project query03)	145

TABLE OF CONTENTS

17.2 Custom queries in our Repository class	146
17.3 Making wildcard a sticky form	148
V Symfony code generation	149
18 CRUD controller and templates generation	151
18.1 Symfony's CRUD generator (project crud-01)	151
18.2 What you need to add to your project	151
18.3 Generating new Entity class Category	152
18.4 Generating CRUD for a new Entity class	152
18.5 The generated routes	153
18.6 The generated CRUD controller	154
18.7 The generated index (a.k.a. list) controller method	155
18.8 The generated new() method	158
18.9 The generated show() method	158
18.10The generated edit() method	159
18.11The generated delete() method	160
VI Sessions	163
19 Introduction to Symfony sessions	165
19.1 Create a new project from scratch (project sessions01)	165
19.2 Default controller - hello world	165
19.3 Twig foreground/background colours (sessions02)	166
19.4 Working with sessions in Symfony Controller methods (project session03)	169
19.5 Symfony's 2 session 'bags'	170
19.6 Storing values in the session in a controller action	171
19.7 Twig function to retrieve values from session	172
19.8 Attempt to read colors array property from the session	172
19.9 Applying colours in HTML head <style> element (project session04)	174
19.10Testing whether an attribute is present in the current session	176
19.11Removing an item from the session attribute bag	176
19.12Clearing all items in the session attribute bag	176
20 Working with a session 'basket' of products	177
20.1 Shopping cart of products (project session05)	177
20.2 Create a new project with the required packages	177
20.3 Create a Product entity & generate its CRUD	178
20.4 Homepage - link to products home	178
20.5 Basket index: list basket contents (project sessions07)	179
20.6 Controller method - clear()	180

20.7 Debugging sessions in Twig	181
20.8 Adding a object to the basket	182
20.9 The delete action method	183
20.10The Twig template for the basket index action	184
20.11Adding useful links to our <code>base.html.twig</code> template	187
20.12Adding the ‘add to basket’ link in the list of products	187
 VII Security and Authentication	 189
 21 Quickstart Symfony security	 191
21.1 Learn about Symfony security	191
21.2 New project with open and secured routes (project <code>security01</code>)	192
21.3 Create new project and add the security bundle library	192
21.4 Make a Default controller	192
21.5 Make a secured Admin controller	193
21.6 Core features about Symfony security	194
21.7 Generating the special <code>User</code> Entity class (project <code>security02</code>)	196
21.8 Review the changes to the <code>/config/packages/security.yml</code> file	196
21.9 Migrate new <code>User</code> class to your database	197
21.10Make some <code>User</code> fixtures	197
21.11Run and check your fixtures	199
21.12Creating a Login form	200
21.13Check the new routes	200
21.14Allow <code>any</code> user to view the login form	201
21.15Clear cache & visit <code>/admin</code>	202
21.16Using the <code>/logout</code> route	204
21.17Finding and using the internal login/logout route names in <code>SecurityController</code> . .	204
 22 Security users from database	 207
22.1 Improving <code>UserFixtures</code> with a <code>createUser(...)</code> method (project <code>security03</code>) . .	207
22.2 Loading the fixtures	208
22.3 Using SQL from CLI to see users in DB	209
 23 Custom login page	 211
23.1 A D.I.Y. (customisable) login form (project <code>security04</code>)	211
23.2 Simplifying the generated login Twig template	211
23.3 CSRF (Cross Site Request Forgery) protection	212
23.4 Display any errors	213
23.5 Custom login form when attempting to access <code>/admin</code>	214
23.6 Path for successful login	214
 24 Custom <code>AccessDeniedException</code> handler	 217

TABLE OF CONTENTS

24.1	Symfony documentation for 403 access denied exception	217
24.2	Declaring our handler (project security05)	217
24.3	The exception handler class	218
25	Twig and logging	221
25.1	Getting reference to Twig and Logger objects	221
25.2	Using Twig for access denied message (project security06)	221
25.3	The Twig page	222
25.4	Terminal log	224
25.5	Learn more about logger and exceptions	224
26	User roles and role hierarchies	225
26.1	Simplifying roles with a hierarchy (project security07)	225
26.2	Modify fixtures	226
26.3	Removing default adding of ROLE_USER if using a hierarchy	226
26.4	Allowing easy switching of users when debugging	227
27	Customising view based on logged-in user	229
27.1	Twig nav links when logged in (project security08)	229
27.2	Getting reference to the current user in a Controller	231
28	Simplifying roles and adding secure User CRUD (project security09)	233
28.1	User CRUD PROBLEM 1: an array of ROLES	233
28.2	User CRUD PROBLEM 1: a solution	233
28.3	User CRUD PROBLEM 2: plain test password stored in DB	235
28.4	User CRUD PROBLEM 2: a solution	235
28.5	Securing the User CRUD for ROLE_ADMIN only	237
28.6	Further steps	237
VIII	Entity associations (one-to-many relationships etc.)	239
29	Database relationships (Doctrine associations)	241
29.1	Information about Symfony 4 and databases	241
29.2	Create a new project from scratch (project associations01)	241
29.3	Categories for Products	242
29.4	Defining the many-to-one relationship from Product to Category	242
29.5	How to allow null for a Product's category	243
29.6	Adding the optional one-to-many relationship from Category to Product	244
29.7	Create and migrate DB schema	244
29.8	Generate CRUD for Product and Category	245
29.9	Add Category selection in Product form	245
29.10	Add small and large item Category	247

29.11 Drop-down menu of categories when creating/editing Products	247
29.12 Adding display of Category to list and show Product	248
29.13 <code>toString()</code> method	250
29.14 Setup relationship via make	250
30 Many-to-one (e.g. Products for a single Category)	253
30.1 Basic list products for current Category (project associations02)	253
30.2 Add getProducts() for Entity Category	253
30.3 Add a __toString() for Entity Products	254
30.4 Make Category form type add products property	254
30.5 Adding a nicer list of Products for Category show page	255
30.6 Improving the Edit form (project associations03)	258
30.7 Creating related objects as Fixtures (project associations04)	261
30.8 Using Joins in custom Repository classes	262
31 Logged-in user stored as item author	265
31.1 Getting User object for currently logged-in user	265
31.2 Simple example: Users and their county (associations05)	266
31.3 Add <code>toString</code> method to User	268
31.4 Use currently logged-in user as author	268
31.5 Protect CRUD so must be logged in	269
IX PHPDocumentor (2)	271
32 PHPDocumentor	273
32.1 Why document code?	273
32.2 Self-documenting code	273
32.3 PHPDocumentor 2	274
32.4 Installing PHPDocumentor 2 - the PHAR	274
32.5 Installing PHPDocumentor 2 - via Composer	274
32.6 DocBlock comments	274
32.7 Generating the documentation	275
32.8 Using an XML configuration file phpdoc.dist.xml	275
32.9 WARNING - PHPStorm default comments	276
32.10 TODO - special treatment	277
X Symfony Testing with Codeception	279
33 Unit testing in Symfony with Codeception	281
33.1 Codeception Open Source BDD project	281
33.2 Adding Codeception to an existing project (project codeception01)	281

TABLE OF CONTENTS

33.3	Installing the required modules	284
33.4	What Codeception has added to our project	285
34	Acceptance Tests	287
34.1	Test for home page text at / (project <code>codeception02</code>)	287
34.2	Run the test (fail - server not running)	288
34.3	Run the test (pass, when server running)	289
34.4	From red to green	289
34.5	Make green - add link to about page in base Twig template	291
34.6	Annotation style data provider to test multiple data (project <code>codeception03</code>)	291
34.7	Traditional Data Provider syntax	293
34.8	Common assertions for Acceptance tests	293
35	Filling out forms	295
35.1	Setup database (project <code>codeception04</code>)	295
35.2	View the Twig template for a new Recipe	295
35.3	Cest to enter a new recipe	296
36	Codeception Symfony DB testing	299
36.1	Adding Symfony and Doctrine to the settings (project <code>codeception05</code>)	299
36.2	Test Users in DB from Fixtures	300
36.3	Check DB reset after each test	301
36.4	Counting items retrieved from DB (project <code>codeception06</code>)	302
36.5	Create a Recipe fixture	302
36.6	Comparing Repository counts BEFORE and AFTER DB actions	303
36.7	Testing properties of items added to the database	304
37	Testing user role authentication	307
37.1	Automating user login (project <code>codeception07</code>)	307
37.2	Testing login & link visibility with helper methods	307
37.3	Testing access denied messages	310
38	Testing roles by creating and testing new users	313
38.1	Automatic User Crud (project <code>codeception08</code>)	313
38.2	Testing number of users in DB repository increases by 1 after creating	314
38.3	Test we can only see new user in DB Repository AFTER creating it	315
38.4	Testing we can login with created users	316
38.5	Test new user with <code>ROLE_ADMIN</code> can visit secured admin home page	317
XI	Publishing Symfony websites	319
39	Publishing your Symfony website	321

39.1 Requirements for Symfony publishing	321
39.2 Simplest ways to host Symfony projects	322
40 Setting up project ready for Fortrabbit	325
40.1 Updating the names of our MySQL variables to match Fortrabbit ones	325
40.2 Create new DB locally, and make fresh migrations	326
40.3 Creating the <code>/public/.htaccess</code> Apache server routing file	326
41 Publishing with Fortrabbit.com	329
41.1 Main steps for PAAS publishing with Fortrabbit	329
41.2 Create a new Fortrabbit Symfony project	330
41.3 Temporarily set project environment to <code>dev</code> so we can load DB fixtures	332
41.4 Getting a linked Git project on your local computer	333
41.5 Visit site to see if published (although may be DB errors)	336
41.6 Connect command-line terminal to Fortrabbit project via SSH	338
41.7 Use SSH to run DB migrations	338
41.8 Use SSH to load fixtures	339
41.9 Use Doctrine query to check DB contents	340
41.10MySQL queries using SSH tunnel	340
41.11Published website	341
XII Symfony Testing	343
42 Unit testing in Symfony	345
42.1 Testing in Symfony	345
42.2 Installing Simple-PHPUnit (project <code>test01</code>)	345
42.3 Completing the installation	346
42.4 Running Simple-PHPUnit	347
42.5 Testing other classes (project <code>test02</code>)	348
42.6 The class to test our calculator	348
42.7 Using a data provider to test with multiple datasets (project <code>test03</code>)	350
42.8 Configuring testing reports (project <code>test04</code>)	351
42.9 Testing for exceptions (project <code>test07</code>)	353
42.10PHPUnit <code>expectException(...)</code>	355
42.11PHPUnit annotation comment <code>@expectedException</code>	356
42.12Testing for custom Exception classes	356
42.13Checking Types with assertions	358
42.14Same vs. Equals	358
43 Code coverage and xDebug	361
43.1 Code Coverage	361
43.2 Generating Code Coverage HTML report	362

TABLE OF CONTENTS

43.3 Tailoring the ‘whitelist’	364
44 Web testing	365
44.1 Testing controllers with <code>WebTestCase</code> (project <code>test05</code>)	365
44.2 Automating a test for the home page contents	366
44.3 Normalise content to lowercase (project <code>test06</code>)	368
44.4 Test multiple pages with a data provider	369
44.5 Testing links (project <code>test08</code>)	370
44.6 Issue with routes that end with a forward slash /	373
44.6.1 Solution 1: Ensure url pattern in test method exactly matches router url pattern	374
44.6.2 Solution 2: Instruct client to ‘follow redirects’	374
45 Testing web forms	377
45.1 Testing forms (project <code>test09</code>)	377
45.2 Test we can get a reference to the form	381
45.3 Submitting the form	382
45.4 Entering form values then submitting	382
45.5 Testing we get the correct result via form submission	384
45.6 Selecting form, entering values and submitting in one step	385
 XIII Appendices	 387
A Software required for Symfony development	389
A.1 Don’t confuse different software tools	389
A.2 Software tools	390
A.3 Test software by creating a new Symfony 4 project	390
 B PHP Windows setup	 391
B.1 Check if you have PHP installed and working	391
B.1.1 Download the latest version of PHP	391
B.2 Add the <code>path</code> to <code>php.exe</code> to your System environment variables	394
B.3 Check your <code>php.ini</code> file	396
B.4 PHP Info & SQL driver test	396
 C The Composer andd Symfonhy command line tools	 399
C.1 Composer	399
C.1.1 Windows Composer install	399
C.1.2 Windows Composer install	400
C.2 Symfony command line tool	400
 D Software tools	 401

D.1	PHPStorm editor	401
D.2	MySQL Workbench	401
D.3	Git	402
D.4	Git Windows installation	403
E	The fully-featured Symfony 4 demo	405
E.1	Visit Github repo for full Symfony demo	405
E.2	Git steps for download (clone)	405
E.3	Non-git download	406
E.4	Install dependencies	406
E.5	Run the demo	406
E.6	View demo in browser	407
E.7	Explore the code in PHPStorm	407
E.8	Switch demo from SQLite to MySQL	407
E.9	Running the tests in the SF4 demo	409
E.10	Run the tests	409
E.11	Explore directory <code>/tests</code>	410
E.12	Learn more	411
F	Solving problems with Symfony	413
F.1	No home page loading	413
F.2	“Route not Found” error after adding new controller method	414
F.3	Issues with timezone	414
F.4	Issues with Symfony 3 and PHPUnit.phar	414
F.5	PHPUnit installed via Composer	415
G	Publish via Fortrabbbit (PHP as a service)	417
G.1	SSH key	417
G.1.1	Windows SSH key setup	417
G.1.2	Mac SSH key setup	417
G.1.3	Linux SSH key setup	417
G.1.4	Fortrabbbit	418
G.2	Creating a new web App	420
G.3	Cloning and populating your Git repo	422
G.4	Fixing the Fixtures issue	424
G.5	Fixing the Apache <code>.htaccess</code> issue	425
G.6	Adding, committing and pushing the project files to the repo	425
G.7	SSH CLI Terminal to migrate and install DB fixtures	428
G.8	Symfony project should now be fully deployed	431
H	Quick setup for new ‘blog’ project	433
H.1	Create a new project, e.g. ‘blog’	433

TABLE OF CONTENTS

H.2	Set up your localhost browser shortcut to for <code>app_dev.php</code>	433
H.3	Add <code>run</code> shortcut to your <code>Composer.json</code> scripts	433
H.4	Change directories and run the app	434
H.5	Remove default content	434
I	Steps to download code and get website up and running	435
I.1	First get the source code	435
I.1.1	Getting code from a zip archive	435
I.1.2	Getting code from a Git repository	435
I.2	Once you have the source code (with vendor) do the following	436
I.3	Run the webserver	436
J	About <code>parameters.yml</code> and <code>config.yml</code>	437
J.1	Project (and deployment) specific settings in (<code>/app/config/parameters.yml</code>) . . .	437
J.2	More general project configuration (<code>/app/config/config.yml</code>)	438
K	Setting up for MySQL Database	439
K.1	Declaring the parameters for the database (<code>parameters.yml</code>)	439
L	Setting up for SQLite Database	441
L.1	NOTE regarding FIXTURES	441
L.2	SQLite suitable for most small-medium websites	441
L.3	Create directory where SQLite database will be stored	442
L.4	Declaring the parameters for the database (<code>parameters.yml</code>)	442
L.5	Setting project configuraetion to work with the SQLite database driver and path (<code>/app/config/config.yml</code>)	442
M	Setting up Adminer for DB GUI interaction	445
M.1	Adminer - small and simple DB GUI	445
M.2	Getting Adminer	446
M.3	Setting up	446
M.4	Running Adminer	446
N	Avoiding issues of SQL reserved words in entity and property names	449
O	Transcript of interactive entity generation	451
P	Killing ‘php’ processes in OS X	453
Q	Docker and Symfony projects	455
Q.1	Setup	455
Q.2	Dockerfile	455
Q.3	Build your Docker image	456
Q.4	Run a Container process based on your image (exposing HTTP port 80)	456

Q.5	Alternative Dockerfile for a basic PHP application, using Apache	456
Q.6	Create config file for Apache	457
Q.7	Other Docker reference stuff	457
Q.7.1	Docker Images	457
Q.7.2	Containers	458
Q.7.3	New Image from current (changed) state of a running Container	458
Q.7.4	Exposing HTTP ports for Containers running web application servers	459
Q.8	Useful reference sites	459
R	xDebug for Windows	461
R.1	Steps for Windows	461
R.2	Steps for Linux/Mac	461
R.3	Use the xDebug wizard!	462
R.4	PHP Function phpinfo()	463
R.5	More information	464
	List of References	465

Part I

Introduction to Symfony

1

Introduction

1.1 What is Symfony 5?

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

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

I recommend you install the following:

- PHP 8.1 (download/install from [php.net](https://www.php.net))
 - (it will work with 8.0, but 8.1 allows enumerations which is handy ...)
- a good text editor (I like [PHPStorm](#), but then it's free for educational users...)
- [Composer](#) (PHP package manager - a PHP program)
- the Symfony command-line tool
 - <https://symfony.com/download>

The following are also a good idea: - a MySQL database server - e.g. MySQLWorkbench Community is free and cross-platform - Git - see [GitforWindows](#)

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

or Symfony's (not free) SymfonyCloud PHP-as-a-Service (PaaS):

- <https://symfony.com/cloud/>

Learn more about the software needed for Symfony development in Appendix A. For steps in installing PHP and the other software, see Appendices B and D.

1.3 Check system requirements

Once you've installed the Symfony command-line tool, check your system setup with the `symfony check:requirements` command:

```
$ symfony check:requirements
```

```
Symfony Requirements Checker
```

```
~~~~~
```

```
> PHP is using the following php.ini file:
```

```
/usr/local/etc/php/8.1/php.ini
```

```
> Checking Symfony requirements:
```

```
.....
```

```
[OK]
```

```
Your system is ready to run Symfony projects
```

1.4 How to I get started with a new Symfony project

In a CLI (Command Line Interface) terminal window, `cd` into the directory where you want to create your Symfony project(s). Then create a new Symfony empty web application project, named `project01` (or whatever you wish) by typing:

```
$ symfony new --full project01
```

NOTE: If for some reason you don't have the Symfony command line tool installed, you can also create a project using Composer:

```
$ composer create-project symfony/website-skeleton project01
```

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

```
$ symfony new --full project01
* Creating a new Symfony project with Composer
  (running /usr/local/bin/composer create-project symfony/website-skeleton /Users/matt/project01)

* Setting up the project under Git version control
  (running git init /Users/matt/project01)

[OK] Your project is now ready in /Users/matt/project01
```

Another way to get going quickly with Symfony is to download one of the projects accompanying this book ...

1.5 Where are the projects accompanying this book?

All the projects in this book are freely available, as public repositories on Github as follows:

- <https://github.com/dr-matt-smith/php-symfony-6-book/codes>

To retrieve and setup a sample project follow these steps:

1. download the project to your local computer (e.g. `git clone URL`)
2. change (`cd`) into the created directory
3. type `composer install` to download any required 3rd-party packages into a `/vendor` folder
 - NOTE: `composer install` installs the **same** component versions as defined in the `composer.lock` file. `composer update` will attempt to install the **most up-to-date stable** versions of the components in the `composer.json` file.
4. Then run your web server (see below) and explore via a web browser

1.6 How to I run a Symfony webapp?

1.6.1 From the CLI

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

```
$ symfony serve
```

NOTE: This is short for `symfony server:start`

If you don't have the Symfony command line tool installed you could also use the PHP built-in web server:

```
$ php -S localhost:8000 -t public
```

Then open a web browser and visit the website root at `http://localhost:8000`.

See Figure 1.1 for a screenshot of the default Symfony 5 home page (with a message saying you've not configured a home page!).

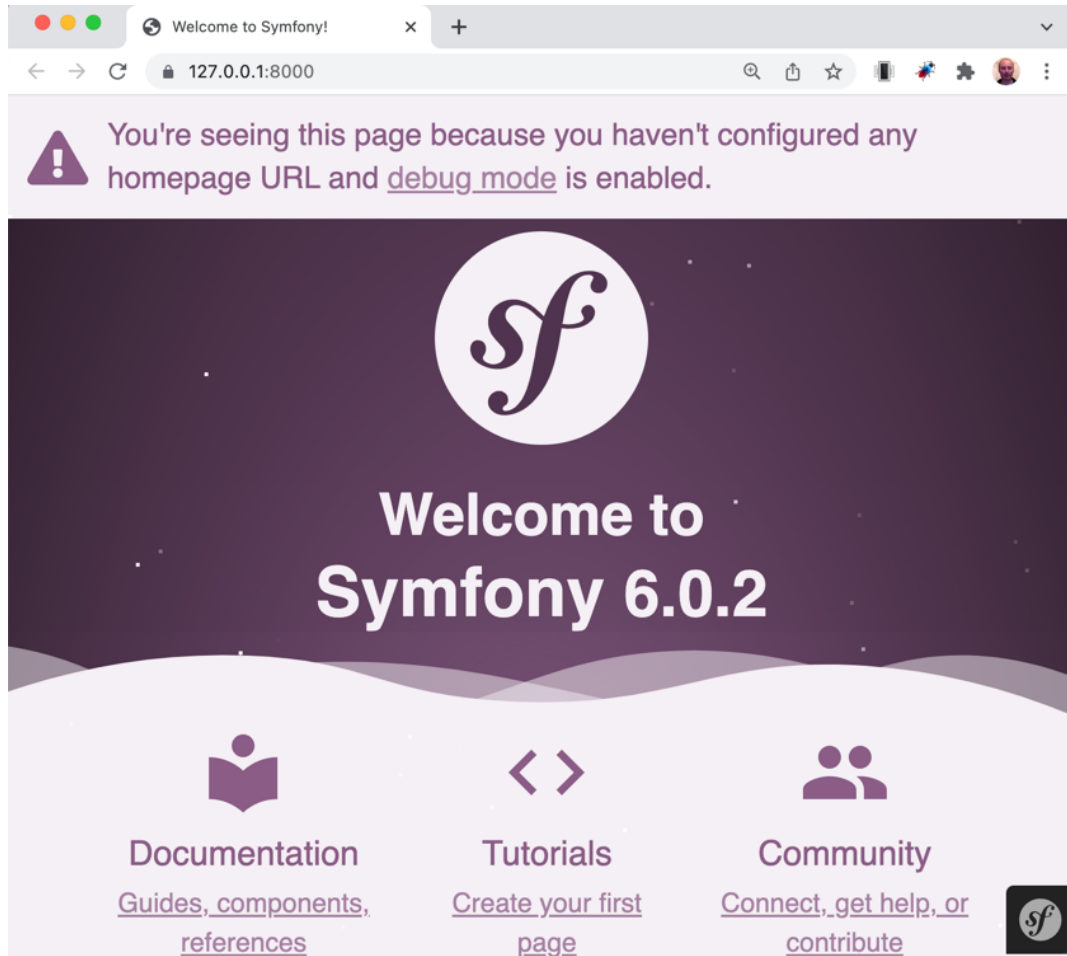


Figure 1.1: Screenshot default Symfony empty new project home page.

1.6.2 From a Webserver application (like Apache or XAMPP)

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

1.7 It isn't working! (Problem Solving)

If you have trouble with running Symfony, take a look at Appendix F, which lists some common issues and how to solve them.

1.8 Can I see a demo project with lots of Symfony features?

Yes! There is a full-featured Symfony demo project. Checkout Appendix E for details of downloading and running the demo and its associated automated tests.

1.9 Any free videos about SF to get me going?

Yes! Those nice people at Symfonycasts have released a bunch of free videos all about Symfony (and OO PHP in general).

So plug in your headphones and watch them, or read the transcripts below the video if you're no headphones. A good rule is to watch a video or two **before** trying it out yourself.

You'll find the video tutorials at:

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

(ask Matt to ask his contacts in Symfonycasts to try to get his students a month's free access ... if your Github Education Pack free access has expired ...)

2

First steps

2.1 What we'll make (basic01)

See Figure 2.1 for a screenshot of the new homepage we'll create in our first project (after some setup steps).



Figure 2.1: New home page.

There are 3 things Symfony needs to serve up a page:

1. a route
2. a controller class and method
3. a Response object to be returned to the web client

The first 2 can be combined, through the use of 'attributes' , which declare the route in a line beginning # immediately before the controller method defining the 'action' for that route. See this example:

```
#[Route('/', name: 'homepage')]
public function indexAction()
{
    ... build and return Response object here ...
}
```

For example the code below defines:

- a attribute Route comment for URL pattern / (i.e. website route)
 - `#[Route('/', name: 'homepage')]`
 - the Symfony “router” system attempts to match pattern / in the URL of the HTTP Request received by the server
- controller method `indexAction()`
 - this method will be involved if the route matches
 - controller method have the responsibility to create and return a Symfony **Response** object
- note, Symfony allows us to declare an internal name for each route (in the example above `homepage`)
 - we can use the internal name when generating URLs for links in out templating system
 - the advantage is that the route is only defined once (in the annotation comment), so if the route changes, it only needs to be changed in one place, and all references to the internal route name will automatically use the updated route
 - for example, if this homepage route was changed from / to `/default` all URLs generated using the `homepage` internal name would now generated `/default`

2.2 Create a new Symfony project

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

```
$ symfony new --full project01
```

```
* Creating a new Symfony project with Composer
... etc. ...
```

```
[OK] Your project is now ready in /Users/matt/Documents/Books/php-symfony-6-book/codes/;
```

```
$ cd basic01
```

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

```
$ symfony serve
Tailing Web Server log file (/Users/matt/.symfony/log/ec56398112e31dba20d3fec928509d0cec5c3764.1
Tailing PHP-FPM log file (/Users/matt/.symfony/log/ec56398112e31dba20d3fec928509d0cec5c3764/53f
WARNING read /Users/matt/.symfony/var/ec56398112e31dba20d3fec928509d0cec5c3764: is a directory

[OK] Web server listening
    The Web server is using PHP FPM 8.1.1
    https://127.0.0.1:8000

[Web Server ] Jan  2 18:53:06 |INFO    | PHP    listening path="/usr/local/Cellar/php/8.1.1/sbin
[PHP-FPM    ] Jan  2 18:53:06 |NOTICE  | FPM    ready to handle connections

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

Figure 2.2 shows a screenshot of the default page for the web root (path /), when we have no routes set up and we are in development mode (i.e. our `.env` file contains `APP_ENV=dev`).

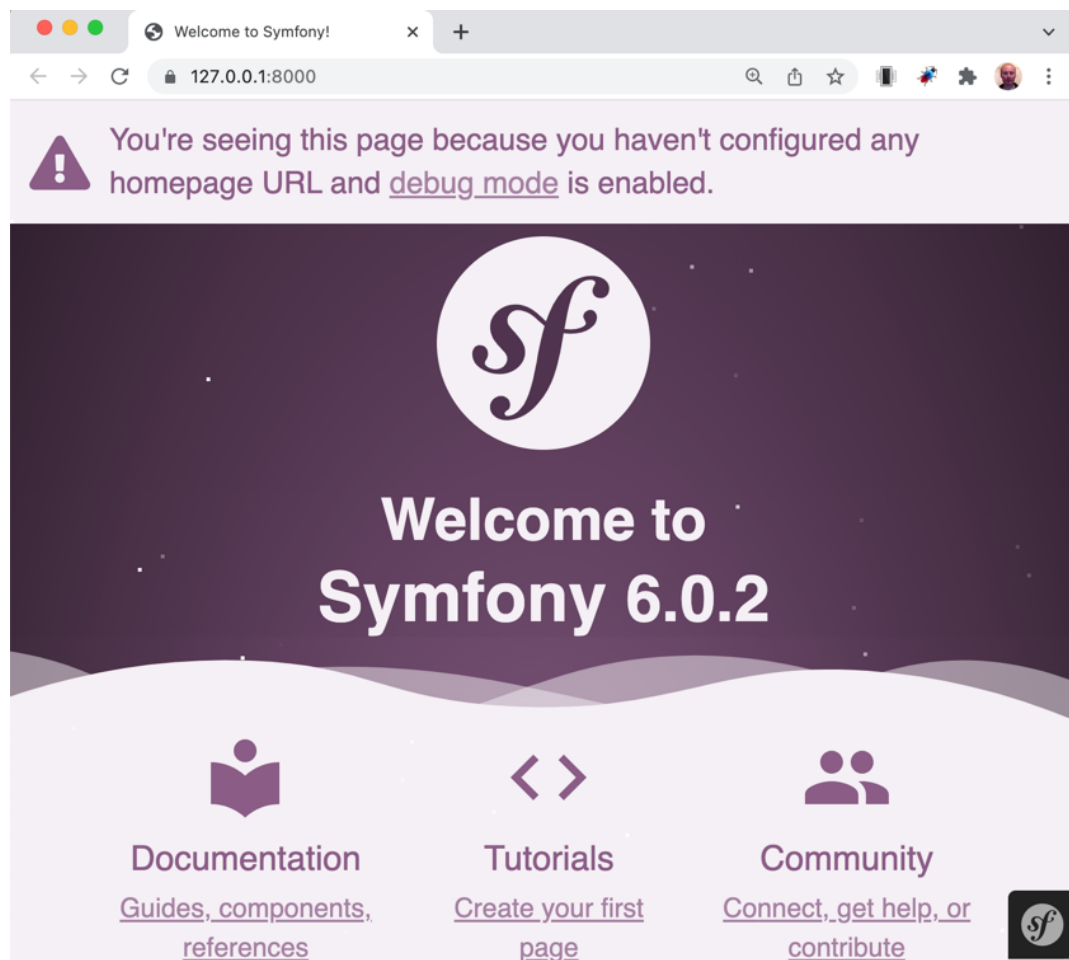


Figure 2.2: Screenshot default Symfony 4 page for web root (when no routes defined).

2.3 List the routes

There should not be any (non-debug) routes yet. All routes starting with an underscore `_` symbol are debugging routes used by the very useful Symfony profiler - this creates the information footer at the bottom of our pages when we are developing Symfony applications.

but let's check at the console by typing `php bin/console debug:router`:

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

Name	Method	Scheme	Host	Path
<code>_wdt</code>	ANY	ANY	ANY	<code>/_wdt/{token}</code>
<code>_profiler_home</code>	ANY	ANY	ANY	<code>/_profiler/</code>
<code>_profiler_search</code>	ANY	ANY	ANY	<code>/_profiler/search</code>
<code>_profiler_search_bar</code>	ANY	ANY	ANY	<code>/_profiler/search_bar</code>
<code>_profiler_phpinfo</code>	ANY	ANY	ANY	<code>/_profiler/phpinfo</code>
<code>_profiler_search_results</code>	ANY	ANY	ANY	<code>/_profiler/{token}/search/results</code>
<code>_profiler_open_file</code>	ANY	ANY	ANY	<code>/_profiler/open</code>
<code>_profiler</code>	ANY	ANY	ANY	<code>/_profiler/{token}</code>
<code>_profiler_router</code>	ANY	ANY	ANY	<code>/_profiler/{token}/router</code>
<code>_profiler_exception</code>	ANY	ANY	ANY	<code>/_profiler/{token}/exception</code>
<code>_profiler_exception_css</code>	ANY	ANY	ANY	<code>/_profiler/{token}/exception.css</code>
<code>_preview_error</code>	ANY	ANY	ANY	<code>/_error/{code}.{_format}</code>

NOTE:

- you can usually shorten the Symfony CLI commands to 1 of 2 letters, e.g. `debug:router` could be written `de:ro` ...

The only routes we can see all start with an underscore (e.g. `_preview_error`), so no application routes have been declared yet ...

2.4 Create a controller

We could write a new class for our homepage controller, but ... let's ask Symfony to make it for us. Typical pages seen by non-logged-in users like the home page, about page, contact details etc. are often referred to as 'default' pages, and so we'll name the controller class for these pages our `DefaultController`.

1. Tell Symfony to create a new homepage (default) controller. A since a class will be created

starting with the controller name, ensure your controller name starts with a CAPITAL letter, e.g. `Default` not `default`:

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

created: src/Controller/DefaultController.php
created: templates/default/index.html.twig

Success!
```

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

Symfony controller classes are stored in directory `/src/Controller`. We can see that a new controller class has been created named `DefaultController.php` in folder `/src/Controller`.

A second file was also created, a view template file `templates/default/index.html.twig`,

Look inside the generated class `/src/Controller/DefaultController.php`. It should look something like this:

```
<?php
namespace App\Controller;

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

class DefaultController extends AbstractController
{
    #[Route('/default', name: 'default')]
    public function index(): Response
    {
        return $this->render('default/index.html.twig', [
            'controller_name' => 'DefaultController',
        ]);
    }
}
```

This default controller uses a **Twig** template to return an HTML page:

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

{% block title %}Hello DefaultController!{% endblock %}

{% block body %}
```



```
<style>
    .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font: 18px/1.5 sans-serif; }
    .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
</style>

<div class="example-wrapper">
    <h1>Hello {{ controller_name }}! TICK</h1>

    This friendly message is coming from:
    <ul>
        <li>Your controller at <code><a href="{{ '/Users/matt/Documents/Books/php-symfony-6-book/code" data-bbox="146 160 981 404" data-label="Text">

```
Your controller at <code><a href="{{ '/Users/matt/Documents/Books/php-symfony-6-book/code
 Your template at <code><a href="{{ '/Users/matt/Documents/Books/php-symfony-6-book/codes/

</div>
{% endblock %}
```


```

Let's 'make this our own' by changing the contents of the Response returned to a simple text response. Do the following:

- comment-out the body of the `index()` method
 - at the top of the class add a `use` statement, so we can make use of the Symfony HTTPFoundation class `Response`
- ```
use Symfony\Component\HttpFoundation\Response;
```
- write a new body for the `index()` method to output a simple text message response:

```
 return new Response('Welcome to your new controller!');
```

So the listing of your `DefaultController` should look as follows:

```
<?php
namespace App\Controller;

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

class DefaultController extends AbstractController
{
 #[Route('/', default, name: 'default')]
 public function index(): Response
 {
 return new Response('Welcome to your new controller!');
 // return $this->render('default/index.html.twig', [
```

```
// 'controller_name' => 'DefaultController',
// });
 }
}
```

## 2.5 Run web server to visit new default route

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

But we see that default Symfony welcome page, not our custom response text message!

Since we **have** defined a route, we don't get the default page any more. However, since we named our controller **Default**, then this is the route that was defined for it:

Name	Method	Scheme	Host	Path
-----				
_...(all those debug routes starting with _ )				
default	ANY	ANY	ANY	/default

If we look more closely at the generated code, we can see this route `/default` in the **attribute** preceding controller method `index()` in `src/Controllers/DefaultController.php`

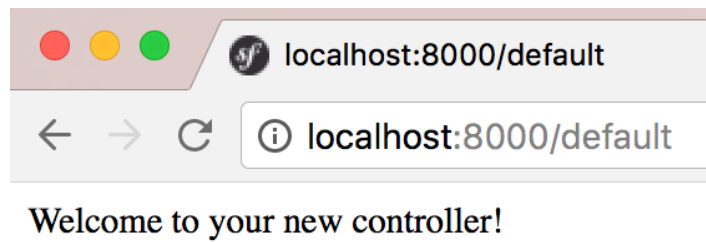
```
#[Route('/default', name: 'default')]
```

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

NOTE:

- if you still don't see our custom welcome page, then try first clearing the 'cache' to see the result of code changes we have just made. To speed things up Syfony uses a cache (memory) of recent Responses - but if you've made code change the cached pages and routes may be out of date...
- to clear the cache using a Symfony CLI comment type `php app/console cache:clear`
- to clear the cache by deleting the files themselves, DELETE the `/var` folder
  - you can safely delete this folder at any time (unless you are using SQLite and storing your DB files there...)

Figure 2.3 shows a screenshot of the message created from our generated default controller method.

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

## 2.6 Other types of Response content

We could also have asked our Controller function to return JSON rather than text. We can create JSON either using Twig, or with the inherited `->json(...)` method. For example, try replacing the body of your `index()` method with the following:

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

## 2.7 The default Twig page

If we return our `index()` method back to what was first automatically generated for us, we can see an HTML page in our browser that is output from the Twig template:

```
public function index()
{
 return $this->render('default/index.html.twig', [
 'controller_name' => 'DefaultController',
]);
}
```

Figure 2.4 shows a screenshot of the Twig HTML page that was automatically generated.

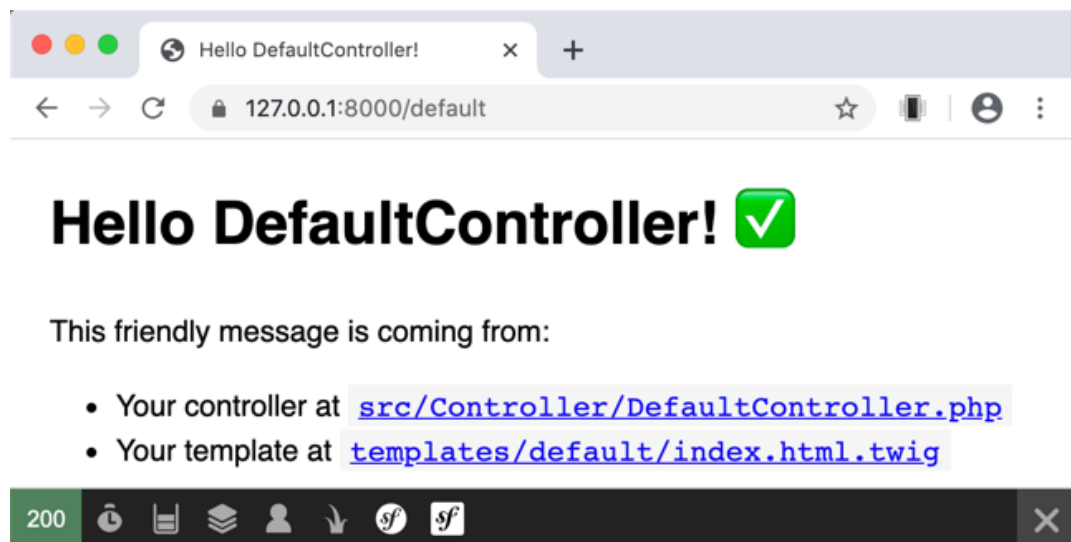


Figure 2.4: Screenshot of generated Twig page for URL path /default.

# 3

## Twig templating

### 3.1 Customizing the Twig output (basic02)

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

```
namespace App\Controller;

use Symfony\...

class DefaultController extends AbstractController
{
 #[Route('/default', name: 'default')]
 public function index(): Response
 {
 return $this->render('default/index.html.twig', [
 'controller_name' => 'DefaultController',
]);
 }
}
```

As you can see, the controller method now returns the output of method `$this->render(...)` rather than directly creating a `Response` object. With the Twig bundle added, each controller class now has access to the Twig `render(...)` method.

Figure 3.1 shows a screenshot of the message created from our generated default controller method with Twig.

NOTE: The actual look of the default generated Twig content may be a little different (e.g. 19 Feb 2019 it now says `Hello DefaultController!`)...

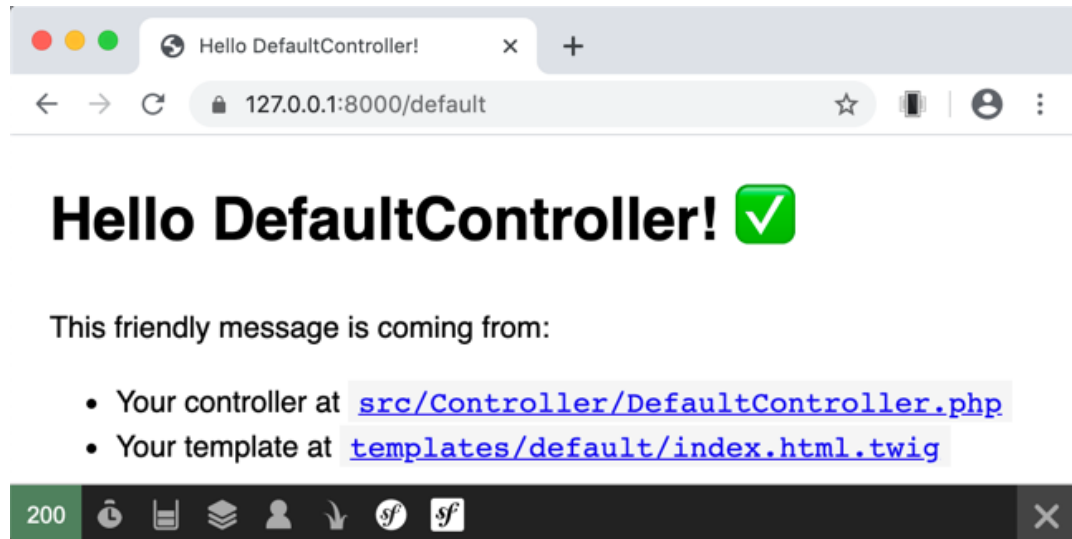


Figure 3.1: Screenshot of generated page for URL path `/default`.

### 3.2 Specific URL path and internal name for our default route method

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

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

Now the route is as follows (from typing `php bin/console de:ro`):

Name	Method	Scheme	Host	Path
homepage	ANY	ANY	ANY	/

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

Luckily, if we are using a PHP-friendly editor like PHPStorm, as we start to type the name of a class, the IDE will popup a suggestion of namespaced classes to choose from. Figure 3.2 shows a

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

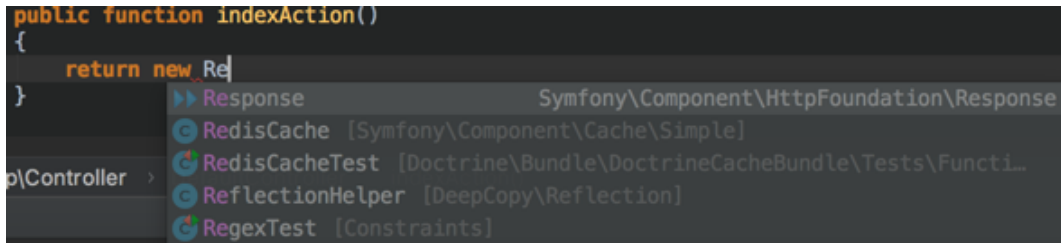


Figure 3.2: Screenshot of PHPStorm IDE suggesting namespaces classes.

Here is a complete `DefaultController` class:

```
namespace App\Controller;

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

class DefaultController extends AbstractController
{
 #[Route('/', name: 'homepage')]
 public function index(): Response
 {
 return new Response('Hello there!');
 }
}
```

Figure 3.3 shows a screenshot of the message created from our `Response()` object.

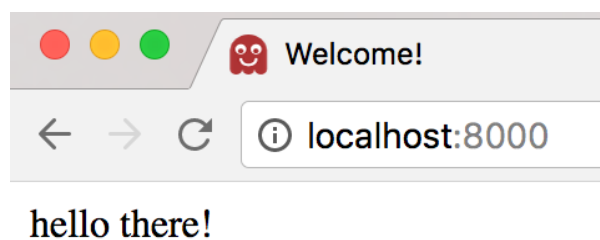


Figure 3.3: Screenshot of page seen for `new Response('hello there!')`.

### 3.3 Clearing the cache

Sometimes, when we've added a new route, we still get an error saying the route was not found, or showing us out-of-date content. This can be a problem of the Symfony **cache**.

So clearing the cache is a good way to resolve this problem (you may get in the habit of clearing the cache each time you add/change any routes).

You can clear the cache in 2 ways:

1. Simply delete directory `/var/cache`
2. Use the CLI command to clear the cache:

```
$ php bin/console cache:clear

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

$
```

### 3.4 Let's create a nice Twig home page

We are (soon) going to create Twig template in `templates/default/homepage.html.twig`. So we need to ask the Twig object in our Symfony project to create an HTTP response via its `render()` method. Part of the 'magic' of PHP Object-Oriented inheritance (and the **Dependency Injection** design pattern), is that since our controller class is a subclass of `Symfony\Bundle\FrameworkBundle\Controller\Controller`, then objects of our controller automatically have access to a `render(...)` method for an automatically generated Twig object.

In a nutshell, to output an HTTP response generated from Twig, we just have to specify the Twig template name, and relative location<sup>1</sup>, and supply an array of any parameters we want to pass to the template.

So we can simply write the following to ask Symfony to generate an HTTP response from Twig's text output from rendering the template that can (will soon!) be found in `/templates/default/homepage.html.twig`:

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

---

<sup>1</sup>The 'root' of Twig template locations is, by default, `/templates`. To keep files well-organised, we should create subdirectories for related pages. For example, if there is a Twig template `/templates/admin/loginForm.html.twig`, then we would need to refer to its location (relative to `/templates`) as `admin/loginForm.html.twig`.



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

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

- Replace the contents of file `index.html.twig` with the following:

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

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

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

Note that Twig paths searches from the Twig root location of `/templates`, not from the location of the file doing the inheriting, so do **NOT** write `{% extends 'default/base.html.twig' %}`...

Figure 3.4 shows a screenshot our Twig-generated page in the web browser.



Figure 3.4: Screenshot of page from our Twig template.



# 4

## Creating our own classes

### 4.1 Goals

Our goals are to:

- create a simple Student entity class (by hand - not using the **make** tool)
- create a route / controller / template to show one student on a web page
- create a repository class, to manage an array of Student objects
- create a route / controller / template to list all students as a web page
- create a route / controller / template to show one student on a web page for a given Id

### 4.2 Let's create an Entity Student (basic03)

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

```
<?php
namespace App\Entity;

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

```
 private string $surname;
 }
```

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

So you should end up with accessor method for each property such as:

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

/**
 * @param int $id
 */
public function setId(int $id): void
{
 $this->id = $id;
}

... etc... for the other properties ...
```

## 4.3 Create a StudentController class

Generate a StudentController class:

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

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

```
<?php

namespace App\Controller;

use Symfony\...

class StudentController extends AbstractController
{
 #[Route('/student', name: 'student')]
}
```

```
public function index(): Response
{
 return ... default code here ...
}
}
```

NOTE:

- as well as creating the class `/src/Controller/StudentController.php`, a folder and Twig template has also been created for you in `/templates/student/index.html.twig`

NOTE!!!!: When adding new routes, it's a good idea to **CLEAR THE CACHE**, otherwise Symfony may not recognise the new or changed routes ... Either manually delete the `/var/cache` directory, or run the `cache:clear` console command (you can shorten to `ca:cl`)

```
$ php bin/console cache:clear
```

```
// Clearing the cache for the dev environment with debug true
```

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

Let's make this new controller index method create a student (1, matt, smith) and display it with a Twig template (which we'll write next!). We will also improve the route internal name, changing it to `student_show`, and change the method name to `show()`. So your class (with its `use` statements, especially for `App\Entity\Student`) looks as follows now:

```
<?php

namespace App\Controller;

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

use App\Entity\Student;

class StudentController extends AbstractController
{
 #[Route('/student', name: 'student')]
 public function index(): Response
 {
 $student = new Student();
 $student->setId(99);
 $student->setFirstName('matt');
 $student->setSurname('Smith');
 }
}
```

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

NOTE:: Ensure your code has the appropriate `use` statement added for the `App\Entity\Student` class (since it's not in the same namespace as the controller, we have to add a `use` statement) - a nice IDE like PHPStorm will add this for you...

## 4.4 The show student template `/templates/student/show.html.twig`

In folder `/templates/student` create a new Twig template `show.html.twig` containing the following:

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

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

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

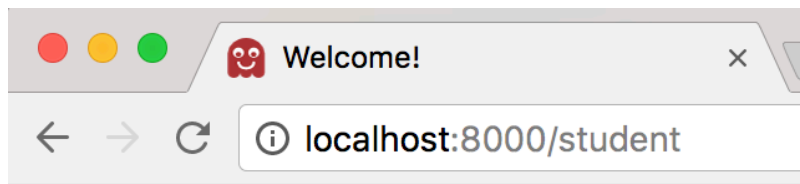
 name = {{ student.firstName }} {{ student.surname }}
 </p>
{% endblock %}
```

Do the following:

- Run the web server

```
symfony serve
```
- Visit `/student`
  - you should see our student details displayed as a nice HTML page.

Figure 4.1 shows a screenshot our student details web page.



# Student SHOW page

id = 1  
name = matt smith

Figure 4.1: Screenshot of student show page.

## 4.5 Twig debug `dump(...)` function

A very useful feature of Twig is its `dump(...)` function. This outputs to the web page a syntax colored dump of the variable its passed. It's similar to the PHP `var_dump(...)` function. Figure 4.2 shows a screenshot of adding the following to our `show.html.twig` template:

```
{% block body %}
 <h1>Student SHOW page</h1>
 <p>
 id = {{ student.id }}

 name = {{ student.firstName }} {{ student.surname }}
 </p>

 {{ dump (student) }}
{% endblock %}
```

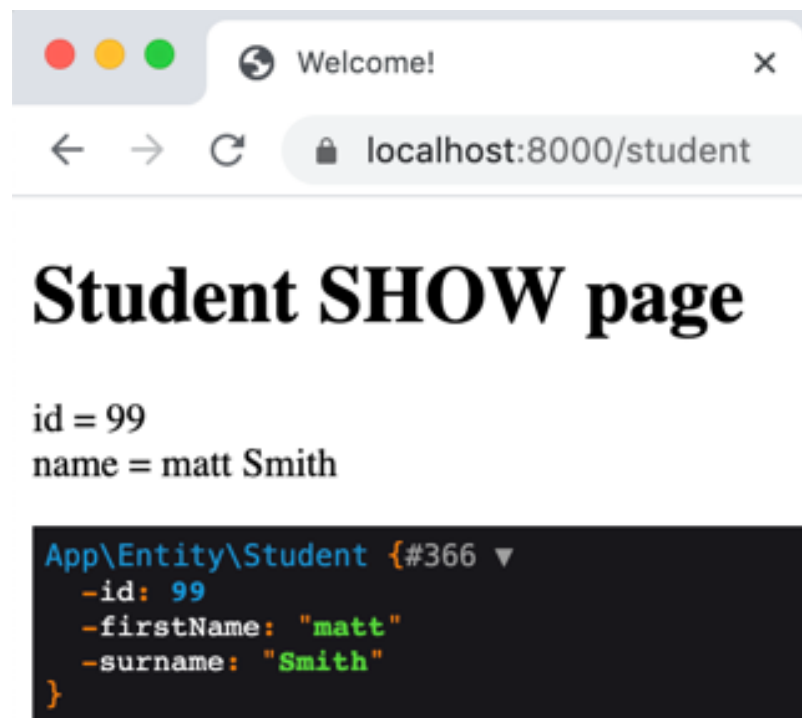


Figure 4.2: Screenshot of student show page.



## 4.6 Creating an Entity Repository (basic04)

We will now move on to work with an **array** of **Student** objects, which we'll make easy to work with by creating a **Repository** class. Let's create the **StudentRepository** class to work with collections of **Student** objects. Create PHP class file **StudentRepository.php** in directory **/src/Repository**:

```
<?php
namespace App\Repository;

use App\Entity\Student;

class StudentRepository
{
 private $students = [];

 public function __construct()
 {
 $id = 1;
 $s1 = new Student();
 $s1->setId(1);
 $s1->setFirstName('matt');
 $s1->setSurname('smith');
 $this->students[$id] = $s1;

 $id = 2;
 $s2 = new Student();
 $s2->setId(2);
 $s2->setFirstName('joelle');
 $s2->setSurname('murphy');
 $this->students[$id] = $s2;

 $id = 3;
 $s3 = new Student();
 $s3->setId(3);
 $s3->setFirstName('frances');
 $s3->setSurname('mcguinness');
 $this->students[$id] = $s3;
 }

 public function findAll()
 {
```

```
 return $this->students;
 }
}
```

## 4.7 The student list controller method

Let's replace the contents of our `index()` method in the `StudentController` class, with one that will retrieve the array of student records from an instance of `StudentRepository`, and pass that array to our Twig template. The Twig template will loop through and display each one.

Replace the existing method `index()` of controller class `StudentController` with the following:

```
...

use App\Repository\StudentRepository;

use App\Entity\Student;

class StudentController extends AbstractController
{
 #[Route('/student', name: 'student')]
 public function index(): Response
 {
 $studentRepository = new StudentRepository();
 $students = $studentRepository->findAll();

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

So our routes remain the same, with the URL pattern `/student` being routed to our `StudentController->index()` method:

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

Name	Method	Scheme	Host	Path
------	--------	--------	------	------

```
_... (lots of other debug profiler routes)
homepage ANY ANY ANY /
student ANY ANY ANY /student
```

## 4.8 The list student template `/templates/student/list.html.twig`

In directory `/templates/student` create Twig template `list.html.twig` with the following (you may wish to duplicate the `show` template and edit it to match this):

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

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

 {% for student in students %}

 id = {{ student.id }}

 name = {{ student.firstName }} {{ student.surname }}

 {% endfor %}

{% endblock %}
```

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

Figure 4.3 shows a screenshot our list of students web page.

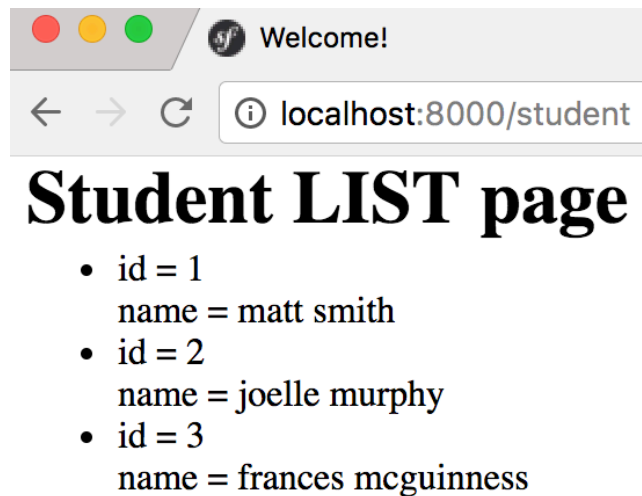


Figure 4.3: Screenshot of student list page.

## 4.9 Refactor show action to show details of one Student object (project basic05)

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

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

NOTE: We'll give this **show** route the internal name `student_show` - these internal route names are used when we create links between pages in our Twig templates, and so it's important to name them meaningfully and consistently to make later coding straightforward.

```
#[Route('/student/{id}', name: 'student_show')]
public function show($id): Response
{
 $studentRepository = new StudentRepository();
 $student = $studentRepository->find($id);

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

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

```

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

```

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

```

#[Route('/student', name: 'student_list')]
public function list(): Response
{
 ... as before
}

```

We can check these routes via the console:

- `/student/{id}` will invoke our `show($id)` method
- `/student` will invoke our `list()` method

Name	Method	Scheme	Host	Path
_... (lots of other debug profiler routes)				
homepage	ANY	ANY	ANY	/
student_list	ANY	ANY	ANY	/student
student_show	ANY	ANY	ANY	/student/{id}

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

It's a good idea to **CLEAR THE CACHE** when adding/changing routes, otherwise Symfony may not recognise the new or changed routes ... Either manually delete the `/var/cache` directory, or run the `cache:clear` console command:

```

$ php bin/console cache:clear

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

```

Symfony caches (stores) routing data and also rendered pages from Twig, to speed up response time. But if you have changed controllers and routes, sometimes you have to manually delete the cache to ensure all new routes are checked against new requests.

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

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

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

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

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

## 4.11 Make each item in list a link to show

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

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

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

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

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

 {% for student in students %}

 id = {{ student.id }}

 name = {{ student.firstName }} {{ student.surname }}

 (details)

 {% endfor %}

</block>
```

```

 {% endfor %}

{% endblock %}
```

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

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

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

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

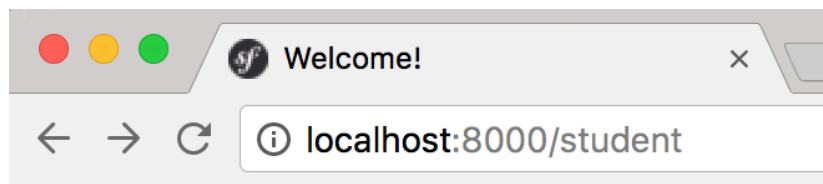
could be represented in Twig as:

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

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

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

Figure 4.4 shows a screenshot our list of students web page, with a `(details)` hypertext link to the show page for each individual student object.



## Student LIST page

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

Figure 4.4: Screenshot of student list page, with links to show page for each student object.



## 4.12 Dealing with not-found issues (project basic06)

If we attempted to retrieve a record, but got back `null`, we might cope with it in this way in our controller method, i.e. by throwing a `NotFound-Exception` (which generates a 404-page in production):

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

Or we could simply create an error Twig page, and display that to the user, e.g.:

```
public function show($id): Response
{
 $studentRepository = new StudentRepository();
 $student = $studentRepository->find($id);

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

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

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

and a Twig template `/templates/error/404.html.twig` looking like this:

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

{% block title %}ERROR PAGE{% endblock %}

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

 <h2>404 - no found error</h2>

 <p>
 sorry - the item/page you were looking for could not be found
 </p>
{% endblock %}
```

```
</p>
{% endblock %}
```

NOTE: We have overridden the `title` Twig block, so that the page title is `ERROR PAGE...`

Figure 4.5 shows a screenshot of our custom 404 error template for when no such student can be found for the given ID.

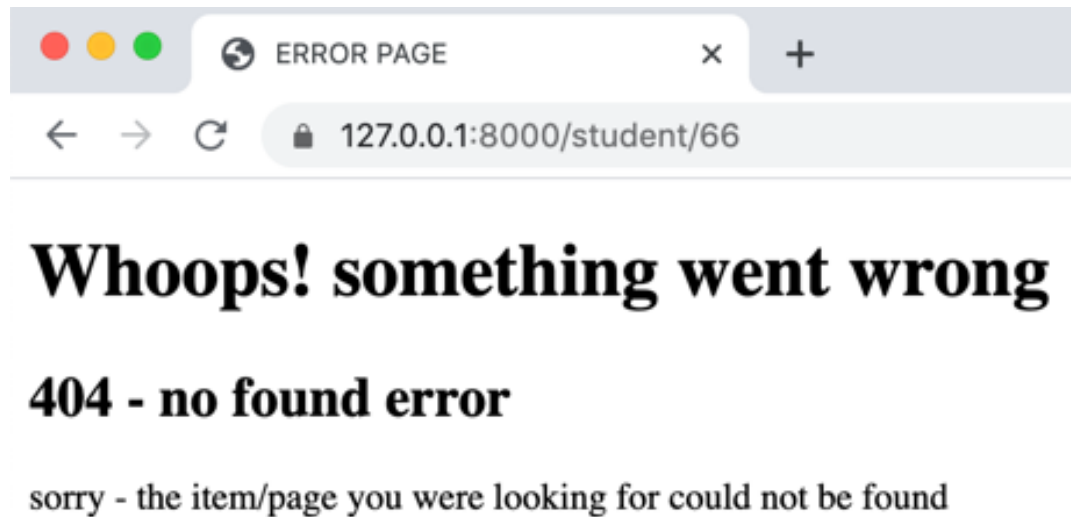


Figure 4.5: Error page for non-existent student ID = 66.