

**AN INTRODUCTION TO SYMFONY 5**  
(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 ...



# Table of Contents

<b>Acknowledgements</b>	<b>i</b>
<b>I Introduction to Symfony</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 What is Symfony 5? . . . . .	3
1.2 What do I need on my computer to get started? . . . . .	3
1.3 How do I get started with a new Symfony project . . . . .	4
1.4 Where are the projects accompanying this book? . . . . .	4
1.5 How do I run a Symfony webapp? . . . . .	5
1.5.1 From the CLI . . . . .	5
1.5.2 From a Webserver application (like Apache or XAMPP) . . . . .	7
1.6 It isn't working! (Problem Solving) . . . . .	7
1.7 Can I see a demo project with lots of Symfony features? . . . . .	7
1.8 Any free videos about SF5 to get me going? . . . . .	7
<b>2 First steps</b>	<b>9</b>
2.1 What we'll make ( <b>basic01</b> ) . . . . .	9
2.2 Create a new Symfony project . . . . .	10
2.3 List the routes . . . . .	11
2.4 Create a controller . . . . .	12
2.5 Run web server to visit new default route . . . . .	14
2.6 Other types of Response content . . . . .	15
2.7 The default Twig page . . . . .	15
<b>3 Twig templating</b>	<b>17</b>
3.1 Customizing the Twig output ( <b>basic02</b> ) . . . . .	17
3.2 Specific URL path and internal name for our default route method . . . . .	18
3.3 Clearing the cache . . . . .	20
3.4 Let's create a nice Twig home page . . . . .	20
<b>4 Creating our own classes</b>	<b>23</b>

4.1	Goals . . . . .	23
4.2	Let's create an Entity Student ( <code>basic03</code> ) . . . . .	23
4.3	Create a <code>StudentController</code> class . . . . .	24
4.4	The show student template <code>/templates/student/show.html.twig</code> . . . . .	25
4.5	Twig debug <code>dump(...)</code> function . . . . .	26
4.6	Creating an Entity Repository ( <code>basic04</code> ) . . . . .	27
4.7	The student list controller method . . . . .	28
4.8	The list student template <code>/templates/student/index.html.twig</code> . . . . .	29
4.9	Refactor show action to show details of one Student object (project <code>basic05</code> ) . . . . .	30
4.10	Adding a <code>find(\$id)</code> method to the student repository . . . . .	32
4.11	Make each item in list a link to show . . . . .	32
4.12	Dealing with not-found issues (project <code>basic06</code> ) . . . . .	35
<b>II</b>	<b>Symfony and Databases</b>	<b>37</b>
<b>5</b>	<b>Doctrine the ORM</b>	<b>39</b>
5.1	What is an ORM? . . . . .	39
5.2	Setting the database connection URL for MySQL . . . . .	40
5.3	Setting the database connection URL for SQLite . . . . .	40
5.4	Quick start . . . . .	41
5.5	Make your database . . . . .	41
<b>6</b>	<b>Working with Entity classes</b>	<b>43</b>
6.1	A <code>Student</code> DB-entity class (project <code>db01</code> ) . . . . .	43
6.2	Using annotation comments to declare DB mappings . . . . .	43
6.3	Declaring types for fields . . . . .	44
6.4	Validate our annotations . . . . .	44
6.5	The <code>StudentRepository</code> class ( <code>/src/Repository/StudentRepository</code> ) . . . . .	45
6.6	Create a migration (a migration <code>diff</code> file) . . . . .	46
6.7	Run the migration to make the database structure match the entity class declarations	46
6.8	Re-validate our annotations . . . . .	48
6.9	Generating entities from an existing database . . . . .	48
6.10	Note - use maker to save time (project <code>db02</code> ) . . . . .	48
6.11	Use maker to create properties, annotations and accessor methods! . . . . .	49
<b>7</b>	<b>Symfony approach to database CRUD</b>	<b>53</b>
7.1	Creating new student records (project <code>db02</code> ) . . . . .	53
7.2	Query database with SQL from CLI server . . . . .	56
7.3	Updating the <code>listAction()</code> to use Doctrine . . . . .	56
7.4	Deleting by id . . . . .	57
7.5	Updating given id and new name . . . . .	58

---

## TABLE OF CONTENTS

7.6	Updating our show action . . . . .	59
7.7	Redirecting to show after create/update . . . . .	60
7.8	Given <code>id</code> let Doctrine find Product automatically (project <code>db03</code> ) . . . . .	61
7.9	Creating the CRUD controller automatically from the CLI (project <code>db04</code> ) . . . . .	64
<b>8</b>	<b>Fixtures - setting up a database state</b>	<b>67</b>
8.1	Initial values for your project database (project <code>db05</code> ) . . . . .	67
8.2	Installing and registering the fixtures bundle . . . . .	67
8.2.1	Install the bundle . . . . .	67
8.3	Writing the fixture classes . . . . .	68
8.4	Loading the fixtures . . . . .	69
8.5	User Faker to generate plausible test data (project <code>db06</code> ) . . . . .	71
<b>III</b>	<b>Froms and form processing</b>	<b>73</b>
<b>9</b>	<b>DIY forms</b>	<b>75</b>
9.1	Preparation . . . . .	75
9.2	Adding a form for new Student creation (project <code>form01</code> ) . . . . .	75
9.3	Refactor our <code>create(...)</code> method . . . . .	77
9.4	Twig new student form . . . . .	77
9.5	Controller method (and annotation) to display new student form . . . . .	78
9.6	Controller method to process POST form data . . . . .	78
9.7	Validating form data, and displaying temporary ‘flash’ messages in Twig . . . . .	79
9.8	Three kinds of flash message: notice, warning and error . . . . .	80
9.9	Adding flash display (with CSS) to our Twig template (project <code>form02</code> ) . . . . .	80
9.10	Adding validation logic to our form processing controller method . . . . .	81
9.11	Postback logic (project <code>form03</code> ) . . . . .	82
9.12	Extra notes . . . . .	86
<b>10</b>	<b>Automatic forms generated from Entities</b>	<b>87</b>
10.1	Using the Symfony form generator (project <code>form04</code> ) . . . . .	87
10.2	The Symfony form generator via Twig . . . . .	87
10.3	Updating <code>StudentController-&gt;new()</code> . . . . .	88
10.4	Postback - form submits to same URL . . . . .	91
10.5	Using form classes (project <code>form05</code> ) . . . . .	92
10.6	Video tutorials about Symfony forms . . . . .	95
<b>11</b>	<b>Customising the display of generated forms</b>	<b>97</b>
11.1	First let’s Bootstrap this project (project <code>form06</code> ) . . . . .	97
11.2	Configure Twig to use the Bootstrap theme . . . . .	97
11.3	Add the Bootstrap CSS import into our base Twig template . . . . .	98
11.4	Add the Bootstrap JavaScript import into our base Twig template. . . . .	98

11.5 Run site and see some Bootstrap styling . . . . .	99
11.6 Adding elements for navigation and page content . . . . .	101
11.7 Add Bootstrap navigation bar . . . . .	102
11.8 Styling list of links in navbar . . . . .	104
11.9 Adding the hamburger-menu and collapsible links . . . . .	105
<b>12 Customizing display of Symfony forms</b>	<b>107</b>
12.1 Understanding the 3 parts of a form (project <code>form07</code> ) . . . . .	107
12.2 Using a Twig form-theme template . . . . .	108
12.3 DIY (Do-It-Yourself) form display customisations . . . . .	108
12.4 Customising display of parts of each form field . . . . .	109
12.5 Specifying a form's <b>method</b> and <b>action</b> . . . . .	110
<b>IV Custom Repository Queries with forms</b>	<b>113</b>
<b>13 Custom database queries</b>	<b>115</b>
13.1 Search for exact property value (project <code>query01</code> ) . . . . .	115
13.2 Fixtures . . . . .	115
13.3 Add new route and controller method for category search . . . . .	117
13.4 Aside: How to the free 'helper' Doctrine methods work? . . . . .	118
13.5 Testing our search by category . . . . .	121
<b>14 Custom database queries</b>	<b>125</b>
14.1 Search Form for exact property value (project <code>query02</code> ) . . . . .	125
14.2 The form in Twig templat . . . . .	127
14.3 Controller method to process form submission . . . . .	128
14.4 Getting rid of the URL search route . . . . .	129
<b>V Symfony code generation</b>	<b>131</b>
<b>15 CRUD controller and templates generation</b>	<b>133</b>
15.1 Symfony's CRUD generator (project <code>crud-01</code> ) . . . . .	133
15.2 What you need to add to your project . . . . .	133
15.3 Generating new Entity class <code>Category</code> . . . . .	134
15.4 Generating CRUD for a new Entity class . . . . .	134
15.5 The generated routes . . . . .	135
15.6 The generated CRUD controller . . . . .	136
15.7 The generated index (a.k.a. list) controller method . . . . .	137
15.8 The generated <code>new()</code> method . . . . .	140
15.9 The generated <code>show()</code> method . . . . .	140
15.10The generated <code>edit()</code> method . . . . .	141

---

## TABLE OF CONTENTS

15.11The generated <code>delete()</code> method . . . . .	142
<b>VI Sessions</b>	<b>145</b>
<b>16 Introduction to Symfony sessions</b>	<b>147</b>
16.1 Create a new project from scratch (project <code>sessions01</code> ) . . . . .	147
16.2 Default controller - hello world . . . . .	147
16.3 Twig foreground/background colours ( <code>sessions02</code> ) . . . . .	148
16.4 Working with sessions in Symfony Controller methods (project <code>session03</code> ) . . . . .	151
16.5 Symfony's 2 session 'bags' . . . . .	152
16.6 Storing values in the session in a controller action . . . . .	153
16.7 Twig function to retrieve values from session . . . . .	154
16.8 Attempt to read <code>colors</code> array property from the session . . . . .	154
16.9 Applying colours in HTML head <code>&lt;style&gt;</code> element (project <code>session04</code> ) . . . . .	156
16.10Testing whether an attribute is present in the current session . . . . .	158
16.11Removing an item from the session attribute bag . . . . .	158
16.12Clearing all items in the session attribute bag . . . . .	158
<b>17 Working with a session 'basket' of products</b>	<b>159</b>
17.1 Shopping cart of products (project <code>session05</code> ) . . . . .	159
17.2 Create a new project with the required packages . . . . .	159
17.3 Create a Product entity & generate its CRUD . . . . .	160
17.4 Homepage - link to products home . . . . .	160
17.5 Basket index: list basket contents (project <code>sessions07</code> ) . . . . .	161
17.6 Controller method - <code>clear()</code> . . . . .	162
17.7 Debugging sessions in Twig . . . . .	163
17.8 Adding a object to the basket . . . . .	164
17.9 The delete action method . . . . .	165
17.10The Twig template for the basket index action . . . . .	166
17.11Adding useful links to our <code>base.html.twig</code> template . . . . .	169
17.12Adding the 'add to basket' link in the list of products . . . . .	169
<b>VII Security and Authentication</b>	<b>171</b>
<b>18 Quickstart Symfony security</b>	<b>173</b>
18.1 Learn about Symfony security . . . . .	173
18.2 New project with open and secured routes (project <code>security01</code> ) . . . . .	173
18.3 Create new project and add the security bundle library . . . . .	173
18.4 Make a Default controller . . . . .	174
18.5 Make a secured Admin controller . . . . .	174
18.6 Core features about Symfony security . . . . .	177

18.7 Generating the special <code>User</code> Entity class (project <code>security02</code> ) . . . . .	178
18.8 Review the changes to the <code>/config/packages/security.yml</code> file . . . . .	179
18.9 Migrate new <code>User</code> class to your database . . . . .	179
18.10 Make some <code>User</code> fixtures . . . . .	179
18.11 Run and check your fixtures . . . . .	182
18.12 Creating a Login form . . . . .	182
18.13 Check the new routes . . . . .	183
18.14 Allow <b>any</b> user to view the login form . . . . .	183
18.15 Clear cache & visit <code>/admin</code> . . . . .	184
18.16 Using the <code>/logout</code> route . . . . .	186
18.17 Finding and using the internal login/logout route names in <code>SecurityController</code> . .	186
<b>19 Security users from database</b>	<b>189</b>
19.1 Improving <code>UserFixtures</code> with a <code>createUser(...)</code> method (project <code>security03</code> ) . .	189
19.2 Loading the fixtures . . . . .	190
19.3 Using SQL from CLI to see users in DB . . . . .	191
<b>20 Custom login page</b>	<b>193</b>
20.1 A D.I.Y. (customisable) login form (project <code>security04</code> ) . . . . .	193
20.2 Simplifying the generated login Twig template . . . . .	193
20.3 CSRF (Cross Site Request Forgery) protection . . . . .	194
20.4 Display any errors . . . . .	195
20.5 Custom login form when attempting to access <code>/admin</code> . . . . .	196
20.6 Path for successful login . . . . .	196
<b>21 Custom AccessDeniedException handler</b>	<b>199</b>
21.1 Symfony documentation for 403 access denied exception . . . . .	199
21.2 Declaring our handler (project <code>security05</code> ) . . . . .	199
21.3 The exception handler class . . . . .	200
<b>22 Twig and logging</b>	<b>203</b>
22.1 Getting reference to Twig and Logger objects . . . . .	203
22.2 Using Twig for access denied message (project <code>security06</code> ) . . . . .	203
22.3 The Twig page . . . . .	204
22.4 Terminal log . . . . .	206
22.5 Learn more about logger and exceptions . . . . .	206
<b>23 User roles and role hierarchies</b>	<b>207</b>
23.1 Simplifying roles with a hierarchy (project <code>security07</code> ) . . . . .	207
23.2 Modify fixtures . . . . .	208
23.3 Removing default adding of <code>ROLE_USER</code> if using a hierarchy . . . . .	208
23.4 Allowing easy switching of users when debugging . . . . .	209

---

## TABLE OF CONTENTS

<b>24 Customising view based on logged-in user</b>	<b>211</b>
24.1 Twig nav links when logged in (project <code>security08</code> ) . . . . .	211
24.2 Getting reference to the current user in a Controller . . . . .	213
<b>VIII Entity associations (one-to-many relationships etc.)</b>	<b>215</b>
<b>25 Database relationships (Doctrine associations)</b>	<b>217</b>
25.1 Information about Symfony 4 and databases . . . . .	217
25.2 Create a new project from scratch (project <code>associations01</code> ) . . . . .	217
25.3 Categories for Products . . . . .	218
25.4 Defining the many-to-one relationship from Product to Category . . . . .	218
25.5 How to allow <code>null</code> for a Product's category . . . . .	219
25.6 Adding the optional one-to-many relationship from Category to Product . . . . .	220
25.7 Create and migrate DB schema . . . . .	220
25.8 Generate CRUD for Product and Category . . . . .	221
25.9 Add Category selection in Product form . . . . .	221
25.10 Add small and large item Category . . . . .	223
25.11 Drop-down menu of categories when creating/editing Products . . . . .	223
25.12 Adding display of Category to list and show Product . . . . .	224
25.13 <code>toString()</code> method . . . . .	226
25.14 Setup relationship via <code>make</code> . . . . .	226
<b>26 Many-to-one (e.g. Products for a single Category)</b>	<b>229</b>
26.1 Basic list products for current Category (project <code>associations02</code> ) . . . . .	229
26.2 Add <code>getProducts()</code> for Entity Category . . . . .	229
26.3 Add a <code>__toString()</code> for Entity Products . . . . .	230
26.4 Make Category form type add <code>products</code> property . . . . .	230
26.5 Adding a nicer list of Products for Category show page . . . . .	231
26.6 Improving the Edit form (project <code>associations03</code> ) . . . . .	234
26.7 Creating related objects as Fixtures (project <code>associations04</code> ) . . . . .	237
26.8 Using Joins in custom Repository classes . . . . .	238
<b>27 Logged-in user stored as item author</b>	<b>241</b>
27.1 Getting User object for currently logged-in user . . . . .	241
27.2 Simple example: Users and their county ( <code>associations05</code> ) . . . . .	242
27.3 Add <code>toString</code> method to <code>User</code> . . . . .	244
27.4 Use currently logged-in user as author . . . . .	244
27.5 Protect CRUD so must be logged in . . . . .	245

<b>IX PHPDocumentor (2)</b>	<b>247</b>
<b>28 PHPDocumentor</b>	<b>249</b>
28.1 Why document code? . . . . .	249
28.2 Self-documenting code . . . . .	249
28.3 PHPDocumentor 2 . . . . .	250
28.4 Installing PHPDocumentor 2 - the PHAR . . . . .	250
28.5 Installing PHPDocumentor 2 - via Composer . . . . .	250
28.6 DocBlock comments . . . . .	250
28.7 Generating the documentation . . . . .	251
28.8 Using an XML configuration file <code>phpdoc.dist.xml</code> . . . . .	251
28.9 WARNING - PHPStorm default comments . . . . .	252
28.10 TODO - special treatment . . . . .	253
<b>X Symfony Testing with Codeception</b>	<b>255</b>
<b>29 Unit testing in Symfony with Codeception</b>	<b>257</b>
29.1 Codeception Open Source BDD project . . . . .	257
29.2 Adding Codeception to an existing project (project <code>codeception01</code> ) . . . . .	257
29.3 What Codeception has added to our project . . . . .	261
<b>30 Check Codeception is working</b>	<b>263</b>
30.1 Run Codeception (with no tests!) . . . . .	263
30.2 Test with a simple Unit test . . . . .	264
30.3 Fixing error message about missing <code>ext-mbstring</code> : . . . . .	266
30.4 Testing other classes (project <code>codeception02</code> ) . . . . .	266
30.5 The class to test our calculator . . . . .	267
<b>31 Acceptance Tests</b>	<b>271</b>
31.1 Test for home page text at / (project <code>codeception03</code> ) . . . . .	271
31.2 Run the test (fail - server not running) . . . . .	272
31.3 Run the test (pass, when server running) . . . . .	273
31.4 From red to green . . . . .	273
31.5 Make green - add link to about page in base Twig template . . . . .	275
31.6 Annotation style data provider to test multiple data . . . . .	275
31.7 Traditional Data Provider syntax . . . . .	277
31.8 Common assertions for Acceptance tests . . . . .	277
<b>32 Filling out forms</b>	<b>279</b>
32.1 Setup database . . . . .	279
32.2 Cest to enter a new recipe (project <code>codeception04</code> ) . . . . .	279

---

## TABLE OF CONTENTS

<b>33 Codeception Symfony DB testing</b>	<b>281</b>
33.1 Adding Symfony and Doctrine to the settings (project <code>codeception05</code> ) . . . . .	281
33.2 Test Users in DB from Fixtures . . . . .	282
33.3 Check DB reset after each test . . . . .	283
33.4 Add DB counts to our form-filling Acceptance test . . . . .	283
<b>XI Symfony Testing</b>	<b>287</b>
<b>34 Unit testing in Symfony</b>	<b>289</b>
34.1 Testing in Symfony . . . . .	289
34.2 Installing Simple-PHPUnit (project <code>test01</code> ) . . . . .	289
34.3 Completing the installation . . . . .	290
34.4 Running Simple-PHPUnit . . . . .	291
34.5 Testing other classes (project <code>test02</code> ) . . . . .	292
34.6 The class to test our calculator . . . . .	292
34.7 Using a data provider to test with multiple datasets (project <code>test03</code> ) . . . . .	294
34.8 Configuring testing reports (project <code>test04</code> ) . . . . .	295
34.9 Testing for exceptions (project <code>test07</code> ) . . . . .	297
34.10 PHPUnit <code>expectException(...)</code> . . . . .	299
34.11 PHPUnit annotation comment <code>@expectedException</code> . . . . .	300
34.12 Testing for custom Exception classes . . . . .	300
34.13 Checking Types with assertions . . . . .	302
34.14 Same vs. Equals . . . . .	302
<b>35 Code coverage and xDebug</b>	<b>305</b>
35.1 Code Coverage . . . . .	305
35.2 Generating Code Coverage HTML report . . . . .	306
35.3 Tailoring the ‘whitelist’ . . . . .	308
<b>36 Web testing</b>	<b>309</b>
36.1 Testing controllers with <code>WebTestCase</code> (project <code>test05</code> ) . . . . .	309
36.2 Automating a test for the home page contents . . . . .	310
36.3 Normalise content to lowercase (project <code>test06</code> ) . . . . .	312
36.4 Test multiple pages with a data provider . . . . .	313
36.5 Testing links (project <code>test08</code> ) . . . . .	314
36.6 Issue with routes that end with a forward slash / . . . . .	317
36.6.1 Solution 1: Ensure url pattern in test method exactly matches router url pattern . . . . .	318
36.6.2 Solution 2: Instruct client to ‘follow redirects’ . . . . .	318
<b>37 Testing web forms</b>	<b>321</b>
37.1 Testing forms (project <code>test09</code> ) . . . . .	321

37.2 Test we can get a reference to the form . . . . .	325
37.3 Submitting the form . . . . .	326
37.4 Entering form values then submitting . . . . .	326
37.5 Testing we get the correct result via form submission . . . . .	328
37.6 Selecting form, entering values and submitting in one step . . . . .	329
<b>XII Appendices</b>	<b>331</b>
<b>A Software required for Symfony development</b>	<b>333</b>
A.1 Don't confuse different software tools . . . . .	333
A.2 Software tools . . . . .	334
A.3 Test software by creating a new Symfony 4 project . . . . .	334
<b>B PHP Windows setup</b>	<b>335</b>
B.1 Check if you have PHP installed and working . . . . .	335
B.1.1 Download the latest version of PHP . . . . .	335
B.2 Add the <b>path</b> to <b>php.exe</b> to your System environment variables . . . . .	338
B.3 Check your <b>php.ini</b> file . . . . .	340
B.4 PHP Info & SQL driver test . . . . .	340
<b>C The Composer andd Symfonhy command line tools</b>	<b>343</b>
C.1 Composer . . . . .	343
C.1.1 Windows Composer install . . . . .	343
C.1.2 Windows Composer install . . . . .	344
C.2 Symfony command line tool . . . . .	344
<b>D Software tools</b>	<b>345</b>
D.1 PHPStorm editor . . . . .	345
D.2 MySQL Workbench . . . . .	345
D.3 Git . . . . .	346
D.4 Git Windows installation . . . . .	347
<b>E The fully-featured Symfony 4 demo</b>	<b>349</b>
E.1 Visit Github repo for full Symfony demo . . . . .	349
E.2 Git steps for download (clone) . . . . .	349
E.3 Non-git download . . . . .	350
E.4 Install dependencies . . . . .	350
E.5 Run the demo . . . . .	350
E.6 View demo in browser . . . . .	351
E.7 Explore the code in PHPStorm . . . . .	351
E.8 Switch demo from SQLite to MySQL . . . . .	351
E.9 Running the tests in the SF4 demo . . . . .	353

---

## TABLE OF CONTENTS

E.10 Run the tests . . . . .	353
E.11 Explore directory /tests . . . . .	354
E.12 Learn more . . . . .	355
<b>F Solving problems with Symfony</b>	<b>357</b>
F.1 No home page loading . . . . .	357
F.2 “Route not Found” error after adding new controller method . . . . .	358
F.3 Issues with timezone . . . . .	358
F.4 Issues with Symfony 3 and PHPUnit.phar . . . . .	358
F.5 PHPUnit installed via Composer . . . . .	359
<b>G Publish via Fortrabbit (PHP as a service)</b>	<b>361</b>
G.1 SSH key . . . . .	361
G.1.1 Windows SSH key setup . . . . .	361
G.1.2 Mac SSH key setup . . . . .	361
G.1.3 Linux SSH key setup . . . . .	361
G.1.4 Fortrabbit . . . . .	362
G.2 Creating a new web App . . . . .	364
G.3 Cloning and populating your Git repo . . . . .	366
G.4 Fixing the Fixtures issue . . . . .	368
G.5 Fixing the Apache .htaccess issue . . . . .	369
G.6 Adding, committing and pushing the project files to the repo . . . . .	369
G.7 SHH CLI Terminal to migrate and install DB fixtures . . . . .	372
G.8 Symfony project should now be fully deployed . . . . .	375
<b>H Quick setup for new ‘blog’ project</b>	<b>377</b>
H.1 Create a new project, e.g. ‘blog’ . . . . .	377
H.2 Set up your localhost browser shortcut to for app_dev.php . . . . .	377
H.3 Add run shortcut to your Composer.json scripts . . . . .	377
H.4 Change directories and run the app . . . . .	378
H.5 Remove default content . . . . .	378
<b>I Steps to download code and get website up and running</b>	<b>379</b>
I.1 First get the source code . . . . .	379
I.1.1 Getting code from a zip archive . . . . .	379
I.1.2 Getting code from a Git repository . . . . .	379
I.2 Once you have the source code (with vendor) do the following . . . . .	380
I.3 Run the webserver . . . . .	380
<b>J About parameters.yml and config.yml</b>	<b>381</b>
J.1 Project (and deployment) specific settings in (/app/config/parameters.yml) . . .	381
J.2 More general project configuration (/app/config/config.yml) . . . . .	382

<b>K Setting up for MySQL Database</b>	<b>383</b>
K.1 Declaring the parameters for the database ( <code>parameters.yml</code> ) . . . . .	383
<b>L Setting up for SQLite Database</b>	<b>385</b>
L.1 NOTE regarding FIXTURES . . . . .	385
L.2 SQLite suitable for most small-medium websites . . . . .	385
L.3 Create directory where SQLite database will be stored . . . . .	386
L.4 Declaring the parameters for the database ( <code>parameters.yml</code> ) . . . . .	386
L.5 Setting project configuration to work with the SQLite database driver and path ( <code>/app/config/config.yml</code> ) . . . . .	386
<b>M Setting up Adminer for DB GUI interaction</b>	<b>389</b>
M.1 Adminer - small and simple DB GUI . . . . .	389
M.2 Getting Adminer . . . . .	390
M.3 Setting up . . . . .	390
M.4 Running Adminer . . . . .	390
<b>N Avoiding issues of SQL reserved words in entity and property names</b>	<b>393</b>
<b>O Transcript of interactive entity generation</b>	<b>395</b>
<b>P Killing ‘php’ processes in OS X</b>	<b>397</b>
<b>Q Docker and Symfony projects</b>	<b>399</b>
Q.1 Setup . . . . .	399
Q.2 Dockerfile . . . . .	399
Q.3 Build your Docker image . . . . .	400
Q.4 Run a Container process based on your image (exposing HTTP port 80) . . . . .	400
Q.5 Alternative <code>Dockerfile</code> for a basic PHP application, using Apache . . . . .	400
Q.6 Create config file for Apache . . . . .	401
Q.7 Other Docker reference stuff . . . . .	401
Q.7.1 Docker Images . . . . .	401
Q.7.2 Containers . . . . .	402
Q.7.3 New Image from current (changed) state of a running Container . . . . .	402
Q.7.4 Exposing HTTP ports for Containers running web application servers . . . . .	403
Q.8 Useful reference sites . . . . .	403
<b>R xDebug for Windows</b>	<b>405</b>
R.1 Steps for Windows . . . . .	405
R.2 Steps for Linux/Mac . . . . .	405
R.3 Use the xDebug wizard! . . . . .	406
R.4 PHP Function <code>phpinfo()</code> . . . . .	407
R.5 More information . . . . .	408

---

## TABLE OF CONTENTS

<b>List of References</b>	<b>409</b>
---------------------------	------------

---

TABLE OF CONTENTS

## **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 7 (download/install from [php.net](#))
- a good text editor (I like [PHPStorm](#), but then it's free for educational users...)
- [Composer](#) (PHP package manager - a PHP program)

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

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

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 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
```

NOTE: - If the first line does not show a Symfony version starting with v4 then you may have an old version of PHP installed. You need PHP 7.2.5 minimum to run Symfony 5.

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

## 1.4 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-symfony5-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

## 1.5 How to I run a Symfony webapp?

### 1.5.1 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
```

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!).

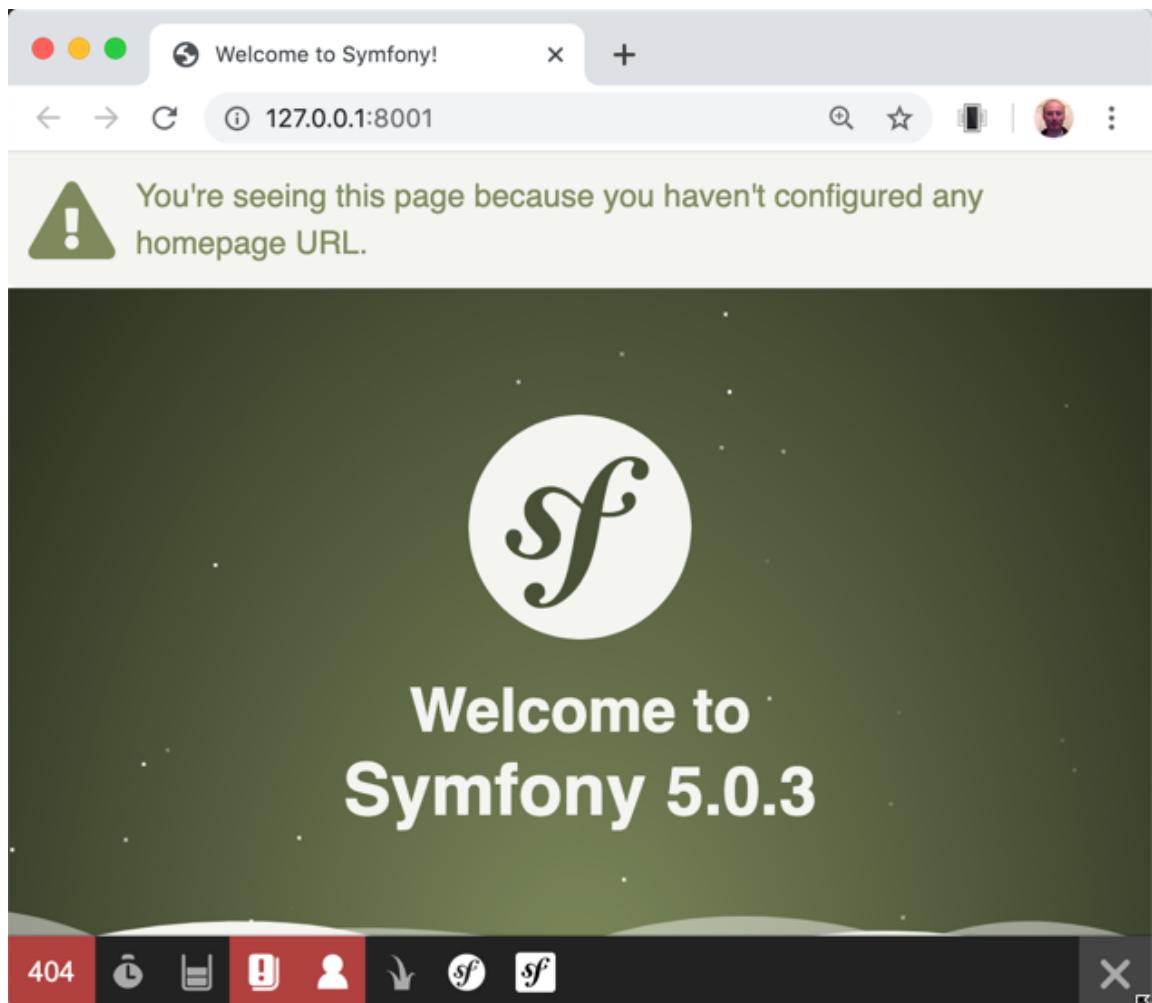


Figure 1.1: Screenshot default Symfony 4 home page.

### 1.5.2 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.

## 1.6 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.

## 1.7 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.

## 1.8 Any free videos about SF5 to get me going?

Yes! Those nice people at Symfonycasts have released a bunch of free videos all about Symfony 5 (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 ...)



# 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 ‘Annotation’ comments, which declare the route in a comment 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:

- an annotation 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 our 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 generate `/default`

## 2.2 Create a new Symfony project

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

```
$ symfony new --full basic01  
Installing symfony/skeleton (v4.0.5)  
  - Installing symfony/skeleton (v4.0.5): Loading from cache  
  
  ... etc. ...  
  
$ 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
[OK] Server listening on http://127.0.0.1:8000
// 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`).

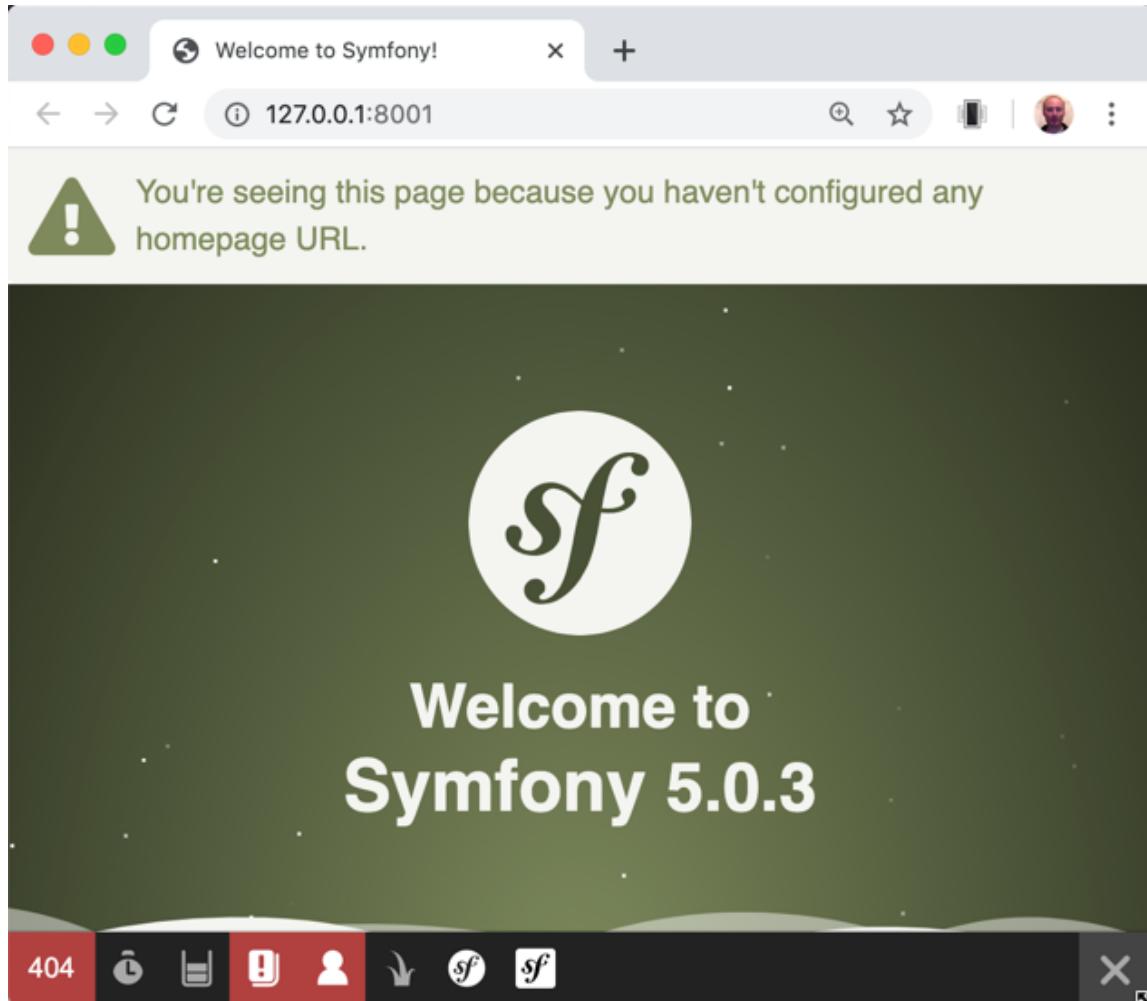


Figure 2.2: Screenshot default Symfony 4 page for web root (when no routes defined).

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

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:

```
$ 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\Routing\Annotation\Route;

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

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

```
{
    "message": "Welcome to your new controller!",
    "path": "src\\Controller\\DefaultController.php",
}
```

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\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends AbstractController
{
    /**
     * @Route("/default", name="default")
     */
    public function index()
    {
        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
<hr/>				
....(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 `annotation` comment 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.

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

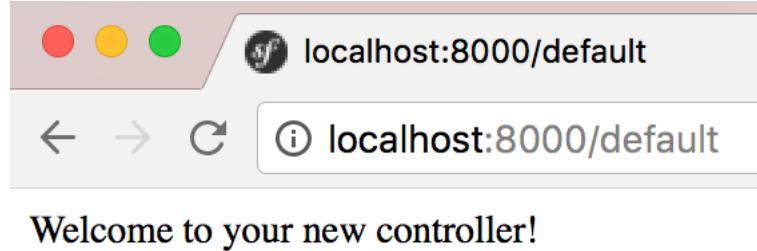


Figure 2.3: Screenshot of generated page for URL path /default.

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

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

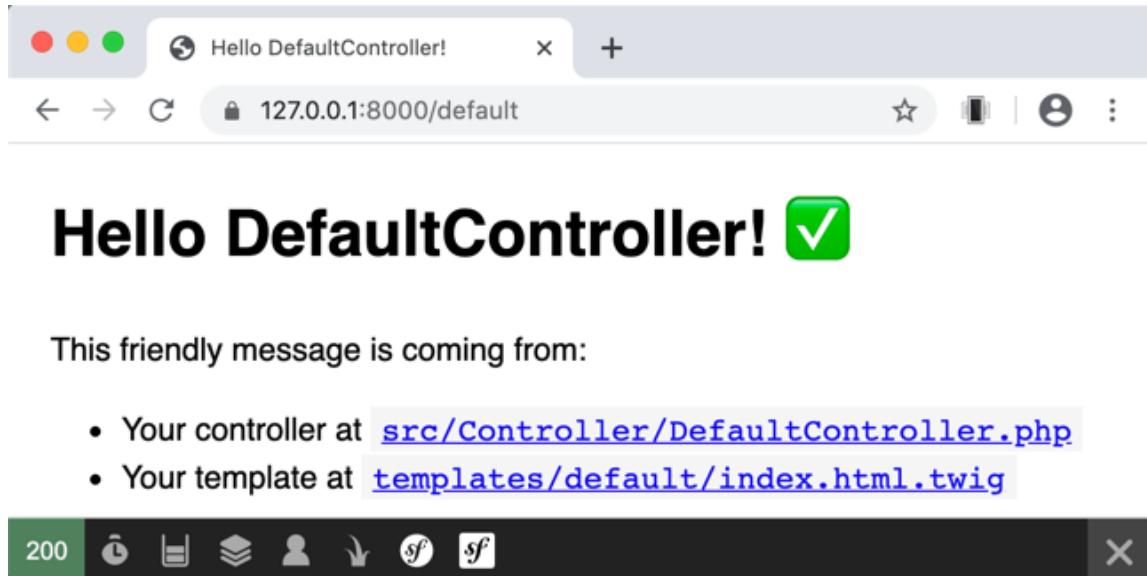


Figure 3.1: Screenshot of generated page for URL path `/default`.

## 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 Controller
{
    /**
     * @Route("/", name="homepage")
     */
    public function index()
```

Now the route is:

Name	Method	Scheme	Host	Path

```
homepage          ANY      ANY      ANY      /
```

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

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

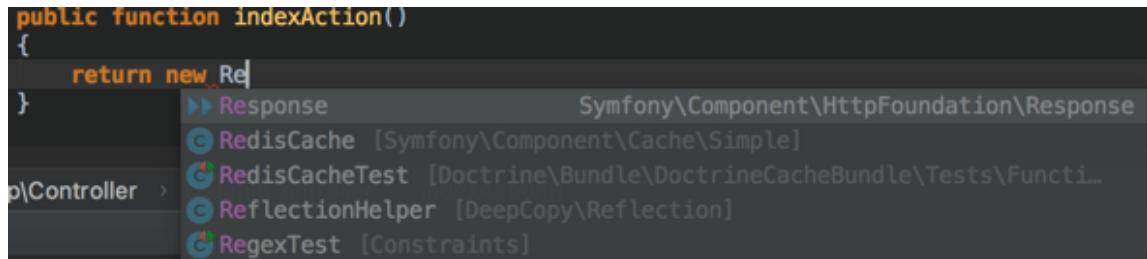


Figure 3.2: Screenshot of PHPStorm IDE suggesting namespaces classes.

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 indexAction()
    {
        return new Response('Hello there!');
    }
}
```

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

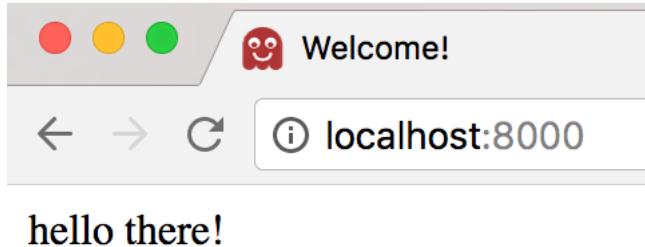


Figure 3.3: Screenshot of page seen for `new Response('hello there!')`.

### 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 hom 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 **Dependancy Injection** design pattern), is that since our controller class is a subclass of `Symfony\Bundle\FrameworkBundle\Controller\Controller`, then objects of our controller automatically have access to a `render(...)` method for an automatically generated Twig object.

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

---

<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`.

the template.

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

```
/*
 * @Route("/", name="homepage")
 */
public function indexAction()
{
    $template = 'default/index.html.twig';
    $args = [];
    return $this->render($template, $args);
}
```

Now let's put our own personal content in that Twig template in `/tempaltes/default/homepage.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 `/tempaltes`, 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 `Student` class:

```
<?php  
namespace App\Entity;  
  
class Student  
{  
    private $id;  
    private $firstName;
```

```
    private $surname;  
}
```

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

### 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\Bundle\FrameworkBundle\Controller\AbstractController;  
use Symfony\Component\Routing\Annotation\Route;  
  
class StudentController extends AbstractController  
{  
    /**  
     * @Route("/student", name="student")  
     */  
    public function index()  
    {  
        ... default code here ...  
    }  
}
```

NOTE!!!!: When adding new routes, it's a good idea to **CLEAR THE CACHE**, otherwise Symfony may not 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.
```

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

and change the method name to `show()`. So your class (with its `use` statements, especially for `App\Entity\Student`) looks as follows now:

```
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use App\Entity\Student;

class StudentController extends AbstractController
{
    /**
     * @Route("/student", name="student")
     */
    public function index()
    {
        $student = new Student();
        $student->setId(99);
        $student->setFirstName('matt');
        $student->setSurname('Smith');

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

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

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

Do the following:

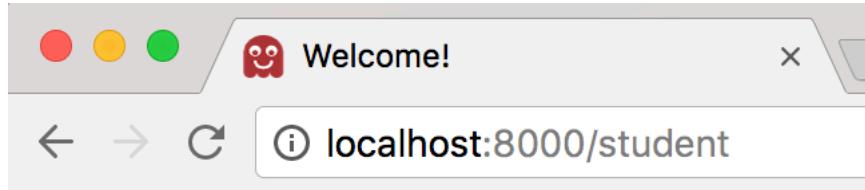
- Run the web server

```
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.

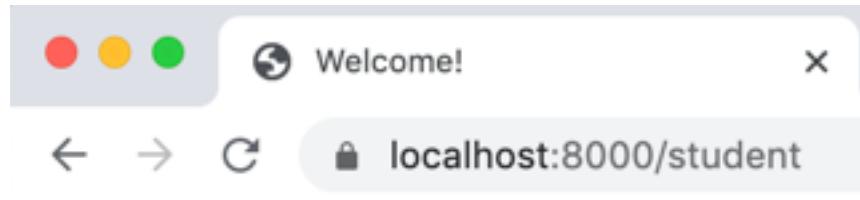
### 4.5 Twig debug dump(...) function

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 `index.html.twig` template:

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

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

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



## Student SHOW page

**id = 99**  
**name = matt Smith**

```
App\Entity\Student {#366 ▾
  -id: 99
  -firstName: "matt"
  -surname: "Smith"
}
```

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:

```
...
class StudentController extends AbstractController
{
    ...
    /**
     * @Route("/student", name="student")
     */
    public function index()
    {
        $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
<hr/>				
_... (lots of other debug profiler routes)				
homepage	ANY	ANY	ANY	/
student	ANY	ANY	ANY	/student

---

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

In directory `/templates/student` replace the contents of Twig template `index.html.twig` with the following:

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

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

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

Run the web server and visit `/student`, 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.

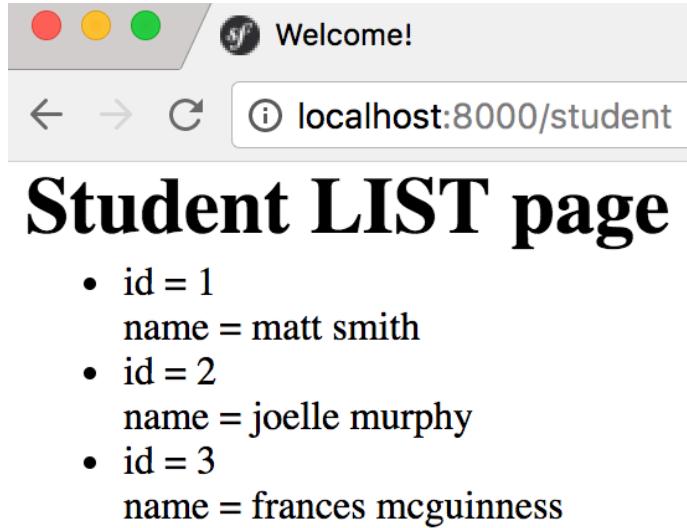


Figure 4.3: Screenshot of student list page.

## 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)
{
    $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()
{
    ... 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
<hr/>				
_... (lots of other debug profiler routes)				

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>

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

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

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

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

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

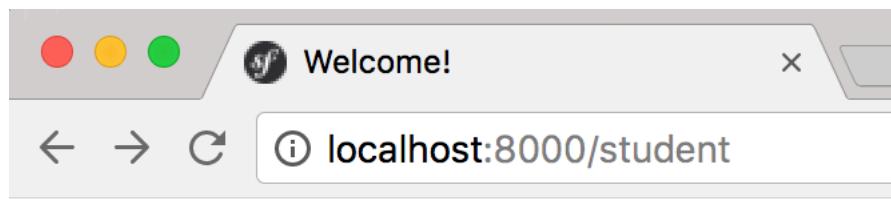
could be represented in Twig as:

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

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

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

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 Not-Found-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 showAction($id)
{
    $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>

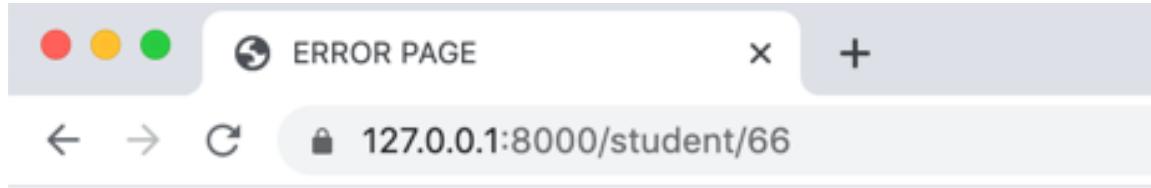
    <p>
        sorry - the item/page you were looking for could not be found
    </p>

```

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

NOTE: We have overriden 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.



# Whoops! something went wrong

## 404 - no found error

sorry - the item/page you were looking for could not be found

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

## **Part II**

# **Symfony and Databases**



# 5

## Doctrine the ORM

### 5.1 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 interact (CRUD<sup>1</sup>) 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:

---

<sup>1</sup>CRUD = Create-Read-Update-Delete

- Doctrine project's ORM page
- Wikipedia's ORM page
- Symfony's Doctrine help pages

## 5.2 Setting the database connection URL for MySQL

NOTE: This chapter assumes you are starting from the Student basic project from the end of the last chapter...

Edit file `.env` to change the default database URL to one that will connect to MySQL server running at port 3306, with username `root` and password `pass`, and working with database schema `web3` (or whatever **you want to name your database ...**)

So change this line in `.env` from:

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

to

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

NOTE: If you prefer to parametrize the database connection, use environment variables and then  `${VAR}` in your URL:

```
DB_USER=root
DB_PASSWORD=pass
DB_HOST=127.0.0.1
DB_PORT=3306
DB_NAME=web3
DATABASE_URL=mysql://${DB_USER}:${DB_PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}
```

## 5.3 Setting the database connection URL for SQLite

If you want a non-MySQL database setup for now, then just use the basic SQLite setup:

So change this line in `.env` from:

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

to

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

This will work with SQLite database file `data.db` in directory `/var`.

## 5.4 Quick start

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

1. `doctrine:database:create`
2. `doctrine:migrations:diff`
3. `doctrine:migrations:migrate` (or possibly `doctrine:schema:update --force`)
4. `doctrine:schema:validate`
5. `doctrine:fixtures:load`
6. `doctrine:query:sql`

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

## 5.5 Make your database

We can now use the settings in the `.env` file to connect to the MySQL server and create our database schema:

```
$ php bin/console doctrine:database:create
```



# 6

## Working with Entity classes

### 6.1 A Student DB-entity class (project db01)

Doctrine expects to find entity classes in a directory named `/src/Entity`, and corresponding repository classes in `/src/Repository`. We already have our `Student` and `StudentRepository` classes in the right places!

Although we'll have to make some changes to these classes of course.

### 6.2 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 ORM repository class `StudentRepository`.

```
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
```

```
* @ORM\Entity(repositoryClass="App\Repository\StudentRepository")
*/
class Student
{
```

## 6.3 Declaring types for fields

We now use annotations to declare the types (and if appropriate, lengths) of each field.

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

/**
 * @ORM\Column(type="string")
*/
private $firstName;

/**
 * @ORM\Column(type="string")
*/
private $surname;
```

## 6.4 Validate 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 Doctrine to create the actual database schema and tables, 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
```

```
-----
```

```
[ERROR] The database schema is not in sync with the current mapping file.
```

## 6.5 The StudenRepository class (`/src/Repository/StudentRepository`)

We need to change our repository class to be one that works with the Doctrine ORM. Unless we are writing special purpose query methods, all we really need for an ORM repository class is to ensure it subclasses `DoctrineBundle\Repository\ServiceEntityRepository` and its constructor points it to the corresponding entity class.

Change class `StudentRepository` as follows:

- remove all methods
- add `use` statements for:

```
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Common\Persistence\ManagerRegistry;
```

- make the class extend class `ServiceEntityRepository`

```
class StudentRepository extends ServiceEntityRepository
```

- add a constructor method:

```
public function __construct(ManagerRegistry $registry)
{
    parent::__construct($registry, Student::class);
}
```

So the full listing for `StudentRepository` is now:

```
namespace App\Repository;

use App\Entity\Student;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
public function __construct(ManagerRegistry $registry)

class StudentRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Student::class);
    }
}
```

## 6.6 Create a migration (a migration diff file)

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

```
$ php bin/console make:migration
```

Success!

```
Next: Review the new migration "src/Migrations/Version20180213082441.php"  
Then: Run the migration with php bin/console doctrine:migrations:migrate  
See https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html
```

NOTE: This is a shorter way of writing the old doctrine command: `php bin/console doctrine:migrations:diff`

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

```
namespace DoctrineMigrations;  
  
use Doctrine\DBAL\Migrations\AbstractMigration;  
use Doctrine\DBAL\Schema\Schema;  
  
/**  
 * Auto-generated Migration: Please modify to your needs!  
 */  
class Version20180213082441 extends AbstractMigration  
{  
    public function up(Schema $schema)  
    {  
        // this up() migration is auto-generated, please modify it to your needs  
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql',  
        'Migration can only be executed safely on \'mysql\'.');  
  
        $this->addSql('CREATE TABLE student (id INT AUTO_INCREMENT NOT NULL,  
        first_name VARCHAR(255) NOT NULL, surname VARCHAR(255) NOT NULL, PRIMARY KEY(id)) DE
```

## 6.7 Run the migration to make the database structure match the entity class declarations

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

ahead and change the database structure:

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

Application Migrations

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

++ migrating 20180201223133

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

++ migrated (0.14s)

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

You can see the results of creating the database schema and creating table(s) to match your ORM entities using a database client such as MySQL Workbench. Figure 6.1 shows a screenshot of MySQL Workbench showing the database's `student` table to match our `Student` entity class.

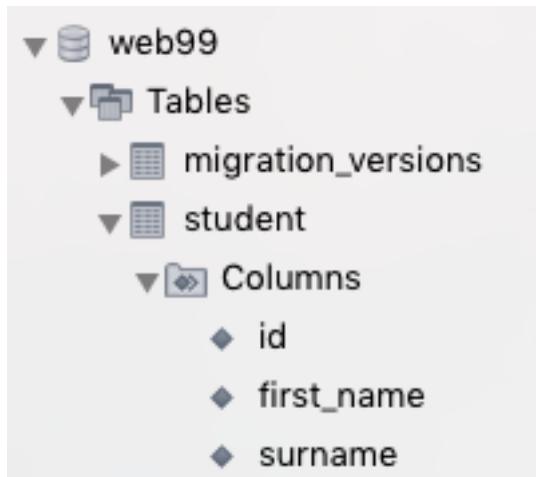


Figure 6.1: Screenshot MySQL Workbench and generated schema and product table.

## 6.8 Re-validate our annotations

We should get 2 “ok”’s if we re-validate our schema now:

```
$ 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
-----
[OK] The database schema is in sync with the mapping files.
```

## 6.9 Generating entities from an existing database

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

- [Symfony docs on inferring entites from existing db tables](#)

## 6.10 Note - use maker to save time (project db02)

We could have automatically created our Student entity and StudentRepository classes from scratch, using the `make` package:

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

created: src/Entity/Student.php
created: src/Repository/StudentRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

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

Success!
```

```
Next: When you're ready, create a migration with make:migration
$
```

In the above <RETURN> was pressed to not add any fields automatically. The Maker bundle created 2 classes for us:

- a Student class `src/Entity/Student.php`, containing just a private `id` property and a public `getId()` method
- and a generic StudentRepository class `src/Repository/StudentRepository.php`

We would then be able to manually add the `firstName` and `surname` properties (and their annotation comments) as we did earlier in the chapter:

```
/**
 * @ORM\Column(type="string")
 */
private $firstName;

/**
 * @ORM\Column(type="string")
 */
private $surname;
```

Finally we would have had to generate getters and setters for these 2 fields, and migrate to the database.

## 6.11 Use maker to create properties, annotations and accessor methods!

We could automatically create our Student entity and StudentRepository classes from scratch, using the `make` package. It will interactively ask you about fields you wish to create, and add the appropriate annotations and accessor (get/set) methods for you!

If you want to try this, first:

- Delete the entity class: `/src/Entity/Student.php`
- Delete the repository class: `/src/Repository/StudentRepository.php`

Then run the CLI command `make:entity Student`, and at the prompt ask it to create our `firstName` and `surname` text properties (all entities get an auto-incremented `Id` field with us having to ask):

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

created: src/Entity/Student.php
created: src/Repository/StudentRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):
> firstName

Field type (enter ? to see all types) [string]:
>

Field length [255]:
>

Can this field be null in the database (nullable) (yes/no) [no]:
> surname

updated: src/Entity/Student.php

Add another property? Enter the property name (or press <return> to stop adding fields):
>

Success!
```

Next: When you're ready, create a migration with `make:migration`

For each property the Maker bundle wants to know 3 things:

- property name (e.g. `firstName` and `surname`)
- property type (default is `string`)
- whether `NUL` can be stored for property

For `string` properties like `firstName` we just need to enter the property name and hit `<RETURN>` for the defaults (`string`, not nullable). For other types of field you can get a list of types by entering `?` at the prompt:. There are quite a few of them:

```
Field type (enter ? to see all types) [string]:
> ?
```

Main types

```
* string  
* text  
* boolean  
* integer (or smallint, bigint)  
* float
```

**Relationships / Associations**

```
* relation (a wizard will help you build the relation)  
* ManyToOne  
* OneToMany  
* ManyToMany  
* OneToOne
```

**Array/Object Types**

```
* array (or simple_array)  
* json  
* object  
* binary  
* blob
```

**Date/Time Types**

```
* datetime (or datetime_immutable)  
* datetimetz (or datetimetz_immutable)  
* date (or date_immutable)  
* time (or time_immutable)  
* dateinterval
```

**Other Types**

```
* json_array  
* decimal  
* guid
```



# 7

## Symfony approach to database CRUD

### 7.1 Creating new student records (project db02)

Let's add a new route and controller method to our `StudentController` class. This will define the `create()` method that receives parameter `$firstName` and `$surname` extracted from the route `/students/create/{firstName}/{surname}`. This is all done automatically for us, through Symfony seeing the route parameters in the `@Route(...)` annotation comment that immediately precedes the controller method. The 'signature' for our new `create(...)` method names 2 parameters that match those in the `@Route(...)` annotation comment `create($firstName, $surname)`:

```
/**
 * @Route("/student/create/{firstName}/{surname}")
 */
public function create($firstName, $surname)
```

Creating a new `Student` object is straightforward, given `$firstName` and `$surname` from the URL-encoded GET name=value pairs:

```
$student = new Student();
$student->setFirstName($firstName);
$student->setSurname($surname);
```

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

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

So the code for our create controller method is:

```
...
// we need to add a 'use' statement so we can create a Response object...
use Symfony\Component\HttpFoundation\Response;
...

/**
 * @Route("/student/create/{firstName}/{surname}")
 */
public function create($firstName, $surname)
{
    $student = new Student();
    $student->setFirstName($firstName);
    $student->setSurname($surname);

    // 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 smith` just visit this URL with your browser:

```
http://localhost:8000/student/create/matt/smith
```

Figure 7.1 shows how a new record `matt smith` is added to the database table via route `/student/create/{firstName}/{surname}`.

We can see these records in our database. Figure 7.2 shows our new `students` table created for us.

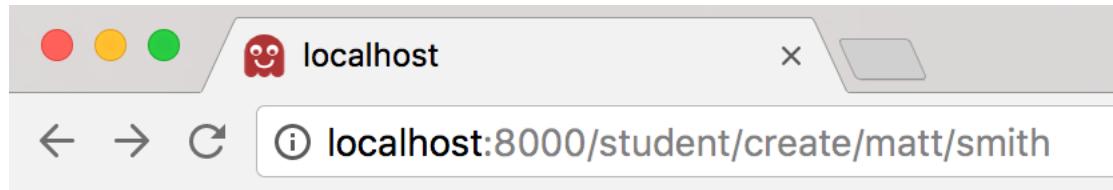


Figure 7.1: Creating new student via route /students/create/{firstName}/{surname}.

A screenshot of the PHPMyAdmin interface. The SQL tab contains the following code:

```
use web6;
select * from student;
```

The Result Grid shows a single row of data:

	id	first_name	surname
▶	1	matt	smith
	NULL	NULL	NULL

Figure 7.2: Controller created records in PHPMyAdmin.

## 7.2 Query database with SQL from CLI server

The `doctrine:query:sql` CLI command allows us to run SQL queries to our database directly from the CLI. Let's request all `Product` rows from table `product`:

```
$ php bin/console doctrine:query:sql "select * from student"

.../vendor/doctrine/common/lib/Doctrine/Common/Util/Debug.php:71:
array (size=1)
  0 =>
    array (size=3)
      'id' => string '1' (length=1)
      'first_name' => string 'matt' (length=4)
      'surname' => string 'smith' (length=5)
```

## 7.3 Updating the `listAction()` to use Doctrine

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

```
$studentRepository = $this->getDoctrine()->getRepository('App: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->findOneByFirstName('matt');

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

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

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

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

Our `listAction()` method now looks as follows:

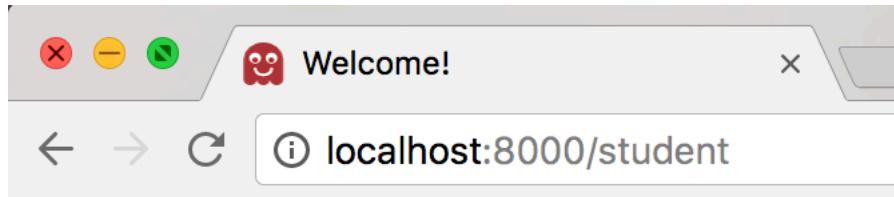
```
public function listAction()
{
```

```
$studentRepository = $this->getDoctrine()->getRepository('App:Student');
$students = $studentRepository->findAll();

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

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

Figure 7.3 shows Twig HTML page listing all students generated from route /student.



## Student LIST page

- id = 1  
name = matt smith  
[\(details\)](#)
- id = 2  
name = fred murphy  
[\(details\)](#)

Figure 7.3: Listing all database student records with route /student.

## 7.4 Deleting by id

Let's define a delete route /student/delete/{id} and a delete() controller method. This method needs to first retrieve 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("/student/delete/{id}")
```

```
/*
public function delete($id)
{
    // entity manager
    $em = $this->getDoctrine()->getManager();
    $studentRepository = $this->getDoctrine()->getRepository('App:Student');

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

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

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

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

## 7.5 Updating given id and new name

We can do something similar to update. In this case we need 3 parameters: the id and the new first and surname. 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("/student/update/{id}/{newFirstName}/{newSurname}")
 */
public function update($id, $newFirstName, $newSurname)
{
    $em = $this->getDoctrine()->getManager();
    $student = $em->getRepository('App:Student')->find($id);

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

```
$student->setFirstName($newFirstName);  
$student->setSurname($newSurname);  
$em->flush();  
  
return $this->redirectToRoute('homepage');  
}
```

Until we write an error handler we'll get Symfony style exception pages, such as shown in Figure 7.4 when trying to update a non-existent student with id=99.

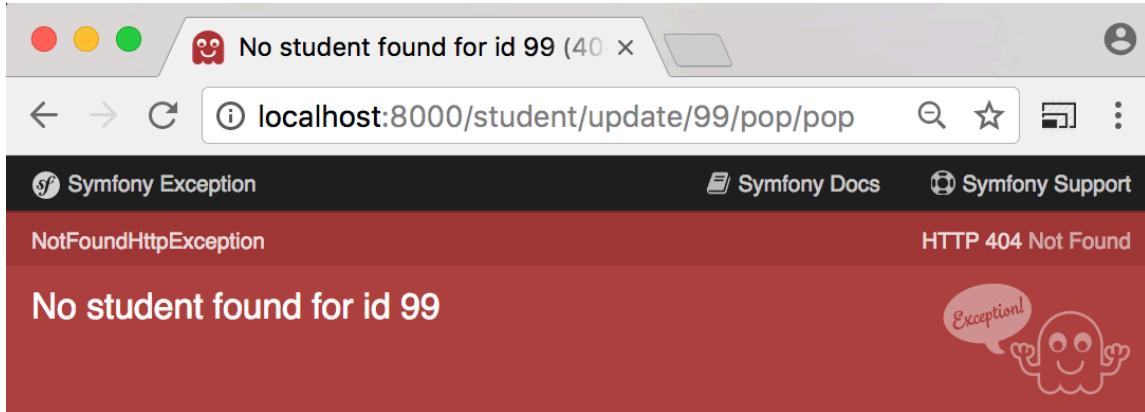


Figure 7.4: Listing all database student records with route /students/list.

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

- we are getting the reference to the repository via the entity manager `$em->getRepository('App:Student')`
- we are ‘chaining’ the `find($id)` method call onto the end of the code to get a reference to the repository (rather than storing the repository object reference and then invoking `find($id)`). This is an example of using the ‘fluent’ interface<sup>1</sup> offered by Doctrine (where methods finish by returning a 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 `delete()` method ...

## 7.6 Updating our show action

We can now update our code in our `show(...)` to retrieve the record from the database:

---

<sup>1</sup>read about it at [Wikipedia](#)

```
public function show($id)
{
    $em = $this->getDoctrine()->getManager();
    $student = $em->getRepository('App:Student')->find($id);
```

So our full method for the show action looks as follows:

```
/**
 * @Route("/student/{id}", name="student_show")
 */
public function shown($id)
{
    $em = $this->getDoctrine()->getManager();
    $student = $em->getRepository('App:Student')->find($id);

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

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

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

We could, if we wish, throw a 404 error exception if no student records can be found for the given id, rather than rendering an error Twig template:

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

## 7.7 Redirecting to show after create/update

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

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

Let's go back to the list page after a create or update action. Tell Symfony to redirect to the `student_show` route for

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

e.g. refactor the `update()` method to be as follows:

```
/**
 * @Route("/student/update/{id}/{newFirstName}/{newSurname}")
 */
public function update($id, $newFirstName, $newSurname)
{
    $em = $this->getDoctrine()->getManager();
    $student = $em->getRepository('App:Student')->find($id);

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

    $student->setFirstName($newFirstName);
    $student->setSurname($newSurname);
    $em->flush();

    return $this->redirectToRoute('student_show', [
        'id' => $student->getId()
    ]);
}
```

## 7.8 Given id let Doctrine find Product automatically (project db03)

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

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

```
/**
```

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

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

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

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

to just:

```
/**
 * @Route("/student/{id}", name="student_show")
*/
public function show(Student $student)
{
    $template = 'student/show.html.twig';
    $args = [
        'student' => $student
    ];

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

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

The Param-Converter will use the Doctrine ORM to go off, find the `ProductRepository`, run a `find(<id>)` query, and return the retrieved object for us!

Note - if there is no record in the database corresponding to the `id` then a 404-not-found error page will be generated.

Learn more about the Param-Converter on the Symfony documentation pages:

- <https://symfony.com/doc/current/doctrine.html#automatically-fetching-objects-paramconverter>
- <http://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/convertisers.html>

Likewise for delete action:

```
/**
 * @Route("/student/delete/{id}")
 */
public function delete(Student $student)
{
    // entity manager
    $em = $this->getDoctrine()->getManager();

    // store ID before deleting, so can report ID later
    $id = $student->getId();

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

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

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

Likewise for update action:

```
/**
 * @Route("/student/update/{id}/{newFirstName}/{newSurname}")
 */
public function update(Student $student, $newFirstName, $newSurname)
{
    $em = $this->getDoctrine()->getManager();

    $student->setFirstName($newFirstName);
    $student->setSurname($newSurname);
    $em->flush();

    return $this->redirectToRoute('student_show', [
        'id' => $student->getId()
    ]);
}
```

```
]);  
}
```

NOTE - we will now get ParamConverter errors rather than 404 errors if no record matches ID through ...

## 7.9 Creating the CRUD controller automatically from the CLI (project db04)

Here is something you might want to look into - automatic generation of controllers and Twig templates (we'll look at this in more detail in a later chapter).

NOTE: If trying out thew CRUD generation below, then make a copy of your current project, and try this out on the copy. Then discard the copy, so you can carry on working on your student project in the next chapter.

To try this out do the following:

1. Delete the `StudentController` class, since we'll be generating one automatically
2. Delete the `templates/student` directory, since we'll be generating those templates automatically
3. Then use the `make crud` command:

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

You should see the following output in the CLI:

```
$ php bin/console make:crud Student  
  
created: src/Controller/StudentController.php  
created: src/Form/StudentType.php  
created: templates/student/_delete_form.html.twig  
created: templates/student/_form.html.twig  
created: templates/student/edit.html.twig  
created: templates/student/index.html.twig  
created: templates/student/new.html.twig  
created: templates/student/show.html.twig
```

Success!

Next: Check your new CRUD by going to `/student/`

## CHAPTER 7. SYMFONY APPROACH TO DATABASE CRUD

You should find that you have now forms for creating and editing Student records, and controller routes for listing and showing records, and Twig templates to support all of this...

NOTE: As usualy, if you get any messages about ‘Route not found’ or whatever, you need to delete the `/var/cache` ...



# 8

## Fixtures - setting up a database state

### 8.1 Initial values for your project database (project db05)

Fixtures play two roles:

- inserting initial values into your database (e.g. the first `admin` user)
- setting up the database to a known state for **testing** purposes

Doctrine provides a Symfony fixtures **bundle** that makes things very straightforward.

Learn more about Symfony fixtures at:

- [Symfony website fixtures page](#)

### 8.2 Installing and registering the fixtures bundle

#### 8.2.1 Install the bundle

Use Composer to install the bundle in the the `/vendor` directory:

```
composer req orm-fixtures
```

You should now see a new directory created `/src/DataFixtures`. Also there is a sample fixtures class provided `AppFixtures`:

```
<?php

namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;

class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        // $product = new Product();
        // $manager->persist($product);

        $manager->flush();
    }
}
```

### 8.3 Writing the fixture classes

We need to locate our fixtures in our `/src` directory, inside a `/DataFixtures` directory. The path for our data fixtures classes should be `/src/DataFixtures/`.

Fixture classes need to implement the interfaces, `Fixture`.

NOTE: Some fixtures will also require your class to include the `ContainerAwareInterface`, for when our code also needs to access the container, by implementing the `ContainerAwareInterface`.

Let's write a class to create 3 objects for entity `App\Entity\Student`. The class will be declared in file `/src/DataFixtures/StudentFixtures.php`. Make a copy of the provided `AppFixtures` class naming the copy `StudentFixtures`, and change the class name inside the code.

We also need to add a `use` statement so that our class can make use of the `Entity\Student` class.

The `make` feature will create a skeleton fixture class for us. So let's make class `StudentFixtures`:

```
$ php bin/console make:fixtures StudentFixtures
```

```
created: src/DataFixtures/StudentFixtures.php
```

```
Success!
```

Next: Open your new fixtures class and start customizing it.

```
Load your fixtures by running: php bin/console doctrine:fixtures:load  
Docs: https://symfony.com/doc/master/bundles/DoctrineFixturesBundle/index.html
```

Since we are going to be creating instance-objects of class `Student` we need to add a `use` statement:

```
...
```

```
use App\Entity\Student;  
  
class StudentFixtures extends Fixture  
{
```

Now we need to implement the details of our `load(...)` method, that gets invoked when we are loading fixtures from the CLI. This method creates objects for the entities we want in our database, and the saves (persists) them to the database. Finally the `flush()` method is invoked, forcing the database to be updated with all queued new/changed/deleted objects:

In the code below, we create 3 `Student` objects and have them persisted to the database.

```
public function load(ObjectManager $manager)  
{  
    $s1 = new Student();  
    $s1->setFirstName('matt');  
    $s1->setSurname('smith');  
    $s2 = new Student();  
    $s2->setFirstName('joe');  
    $s2->setSurname('bloggs');  
    $s3 = new Student();  
    $s3->setFirstName('joelle');  
    $s3->setSurname('murph');  
  
    $manager->persist($s1);  
    $manager->persist($s2);  
    $manager->persist($s3);  
  
    $manager->flush();  
}
```

## 8.4 Loading the fixtures

**WARNING** Fixtures **replace** existing DB contents - so you'll lose any previous data when you load fixtures...

Loading fixtures involves deleting all existing database contents and then creating the data from the fixture classes - so you'll get a warning when loading fixtures. At the CLI type:

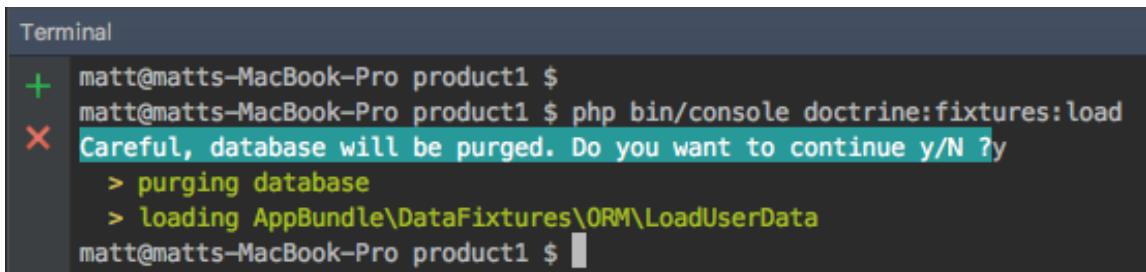
```
php bin/console doctrine:fixtures:load
```

You should then be asked to enter y (for YES) if you want to continue:

```
$ php bin/console doctrine:fixtures:load
```

```
Careful, database will be purged. Do you want to continue y/N ?y
> purging database
> loading App\DataFixtures\LoadStudents
```

Figure 8.1 shows an example of the CLI output when you load fixtures (in the screenshot it was for initial user data for a login system...)



```
Terminal
+ matt@matts-MacBook-Pro product1 $
matt@matts-MacBook-Pro product1 $ php bin/console doctrine:fixtures:load
✖ Careful, database will be purged. Do you want to continue y/N ?y
    > purging database
    > loading AppBundle\DataFixtures\ORM\LoadUserData
matt@matts-MacBook-Pro product1 $
```

Figure 8.1: Using CLI to load database fixtures.

Alternatively, you could execute an SQL query from the CLI using the `doctrine:query:sql` command:

```
$ php bin/console doctrine:query:sql "select * from student"

/.../db06_fixtures/vendor/doctrine/common/lib/Doctrine/Common/Util/Debug.php:71:
array (size=3)
0 =>
    array (size=3)
        'id' => string '13' (length=2)
        'first_name' => string 'matt' (length=4)
        'surname' => string 'smith' (length=5)
1 =>
    array (size=3)
        'id' => string '14' (length=2)
        'first_name' => string 'joe' (length=3)
        'surname' => string 'bloggs' (length=6)
2 =>
    array (size=3)
```

```
'id' => string '15' (length=2)
'first_name' => string 'joelle' (length=6)
'surname' => string 'murph' (length=5)
```

## 8.5 User Faker to generate plausible test data (project db06)

For testing purposes the `Faker` library is fantastic for generating plausible, random data.

Let's install it and generate some random students in our Fixtures class:

1. use Composer to add the Faker package to our `/vendor/` directory:

```
$ composer req fzaninotto/faker

Using version ^1.7 for fzaninotto/faker
./composer.json has been updated
Loading composer repositories with package information
...
Executing script assets:install --symlink --relative public [OK]
```

2. Add a `uses` statement in our `/src/DataFixtures/LoadStudents.php` class, so that we can make use of the `Faker` class:

```
use Faker\Factory;
```

2. refactor our `load()` method in `/src/DataFixtures/LoadStudents.php` to create a Faker 'factory', and loop to generate names for 10 male students, and insert them into the database:

```
public function load(ObjectManager $manager) {
    $faker = Factory::create();

    $numStudents = 10;
    for ($i=0; $i < $numStudents; $i++) {
        $firstName = $faker->firstNameMale;
        $surname = $faker->lastName;

        $student = new Student();
        $student->setFirstName($firstName);
        $student->setSurname($surname);

        $manager->persist($student);
    }
}
```

```
$manager->flush();  
}
```

- use the CLI Doctrine command to run the fixtures creation method:

```
$ php bin/console doctrine:fixtures:load  
Careful, database will be purged. Do you want to continue y/N ?y  
> purging database  
> loading App\DataFixtures\LoadStudents
```

That's it - you should now have 10 'fake' students in your database.

Figure 8.2 shows a screenshot of the DB client showing the 10 created 'fake' students.

The screenshot shows a MySQL Workbench interface. The SQL tab contains the following code:

```
1 • use web6;  
2  
3 • select * from student;
```

The Result Grid shows the following data:

	id	first_name	surname
▶	1	Perry	Spinka
	2	Nat	Satterfield
	3	Efren	Rutherford
	4	Hilton	Macejkovic
	5	Wilhelm	Rolfson
	6	Jose	Ruecker
	7	Janick	Funk
	8	Ole	Kreiger
	9	Kenneth	Feil
	10	Alf	Brown
	NULL	NULL	NULL

Figure 8.2: Ten fake students inserted into DB.

Learn more about the Faker class at its Github project page:

- <https://github.com/fzaninotto/Faker>

## **Part III**

# **Froms and form processing**



# 9

## DIY forms

### 9.1 Preparation

This project assumes you are working with a copy of project db03 - i.e. with a D.I.Y. controller and templates.

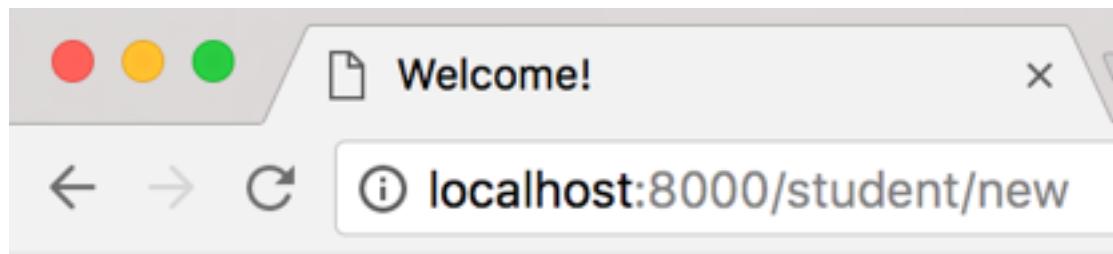
You can start with a copy from the book Github repositories if you wish: - <https://github.com/dr-matt-smith/php-symfony-5-book-codes-databases-03-param-converter>

### 9.2 Adding a form for new Student creation (project `form01`)

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 `/student/new`
    - \* with internal route name `student_new_form`
- a controller method to process the submitted form data
  - route `/student/processNewForm`
    - \* with internal route name `student_process_new_form`

The form will look as show in Figure 9.1.



# Create new student

First Name:

Surname:

Figure 9.1: Form for a new student

### 9.3 Refactor our `create(...)` method

Since we will now be creating new Students using a form, rather than passing the properties directly as GET parameters in the URL, let's refactor our `create(...)` in class `StudentController` to be a private method (no route annotation comment), that accepts 2 parameters and uses them to create a new Student object and store it in the database, then redirect to the list of students page:

```
private function create($firstName, $surname)
{
    $student = new Student();
    $student->setFirstName($firstName);
    $student->setSurname($surname);

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

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

### 9.4 Twig new student form

Here is our new student form '/templates/student/new.html.twig':

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

{% block pageTitle %}new student form{% endblock %}

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

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

## 9.5 Controller method (and annotation) to display new student form

Let's add a `new` action to `StudentController`.

NOTE: This should be the **FIRST** method in this class - Since we don't want `/student/new` being treated as `/student/{id = 'new'}`, so our new form action method should be placed before our show action. <<<<<

Here is our `StudentController` method `newForm()` to display our Twig form:

```
/**  
 * @Route("/student/new", name="student_new")  
 */  
public function newForm()  
{  
    $template = 'student/new.html.twig';  
    $args = [  
    ];  
    return $this->render($template, $args);  
}
```

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

```
(... existing Twig code to show list of students here ...)  
  
<hr>  
<a href="{{ path('student_new_form') }}">  
    create NEW student  
</a>  
{% endblock %}
```

## 9.6 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 `$firstName` and `$surname` the POST `firstName` and `surname` parameters by writing the following:

```
$firstName = $request->request->get('firstName');  
$surname = $request->request->get('surname');
```

We will need access to the HTTP request, so we must declare a method parameter of `Request $request`. Symfony will now automatically provide this method with access to an object `$request`, which we can interrogate for things like the HTTP method of the request, and any name/value variables received in the request:

```
public function processNewFormAction(Request $request)
{
```

Note: We have not **namespaced** class `Request`, so, at the top of our controller class declaration, we need to add an appropriate `use` statement, so PHP knows **which Request class** we are referring to. So we need to add the following before the class declaration:

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

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

```
/**
 * @Route("/student/processNewForm", name="student_process_new_form")
 */
public function processNewForm(Request $request)
{
    // extract name values from POST data
    $firstName = $request->request->get('firstName');
    $surname = $request->request->get('surname');

    // forward this to the createAction() method
    return $this->create($firstName, $surname);
}
```

NOTE: that we then invoke our existing `createAction(...)` method, passing on the extracted `$firsName` and `$surname` strings.

NOTE: This should be **the FIRST** method in this class (or at least before the `show` method )- Since we don't want `/student/processNewForm` being treated as `/student/{id = 'new'}`, so our new form action method should be placed before our show action. If you get a Param Converter exception Student object not found you put this method **after** the show method... <<<<<<

## 9.7 Validating form data, and displaying temporary ‘flash’ messages in Twig

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.

## 9.8 Three kinds of flash message: notice, warning and error

Typically we create 3 different kinds of flash notice:

- notice
- warning
- error

Our Twig template would style these differnly (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 %}
```

## 9.9 Adding flash display (with CSS) to our Twig template (project `form02`)

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 `/public` - remember that `/public` 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 `/public/css/flash.css` containing the following:

```
.flash-error {  
    padding: 1rem;  
    margin: 1rem;
```

```
background-color: pink;  
}
```

Next we need to edit our `/templates/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>
```

## 9.10 Adding validation logic to our form processing controller method

Our form data is valid if **neither** name received was empty:

```
$isValid = !empty($firstName) && !empty($surname);
```

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

```
public function processNewForm(Request $request)  
{  
  // extract name values from POST data  
  $firstName = $request->request->get('firstName');  
  $surname = $request->request->get('surname');  
  
  // valid if neither value is EMPTY  
  $isValid = !empty($firstName) && !empty($surname);  
  if(!$isValid){  
    $this->addFlash(  
      'error',  
      'student firstName/surname cannot be an empty string'  
    );  
  }  
  // forward this to the createAction() method
```

```
    return $this->newForm();
}

// forward this to the createAction() method
return $this->create($firstName, $surname);
}
```

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 `/templates/student/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 ...)
```

## 9.11 Postback logic (project form03)

A common approach (and used in CRUD auto-generated code) is to combine the logic for displaying a form, and processing its submission, in a single method. The logic for this is that if any of the submitted data was invalid (or missing), then the default form processing can go back to re-displaying the form (with an appropriate ‘flash’ error message) to the user.

This approach is known as a ‘postback’ - i.e. that the submission of the form is POSTEd back to the same method that displayed the form.

The logic usually goes something like this:

1. Define a controller method route for both GET and POST HTTP methods
2. Attempt to find values in the POST request body

3. If the form was submitted by the POST method AND the data was all valid THEN
  - invoke the method to create the object/process the data and return an appropriate success response
4. (else) If POST submitted but NOT valid THEN
  - create an appropriate flash error message in the session
5. return a response showing the form via Twig `render(...)` method
  - passing values, if we want a ‘sticky’ form remembering partly valid form values

Let’s name our combined show form & process form controller method `newAction(...)`, name its internal route as `student_new`, and declare that only POST and GET HTTP requests are to be routed to this method<sup>1</sup>:

```
/**  
 * @Route("/student/new", name="student_new", methods={"POST", "GET"})  
 */  
public function newAction(Request $request)  
{
```

Remember, we will need access to the `Request` object to get access to the POST values, and to check with HTTP method the request was sent via.

The simplest request will be for the new student form to be displayed, the logic for that is from our old `newFormAction()`:

```
// render the form for the user  
$template = 'student/new.html.twig';  
$argsArray = [  
];  
  
return $this->render($template, $argsArray);
```

The rest of the logic in this method will relate to when the HTTP request is POST-submission of the form, and its validation. We can check whether the HTTP request was received as follows:

```
$isSubmitted = $request->isMethod('POST');
```

We can attempt to retrieve values from a POST submitted form as follows:

```
// attempt to find values in POST variables  
$firstName = $request->request->get('firstName');  
$surname = $request->request->get('surname');
```

<sup>1</sup>By default a controller method that does not declare any specific HTTP methods will be used for **any** HTTP method matching the route pattern. So it is good practice to start limiting our controller methods to only those HTTP methods that are valid for how we wish our web application to behave...

Note: If there was no named variable in the POST data, the variables `$firstName` and `$surname` will return `null` (and so will register as `true` when tested with `isEmpty(...)`).

If our form validation logic is simply that neither name can be an empty string (or null), then we can write an expression to check that neither is empty as follows:

```
$isValid = !empty($firstName) && !empty($surname);
```

Our core logic for this controller is that **if** the request was an HTTP POST method **and** the values received were value, then we are happy to accept the form data and go off and create a new object (and return an appropriate response). We can write this as follows:

```
// if SUBMITTED & VALID - go ahead and create new object
if ($isSubmitted && $isValid) {
    return $this->createAction($firstName, $surname);
}
```

NOTE: Since our method is invoking a `return`, then no further processing of statements in the method will occur. I.e. we can locate our logic for (re)displaying the form after this `if`-test.

If it was a POST submitted form but the data was **not** valid, then we should create a ‘flash’ error message in the session:

```
if ($isSubmitted && !$isValid) {
    $this->addFlash(
        'error',
        'student firstName/surname cannot be an empty string'
    );
}
```

We can now simply replace the previous 2 methods `processNewFormAction()` and `newFormAction()` with our new single postback method `new(...)` as follows:

```
/**
 * @Route("/student/new", name="student_new_form", methods={"POST", "GET"})
 */
public function new(Request $request) {
    // attempt to find values in POST variables
    $firstName = $request->request->get('firstName');
    $surname = $request->request->get('surname');

    // valid if neither value is EMPTY
    $isValid = !empty($firstName) && !empty($surname);

    // was form submitted with POST method?
    $isSubmitted = $request->isMethod('POST');
```

```
// if SUBMITTED & VALID - go ahead and create new object
if ($isSubmitted && $isValid) {
    return $this->create($firstName, $surname);
}

if ($isSubmitted && !$isValid) { $this->addFlash(
    'error',
    'student firstName/surname cannot be an empty string'
);
}

// render the form for the user
$template = 'student/new.html.twig';
$args = [
    'firstName' => $firstName,
    'surname' => $surname
];
return $this->render($template, $args);
}
```

Finally (!) we can achieve a ‘sticky’ form by passing any value in `$firstName` and `$surname` to our Twig template in its argument array:

```
$argsArray = [
    'firstName' => $firstName,
    'surname' => $surname
];
```

These will either be null, or have the string values from the POST submitted form attempt. We re-display these values (if no null) by adding `value=""` attributed in our Twig form template `/templates/student/new.html.twig` as follows:

```
<form action="/student/new" method="POST">
    First Name:
    <input type="text" name="firstName" value="{{ firstname }}>
<p>
    Surname:
    <input type="text" name="surname" value="{{ surname }}>
<p>
    <input type="submit" value="Create new student">
</form>
```

NOTE: We have **changed** the form action to `"/student/new"`, so that the form POST submission will be routed to the same method (`new()`) as the one to display the form.

## 9.12 Extra notes

Here is how to work with Enum style drop-down combo-boxes:

- Article on Symfony Enums in forms from Maxence POUTORD

# 10

## Automatic forms generated from Entities

### 10.1 Using the Symfony form generator (project `form04`)

Given an object of an Entity class, Symfony can analyse its property names and types, and generate a form (with a little help). That's what we'll do in this chapter.

However, first, let's simplify something for later, we'll make our `createAction()` expect to be given a reference to a `Student` object (rather than expect 2 string parameters `firstName` and `surname`):

```
public function createAction(Student $student)
{
    $em = $this->getDoctrine()->getManager();
    $em->persist($student);
    $em->flush();

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

### 10.2 The Symfony form generator via Twig

In a controller we can create a `$form` object, and pass this as a Twig variable to the template `form`. Twig offers 4 special functions for rendering (displaying) forms, these are:

- `form()` :: display the whole form (i.e. display the whole thing in one line!)

- `form_start()` :: display the beginning of the form
- `form_widget()` :: display all the fields etc.
- `form_end()` :: display the end of the form

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 errors by each form field).

The 'magic' happens in the controller method...

### 10.3 Updating `StudentController->new()`

First, our controller method will need to pass a Twig variable `form` to the `render()` method. This will be created for us by the `createView()` method of a Symfony form object. So `new()` will end as follows:

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

Our method will use Symfony's FormBuilder to create the form for us, based on an instance of class `Student`. First we create a new, empty `Student` object, and then use Symfony's `createFormBuilder()` method to create a form based on the Entity class of our `$student` object:

```
public function new(Request $request)  
{  
    // create a new Student object  
    $student = new Student();  
  
    // create a form with 'firstName' and 'surname' text fields  
    $form = $this->createFormBuilder($student)  
        ->add('firstName', TextType::class)
```

```
->add('surname', TextType::class)
->add('save', SubmitType::class, array('label' => 'Create Student'))->getForm();
```

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 ask Symfony to ‘handle’ the request for us. If the HTTP request was a POST submission, then the submitted values will be used to populate our `$student` object. Otherwise, if GET method, the form will be an empty, new form.

```
// if was POST submission, extract data and put into '$student'
$form->handleRequest($request);
```

Forms have basic validation. The default for text entity properties is NOT NULL, so both name fields will be validated this way - both through HTML 5 validation and on the server side. If the form was submitted (via POST) and is valid, then we’ll go ahead and create a new `Student` object as before:

```
// if SUBMITTED & VALID - go ahead and create new object
if ($form->isSubmitted() && $form->isValid()) {
    return $this->createAction($student);
}
```

If not submitted (or not valid), then the logic falls through to displaying the form via Twig. The full listing for our improved `new()` method is as follows:

```
/*
 * @Route("/student/new", name="student_new", methods={"POST", "GET"})
 */
public function new(Request $request)
{
    // create a task and give it some dummy data for this example
    $student = new Student();

    // create a form with 'firstName' and 'surname' text fields
    $form = $this->createFormBuilder($student)
        ->add('firstName', TextType::class)
        ->add('surname', TextType::class)
        ->add('save', SubmitType::class, array('label' => 'Create Student'))->getForm();

    // if was POST submission, extract data and put into '$student'
    $form->handleRequest($request);
```

```
// if SUBMITTED & VALID - go ahead and create new object
if ($form->isSubmitted() && $form->isValid()) {
    return $this->createAction($student);
}

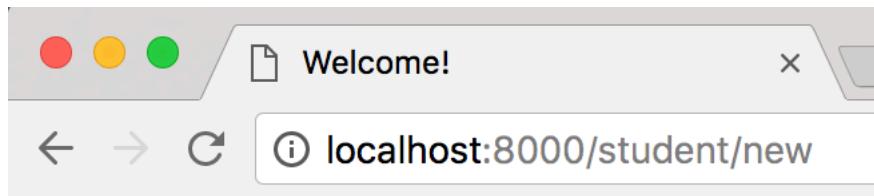
// render the form for the user
$template = 'student/new.html.twig';
$argsArray = [
    'form' => $form->createView(),
];

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

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 **student/new.html.twig**.

Figure 10.1 shows a screenshot of the resulting form.



- [student list](#)

# Create new student

First name

Surname

Figure 10.1: Symfony generated new student form.

## 10.4 Postback - form submits to same URL

If we look at the HTML in the source of our web page (see Figure 10.2), we can see that the form has no `action` attribute, which means that when POST submitted, it will be submitted to the same URL (i.e. a our method `new()`). However, since we've already written our logic to process a **post-back** like this, then our code will work :-)

```
▼<form name="form" method="post">
  ▼<div id="form">
    ▼<div>
      <label for="form(firstName" class="required">First name</label>
      <input type="text" id="form(firstName" name="form[firstName]">
    </div>
    ▼<div>
      <label for="form(surname" class="required">Surname</label>
      <input type="text" id="form(surname" name="form[surname]" requ
    </div>
    ▼<div>
      <button type="submit" id="form_save" name="form[save]">Create
    </div>
      <input type="hidden" id="form_token" name="form[_token]" value=
        gKxeirThZWAfisD6wYffCP5UP8SIx2rYt8">
    </div>
  </form>
```

Figure 10.2: HTML source of generated form

## 10.5 Using form classes (project `form05`)

Although simple forms can be created inside a controller method as above, it's good practice to create a separate from 'type' class to create each form.

Rather than write one from scratch, some of the work can be done for us using the `maker` bundle. To create class `/src/Form/StudentType.php` we first enter CLI command:

```
$ php bin/console make:form
```

You'll then be asked the form class name - by Symfony convention we just add `Type` to the Entity class name:

```
The name of the form class (e.g. VictoriousPuppyType):
> Student
```

You'll then be asked for the Entity name, so we enter `Student`:

```
The name of Entity or fully qualified model class name that the new form will be bound to (e
> Student
```

You'll then see output telling us that the make tool has generated Form class `StudentType` for us in the `src/Form/` directory:

```
created: src/Form/StudentType.php
```

```
Success!
```

Next: Add fields to your form and start using it.

Find the documentation at <https://symfony.com/doc/current/forms.html>

If we look inside `/src/Form/StudentType.php` we see a skeleton class as follows:

```
namespace App\Form;

use App\Entity\Student;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class StudentType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('firstName')
            ->add('surname')
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Student::class,
        ]);
    }
}
```

For the `/Form/StudentType.php` class we need to:

- add `use` statements for the `SubmitType` we want to use (it works out for itself the `TextType` from the Entity annotations)

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

- write a statement to add a submit button to the form:

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('firstName')
        ->add('surname')
```

```
        ->add('save', SubmitType::class, array('label' => 'Create Student'))  
    ;  
  
}
```

That's our `StudentType` form class complete.

For the `/Controller/StudentController.php` class we need to:

- remove the `use` statements for `TextType` and `SubmitType`
- add a `use` statement for the `StudentType` class we have just created:

```
use App\Form\StudentType;
```

- simplify our controller method, which can create the form in a single statement:

```
$form = $this->createForm(StudentType::class, $student);
```

So our refactored `new()` controller method looks as follows:

```
public function new(Request $request)  
{  
    $student = new Student();  
  
    $form = $this->createForm(StudentType::class, $student);  
  
    $form->handleRequest($request);  
  
    if ($form->isSubmitted() && $form->isValid()) {  
        return $this->createAction($student);  
    }  
  
    $template = 'student/new.html.twig';  
    $argsArray = [  
        'form' => $form->createView(),  
    ];  
  
    return $this->render($template, $argsArray);  
}
```

Figure 10.3 shows a screenshot of the HTML validation from the generated form (empty values not accepted due to `=required` attribute in the text `<input>` tags).

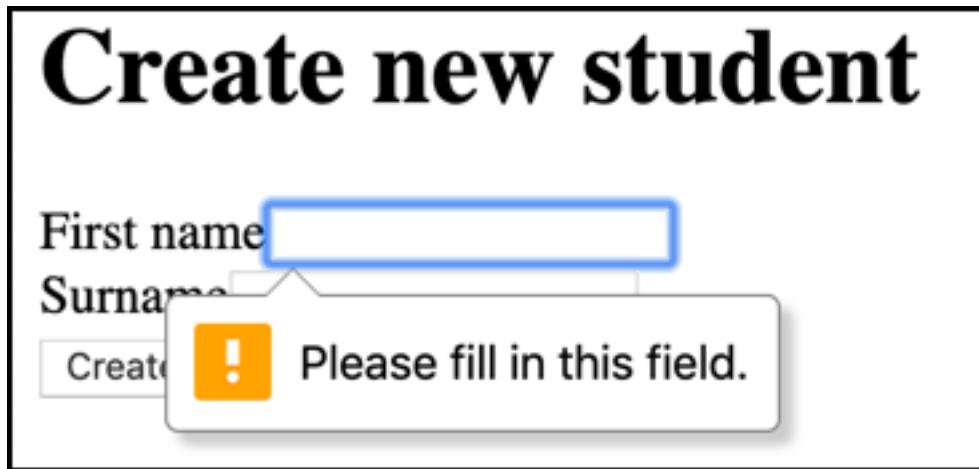


Figure 10.3: HTML validation preventing empty text submissions.

## 10.6 Video tutorials about Symfony forms

Here are some resources on this topic:

- Example of a numeric ‘greater than’ constraint in Entity class
- Video: Code Review form validation with `@Assert`



# 11

## Customising the display of generated forms

### 11.1 First let's Bootstrap this project (project form06)

Since the Twig Symfony component allows custom themes, of which Bootstrap 4 is one of them, it is relatively easy to add Bootstrap to our website.

A great advantage of adding Bootstrap via a Twig theme is that components, such as the Form generation component, know about themes and will use them to decorate their output. So our form fields and buttons will make use of Bootstrap structures and CSS classes once we add this theme.

To add Bootstrap to a Symfony project we need to do 3 things:

1. Configure Twig to use the Bootstrap theme.
2. Add the Bootstrap CSS import into our base Twig template.
3. Add the Bootstrap JavaScript import into our base Twig template.

Learn more about the Bootstrap 4 theme on the Symfony documentation pages:

- <https://symfony.com/doc/current/form/bootstrap4.html>

### 11.2 Configure Twig to use the Bootstrap theme

Well Symfony to generate forms using the Bootstrap theme by adding:

```
form_themes: ['bootstrap_4_layout.html.twig']
```

to `/config/packages/twig.yml` file. So this file should now look as follows;

```
twig:  
    paths: ['%kernel.project_dir%/templates']  
    debug: '%kernel.debug%'  
    strict_variables: '%kernel.debug%'  
    form_themes: ['bootstrap_4_layout.html.twig']
```

### 11.3 Add the Bootstrap CSS import into our base Twig template

The Bootstrap QuickStart tells us to copy the CSS `<link>` tag from here:

- <https://getbootstrap.com/docs/4.4/getting-started/introduction/#css>

into the CSS part of our `/templates/base.html.twig` Twig template. Add this `<link>` tag just before the `stylesheets` block:

```
<!DOCTYPE html> <html>  
  <head>  
    <meta charset="UTF-8" />  
    <title>MGW - {% block pageTitle %}{% endblock %}</title>  
    <style>  
      @import '/css/flash.css';  
    </style>  
  
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.css" type="text/css"/>   
    {% block stylesheets %}{% endblock %}  
  
  </head>  
  ...
```

### 11.4 Add the Bootstrap JavaScript import into our base Twig template.

The Bootstrap QuickStart tells us to copy the JS `<script>` tags from here:

- <https://getbootstrap.com/docs/4.4/getting-started/introduction/#js>

into the last part of the `<body>` element in `/templates/base.html.twig` Twig template. Add these `<script>` tags just after the `javascripts` block:

```
...  
  
<body>  
<nav>  
    <ul>  
        <li>  
            <a href="{{ path('student_list') }}>student actions</a>  
        </li>  
    </ul>  
</nav>  
  
{% block body %}{% endblock %}  
  
{% block javascripts %}{% endblock %}  
  
<script src="https://code.jquery.com/jquery-3.4.1.slim.min.js" integrity="sha384-J6qa4849bleE2+poT4WnyKh  
<script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js" integrity="sha384-Q6E7h  
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js" integrity="sha384-Q6E7h  
  
</body>  
</html>
```

## 11.5 Run site and see some Bootstrap styling

Figure 11.1 shows a screenshot how our new Student form looks now. We can see some basic Bootstrap styling with blue buttons, and sans-serif fonts etc. But the text boxes go right to the left/right edges of the browser window, with no padding etc.

Figure 11.2 shows the HTML source - we can see no page/content `<div>` elements around the form, which are needed as part of the guidelines of using Bootstrap.

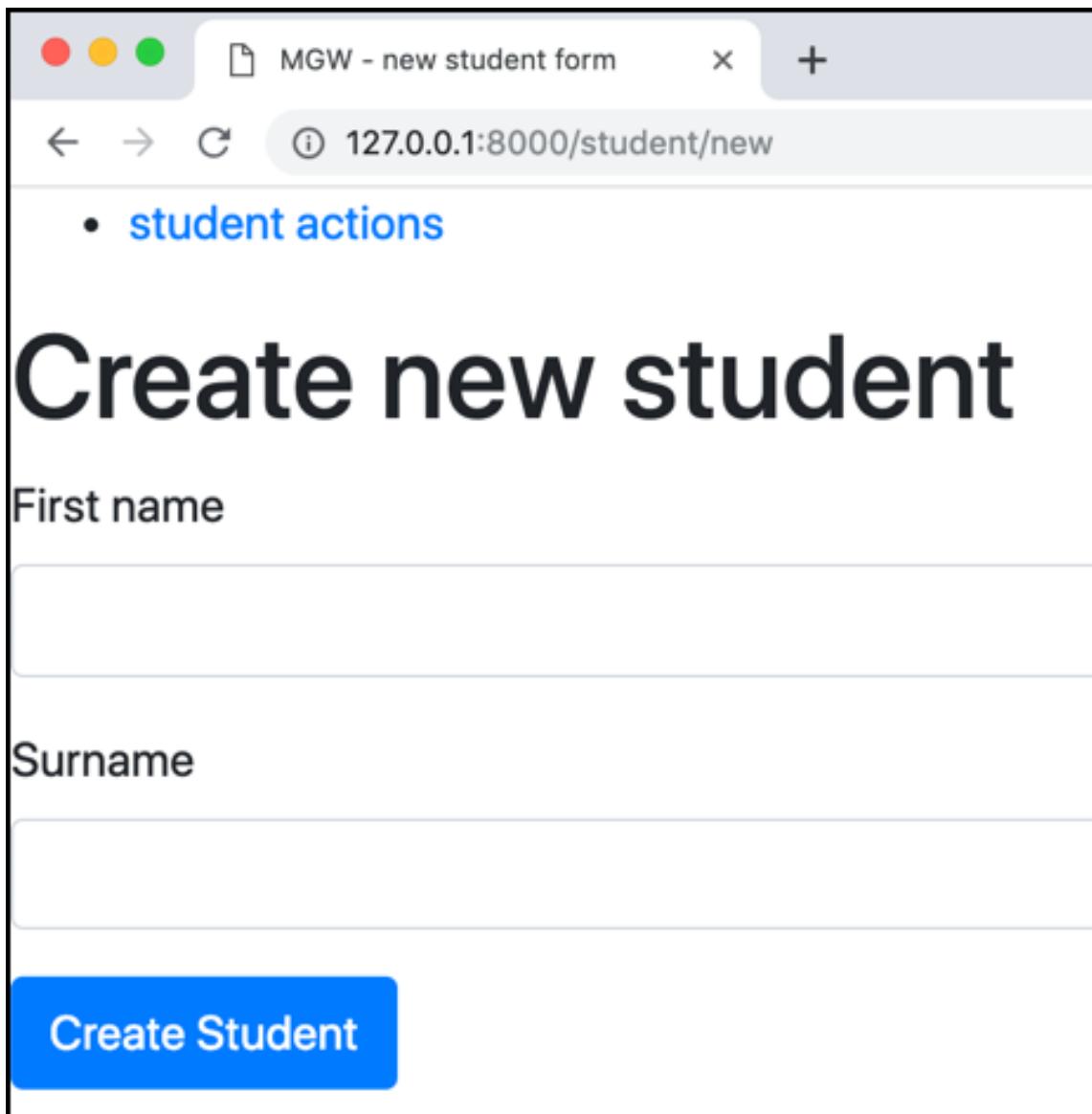


Figure 11.1: Basic Bootstrap styling of generated form.

```
13
14 <body>
15 <nav>
16     <ul>
17         <li>
18             <a href="/student">student actions</a>
19         </li>
20     </ul>
21 </nav>
22
23         <h1>Create new student</h1>
24 <form name="student" method="post"><div id="student"><div
name</label><input type="text" id="student(firstName" name="st
</div><div class="form-group"><label for="student_surname" cla
```

Figure 11.2: Basic HTML source of generated form.

## 11.6 Adding elements for navigation and page content

Let's ensure main body content of every page is inside a Bootstrap 4 element.

We need to wrap a Bootstrap container and row divs around the body Twig block.

Replace the existing body block in template base.html.twig with the following:

```
<div class="container">  
  <div class="row">  
    <div class="col-sm-12">  
      {% block body %}{% endblock %}  
    </div>  
  </div>  
</div>
```

When we visit the site not, as we can see in Figure 11.3, the page content is within a nicely styled Bootstrap container, with associated margins and padding.

• student actions

# Create new student

First name

Surname

**Create Student**

Figure 11.3: Basic HTML source of generated form.

## 11.7 Add Bootstrap navigation bar

Let's add a title to our navigation bar, declaring this site My Great Website. This should be a link to the website root (we can just link to #).

Do the following:

1. Add a new CSS stylesheet to make our navbar background BLACK. Create file /public/css/nav.css containing:

```
nav {
    background-color: black;
}
```

2. Add an @import statement for this stylesheet in the <style> element in our base.html.twig master template:

```
<!DOCTYPE html> <html>
<head>
    <meta charset="UTF-8" />
    <title>MGW - {% block pageTitle %}{% endblock %}</title>
    <style>
        @import '/css/flash.css';
```

```
@import '/css/nav.css';
</style>
```

...

3. Add some Bootstrap classes and a link around text My Great Website ! in base.html.twig:

```
<nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
    <a style="margin-left: 0.1rem;" class="navbar-brand space-brand" href="#">
        My Great Website !
    </a>

    <ul>
        <li>
            <a href="{{ path('student_list') }}>student actions</a>
        </li>
    </ul>
</nav>
```

Figure 11.4 shows our simple black navbar from our base template.

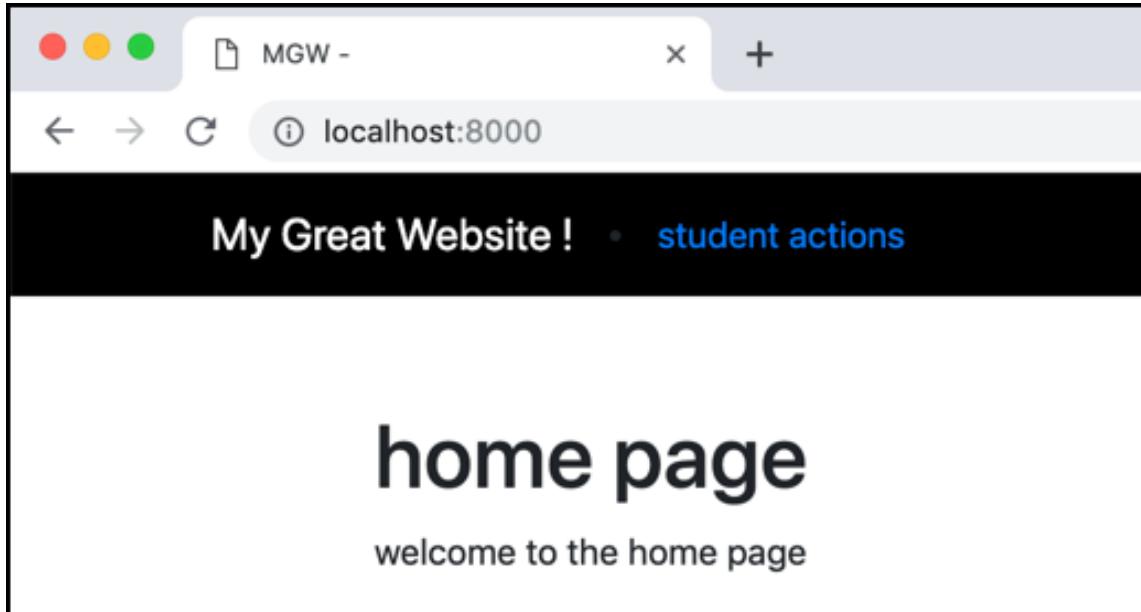


Figure 11.4: Black navbar for all website pages.

## 11.8 Styling list of links in navbar

Let's now have links for list of students and creating a NEW student, properly styled by our Bootstrap theme.

We need to add a Bootstrap styled unordered-list in the `<nav>` element, with links to routes `student_list` and `student_new`:

```

<nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
    <a style="margin-left: 0.1rem;" class="navbar-brand space-brand" href="#">
        My Great Website !
    </a>

    <ul class="navbar-nav ml-auto">
        <li class="nav-item">
            <a class="nav-link" href="{{ url('student_list') }}>
                student list
            </a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="{{ url('student_new_form') }}>
                Create NEW student
            </a>
        </li>
    </ul>

</nav>

```

Figure 11.5 shows the navbar with our 2 styled links.

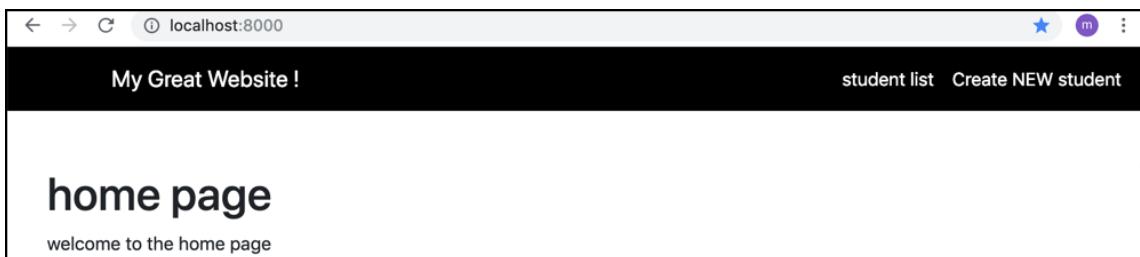


Figure 11.5: Navbar links for all website pages.

## 11.9 Adding the hamburger-menu and collapsible links

While it looks fine in the desktop, these links are lost with a narrow screen. Let's make them be replaced by a 'hamburger-menu' when the browser window is narrow.

We need to add a toggle drop-down button:

```
<button class="navbar-toggler" type="button" data-toggle="collapse"
        data-target="#navbarNavDropdown" aria-controls="navbarNavDropdown"
        aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
</button>
```

We also need to wrap a collapse `<div>` around our unordered list of links, with id of `navbarNavDropdown`, so that it's this list that is replaced by the hamburger-menu:

```
<div class="collapse navbar-collapse" id="navbarNavDropdown">
    <ul class="navbar-nav ml-auto">
        <li>...</li>
        <li>...</li>
    </ul>
</div>
```

So our complete `<nav>` element now looks as follows:

```
<nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
    <a style="margin-left: 0.1rem;" class="navbar-brand space-brand" href="#">
        My Great Website !
    </a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNavDrop
        <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse" id="navbarNavDropdown">
        <ul class="navbar-nav ml-auto">
            <li class="nav-item">
                <a class="nav-link" href="{{ url('student_list') }}> student list
            </a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="{{ url('student_new_form') }}> Create NEW student
            </a>
        </li>
    </div>
</nav>
```

```
</ul>
</div>
</nav>
```

Figure 11.4 shows our simple black navbar from our base template.

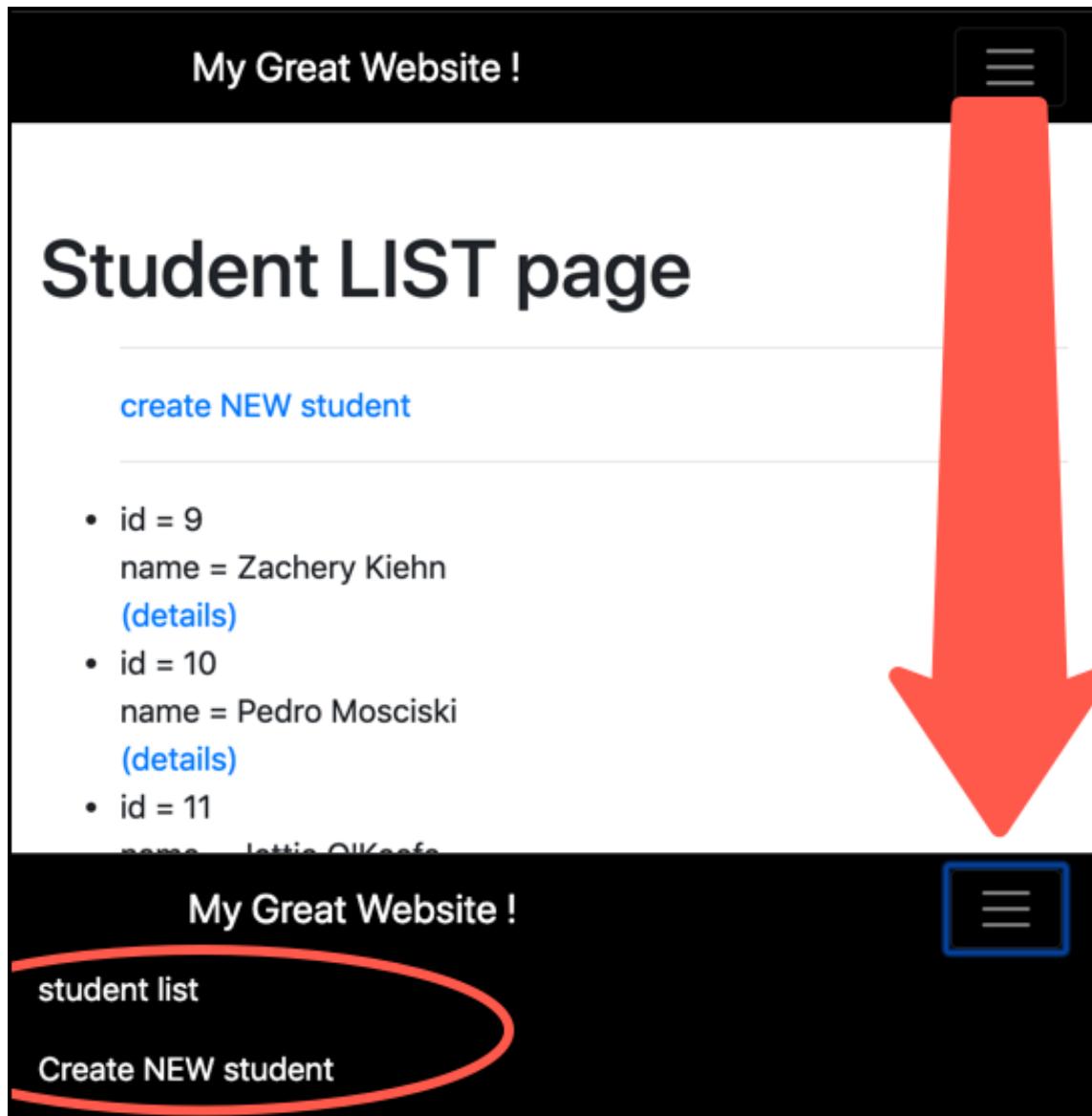


Figure 11.6: Animated hamburger links for narrow browser window.

# 12

## Customizing display of Symfony forms

### 12.1 Understanding the 3 parts of a form (project `form07`)

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.

## 12.2 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
- Bootstrap CSS framework div's and CSS classes
  - bootstrap\_4\_layout.html.twig

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

```
twig:  
    paths: ['%kernel.project_dir%/templates']  
    debug: '%kernel.debug%'  
    strict_variables: '%kernel.debug%'  
    form_themes: ['bootstrap_4_layout.html.twig']
```

## 12.3 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.firstName) }}  
    {{ form_row(form.surname) }}  
    {{ form_row(form.save) }}
```

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

## 12.4 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.firstName) }}  
  
  <div class="errors">  
    {{ form_errors(form.firstName) }}  
  </div>  
  
  {{ form_widget(form.firstName) }}  
</div>
```

So we could display our new student form this way:

```
{% block body %}  
  <h1>Create new student</h1>  
  {{ form_start(form) }}  
  
  <div class="form-group">  
    <div class="errors">  
      {{ form_errors(form.firstName) }}  
    </div>  
  
    {{ form_label(form.firstName) }}  
  
    {{ form_widget(form.firstName) }}  
  </div>  
  
  <div class="form-group">  
    <div class="errors">  
      {{ form_errors(form.surname) }}
```

```
</div>

{{ form_label(form.surname) }}

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

<div class="form-group">
    {{ form_row(form.save) }}
</div>

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

The above would output the following HTML for the `firstName` property (if the errors list was empty):

```
<div class="form-group">
    <div class="errors">

    </div>

    <label for="student(firstName" class="required">First name</label>

    <input type="text" id="student(firstName" name="student(firstName" required="required" clas
    </div>
```

Learn more at:

- [The Symfony form customisation page](#)

## 12.5 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**

# **Custom Repository Queries with forms**



# 13

## Custom database queries

### 13.1 Search for exact property value (project query01)

Let's create a simple database schema for hardware products, and then write some forms to query this database.

Use `make:entity` to create a new Entity class `Product`, with the following properties:

- `description: String`
- `price: Float`
- `category: String`

Use `make:crud` to generate the CRUD pages for `Product` entities.

### 13.2 Fixtures

NOTE: You may need to add the ORM Fixtures library to this project:

```
composer req orm-fixtures --dev
```

Use `make:fixtures ProductFixtures` to create a fixtures class, and write fixtures to enter the following initial data:

```
$p1 = new Product();
$p1->setDescription('bag of nails');
```

```

$p1->setPrice(5.00);
$p1->setCategory('hardware');
$manager->persist($p1);

$p2 = new Product();
$p2->setDescription('sledge hammer');
$p2->setPrice(10.00);
$p2->setCategory('tools');
$manager->persist($p2);

$p3 = new Product();
$p3->setDescription('small bag of washers');
$p3->setPrice(3.00);
$p3->setCategory('hardware');
$manager->persist($p3);

```

Now migrate your updated Entity structure to the database and load those fixtures. Figure 13.1 shows the list of products you should visiting the /product route.

<b>Id</b>	<b>Description</b>	<b>Price</b>	<b>Category</b>	<b>actions</b>
4	bag of nails	5	hardware	<a href="#">show</a> <a href="#">edit</a>
5	sledge hammer	10	tools	<a href="#">show</a> <a href="#">edit</a>
6	small bag of washers	3	hardware	<a href="#">show</a> <a href="#">edit</a>

Figure 13.1: Animated hamburger links for narrow browser window.

### 13.3 Add new route and controller method for category search

Add a new method to the `ProductController` that has the URL route pattern `/product/category/{category}`. We'll name this method `categorySearch(...)` and it will allow us to refine the list of products to only those with the given `Category` string:

```
/**
 * @Route("/category/{category}", name="product_search", methods={"GET"})
 */
public function search($category): Response
{
    $productRepository = $this->getDoctrine()->getRepository('App:Product');
    $products = $productRepository->findByCategory($category);

    $template = 'product/index.html.twig';
    $args = [
        'products' => $products,
        'category' => $category
    ];

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

First, we are getting a string from the URL that follows `/product/category/`. All routes defined in the CRUD generated `ProductController` are prefixed with `/product`, due to the annotation comment that is declared **before** the class declaration:

```
/**
 * @Route("/product")
 */
class ProductController extends AbstractController
{
    ... controller methods here ...
}
```

Whatever appears **after** `/product/category/` in the URL will be put into variabled `$category` by the Symfony routing system, because of the Route annotation comment:

```
/**
 * @Route("/category/{category}", name="product_search", methods={"GET"})
 */
```

We get a reference to an object that is an instance of the `ProductRepository` from this line:

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

We could get an array `products` of **all** `Product` objects from the database by writing:

```
$products = $productRepository->findByCategory($category);
```

But Doctrine repository classes also give us free **helper** methods, that provide `findBy` and `findOneBy` methods for the properties of an Entity class. Since Entity class `Product` has a property `name`, then we get for free the Doctrine query method `findByName(...)` to which we can pass a value of `name` to search for. So we can get the array of `Product` objects whose `name` property matches the parameter `category` as follows:

```
$products = $productRepository->findByCategory($category);
```

Finally, we'll pass both the `$products` array, and the text string `$category` as variables to the `index` list Products Twig template:

```
$template = 'product/index.html.twig';
$args = [
    'products' => $products,
    'category' => $category
];

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

## 13.4 Aside: How to the free ‘helper’ Doctrine methods work?

PHP offers a runtime code reflection (or interpreter pre-processing if you prefer), that can intercept calls to non-existent methods of a class. We use the special **magic** PHP method `__call(...)` which expects 2 parameters, one for the non-existent method name, and one as an array of argument values passed to the non-existent method:

```
public function __call($methodName, $arguments)
{
    ... do something with $methodName and $arguments
}
```

Here is a simple class (put it in `/src/Util/ExampleRepository.php` if you want to try this) that demonstrates how Doctrine uses ‘`__call`’ to identify which Entity property we are trying to query by:

```
<?php
namespace App\Util;

/*
 * class to demonstrate how __call can be used by Doctrine repositories ...
 */
class ExampleRepository
{
    public function findAll()
    {
        return 'you called method findAll()';
    }

    public function __call($methodName, $arguments)
    {
        $html = '';
        $argsString = implode(', ', $arguments) . "\n";

        $html .= "you called method $methodName\n";
        $html .= "with arguments: $argsString\n";

        $result = $this->startsWithFindBy($methodName);
        if($result){
            $html .= "since the method called started with 'findBy'"
            . "\n it looks like you were searching by property '$result'\n";
        }

        return $html;
    }

    private function startsWithFindBy($name)
    {
        $needle = 'findBy';
        $pos = strpos($name, $needle);

        // since 0 would evaluate to FALSE, must use !== not simply !=
        if (($pos !== false) && ($pos == 0)){
            return substr($name, strlen($needle)); // text AFTER findBy
        }

        return false;
    }
}
```

```
    }
}
```

You could add a new method to the `DefaultController` class to see this in action as follows:

```
/**
 * @Route("/call", name="call")
 */
public function call()
{
    // illustrate how __call works
    $exampleRepository = new ExampleRepository();

    $html = "<pre>";
    $html .= "----- calling findAll() -----\\n";
    $html .= $exampleRepository->findAll();

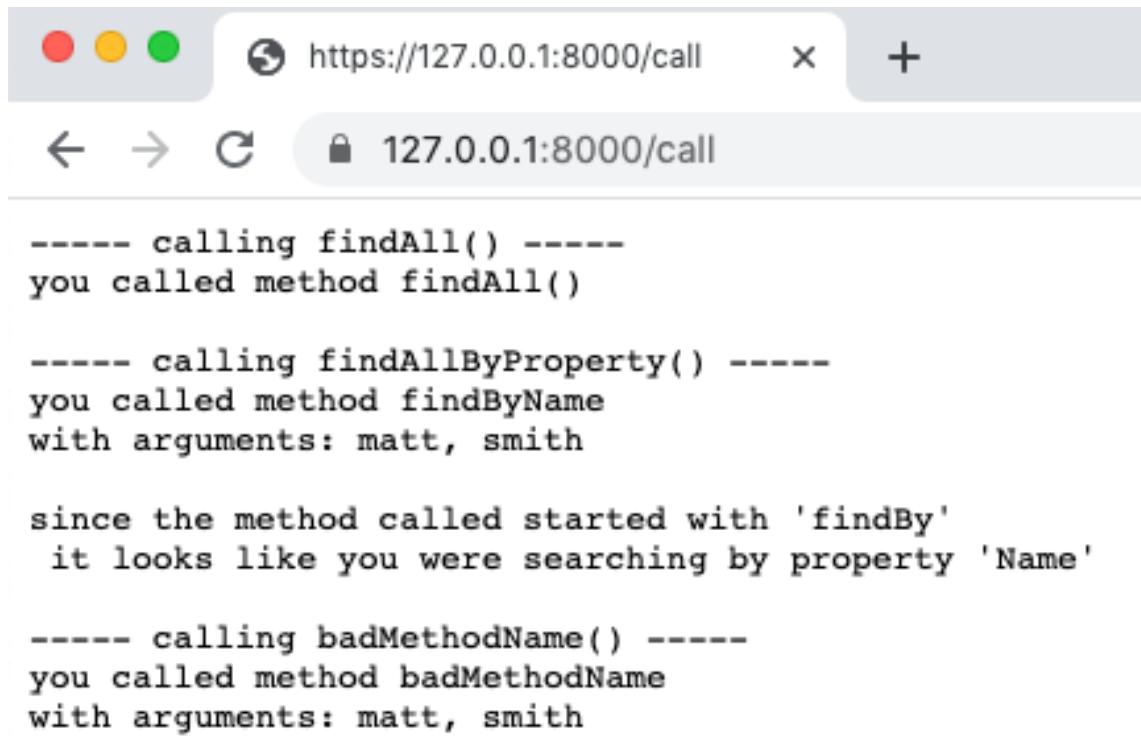
    $html .= "\\n\\n----- calling findAllByProperty() -----\\n";
    $html .= $exampleRepository->findByName('matt', 'smith');

    $html .= "\\n----- calling badMethodName() -----\\n";
    $html .= $exampleRepository->badMethodName('matt', 'smith');

    return new Response($html);
}
```

See Figure 13.2 shows the `ExampleRepository` output you should visiting the `/call` route. We can see that:

- a call to `findAll()` works fine, since that is a defined public method of the class
- a call to `findByName(...)` would work fine, since we can use `__call(...)` to identify that this was a call to a helper `findBy<property>(...)` method
  - and we could add logic to check that this is a property of the Entity class and build an appropriate query from the arguments
- a call to `badMethodName(...)` is caught by `__call(...)`, but fails our test for starting with `findBy`, and so we can ignore it
  - or log error or throw Exception or whatever our program spec says to do in these cases...



The screenshot shows a browser window with the address bar displaying `https://127.0.0.1:8000/call`. The page content is a series of log messages from a Symfony controller action:

```
----- calling findAll() -----
you called method findAll()

----- calling findAllByProperty() -----
you called method findByName
with arguments: matt, smith

since the method called started with 'findBy'
it looks like you were searching by property 'Name'

----- calling badMethodName() -----
you called method badMethodName
with arguments: matt, smith
```

Figure 13.2: Output from our ExampleRepository `__call` demo.

### 13.5 Testing our search by category

If we now visit `/products/category/tools` we should see a list of only those Products with category = tools. See Figure 13.3 for a screenshot of this.

Likewise, for `/products/category/hardware` - see Figure 13.4.

If we try to search with a value that does not appear as the `category` String property for any Products, no products will be listed. See Figure 13.5.

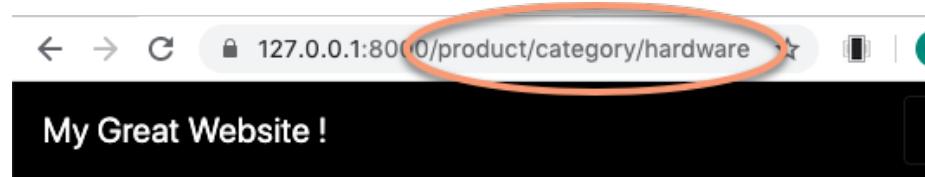


## Product index

<b>Id</b>	<b>Description</b>	<b>Price</b>	<b>Category</b>	<b>actions</b>
5	sledge hammer	10	tools	<a href="#">show</a> <a href="#">edit</a>

[Create new](#)

Figure 13.3: Only tools Products.



## Product index

<b>Id</b>	<b>Description</b>	<b>Price</b>	<b>Category</b>	<b>actions</b>
4	bag of nails	5	hardware	<a href="#">show</a> <a href="#">edit</a>
6	small bag of washers	3	hardware	<a href="#">show</a> <a href="#">edit</a>

[Create new](#)

Figure 13.4: Only hardware Products.



## Product index

Id	Description	Price	Category	actions
no records found				

[Create new](#)

Figure 13.5: Only abc Products (i.e none!).



# 14

## Custom database queries

### 14.1 Search Form for exact property value (project query02)

Searching by having to type values in the URL isn't ideal. So let's add an HTML form in the list of projects page, allowing users to enter the category that way. Figure 14.1 illustrates what we are going to create.

Category

hardware  Submit

hardware

tools

hardware

→ C 127.0.0.1:8000/product/category/hardware

## Product index

<b>Id</b>	<b>Description</b>	<b>Price</b>	<b>Category</b>
			<input type="text"/> Submit
4	bag of nails	5	hardware
6	small bag of washers	3	hardware

Figure 14.1: Form to search for category.

## 14.2 The form in Twig templat

Let's write the HTML code for the submission form for the Products list Twig template in `/templates/product/index.html.twig`.

At present we have a table `<thead>` with a row of column headers, and then a loop for each Product:

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

{% block title %}Product index{% endblock %}

{% block body %}
    <h1>Product index</h1>

    <table class="table">
        <thead>
            <tr>
                <th>Id</th>
                <th>Description</th>
                <th>Price</th>
                <th>Category</th>
                <th>actions</th>
            </tr>
        </thead>
        <tbody>

        <<< TABLE ROW WITH FORM TO GO HERE >>>

        {% for product in products %}
            <tr>
                <td>{{ product.id }}</td>
```

We need to add a new table row between the table headers and the loop of Products:

```
<tr>
    <th></th>
    <th></th>
    <th></th>
    <th>
        <form action="{{ url('search_category') }}" method="post">
            <input name="category">
            <input type="submit">
        </form>
```

```
</th>
<th></th>
</tr>

{% for product in products %}
<tr>
    ... as before
```

The row has empty cells, except for the 4th cell (the Category column), where we create a simple form. The form has:

- a method of `post`
- an action of `url('search_category')`
  - we'll have to create this new route in the `ProductController` to process submission of this form
- a text box named `category`
  - since this text box will appear in the Category column, we don't need to give a text prompt
  - the HTML default `<input>` type is `text`, so we don't need to specify this either
- a Submit button

### 14.3 Controller method to process form submission

Here is the new method in `ProductController` to process submission of this form - implementing the route `search_category`:

```
/**
 * @Route("/searchCategory", name="search_category", methods={"POST"})
 */
public function searchCategory(Request $request): Response
{
    $category = $request->get('category');

    if(empty($category)){
        return $this->redirectToRoute('product_index');
    }

    return $this->redirectToRoute('product_search', ['category' => $category]);
}
```

The annotation comments specify the URL route `/searchCategory`, the internal route name `search_category`, and that we expect the request to be submitted using the `POST` method:

```
/**
 * @Route("/searchCategory", name="search_category", methods={"POST"})
 */
```

We need to extract the `category` variable submitted in the HTTP Request, so we need access to the Symfony Request object. The simnplest way to get a reference to this object is via the Symfony **param converter**, by adding `(Request $request)` as a method parameter. This means we now have Request object variable `$request` available to use in our method:

```
public function searchCategory(Request $request): Response
```

We can retrieve a value from the submitted POST variables int the request using the `get->` method naming the variable `category`. NOTE: In this instance `get` is a **getter** (accessor method) - not to be confused with the HTTP GET Request method...

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

Finally, we can do some logic based on the value of form submitted variable `$category`. If this variable is an emptuy string, let's just redirect Symfonhy to run the method to list all products, route `product_index`:

```
if(empty($category)){
    return $this->redirectToRoute('product_index');
}
```

If `$category` was **not** empty then we can redirect to our category search route, passing the value to this route:

```
return $this->redirectToRoute('product_search', ['category' => $category]);
```

## 14.4 Getting rid of the URL search route

If we no longer wanted the URL search route, we could replace the final statement in our `searchCategory(...)` method to the following (and remove method `search(...)` altogether):

```
/**
 * @Route("/searchCategory", name="search_category", methods={"POST"})
 */
public function searchCategory(Request $request): Response
{
    $category = $request->request->get('category');

    if(empty($category)){
```

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

// if get here, not empty - so use value to search...
$productRepository = $this->getDoctrine()->getRepository('App:Product');
$products = $productRepository->findByCategory($category);

$template = 'product/index.html.twig';
$args = [
    'products' => $products,
    'category' => $category
];

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

# **Part V**

## **Symfony code generation**



# 15

## CRUD controller and templates generation

### 15.1 Symfony's CRUD generator (project crud-01)

After a delay (and a contribution from me about the sequence of methods in generated controllers - new before show), Symfony 4 now has a powerful CRUD generator. Given just an Entity class, the maker-bunder can now generate for you:

- controller class, for CRUD routes (list / show / new / edit / delete)
- Form class for the entity
- templates for: list / show / new / edit / delete

### 15.2 What you need to add to your project

The CRUD maker code needs you to have added 3 other libraries:

- security-csrf
- form
- validator

So first, require these into your Symfony project, and then require in `make` if you haven't already done so:

```
$ composer req security-csrf form validator
$ composer req make
```

### 15.3 Generating new Entity class Category

Generate a new Entity class with a single field `name` - use `make:entity Category`, and add field `name` with defaults of string, length, and not null.

You should now have a basic entity:

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

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $name;

    ... getters and setters
}
```

Migrate this to your database with `doctrine:migrations:diff/migrate`.

### 15.4 Generating CRUD for a new Entity class

Let's create a new Entity class

You can now create Symfony CRUD for a given entity as follows (in this example the `Category` entity is used):

```
$ php bin/console make:crud Category
```

```
created: src/Controller/CategoryController.php
created: src/Form/CategoryType.php
```

```
created: templates/category/_delete_form.html.twig
created: templates/category/_form.html.twig
created: templates/category/edit.html.twig
created: templates/category/index.html.twig
created: templates/category/new.html.twig
created: templates/category/show.html.twig
```

Success!

Next: Check your new CRUD by going to /category/

With the single command above Symfony will generate a CRUD controller (`CategoryController`) and also create a directory containing Twig templates (`/templates/category/index.html.twig` etc.). The list of new files is:

`/src/Controller/CategoryController.php`

`/src/Form/CategoryType.php`

```
/templates/category/_form.html.twig
/templates/category/_delete_form.html.twig
/templates/category/edit.html.twig
/templates/category/index.html.twig
/templates/category/new.html.twig
/templates/category/show.html.twig
```

The list of new (annotation-defined routes):

```
/category --> CategoryController->indexAction()
/category/new --> CategoryController->newAction()
/category/show/{id} --> CategoryController->showAction(Category $category)
/category/delete/{id} --> CategoryController->deleteAction(Category $category)
```

## 15.5 The generated routes

Let's see the new routes generated magically for us:

Name	Method	Scheme	Host	Path
category_index	ANY	ANY	ANY	/category/
category_new	GET POST	ANY	ANY	/category/new
category_show	GET	ANY	ANY	/category/{id}

category_edit	GET POST	ANY	ANY	/category/{id}/edit
category_delete	DELETE	ANY	ANY	/category/{id}
...				

NOTE: The **sequence** of these routes is important (this was the error I fixed for this project). a GET request with a URL looking like this: /category/1 should be matched to the show action, i.e. /category/{id = 1}. But a URL like this /category/new we want to match with the action. If the show route attempts to match **before** the new route, then /category/new is matched to the show route as /category/{id = new}, which will then throw a 404 error, since new is not a valid id for a database Category object.

Once solution is to ensure the new action method appears **before** the show action method in our controller class. If you don't like this solution (I'm not sure myself), then another solution is to design URL routes that cannot get mixed up like this - but that means adding **verbs** for every route. E.g. our category routes could be defined as follows:

Name	Method	Scheme	Host	Path
category_index	ANY	ANY	ANY	/category/list
category_new	GET POST	ANY	ANY	/category/new
category_show	GET	ANY	ANY	/category/show/{id}
category_edit	GET POST	ANY	ANY	/category/edit/{id}
category_delete	DELETE	ANY	ANY	/category/delete/{id}
...				

So, for each request, the entity **category** is followed by its action verb (**list**, **show**, **new** etc.), and then finally, if required an **{id}** parameter. This approach has the advantage of being simple and unambiguous (the sequence of methods in our controller class no longer matters), But (a) it breaks with common conventions in routes in Symfony projects, and (b) it means the URLs are getting longer, and simple, short URLs are one aim (benefit!) of a well designed web framework.

But for **your** personal projects, choose a route pattern scheme that **you** prefer, so this **verb** approach might be something you are happier with. It would also mean you could use **GET** method for **delete** requests, rather than emulating a **DELETE** HTTP request ...

## 15.6 The generated CRUD controller

A controller class was geneated for Category objects in `/src/Controller/CategoryController.php`. Let's first look at the namespaces and class declaration line:

```
namespace App\Controller;
```

```
use App\Entity\Category;
use App\Form\CategoryType;
use App\Repository\CategoryRepository;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

/**
 * @Route("/category")
 */
class CategoryController extends Controller
```

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

If we look in directory `/templates/category/` we’ll see the following generated main templates:

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

and the 2 partial templates (that are included in other pages):

```
_form.html.twig
_delete_form.html.twig
```

Note that all these generated templates extend Twig class `base.html.twig`.

## 15.7 The generated index (a.k.a. list) controller method

Below we can see the code for `indexAction()` that retrieves and then passes an array of `Category` objects to template ‘`category/index.html.twig`’.

```
/**
 * @Route("/", name="category_index", methods={"GET"})
 */
public function index(CategoryRepository $categoryRepository): Response
{
    return $this->render('category/index.html.twig', [
        'categories' => $categoryRepository->findAll(),
    ]);
}
```

```
]);  
}
```

Note how this uses the ‘magic’ of the Param Converter to get a reference to the `StudentRepository` as a method parameter. This makes it a one-liner to `findAll()` objects in the database and pass them on to the Twig template.

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

```
$argsArray = [  
    'categories' => $categoryRepository->findAll()  
];  
  
$template = 'category/index.html.twig';  
return $this->render($template, $argsArray);
```

Twig template `category/index.html.twig` loops through array `categories`, wrapping HTML table row tags around each entity’s content:

```
{% for category in categories %}  
    <tr>  
        <td>{{ category.id }}</td>  
        <td>{{ category.name }}</td>  
        <td>  
            <a href="{{ path('category_show', {'id': category.id}) }}">show</a>  
            <a href="{{ path('category_edit', {'id': category.id}) }}">edit</a>  
        </td>  
    </tr>  
{% else %}  
    <tr>  
        <td colspan="3">no records found</td>  
    </tr>  
{% endfor %}
```

Let’s create a CSS file for table borders and padding in a new file `/public/css/table.css`:

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

Remember in `/templates/base.html.twig` there is a block for style sheets:

```
<head>  
    <meta charset="UTF-8">  
    <title>{% block title %}Welcome!{% endblock %}</title>
```

```
{% block stylesheets %}{% endblock %}
</head>
```

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

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

Figure 15.1 shows a screenshot of how our list of categories looks, rendered by the `categories/index.html.twig` template.

<b>Id</b>	<b>Name</b>	<b>actions</b>
1	hardware	<a href="#">show</a> <a href="#">edit</a>
2	garden	<a href="#">show</a> <a href="#">edit</a>
3	brick-a-brack	<a href="#">show</a> <a href="#">edit</a>

[Create new](#)

Figure 15.1: List of categories in HTML table.

## 15.8 The generated `new()` method

The method and Twig template for a new Category work just as you might expect. For GET requests (and invalid POST submissions) a form will be displayed. Upon valid POST submission the `$category` object populated with the form data will be persisted to the database, and then the user will be redirected to the `edit` action form for the newly created entity.

```
/**
 * @Route("/new", name="category_new", methods={"GET", "POST"})
 */
public function new(Request $request): Response
{
    $category = new Category();
    $form = $this->createForm(CategoryType::class, $category);
    $form->handleRequest($request);

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

        return $this->redirectToRoute('category_index');
    }

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

Note that it redirects to the edit method (`category_index`) after a successful object creation and saving to the database.

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

## 15.9 The generated `show()` method

Initially, the generated ‘show’ method looks just as we might write ourselves:

```
/**
 * @Route("/{id}", name="category_show", methods={"GET"})
 */
```

```
public function show(Category $category): Response
{
    return $this->render('category/show.html.twig', [
        'category' => $category,
    ]);
}
```

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

This magic is the work of the Symfony ‘Param Converter’. Also, of course, if there is no record found in table `category` that corresponds to the received ‘`id`’, then a 404 not-found-exception will be thrown.

Learn more about the ‘param converter’ at the Symfony documentation pages:

- [https://symfony.com/doc/current/best\\_practices/controllers.html#using-the-paramconverter](https://symfony.com/doc/current/best_practices/controllers.html#using-the-paramconverter)

## 15.10 The generated `edit()` method

The ‘`edit`’ generated method is as you might expect. The `edit` method creates a form, and also include code to process valid submission of the edited entity.

Note that it redirects to itself upon successful save of edits. You could change this to redirect to the `show` route as described above for the new action.

```
/**
 * @Route("/{id}/edit", name="category_edit", methods={"GET", "POST"})
 */
public function edit(Request $request, Category $category): Response
{
    $form = $this->createForm(CategoryType::class, $category);
    $form->handleRequest($request);

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

        return $this->redirectToRoute('category_index', [
            'id' => $category->getId(),
        ]);
    }
}
```

```
    return $this->render('category/edit.html.twig', [
        'category' => $category,
        'form' => $form->createView(),
    ]);
}
```

## 15.11 The generated `delete()` method

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

```
/**
 * @Route("/{id}", name="category_delete", methods={"DELETE"})
 */
public function delete(Request $request, Category $category): Response
{
    if ($this->isCsrfTokenValid('delete', $category->getId(), $request->request->get('_token')))
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->remove($category);
        $entityManager->flush();
}

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

The delete form is reached via a Twig include from the edit template (`templates/category/edit.html.twig`):

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

{% block title %}Edit Category{% endblock %}

{% block body %}
<h1>Edit Category</h1>

{{ include('category/_form.html.twig', {'button_label': 'Update'}) }}

<a href="{{ path('category_index') }}">back to list</a>

{{ include('category/_delete_form.html.twig') }}
{% endblock %}
```

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

```
<form method="post" action="{{ path('category_delete', {'id': category.id}) }}" onsubmit="return co
    <input type="hidden" name="_method" value="DELETE">
    <input type="hidden" name="_token" value="{{ csrf_token('delete' ~ category.id) }}">
    <button class="btn">Delete</button>
</form>
```



# **Part VI**

# **Sessions**



# 16

## Introduction to Symfony sessions

### 16.1 Create a new project from scratch (project sessions01)

Let's start with a brand new project to learn about Symfony sessions:

```
$ composer create-project symfony/skeleton session01
```

Let's add to our project the Twig and annotations packages<sup>1</sup>:

```
$ composer req --dev server make  
$ composer req twig annotations
```

It's also a good idea to also include the **Debug** pacakge:

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

### 16.2 Default controller - hello world

Create a new Default controller that renders a Twig template to say **Hello World** to us.

So the controller should look as this (you can speed things up using **make** and then editing the created file). If editing a generated controlle, don't forget to change the route pattern from `/default` to the website root of \ in the annotation comment :

---

<sup>1</sup>And add the server package to `--dev` if that's how you are testing locally.

```
namespace App\Controller;

use Symfony\Component\Routing\Annotation\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    /**
     * @Route("/", name="default")
     */
    public function indexAction()
    {
        $template = 'default/index.html.twig';
        $args = [];
        return $this->render($template, $args);
    }
}
```

Our home page default template `default/index.html.twig` can be this simple<sup>2</sup>:

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

{% block body %}
<p>
    Hello World
</p>
{% endblock %}
```

### 16.3 Twig foreground/background colours (`sessions02`)

Let's start out Symfony sessions learning with the ability to store (and remember) foreground and background colours<sup>3</sup>.

First, let's just pass in a Twig variable from our controller, so that we can write some Twig to work with these variables. Later we'll not receive this variable from the controller, instead we'll use Twig to search for colors in the `session` and set these variables accordingly. But for now, we'll pass a variable from our controller to Twig:

- `colors`: an array holding 2 colors for foreground (text color) and background color

---

<sup>2</sup>If we have a suitable HTML skeleton base template.

<sup>3</sup>I'm not going to get into a colo(u)rs naming discussion. But you may prefer to just always use US-English spelling (*sans 'u'*) since most computer language functions and variables are spelt the US-English way

```
$colors = [  
    'foreground' => 'white',  
    'background' => 'black'  
];
```

So our controller needs to create this variable and pass it on to Twig:

```
public function index()  
{  
    $colors = [  
        'foreground' => 'white',  
        'background' => 'black'  
    ];  
  
    $template = 'default/index.html.twig';  
    $args = [  
        'colors' => $colors,  
    ];  
    return $this->render($template, $args);  
}
```

Next let's add some HTML in our `default/index.html.twig` page to display the value of our 2 stored values.

```
<ul>  
    {% for property, color in colors %}  
        <li>  
            {{ property }} = {{ color }}  
        </li>  
    {% endfor %}  
</ul>
```

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

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

Figure 17.1 shows a screenshot of our home page listing these Twig variables.

Now, let's add a second controller method, named `pinkblue()` that passes 2 different colours to our Twig template:

```
/**  
 * @Route("/pinkblue", name="pinkblue")  
 */  
public function pinkblue()  
{
```

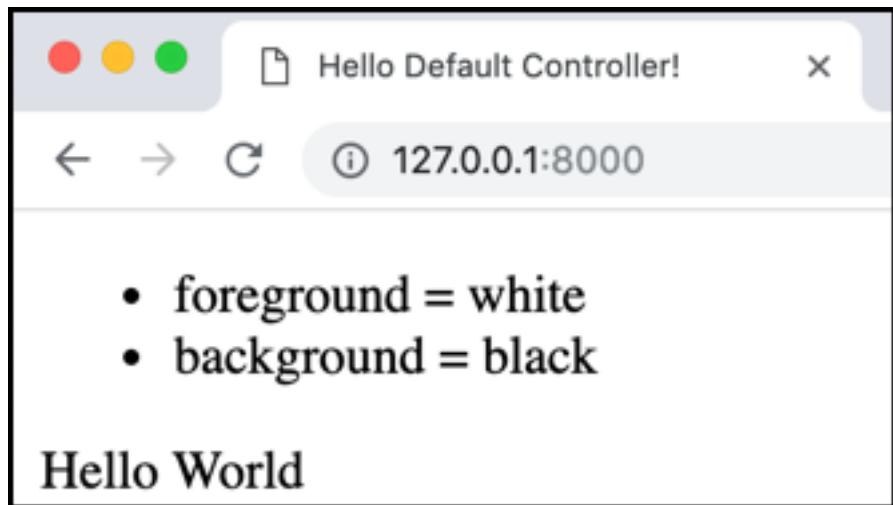


Figure 16.1: Screenshot of home page listing Twig color array variable.

```
$colors = [
    'foreground' => 'blue',
    'background' => 'pink'
];

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

Figure 16.2 shows a screenshot of our second route, passing pink and blue colors to the Twig template.

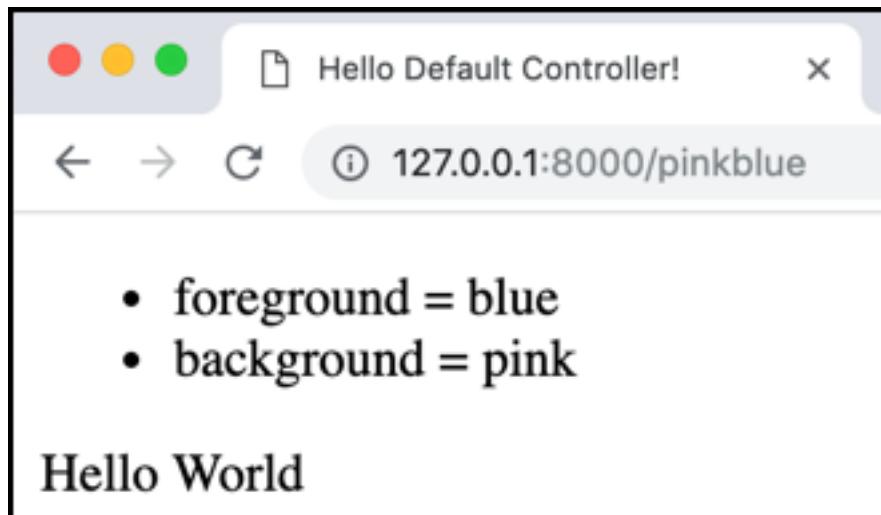


Figure 16.2: Screenshot of /pinkblue route passing different colours to Twig.

## 16.4 Working with sessions in Symfony Controller methods (project session03)

First, since we are going to be using sessions, let's return our default `index()` controller method to pass no arguments to the Twig template. This is because any color variables will be stored in the session and set by other controllers:

```
/**
 * @Route("/", name="default")
 */
public function index()
{
    $template = 'default/index.html.twig';
    $args = [
    ];
    return $this->render($template, $args);
}
```

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

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

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

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

Note - do **not** use any of the standard PHP command for working with sessions. Do all your

Symfony work through the Symfony session API. So, for example, do not use either of these PHP functions:

```
session_start(); // ----- do NOT use this in Symfony -----
session_destroy(); // ----- do NOT use this in Symfony -----
```

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

Note: You may wish to read about **how to start a session in Symfony**<sup>4</sup>.

## 16.5 Symfony's 2 session 'bags'

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

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

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

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

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

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

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

Note - also learn how to 'unset' values when you learn to set them. We can clear everything in a session by writing:

```
$session = new Session();
$session->clear();
```

---

<sup>4</sup>While a session will be started automatically if a session action takes places (if no session was already started), the Symfony documentation recommends your code starts a session if one is required. Here is the code to do so: `$session->start()`, but to be honest it's simpler to rely on Symfony to decide when to start a new session, since sometimes integrating this into your controller logic can be tricky (especially with controller redirects). You'll get errors if you try to start an already started session ...

## 16.6 Storing values in the session in a controller action

Let's refactor DefaultController method pinkBlue() which has route /pinkblue with logic to store colours in the session and then re-direct Symfony to the home page route:

```
/**
 * @Route("/pinkblue", name="pinkblue")
 */
public function pinkBlue()
{
    // create colors array
    $colors = [
        'foreground' => 'blue',
        'background' => 'pink'
    ];

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

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

If you add the Symfony Profiler (composer req --dev profiler) you can view session values in its session tab,, as show in Figure 16.3.

The screenshot shows the Symfony Profiler interface. On the left is a sidebar with icons for Request / Response, Performance, Exception, Logs, and Events. The main area is titled "DefaultController :: index". Below the title is a navigation bar with tabs: Request, Response, Cookies, Session (which is highlighted with a red circle), Flashes, and Server Parameters. Under the "Session" tab, there is a section titled "Session Attributes". A table shows one attribute, "colors", with a value of an array: [foreground => "blue", background => "pink"].

Attribute	Value
colors	[ "foreground" => "blue" "background" => "pink" ]

Figure 16.3: Homepage with session colours applied via CSS.

Learn more at about Symfony sessions at:

- [Symfony and sessions](#)

## 16.7 Twig function to retrieve values from session

Twig offers a function to attempt to retrieve a named value in the session:

```
app.session.get('<attribute_key>')
```

If fact the `app` Twig variable allows us to read lots about the Symfony, including:

- `request (app.request)`
- `user (app.user)`
- `session (app.session)`
- `environment (app.environment)`
- `debug mode (app.debug)`

Read more about Twig `app` in the Symfony documentation pages:

- [https://symfony.com/doc/current/templating/app\\_variable.html](https://symfony.com/doc/current/templating/app_variable.html)

## 16.8 Attempt to read `colors` array property from the session

We can store values in Twig variables using the `set <var> = <expression>` statement. So let's try to read an array of colours from the session named `colors`, and store in a local Twig variable names `colors`:

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

After this statement, `colors` either contains the array retrieved from the session, or it is `null` if no such variable was found in the session.

So we can test for `null`, and if `null` is the value of `colors` then we can set `colors` to our default (black/white) values:

```
{# ----- attempt to read 'colors' from session ----- #}
{% set colors = app.session.get('colors') %}

{# ----- if 'null' then not found in session ----- #}
{# ----- so set to black/white default values ----- #}
{% if colors is null %}
    {# ----- set our default colours array ----- #}
    {% set colors = {
        'foreground': 'black',
        'background': 'white'
    %}
}
```

```
{% endif %}
```

So at this point we know `colors` contains an array, either from the session or our default values (black/white) set in the Twig template.

The full listing for our Twig template `default/index.html.twig` looks as follows: first part logic testing session, second part outputting details about the variables:

```
{# ----- attempt to read 'colors' from session ----- #}
{% set colors = app.session.get('colors') %}

{# ----- if 'null' then no found in session ----- #}
{% if colors is null %}
    {# ----- set our default colours array ----- #}
    {% set colors = {
        'foreground': 'black',
        'background': 'white'
    %}
    %}
{% endif %}

<ul>
    {% for property, color in colors %}
        <li>
            {{ property }} = {{ color }}
        </li>
    {% endfor %}
</ul>

<p>
    Hello World
</p>
```

Finally, we can add another route method in our controller to **clear the session**, i.e. telling our site to reset to the default colors defined in our Twig template:

```
/**
 * @Route("/default", name="default_colors")
 */
public function defaultColors()
{
    // store colours in session 'colours'
    $session = new Session();
    $session->clear();
```

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

```
}
```

## 16.9 Applying colours in HTML head <style> element (project session04)

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

So we remove the Twig code from template `index.html.twig`. So this template just adds our Hello World paragraph to the body block:

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

{% block title %}Hello Default Controller!{% endblock %}

{% block body %}
<p>
    Hello World
</p>
{% endblock %}
```

We'll place our (slightly edited) Twig code into `base.html.twig` as follows. Add the following **before** we start the HTML doctype etc.

```
{# ----- attempt to read 'colors' from session ----- #}
{% set colors = app.session.get('colors') %}

{# ----- if 'null' then no found in session ----- #}
{% if colors is null %}
    {# ----- set our default colours array ----- #}
    {% set colors = {
        'foreground': 'black',
        'background': 'white'
    %}
    %}
{% endif %}
```

So now we know we have our Twig variable `colors` assigned values (either from the session, or from the defaults. Now we can update the `<head>` of our HTML to include a new `body {}` CSS rule,

that pastes in the values of our Twig array `colours['foreground']` and `colours['background']`:

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

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

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

Figure 16.4 shows our text and background colours applied to the CSS of the website homepage.



Figure 16.4: Homepage with session colours applied via CSS.

## 16.10 Testing whether an attribute is present in the current session

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

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

## 16.11 Removing an item from the session attribute bag

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

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

## 16.12 Clearing all items in the session attribute bag

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

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

# 17

## Working with a session ‘basket’ of products

### 17.1 Shopping cart of products (project `session05`)

When you’re leaning sessions, you need to build a ‘shopping cart’! Let’s create CRUD for some Products and then offer a shopping basket.

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

### 17.2 Create a new project with the required packages

Let’s start with a brand new project to work with for shopping baskets in sessions:

```
$ composer create-project symfony/skeleton session05
```

Let’s add to our project the Twig and annotations packages:

```
$ composer req twig annotations
```

Let’s add to our project the server, make and debug developer packages:

```
$ composer req --dev server debug
```

Now let’s add the packages for working with databases and CRUD generation:

```
$ composer req doctrine security-csrf validator form
```

## 17.3 Create a Product entity & generate its CRUD

Make a new Product entity:

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

In in the interactive mode add the following properties:

- `description` (defaults: string/255/not nullable)
- `image` (defaults: string/255/not nullable)
- `price` (float)

You should now have an entity class `src/Entity/Product` with accessor methods and database annotation comments for each property. You should also have a repository class `src/Repository/ProductRepository`.

Configure your `.env` database settings, e.g. to setup for MySQL database `sessions01` have the following:

```
DB_USER=root
DB_PASSWORD=pass
DB_HOST=127.0.0.1
DB_PORT=3306
DB_NAME=sessions01
DATABASE_URL=mysql://${DB_USER}:${DB_PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}
```

Generate the database, and migrations and migrate:

```
$ php bin/console doctrine:database:create
$ php bin/console doctrine:migrations:diff
$ php bin/console doctrine:migrations:migrate
```

Then generate CRUD for this entity (i.e. a `ProductController`, templates in `/templates/product/`, and a form class `src/Form/ProductType.php`):

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

## 17.4 Homepage - link to products home

Create a default controller:

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

Set this controller’s route to the website root `/` (rather than `/default`), and make the Twig template for the default homepage be a link to generated route `product_index`:

```
<p>
Hello World
```

```
</p>
```

```
<a href="{{ url('product_index') }}>list of products</a>
```

Run the server and use your CRUD to add a few products into the database, e.g.:

Id	Description	Image	Price
1	hammer	hammer.png	5.99
2	ladder	ladder.png	19.99
3	bucket of nails	nails.png	0.99

## 17.5 Basket index: list basket contents (project sessions07)

We'll write our code in a new controller class `BasketController.php` in directory `/src/Controller/`.

Generate our new controller:

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

Here is our class (with a couple of changes to routes, and a `use` statement for `Session` class we need to use in a minute):

```
<?php

namespace App\Controller;

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

/**
 * @Route("/basket", name="basket_")
 */
class BasketController extends AbstractController
{
    /**
     * @Route("/", name="index")
     */
    public function index()
    {
        $template = 'basket/index.html.twig';
        $args = [];
        return $this->render($template, $args);
    }
}
```

```
}
```

Note:

- we have added the `@Route` prefix `/basket` to all controller actions in this class by writing a `@Route` annotation comment for the class declaration.
- the basket index controller action is very simple, since all the work extracting values from the session will be done by our Twig template. So our index action simply returns the Twig rendering of template `basket/index.html.twig`

## 17.6 Controller method - `clear()`

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

```
/**  
 * @Route("/clear", name="clear")  
 */  
public function clear()  
{  
    $session = new Session();  
    $session->remove('basket');  
  
    return $this->redirectToRoute('basket_index');  
}
```

Plus, as usualy, we must add a `use` statement to declare the namespace in which the `Session` class we are using belongs to:

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

Let’s see how each route is prefixed with `/basket` and each route name is prefixed with `basket_` by listing routes at the CLI:

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

---

Name	Method	Scheme	Host	Path
basket_index	ANY	ANY	ANY	/basket/
basket_clear	ANY	ANY	ANY	/basket/clear
default	ANY	ANY	ANY	/
product_index	ANY	ANY	ANY	/product/
product_new	GET POST	ANY	ANY	/product/new

---

product_show	GET	ANY	ANY	/product/{id}
product_edit	GET POST	ANY	ANY	/product/{id}/edit
product_delete	DELETE	ANY	ANY	/product/{id}

## 17.7 Debugging sessions in Twig

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

Add the Symfony Twig dumper to your project using Composer:

```
$ composer req --dev var-dumper
```

You can either dump **every** variable that Twig can see, with `dump()`. This will list arguments passed to Twig by the controller, plus the `app` variable, containing session data and other application object properties.

Or you can be more specific, and dump just a particular object or variable. For example we’ll be building an attribute stack session array named `basket`, and the contents of this array can be dumped in Twig with the following statement:

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

You might put this at the bottom of the HTML

element in your `base.html.twig` main template while debugging this shopping basket application:

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

        <hr>
        contents of session 'basket'

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

    </body>
</html>
```

Figure ?? shows an example of what we’d see on the home page from this Twig `dump()` statement if there is one item (Product 1) in the session basket:

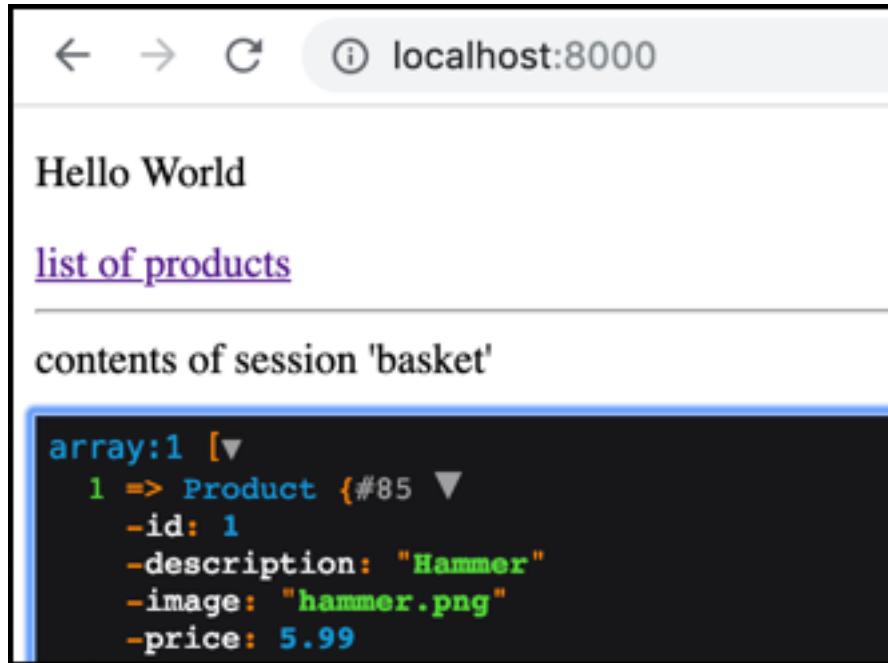


Figure 17.1: Screenshot of home page dumping session basket variable contents.

## 17.8 Adding a object to the basket

The logic to add an object into our session `basket` array requires a little work.

We’ll make things easy for ourselves - using the Symfony Param-Converter. So a product `id` is in the URL `/add/<id>`, but for our method declaration we say we are expecting a reference to a `Product` record `$product`. Symfony will go off and retrieve the row from the database corresponding to the `id`, and return us a reference to a `Product` object containing the properties from the database.

Since we’ll be working with `Product` objects, we need to add a `use` statement at the top of our `BasketController` class:

```
use App\Entity\Product;
```

We need to get a PHP array `$products`, that is either what is currently in the session, or a new empty array if no such array was found in the session:

```
/**
 * @Route("/add/{id}", name="add")
 */
public function addToBasket(Product $product)
```

```
{  
    // default - new empty array  
    $products = [];  
  
    // if 'products' array in session, retrieve and store in $products  
    $session = new Session();  
    if ($session->has('basket')) {  
        $products = $session->get('basket');  
    }  
}
```

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

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

```
// get ID of product  
$id = $product->getId();  
  
// only try to add to array if not already in the array  
if (!array_key_exists($id, $products)) {  
    // append $product to our list  
    $products[$id] = $product;  
  
    // store updated array back into the session  
    $session->set('basket', $products);  
}
```

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

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

## 17.9 The delete action method

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

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

```
/**  
 * @Route("/delete/{id}", name="delete")
```

```
/*
public function deleteAction(int $id)
{
    // default - new empty array
    $products = [];

    // if 'products' array in session, retrieve and store in $products
    $session = new Session();
    if ($session->has('basket')) {
        $products = $session->get('basket');
    }
}
```

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

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

    if (sizeof($products) < 1) {
        return $this->redirectToRoute('basket_clear');
    }

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

Note - if there are no items left in the basket, we redirect to the clear action to remove the basket attribute completely from the session.

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

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

## 17.10 The Twig template for the basket index action

The work extracting the array of products in the basket and displaying them is the task of template `index.html.twig` in `/templates/basket/`.

First, after nice big `<h1>` heading, we attempt to retrieve item `basket` from the session:

```
<h1>Basket contents</h1>
```

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

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

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

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

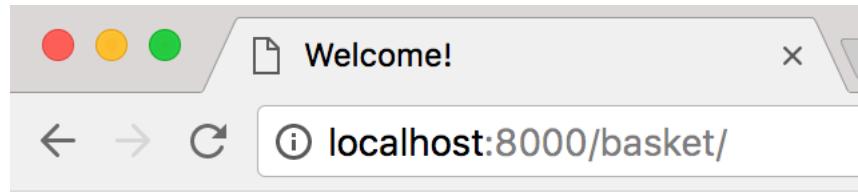
```
{% else %}
<ul>
    {% for product in basket %}
        <li>
            <hr>
            {{ product.id }} :: {{ product.description }}
            <a href="{{ path('basket_delete', { 'id': product.id }) }}>(remove)</a>
        </li>
    {% endfor %}
</ul>
{% endif %}
```

Note that a link to the `delete` action is offered at the end of each list item as text (`remove`).

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

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

Figure 17.2 shows a screenshot of the basket index page - listing the basket contents.



- [home](#)
- [list of products](#)
- [basket](#)

## Basket contents

---

- 3 :: bucket of nails ([remove](#))

[CLEAR all items in basket](#)

Figure 17.2: Shopping basket of elective modules.

## 17.11 Adding useful links to our `base.html.twig` template

Let’s add those useful navigation links to the top of every page. Add the following just before the `body` block is defined in template `base.html.twig`:

```
<nav>
  <ul>
    <li><a href="{{ url('default') }}>home</a></li>
    <li><a href="{{ url('product_index') }}>list of products</a></li>
    <li><a href="{{ url('basket_index') }}>basket</a></li>
  </ul>
</nav>
```

Every page on the website should now show these links.,

## 17.12 Adding the ‘add to basket’ link in the list of products

To link everything together, we can now add a link to ‘add to basket’ in our products CRUD index template. So when we see a list of products we can add one to the basket, and then be redirected to see the updated basket of products. We see below an extra list item for path `basket_add` in template `index.html.twig` in directory `/templates/product/`.

We add this line:

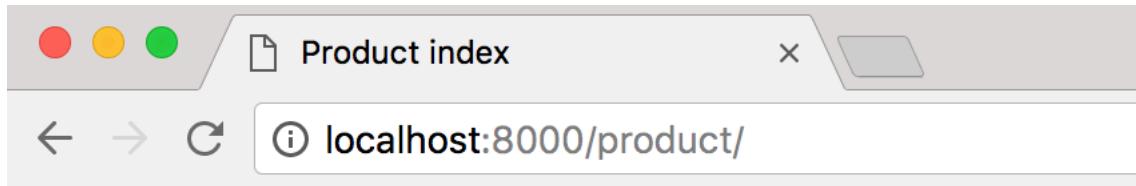
```
<a href="{{ path('basket_add', { 'id': product.id }) }}>add to basket</a>
```

to the end of the table cell displaying each Product

```
{% for product in products %}
  <tr>
    <td>{{ product.id }}</td>
    <td>{{ product.description }}</td>
    <td>{{ product.image }}</td>
    <td>{{ product.price }}</td>
    <td>
      <a href="{{ path('product_show', { 'id': product.id }) }}>show</a>
      <a href="{{ path('product_edit', { 'id': product.id }) }}>edit</a>
      <a href="{{ path('basket_add', { 'id': product.id }) }}>add to basket</a>
    </td>
  </tr>
{% else %}
  <tr>
    <td colspan="5">no records found</td>
```

```
</tr>
{% endfor %}
```

Figure 17.3 shows a screenshot of the list of products page, each with an ‘add to basket’ link.



	<b>Id</b>	<b>Description</b>	<b>Image</b>	<b>Price</b>	<b>actions</b>
1	hammer		hammer.png	5.99	<a href="#">show</a> <a href="#">edit</a> <a href="#">add to basket</a>
2	ladder		ladder.png	19.99	<a href="#">show</a> <a href="#">edit</a> <a href="#">add to basket</a>
3	bucket of nails	nails	nails.png	0.99	<a href="#">show</a> <a href="#">edit</a> <a href="#">add to basket</a>

Figure 17.3: List of Products with ‘add to basket’ link.

## **Part VII**

# **Security and Authentication**



# 18

## Quickstart Symfony security

### 18.1 Learn about Symfony security

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

- [Introduction to security](#)
- [How to build a traditional login form](#)
- [Using CSRF protection](#)

### 18.2 New project with open and secured routes (project `security01`)

We are going to quickly create a 2-page website, with an open home page (url `/`) and a secured admin page (at url `/admin`).

### 18.3 Create new project and add the security bundle library

Create a new project:

```
symfony new --full security01
```

Add the security bundle:

```
composer req symfony/security-bundle
```

Add the fixtures bundle (we'll need this later):

```
composer require orm-fixtures --dev
```

## 18.4 Make a Default controller

Let's make a Default controller `/src/Controller/DefaultController.php`:

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

Edit the route to be `/` and the internal name to be `homepage`:

```
/**
 * @Route("/", name="homepage")
 */
public function indexAction()
{
    $template = 'default/index.html.twig';
    $args = [];
    return $this->render($template, $args);
}
```

Change the template `/templates/default/index.html.twig` to be something like:

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

{% block body %}
    welcome to the home page
{% endblock %}
```

This will be accessible to everyone.

## 18.5 Make a secured Admin controller

Let's make a Admin controller:

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

This will be accessible to only to users logged in with `ROLE_ADMIN` security.

Edit the new `AdminController` in `/src/Controller/AdminController.php`. Add a `use` statement, to let us use the `@IsGranted` annotation:

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

Now we'll restrict access to the index action of our Admin controller using the `@IsGranted` annotation. Symfony security expects logged-in users to have one or more 'roles', these are simple text Strings in the form `ROLE_xxxx`. The default is to have all logged-in users having `ROLE_USER`, and they can have additional roles as well. So let's restrict our admin home page to only logged-in users that have the authentication `ROLE_ADMIN`:

```
/*
 * @Route("/admin", name="admin")
 * @IsGranted("ROLE_ADMIN")
 */
public function index()
{
    $template = 'admin/index.html.twig';
    $args = [];
    return $this->render($template, $args);
}
```

NOTE: We can **make up** whatever roles are appropriate for our application, e.g.:

```
ROLE_ADMIN
ROLE_STUDENT
ROLE_PRESIDENT
ROLE_TECHNICIAN
... etc.
```

Change the template `/templates/admin/index.html.twig` to be something like the following - a secret code we can only see if logged in:

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

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

    here is the secret code to the safe:
    007123
{% endblock %}
```

That's it!

Run the web sever:

- visiting the Default page at `/` is fine, even though we have not logged in ag all
- however, visiting the the `/admin` page should result in an HTTP 401 error (Unauthorized) due to insufficient authentication. See Figure 18.1.

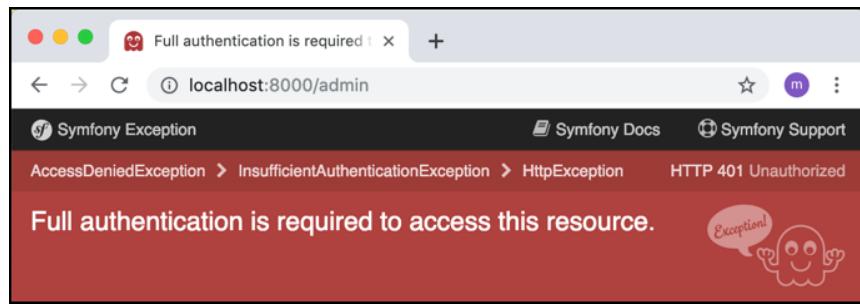


Figure 18.1: Screenshot of error attempting to visit /admin.

Of course, we now need to add a way to login and define different user credentials etc...

## 18.6 Core features about Symfony security

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

- **firewalls**
- **providers** and **encoders**
- **route protection** (we met this with `@IsGranted` controller method annotation comment above...)
- user **roles** (we met this as part of `@IsGranted` above ("ROLE\_ADMIN") ...)

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

```
security:
    providers:
        users_in_memory: { memory: null }

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: lazy
            provider: users_in_memory

    access_control:
```

Symfony considers **every** request to have been authenticated, so if no login action has taken place then the request is considered to have been authenticated to be **anonymous** user `anon`. We can see in this `anon` user in Figure 18.2 this looking at the user information from the Symfony debug bar when visiting the default home page.

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

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

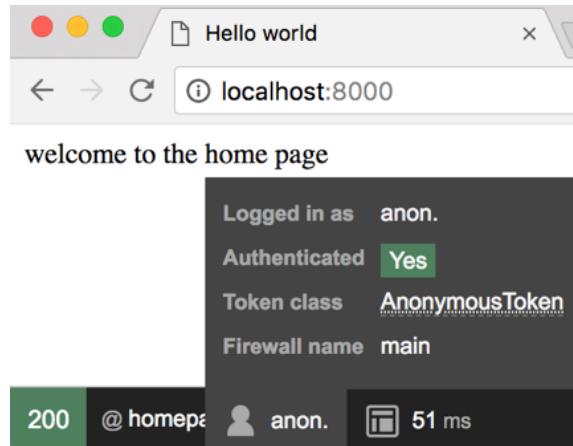


Figure 18.2: Symfony profiler showing anonymous user authentication.

## 18.7 Generating the special User Entity class (project security02)

Let's use the special `make:user` console command to create a `User` entity class that meets the requirements of providing user objects for the Symfony security system.

Enter the following at the command line, then just keep pressing <RETURN> to accept all the defaults:

```
$ php bin/console make:user

The name of the security user class (e.g. User) [User]:
> // press <RETURN> to accept default

Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
> // press <RETURN> to accept default

Enter a property name that will be the unique "display" name for the user (e.g. email, user)
> // press <RETURN> to accept default

Will this app need to hash/check user passwords? Choose No if passwords are not needed or w
Does this app need to hash/check user passwords? (yes/no) [yes]:
> // press <RETURN> to accept default

created: src/Entity/User.php
created: src/Repository/UserRepository.php
updated: src/Entity/User.php
updated: config/packages/security.yaml
```

```
Success!
```

## 18.8 Review the changes to the /config/packages/security.yml file

If we look at `security.yml` it now begins as follows, taking into account our new `User` class:

```
security:
    encoders:
        App\Entity\User:
            algorithm: auto

    # https://symfony.com/doc/current/security.html#where-do-users-come-from-user-providers
    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
```

## 18.9 Migrate new User class to your database

Since we've changed our Entity classes, we should migrate these changes to the database (and, of course, first create your database if you havce not already done so):

```
php bin/console make:migration
php bin/console doctrine:migrations:migrate
```

## 18.10 Make some User fixtures

Let's make some users with the `make:fixture` command:

```
php bin/console make:fixture UserFixtures
```

We'll use the Symfony sample code so that the plain-text passwords can be encoded (hashed) when stored in the database, see:

- <https://symfony.com/doc/current/security.html#c-encoding-passwords>

Edit your class `UserFixtures` to make use of the `PasswordEncoder`:

```
<?php
namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;

use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
use App\Entity\User;

class UserFixtures extends Fixture
{
    private $passwordEncoder;

    public function __construct(UserPasswordEncoderInterface $passwordEncoder)
    {
        $this->passwordEncoder = $passwordEncoder;
    }

    public function load(ObjectManager $manager)
    {
        // (1) create object
        $user = new User();
        $user->setEmail('matt.smith@smith.com');
        $user->setRoles(['ROLE_ADMIN', 'ROLE_TEACHER']);

        $plainPassword = 'smith';
        $encodedPassword = $this->passwordEncoder->encodePassword($user, $plainPassword);

        $user->setPassword($encodedPassword);

        // (2) queue up object to be inserted into DB
        $manager->persist($user);

        // (3) insert objects into database
        $manager->flush();
    }
}
```

From the template class generated for us, the first thing we need to do is add 2 `use` statements, to allow us to make use of the `User` entity class, and the `UserPasswordEncoderInterface` class:

```
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
```

```
use App\Entity\User;
```

Next, to make it easy to encode passwords we'll add a new private instance variable `$passwordEncoder`, and a constructor method to initialise this object:

```
private $passwordEncoder;

public function __construct(UserPasswordEncoderInterface $passwordEncoder)
{
    $this->passwordEncoder = $passwordEncoder;
}
```

Finally, we can write the code to create a new `User` object, set its `email` and `roles` properties, encode a plain text password and set the encoded value to the object. This `$user` object needs to then be added to the queue of objects for the database (`persist(...)`), and then finally inserted into the database (`flush()`):

```
public function load(ObjectManager $manager)
{
    // (1) create object
    $user = new User();
    $user->setEmail('matt.smith@smith.com');
    $user->setRoles(['ROLE_ADMIN', 'ROLE_TEACHER']);

    $plainPassword = 'smith';
    $encodedPassword = $this->passwordEncoder->encodePassword($user, $plainPassword);

    $user->setPassword($encodedPassword);

    // (2) queue up object to be inserted into DB
    $manager->persist($user);

    // (3) insert objects into database
    $manager->flush();
}
```

NOTE: The `roles` property expects to be given an array of String roles, in the form `['ROLE_ADMIN', 'ROLE_SOMETHINGELSE', ...]`. These roles can be whatever we want for user:

```
$user->setRoles(['ROLE_ADMIN', 'ROLE_TEACHER']);
```

## 18.11 Run and check your fixtures

Load the fixtures into the database (with `doctrine:fixtures:load`), and check them with a simple SQL query `select * from user`:

```
php bin/console doctrine:query:sql "select * from user"
Cannot load Xdebug - it was already loaded

/php-symfony-5-book-codes-security-02-user/vendor/doctrine/dbal/lib/Doctrine/DBAL/Tools/Dump
array (size=1)
  0 =>
    array (size=4)
      'id' => string '1' (length=1)
      'email' => string 'matt.smith@smith.com' (length=20)
      'roles' => string '["ROLE_USER", "ROLE_ADMIN"]' (length=27)
      'password' => string '$2y$13$BInaG05FUpAHqcEBtGG05.G.qDbT5SNHoCI1nBHb58FILxJxFUmPu' (1
```

We can see the encoded password and roles `ROLE_USER` and `ROLE_ADMIN`

## 18.12 Creating a Login form

One new addition to the maker tool in Symfony 5 is automatic generation of a login form. Enter the following at the command line:

```
php bin/console make:auth
```

When prompted choose option 1, a Login Form Authenticator:

```
What style of authentication do you want? [Empty authenticator]:
[0] Empty authenticator
[1] Login form authenticator
> 1
```

Next, give the name `LoginFormAuthenticator` for this new authenticator:

```
The class name of the authenticator to create (e.g. AppCustomAuthenticator):
> LoginFormAuthenticator
```

Accept the default (press <RETURN>) for the name of your controller class (`SecurityController`):

```
Choose a name for the controller class (e.g. SecurityController) [SecurityController]:
>
```

Accept the default (press <RETURN>) for creating a `logout` route (`yes`):

```
Do you want to generate a '/logout' URL? (yes/no) [yes]:  
>
```

You should now have a new controller `SecurityController`, a login form `templates/security/login.html.twig`, an authenticator class `LoginFormAuthenticator`, and an updated set of security settings `config/packages/security.yaml`:

```
created: src/Security/LoginFormAuthenticator.php  
updated: config/packages/security.yaml  
created: src/Controller/SecurityController.php  
created: templates/security/login.html.twig
```

Success!

### 18.13 Check the new routes

We can check we have new login/logout routes from with the `debug:router` command:

```
php bin/console debug:router  
Cannot load Xdebug - it was already loaded  
-----  
Name           Method  Scheme  Host   Path  
-----  
_preview_error      ANY     ANY     ANY    /_error/{code}.{_format}  
.... other _profiler debug routes here ...  
admin            ANY     ANY     ANY    /admin  
homepage         ANY     ANY     ANY    /  
app_login         ANY     ANY     ANY    /login  
app_logout        ANY     ANY     ANY    /logout
```

### 18.14 Allow any user to view the login form

Finally, we now have to edit our security firewall to allow **all** users, especially those not yet logged-in!, to access the `/login` route. Add the following line to the end of your `/config/packages/security.yaml` configuration file:

```
- { path: ^/login$, roles: IS_AUTHENTICATED_ANONYMOUSLY }
```

So the full `security.yaml` file should look as follows (with comments removed):

```
security:  
  encoders:  
    App\Entity\User:
```

```
algorithm: auto

providers:
    app_user_provider:
        entity:
            class: App\Entity\User
            property: email

firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        anonymous: lazy
        provider: app_user_provider
        guard:
            authenticators:
                - App\Security\LoginFormAuthenticator
        logout:
            path: app_logout

access_control:
    - { path: ^/login$, roles: IS_AUTHENTICATED_ANONYMOUSLY }
```

## 18.15 Clear cache & visit /admin

Clear the cache (e.g. delete `/var/cache`), and open your browser to `/admin`. Since you are not currently logged-in, you should now be presented with a login form.

After we login with `matt.smith@smith.com` password = `smith`, we should now be able to see in the Symfony Profiler footer that we are logged in, and if we click this profiler footer, and then the **Security** link, we see this user has roles `ROLE_USER` and `ROLE_ADMIN`.

See Figure 18.2 this looking at the user information from the Symfony debug bar when visiting the default home page.

The screenshot shows two stacked web pages. The top page is the 'Admin home' page at `127.0.0.1:8000/admin`. It displays a message: 'here is the secret code to the safe: 007123'. To the right is a sidebar with user information: 'Logged in as matt.smith@smith.com', 'Authenticated Yes', 'Token class PostAuthenticationGuardToken', 'Firewall name main', and 'Actions Logout'. Below this is a footer bar with a link to the profiler (`https://127.0.0.1:8000/_profiler/e82620?panel=security`), the user's email ('matt.smith@smith.com'), and performance metrics ('3 ms'). The bottom page is the 'Symfony Profiler' interface, specifically the 'Security Token' panel. It shows the 'Username' field filled with 'matt.smith@smith.com' and the 'Authenticated' status as 'Yes' (indicated by a green checkmark). A table below lists the 'Roles' property with values: 'ROLE\_USER' and 'ROLE\_ADMIN'.

Figure 18.3: Symfony profiler showing ROLE\_USER and ROLE\_ADMIN authentication.

## 18.16 Using the /logout route

A logout route `/logout` was automatically added when we used the `make:auth` tool. So we can now use this route to logout the current user in several ways:

1. We can enter the route directly in the browser address bar, e.g. via URL:

```
http://localhost:8000/logout
```

2. We can also logout via the Symfony profile toolbar. See Figure 18.4.

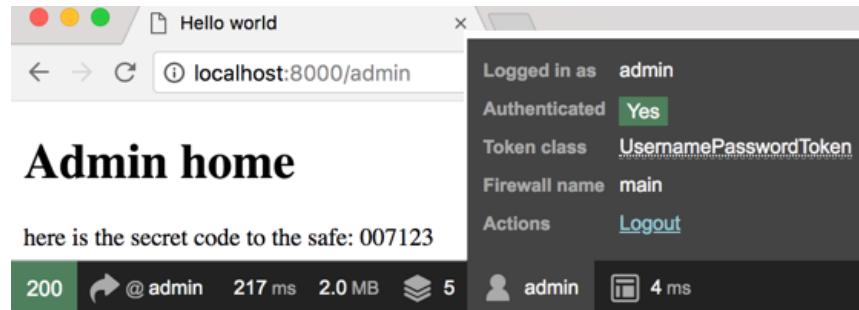


Figure 18.4: Symfony profiler user logout action.

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

## 18.17 Finding and using the internal login/logout route names in SecurityController

Look inside the generated `/src/controller/SecurityController.php` file to see the annotation route comments for our login/lgout routes:

```
...
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    public function login(AuthenticationUtils $authenticationUtils): Response
    {
        ...
    }
}
```

```
/**  
 * @Route("/logout", name="app_logout")  
 */  
public function logout()  
{  
    ...  
}
```

We can add links for the user to login/logout on any page in a Twig template, by using the Twig `url(...)` function and passing it the internal route name for our logout route `app_logout`, e.g.

```
<a href="{{ url('app_logout') }}>  
    logout  
</a>
```



# 19

## Security users from database

### 19.1 Improving UserFixtures with a createUser(...) method (project security03)

Since making users in our `UserFixtures` class is very important, let's add a `helper` method to make it very clear what the properties of each new `User` object will be. See how clear the following is, if we have an extra method `createUser(...)`:

We need a `load(...)` method, that gets invoked when we are loading fixtures from the CLI. This method creates objects for the entities we want in our database, and then saves (persists) them to the database:

```
public function load(ObjectManager $manager)
{
    // create objects
    $userUser = $this->createUser('user@user.com', 'user');
    $userAdmin = $this->createUser('admin@admin.com', 'admin', ['ROLE_ADMIN']);
    $userMatt = $this->createUser('matt.smith@smith.com', 'smith', ['ROLE_ADMIN', 'ROLE_SUPER_ADMIN']);

    // add to DB queue
    $manager->persist($userUser);
    $manager->persist($userAdmin);
    $manager->persist($userMatt);
```

```
// send query to DB
$manager->flush();
}
```

Rather than put all the work in the `load(...)` method, we can create a helper method to create each new object. Method `createUser(...)` creates and returns a reference to a new `User` object given some parameters:

```
private function createUser($username, $plainPassword, $roles = ['ROLE_USER']):User
{
    $user = new User();
    $user->setUsername($username);
    $user->setRoles($roles);

    // password - and encoding
    $encodedPassword = $this->encodePassword($user, $plainPassword);
    $user->setPassword($encodedPassword);

    return $user;
}
```

NOTE: The default role is `ROLE_USER` if none is provided.

## 19.2 Loading the fixtures

Loading fixtures involves deleting all existing database contents and then creating the data from the fixture classes - so you'll get a warning when loading fixtures. At the CLI type:

```
php bin/console doctrine:fixtures:load
```

That's it!

You should now be able to access `/admin` with either the `matt.smith@smith.com/smith` or `admin@admin.com/admin` users. You will get an Access Denied exception if you login with `user@user.com/user`, since that only has `ROLE_USER` privileges, and `ROLE_ADMIN` is required to visit `/admin`.

See Figure 21.1 to see the default Symfony (dev mode) Access Denied exception page.

The next chapter will show you how to deal with (and log) access denied exceptions ...

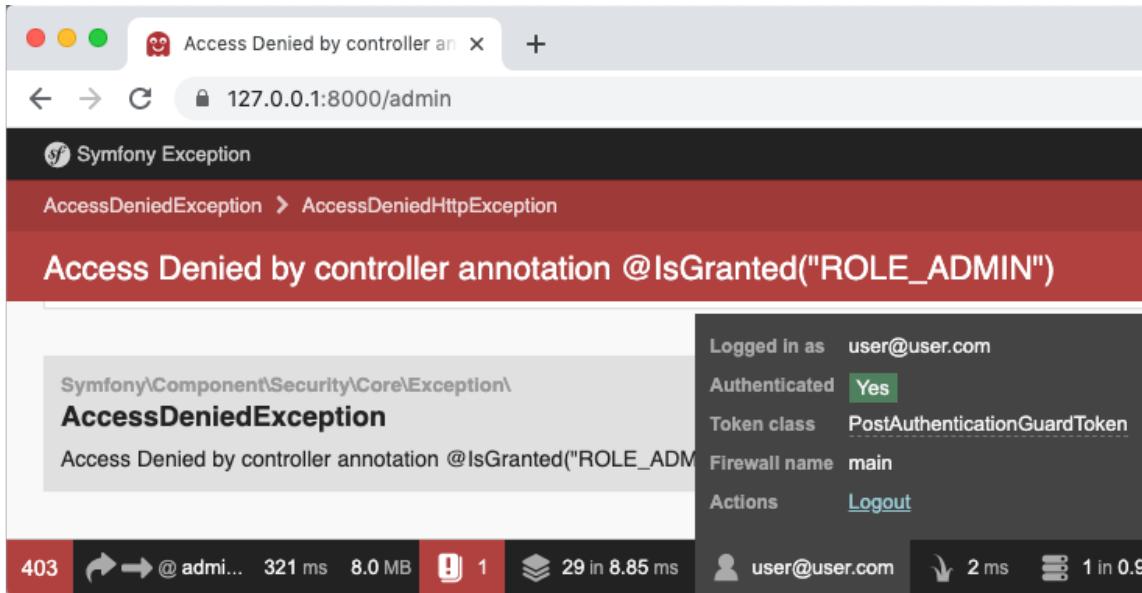


Figure 19.1: Screenshot of Default Symfony access denied page.

### 19.3 Using SQL from CLI to see users in DB

To double check your fixtures have been created correctly in the database, you could run an SQL query from the CLI:

```
$ php bin/console doctrine:query:sql "SELECT * FROM user"
Cannot load Xdebug - it was already loaded

array (size=3)
0 =>
array (size=4)
  'id' => string '2' (length=1)
  'email' => string 'user@user.com' (length=13)
  'roles' => string '['"ROLE_USER"]' (length=13)
  'password' => string '$2y$13$yfMogZlZfDQ3cJeib6Q2k0qXemYBs.4/AnyK/RbAFp69.360N60ai' (length=60)
1 =>
array (size=4)
  'id' => string '3' (length=1)
  'email' => string 'admin@admin.com' (length=15)
  'roles' => string '['"ROLE_ADMIN"]' (length=14)
  'password' => string '$2y$13$9UyVwr0lu0kxLaH57IJM7uPF/NN7iKdBBy.z9im2vx4531elfT80a' (length=60)
2 =>
array (size=4)
```

```
'id' => string '4' (length=1)
'email' => string 'matt.smith@smith.com' (length=14)
'roles' => string '['"ROLE_ADMIN", "ROLE_SUPER_ADMIN"]' (length=34)
'password' => string '$2y$13$4/yo6pKgUgECygZHbawemOSeANK78Cu6bGtKKbSgByFLFxASS1C3u' (1
```

# 20

## Custom login page

### 20.1 A D.I.Y. (customisable) login form (project security04)

When we created the Authenticator it created a login form Twig template for us:

```
$ php bin/console make:auth  
...  
created: src/Controller/SecurityController.php  
created: templates/security/login.html.twig
```

This is just a Twig template, and we should feel free to look inside and edit it ourselves ...

### 20.2 Simplifying the generated login Twig template

The generated Twig login page is fine - but you should become confident in making it your own.

Start by replacing it with this simple, standard HTML login form:

```
<form method="post">  
    <h1>Login</h1>
```

Username:

```
<input value="{{ last_username }}" name="email" id="inputEmail" autofocus>

<p>
    Password:
    <input type="password" name="password" id="inputPassword">

    <input type="submit" value="Login">

</form>
```

The form is shown when the `/login` URL is visited, or Symfony is redirected to internal route `app_login`, with the HTTP GET method. There is not `action` attribute for the `<form>` element, so the form is submitted to the same router, but using the `post` method.

Two name/value form variables are submitted:

- `email` - the email address being used as the unique username
- `password` - the password

### 20.3 CSRF (Cross Site Request Forgery) protection

Although this Twig template will present a login form to the user, it will **not** be accepted by the Symfony security system, due to an exposure to CSRF security vulnerability.

NOTE: For any public **production** site you should always implement CSRF protection. This is implemented using CSRF ‘tokens’ created on the server and exchanged with the web client and form submissions. CSRF tokens help protect web applications against cross-site scripting request forgery attacks and forged login attacks.

Symfony expects forms to submit a special form variable `_csrf_token`. In Symfony this token can be generated using Twig function `csrf_token('authenticate')`. So we need to add this as a hidden form variable for our D.I.Y. form to work:

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

    ... as before
</form>
```

Learn more about CSRF threats and security:

- [Symfony CSRF protection](#)
- [Wikipedia](#)

When using the Symfony generated login form (as we created in this chapter) the CSRF token protection is built-in automatically.

## 20.4 Display any errors

We are only missing one more important set of data from Symfony - any errors to be displayed due to a previous invalid form submission. We should always check for an `error` object, and if present display its `messageData` values as follows (here I've added some CSS to add some padding and a pink background colour):

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

    {% if error %}
        <div style="background-color: pink; padding: 1rem;">
            {{ error.messageKey|trans(error.messageData, 'security') }}
        </div>
    {% endif %}

    <h1>Login</h1>

    Username:
    <input value="{{ last_username }}" name="email" id="inputEmail" autofocus>

    <p>
        Password:
        <input type="password" name="password" id="inputPassword">

        <input type="submit" value="Login">
    </p>
</form>
```

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

- the HTML `<form>` open tag, which we see submits via HTTP `POST` method
  - no action is given, so the form will submit to the same URL as displayed the form (`/login`), but
- add the security CSRF token as a hidden form variable
- display of any Twig `error` variable received

- the `username` label and text input field
  - with 'sticky' form last username value (`last\_username`) if any found in the Twig variable
- the `password` label and password input field
- the submit button named `Login`

## 20.5 Custom login form when attempting to access /admin

See Figure 20.1 to see our custom login form in action.

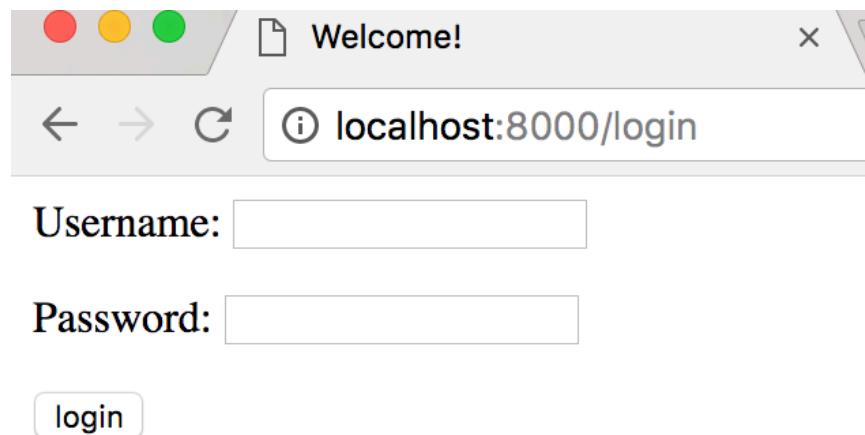


Figure 20.1: Screenshot of custom login form.

## 20.6 Path for successful login

If the user visits the path /login directly in the browser, Symfony needs to know where to direct the user if login is successful. This is defined in method `onAuthenticationSuccess` in class `Security/LoginFormAuthenticator`. If no redirect is defined, then the `TODO` Exception will be thrown:

```
throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
```

Since we have a secure `admin` page, then let's redirect to route `admin`:

```
public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
{
    if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
        return new RedirectResponse($targetPath);
```

```
}

    return new RedirectResponse($this->urlGenerator->generate('admin'));
}
```

If you want to redirect to different pages, depending on the **role** of the newly logged-in user, then do the following:

- get the array of string roles from `$token` with `$token->getRoles()`
- add IF-statement(s) returning a different named route depending on their role, e.g. something like:

```
if(in_array('ROLE_ADMIN', $roles){
    return new RedirectResponse($this->urlGenerator->generate('index_admin'));
}

// else direct to basic staff home page - or whatever ...
return new RedirectResponse($this->urlGenerator->generate('index_staff'));
```



# 21

## Custom AccessDeniedException handler

### 21.1 Symfony documentation for 403 access denied exception

For details about this topic visit the Symfony documentation:

- [https://symfony.com/doc/current/security/access\\_denied\\_handler.html](https://symfony.com/doc/current/security/access_denied_handler.html)

### 21.2 Declaring our handler (project security05)

In `/config/packages/security.yml` we need to declare that the class we'll write below will handle access denied exceptions.

So we add this line to the end of our `main` firewall in `security.yml`:

```
access_denied_handler: App\Security\AccessDeniedHandler
```

So the full listing for our `security.yml` is now:

```
security:  
    encoders:  
        App\Entity\User:  
            algorithm: bcrypt
```

```
providers:
    our_db_provider:
        entity:
            class: App\Entity\User
            property: username

firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        anonymous: true
        provider: our_db_provider
        form_login:
            login_path: login
            check_path: login
        logout:
            path: /logout
            target: /
        access_denied_handler: App\Security\AccessDeniedHandler
```

### 21.3 The exception handler class

Now we need to write our exception handler class in `/src/Security`.

Create new class `AccessDeniedHandler` in file `/src/Security/AccessDeniedHandler.php`:

```
namespace App\Security;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Symfony\Component\Security\Http\Authorization\AccessDeniedHandlerInterface;

class AccessDeniedHandler implements AccessDeniedHandlerInterface
{
    public function handle(Request $request, AccessDeniedException $accessDeniedException)
    {
        return new Response('sorry - you have been denied access', 403);
    }
}
```

That's it!

Now if you try to access `/admin` with `user/user` you'll see the message 'sorry - you have been denied access' on screen. See Figure 21.1.

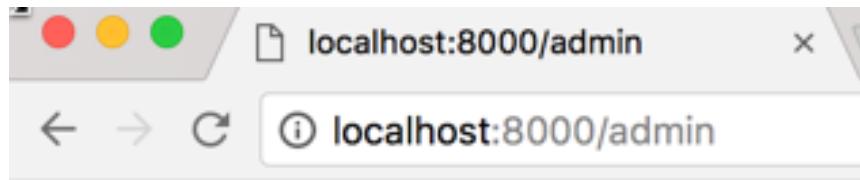


Figure 21.1: Screenshot of Custom Twig access denied page.

Although it won't be generated through the Twig templating system - we'll learn how to do that next ...



# 22

## Twig and logging

### 22.1 Getting reference to Twig and Logger objects

There are many useful service objects available in the Symfony system via the ‘Service Container’. This is a design pattern known as **Dependency Injection**. In Symfony we get access to a service object by **Type Hinting** with the server or interface class name, in the parameter parentheses of the method or constructor of the class.

In this chapter we’ll use this technique to get a reference to the Twig and Logger service objects.

Learn more in the Symfony documentation:

- [https://symfony.com/doc/current/service\\_container.html](https://symfony.com/doc/current/service_container.html)
- [https://symfony.com/doc/current/components/dependency\\_injection.html](https://symfony.com/doc/current/components/dependency_injection.html)

### 22.2 Using Twig for access denied message (project security06)

Let’s improved our Access Denied exception handler in 2 ways:

- display a nice Twig template
- log the exception using the standard Monolog logging system

First add Monolog to our project with Composer:

```
$ composer req logger
```

Now we will refactor class `AccessDeniedHandler` to

```
namespace App\Security;

use Psr\Log\LoggerInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Symfony\Component\Security\Http\Authorization\AccessDeniedHandlerInterface;

class AccessDeniedHandler implements AccessDeniedHandlerInterface
{
    private $twig;
    private $logger;

    public function __construct(ContainerInterface $container, LoggerInterface $logger)
    {
        $this->twig = $container->get('twig');
        $this->logger = $logger;
    }

}
```

Now we can re-write method `handle(...)` to log an error message, and

```
public function handle(Request $request, AccessDeniedException $accessDeniedException)
{
    $this->logger->error('access denied exception');

    $template = 'error/accessDenied.html.twig';
    $args = [];
    $html = $this->twig->render($template, $args);
    return new Response($html);
}
```

## 22.3 The Twig page

Create a new folder `error` in our `/templates` folder, and in that create new Twig template `accessDenied.html.twig` for our nicer looking error page:

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

{% block title %}error{% endblock %}

{% block body %}
    sorry - access is denied for your request
    <p>
        <a href="{{ url('homepage') }}>home</a>
    </p>
{% endblock %}
```

Now, login in as `user@user.com` and try to visit `/admin`. We should get that access denied exception again, since this user does not have the required `ROLE_ADMIN` role privilege. See Figure 22.1 to see the error log register in the Symfony profiler footer, at the bottom of our custom error page.

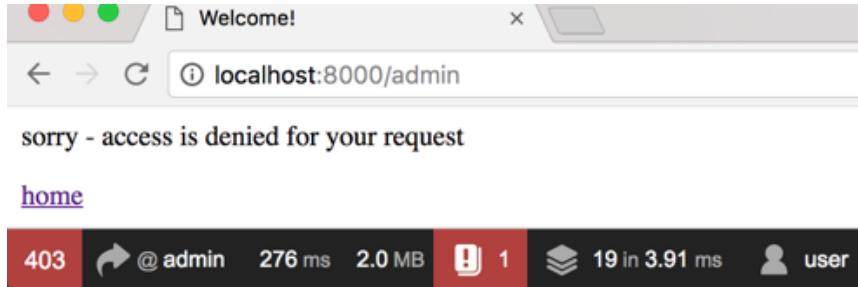


Figure 22.1: Screenshot of Custom Twig access denied page.

If you click on the red error you'll see details of all logged messages during the processing of this request. See Figure 22.2.

Level	Channel	Message
INFO	request	Matched route "admin". 21:18:46 <a href="#">Show context</a>
ERROR	app	access denied exception 21:18:46

Figure 22.2: Screenshot of Profiler log entries.

## 22.4 Terminal log

You'll also see a red highlighted error appear in the terminal window if you are serving this website project with the Symfony web server:

```
[OK] Web server listening on https://127.0.0.1:8000 (PHP FPM 7.3.8)

Mar 10 17:11:55 |WARN| SERVER GET (403) /admin ip="127.0.0.1"
Mar 10 18:11:54 |INFO| REQUEST Matched route "admin". method="GET" request_uri="https://127.0.0.1/admin"
Mar 10 18:11:55 |DEBUG| SECURI Checking for guard authentication credentials. authenticators
    ... a bunch more DEBUG logs ....
Mar 10 18:11:55 |DEBUG| SECURI Access denied, the user is neither anonymous, nor remember-me

Mar 10 18:11:55 |ERROR| APP      access denied exception <<<<< here is our access denied loggg
```

## 22.5 Learn more about logger and exceptions

Learn more about Symfony and the Monolog logger:

- [Logging with Monolog](#)

Learn more about custom exception handlers and error pages:

- [Access Denied Handler](#)
- [Custom Error pages](#)

# 23

## User roles and role hierarchies

### 23.1 Simplifying roles with a hierarchy (project `security07`)

Let's avoid repeating roles in our program logic (e.g. IF `ROLE_USER OR ROLE_ADMIN`) by creating a hierarchy, so we can give `ROLE_ADMIN` all properties of `ROLE_USER` as well. We can easily create a role hierarchy in `/config/packages/security.yml`:

```
security:  
    role_hierarchy:  
        ROLE_ADMIN:      ROLE_USER  
  
        ... rest of 'security.yml' as before ...
```

In fact let's go one further - let's create a 3rd user role (`ROLE_SUPER_ADMIN`) and define that as having all `ROLE_ADMIN` privileges plus the `ROLE_USER` privileges that were inherited by `ROLE_ADMIN`:

```
security:  
    role_hierarchy:  
        ROLE_ADMIN:      ROLE_USER  
        ROLE_SUPER_ADMIN: ROLE_ADMIN  
  
        ... rest of 'security.yml' as before ...
```

Now if we log in as a user with `ROLE_SUPER_ADMIN` we also get `ROLE_ADMIN` and `ROLE_USER` too!

## 23.2 Modify fixtures

Now we can modify our fixtures to make user `matt` have just `ROLE_SUPER_ADMIN` - the other roles should be inherited through the hierarchy:

Change `/src/DataFixtures/UserFixtures.php` as follows:

```
public function load(ObjectManager $manager)
{
    ...

    $userMatt = $this->createUser('matt.smith@smith.com', 'smith', ['ROLE_SUPER_ADMIN'])

    ...
}
```

## 23.3 Removing default adding of `ROLE_USER` if using a hierarchy

If we are using a hierarchy, we don't need always add `ROLE_USER` in code, so we can simplify our getter in our `User` Entity in `/src/Entity/User.php`:

```
```php
public function getRoles()
{
    return $this->roles;
}
```

```

We'll still see `ROLE_USER` for admin and super users, but in the list of **inherited** roles from the hierarchy. This is show in Figure 23.1.

Learn about user role hierarchies at:

- [Symfony hierarchical roles](#)

The screenshot shows the 'Security Token' interface from the Symfony Admin Generator. On the left, a sidebar lists various system components: Exception, Logs, Events, Routing, Cache, Translation, and Security. The 'Security' item is selected. The main panel displays a 'Security Token' for the user 'matt.smith@smith.com'. A green checkmark icon indicates the user is 'Authenticated'. Below this, a table shows the user's properties:

| Property               | Value                                      |
|------------------------|--|
| <b>Roles</b>           | <code>[ "ROLE_SUPER_ADMIN" ]</code>        |
| <b>Inherited Roles</b> | <code>[ "ROLE_ADMIN", "ROLE_USER" ]</code> |

Figure 23.1: Super admin user inheriting ROLE\_USER.

## 23.4 Allowing easy switching of users when debugging

If you wish to speed up testing, you can allow easy switching between users just by adding a but at the end of your request URL, if you add the following to your firewall:

```
switch_user: true
```

Now you can switch users bu adding the following at the end of the URL:

```
?_switch_user=<username>
```

You stop impersonating users by adding `?_switch_user=_exit` to the end of a URL.

For example to visit the home page as user `user` you would write this URL:

```
http://localhost:8000/?_switch_user=user
```

In your Twig you can allow this user to see special content (e.g. a link to exit impersonation) by testing for the special (automatically created role) `ROLE_PREVIOUS_ADMIN`:

```
{% if is_granted('ROLE_PREVIOUS_ADMIN') %}
    <a href="{{ path('admin_index', {'_switch_user': '_exit'}) }">>Exit impersonation & return to a
```

Learn more at:

- [Impersonating users](#)



# 24

## Customising view based on logged-in user

### 24.1 Twig nav links when logged in (project security08)

The [Symfony security docs](#) give us the Twig code for a conditional statement for when the current user has logged in:

```
{% if is_granted('IS_AUTHENTICATED_FULLY') %}  
    <p>Username: {{ app.user.username }}</p>  
{% endif %}
```

We can also test for which **role** a user may have granted when logged-in, e.g.:

```
{% if is_granted('ROLE_ADMIN') %}  
    Welcome to the Admin home page ...  
{% endif %}
```

We can use such conditionals in 2 useful and common ways:

1. Confirm the login username and offer a `logout` link for users who are logged in
2. Have navbar links revealed only for logged-in users (of particular roles)

So let's add such code to our `base.html.twig` master template (in `/templates`).

First, let's add a `<header>` element to either show the username and a logout link, or a link to login if the user is not logged-in yet:

```
<header>
```

```
{% if is_granted('IS_AUTHENTICATED_FULLY') %}  
    Username:  
    <strong>{{ app.user.username }}</strong>  
    <br>  
    <a href="{{ url('app_logout') }}">logout</a>  
{% else %}  
    <a href="{{ url('app_login') }}">login</a>  
{% endif %}  
</header>
```

We can right align it and have a black bottom border with a little style in the `<head>`:

```
<!DOCTYPE html>  
<html>  
    <head>  
        <meta charset="UTF-8">  
        <title>{{ block title }}Welcome!{{ endblock }}</title>  
  
        <style>  
            header {  
                text-align: right;  
                border-bottom: 0.5rem solid black;  
                padding: 1rem;  
            }  
        </style>  
    </head>
```

Next, let's define a `<nav>` element, so that **all** users see a link to the homepage on every page on the website (at least those that extend `base.html.twig`). We will also add a conditional navigation link - to that users logged-in with `ROLE_ADMIN` can also see a link to the admin home page:

```
<nav>  
    <ul>  
        <li>  
            <a href="{{ url('homepage') }}">home</a>  
        </li>  
  
        {% if is_granted('ROLE_ADMIN') %}  
            <li>  
                <a href="{{ url('admin') }}">admin home</a>  
            </li>  
        {% endif %}  
    </ul>  
</nav>
```

So when a user first visits our website homepage, they are not logged-in, so will see a `login` link in the header, and the navigation bar will only show a link to this homepage. See Figure 24.1.

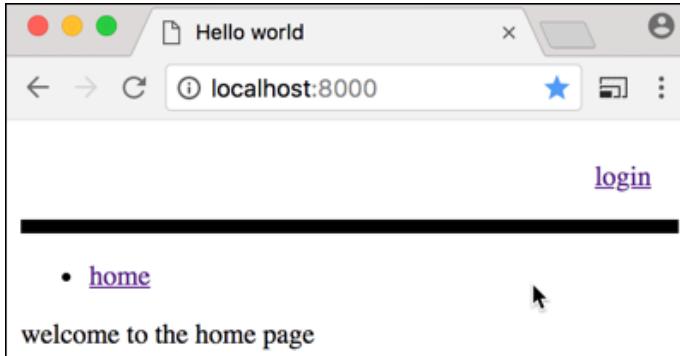


Figure 24.1: Screenshot of homepage before logging-in.

If the user has successfully logged-in with a `ROLE_ADMIN` privilege account, they will now see their username and a `logout` link in the header, and they will also see revealed a link to the admin home page. See Figure 24.2.

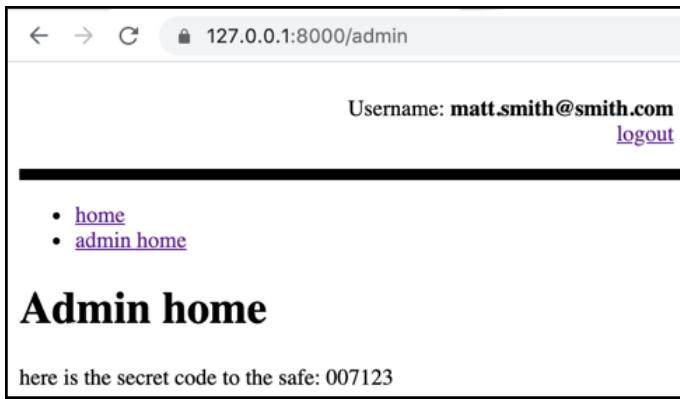


Figure 24.2: Screenshot of homepage after `ROLE_ADMIN` has logged-in.

## 24.2 Getting reference to the current user in a Controller

in PHP (e.g. a controller) you can get the user object as follows:

```
$user = $this->getUser();
```

or you can type-hint in a controller method declaration, and the param converter will provide the `$security` object for your to interrogate:

```
use Symfony\Component\Security\Core\Security;
```

```
public function indexAction(Security $security)
```

```
{  
    $user = $security->getUser();  
}
```

see:

- <https://symfony.com/doc/current/security.html#a-fetching-the-user-object>

## **Part VIII**

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



# 25

## Database relationships (Doctrine associations)

### 25.1 Information about Symfony 4 and databases

Learn about Doctrine relationships and associates at the Symfony documentation pages:

- <https://symfony.com/doc/current/doctrine.html#relationships-and-associations>
- <https://symfony.com/doc/current/doctrine/associations.html>

### 25.2 Create a new project from scratch (`project associations01`)

Create a new project, adding the usual packages for database and CRUD generation:

- server
- make
- twig
- annotations
- doctrine
- form
- validation
- annotations
- security-csrf
- orm-fixtures

## 25.3 Categories for Products

Let's work with a project where we have `Products`, and two categories of Product:

- large items
- small items

So we need to generate a Entity `Category` , with a `name` property:

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

created: src/Entity/Category.php
created: src/Repository/CategoryRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

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

... (hit <RETURN> for defaults and to end generation)
```

Now generate a Product Entity, with properties for `description` (text), `image` (text) and `price` (float):

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

created: src/Entity/Product.php
created: src/Repository/ProductRepository.php

... etc. etc.
```

## 25.4 Defining the many-to-one relationship from Product to Category

We now edit our `Product` entity, declaring a property `category` that has a many-to-one relationship with entity `Category`. I.e., many products relate to one category.

Add the following field and setter in `/src/Entity/Product.php`:

```
class Product
{
```

```

... properties and accessor methods for description / image / price ...

/**
 * @ORM\ManyToOne(targetEntity="App\Entity\Category", inversedBy="products")
 * @ORM\JoinColumn(nullable=true)
 */
private $category;

public function getCategory(): Category
{
    return $this->category;
}

public function setCategory(Category $category)
{
    $this->category = $category;
}

```

## 25.5 How to allow null for a Product's category

We need to allow `null` for a Product's category:

- when it is first created (to generate a form the easy way)
- to allow a category to be removed from a Product

We need to allow our 'getter' to return `null` or a reference to a `Category`, so we change the return type to `?Category`:

```

// allow null - ?Category vs Category
public function getCategory(): ?Category
{
    return $this->category;
}

```

We need set the default value for our 'setter' to `null`:

```

// default Category to null
public function setCategory(Category $category = null)
{
    $this->category = $category;
}

```

This ‘nullable’ parameter/return value is one of the new features from PHP 7.1 onwards:

- [PHP.net guide to migrating to PHP 7.1](#)

## 25.6 Adding the optional one-to-many relationship from Category to Product

Each `Category` relates to many `Products`. Symfony with Doctrine makes it very easy to get an array of `Product` objects for a given `Category` object, without having to write any queries. We just declare a `products` property in the `Category` entity class, and use annotations to declare that it is the reciprocal one-to-many relationship with entity `Product`.

Add the following field and setter in `/src/Entity/Product.php` (don’t forget to add the `use` statement for class `ArrayCollection`):

```
use Doctrine\ORM\Mapping as ORM;
use Doctrine\Common\Collections\ArrayCollection;

/**
 * @ORM\Entity(repositoryClass="App\Repository\CategoryRepository")
 */
class Category
{
    ... properties and accessor methods for name ...

    /**
     * @ORM\OneToMany(targetEntity="App\Entity\Product", mappedBy="category")
     */
    private $products;

    public function __construct()
    {
        $this->products = new ArrayCollection();
    }
}
```

## 25.7 Create and migrate DB schema

Configure your `.env` database settings:

```
DB_USER=root
DB_PASSWORD=pass
```

```
DB_HOST=127.0.0.1
DB_PORT=3306
DB_NAME=web7
DATABASE_URL=mysql://${DB_USER}:${DB_PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}
```

Generate the database, and migrations and migrate:

```
$ php bin/console doctrine:database:create
$ php bin/console doctrine:migrations:diff
$ php bin/console doctrine:migrations:migrate
```

## 25.8 Generate CRUD for Product and Category

Then generate CRUD for this entity (i.e. a `ProductController` and some templates in `/templates/product/`):

```
$ php bin/console make:crud Product
$ php bin/console make:crud Category
```

## 25.9 Add Category selection in Product form

Our generated CRUD for Product creates a Symfony form using method `buildForm(...)` in generated form class `/src/Form/ProductType`:

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('description')
        ->add('image')
        ->add('price')
        ->add('category');
}
```

We need to refine this form builder a declaration that the `Category` to be associated with this Product can be set in the form. So we need to refine the `category` property to our builder as follows:

```
->add('category', EntityType::class,
    // list objects from this class
    'class' => 'App:Category',

    // use the 'Category.name' property as the visible option string
```

```

    'choice_label' => 'name',
]);

```

This references the `EntityType` class, so we need to add a `use` statement for this class:

```

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

```

So the full listing for our updated `ProductType` class is:

```

namespace App\Form;

use App\Entity\Product;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

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

class ProductType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('description')
            ->add('image')
            ->add('price')
            ->add('category', EntityType::class, [
                // list objects from this class
                'class' => 'App:Category',

                // use the 'Category.name' property as the visible option string
                'choice_label' => 'name',
            ]);
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Product::class,
        ]);
    }
}

```

## 25.10 Add small and large item Category

Let's create two categories:

- small items
- large items

You could run the server and manually created these at CRUD page /category/new. Alterntively you could create a `CategoryFixtures` class to automatially add these to the database:

```
namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;
use App\Entity\Category;

class CategoryFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        $cat1 = new Category();
        $cat1->setName('small items');

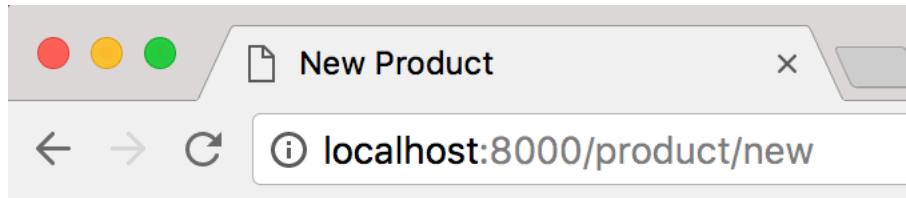
        $cat2 = new Category();
        $cat2->setName('large items');

        $manager->persist($cat1);
        $manager->persist($cat2);

        $manager->flush();
    }
}
```

## 25.11 Drop-down menu of categories when creating/editing Products

Now we are automatically given a drop-down list of `Category` items to choose from when we visit /product/new to create a new `Product` object. See Figure 25.1.



## Create new Product

Description

Image

Price

Category

Figure 25.1: Screenshot of Category dropdown for new Product form.

### 25.12 Adding display of Category to list and show Product

Remember, with the Doctrine ORM (Object-Relational Mapper), if we have a reference to a `Product` object, in PHP we can get its `Category` as follows:

```
$category = $product->getCategory();

if(null != $category)
    // do something with $category
```

In Twig its even simpler, since the dot-syntax finds the public `getter` automatically:

```
Category = {{ product.category }}
```

So we can update the Product list Twig template to show Category as follows (`/templates/product/index.html.twig`)

```
{% for product in products %}
<tr>
    <td>{{ product.id }}</td>
    <td>{{ product.description }}</td>
    <td>{{ product.image }}</td>
    <td>{{ product.price }}</td>
    <td>{{ product.category.name }}</td>
    ...

```

and add a new column header:

```
<tr>
  <th>Id</th>
  <th>Description</th>
  <th>Image</th>
  <th>Price</th>
  <th>Category</th>
  <th>actions</th>
```

And we can update the Product show Twig template to show Category as follows (`/templates/product/show.html.twig`):

```
<tr>
  <th>Id</th>
  <td>{{ product.id }}</td>
</tr>
<tr>
  <th>Description</th>
  <td>{{ product.description }}</td>
</tr>
<tr>
  <th>Image</th>
  <td>{{ product.image }}</td>
</tr>
<tr>
  <th>Price</th>
  <td>{{ product.price }}</td>
</tr>

<tr>
  <th>Category</th>
  <td>{{ product.category.name }}</td>
</tr>
```

See Figure 25.1 to see Category for each Product in the list.

<b>Id</b>	<b>Description</b>	<b>Image</b>	<b>Price</b>	<b>Category</b>	<b>actions</b>
1	hammer	hammer.png	9.99	small items	<a href="#">show</a> <a href="#">edit</a>
3	bag of nails	nails.png	0.99	small items	<a href="#">show</a> <a href="#">edit</a>
4	ladder	ladder.jpg	19.99	large items	<a href="#">show</a> <a href="#">edit</a>

[Create new](#)

Figure 25.2: Screenshot of list of Products with their Category names.

### 25.13 `toString()` method

It is a good idea to have a default `__toString()` method in our `Category` Entity class, so we can write `product.category`. So add the following method to Entity class `src/Category.php`:

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

Now change the `/templates/product/show.html.twig` template to just output `product.category`:

```
<tr>
    <th>Category</th>
    <td>{{ product.category }}</td>
</tr>
```

### 25.14 Setup relationship via `make`

The new improved command line `make` tool can actually do a lot of the above work for us automatically, by defining a `category` property when making Entity class `Product`, with a `Field` type of

relation (rather than string etc.).

```
... (make entity Product)
```

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

```
Field type (enter ? to see all types) [string]:  
> relation
```

```
What class should this entity be related to?:  
> Category
```

```
Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:  
> ManyToOne
```

```
Is the Product.category property allowed to be null (nullable)? (yes/no) [yes]:  
> no
```

```
Do you want to add a new property to Category so that you can access/update  
getProducts()? (yes/no) [yes]:  
> yes
```

```
New field name inside Category [products]:  
> products
```

```
Do you want to automatically delete orphaned App\Entity\Product objects  
(orphanRemoval)? (yes/no) [no]:  
> no
```

```
to stop adding fields):  
>  
(press enter again to finish)
```

Learn more in the Symfony documentation:

- <https://symfony.com/doc/current/doctrine/associations.html>



# 26

Many-to-one (e.g. Products for a single Category)

## 26.1 Basic list products for current Category (project associations02)

First we'll do the minimum to add a list of all the Projects associated with a single Category, then later we'll do it in a nicer way ...

## 26.2 Add `getProducts()` for Entity Category

We need to add a getter for products in `/src/Entity/Category.php` that returns a Doctrine Collection of objects:

```
public function getProducts():Collection
{
    return $this->products;
}
```

We need to add an appropriate `use` statement for

```
use Doctrine\Common\Collections\Collection;
```

### 26.3 Add a `__toString()` for Entity Products

We need to add a ‘magic method’ `__toString()` to Entity Product, since our form builder will need a string for each Product in its list to display:

Add `__toString()` to `/src/Entity/Product.php`. We’ll just list `id` and `description`:

```
public function __toString()
{
    return $this->id . ':' . $this->getDescription();
}
```

### 26.4 Make Category form type add `products` property

Earlier we added the special `products` property to entity Category, which is the ‘many’ link to all the Products for the current Category object. We will now add this property to our Category form class `CategoryType`, so that the form created will display all Products found by automatically following that relationship<sup>1</sup>.

In `/src/Form/CategoryType.php` add `add('products')` to our `buildForm(``)` method:

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('name')
        ->add('products')
    ;
}
```

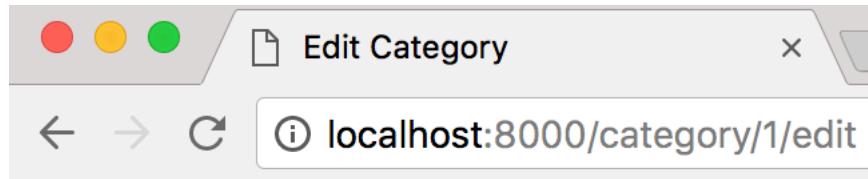
If we visit the ‘edit’ page for a Category now, we can see a read-only multiple value list box displayed for `products`, with all Products for the current Category selected.

While it doesn’t look very nice, our inverse-relationship is all working fine.

See Figure 26.1.

---

<sup>1</sup>I.e. Doctrine will magically run something like ‘SELECT \* FROM product WHERE product.category = category.id’ for the current Category object.



# Edit Category

Name

Products

- 1: hammer
- 3: bag of nails
- 4: ladder

Figure 26.1: Screenshot of products list for Category edit.

## 26.5 Adding a nicer list of Products for Category show page

Let's add a **nice** list of Products for a Category on the Category show page.

In the Twig `/src/templates/show.html.twig` we have a reference to the `category` object. We can get an array of associated Product objects by writing simply `category.products`, so we can loop through this:

```
{% for product in category.products %}  
    {{ product.id }} :: {{ product.description }}  
    <br>  
{% else %}  
    (no products for this category)  
{% endfor %}
```

This will output some HTML like this:

```
1 :: hammer  
<br>  
3 :: bag of nails  
<br>
```

So we can simply add a new HTML table row in our `show.html.twig` template, listing Products as follows:

```
<tr>
    <th>Id</th>
    <td>{{ category.id }}</td>
</tr>
<tr>
    <th>Name</th>
    <td>{{ category.name }}</td>
</tr>

<tr>
    <th>Products for this Category</th>
    <td>
        {% for product in category.products %}
            {{ product.id }} :: {{ product.description }}
            <br>
        {% else %}
            (no products for this category)
        {% endfor %}
    </td>
</tr>
```

See Figure 26.2.

The screenshot shows a web browser window with the following details:

- Header:** The title bar says "Category".
- Address Bar:** The URL is "localhost:8000/category/1".
- Content Area:** The main content is titled "Category". It displays the following information:
  - Id:** 1
  - Name:** small items
  - Products for this Category:** 1 :: hammer  
3 :: bag of nails

Figure 26.2: Screenshot of improved Category show page.

## 26.6 Improving the Edit form (project associations03)

That multi-selection form element was not very nice for our Edit/New forms.

Let's refactor template `/templates/category/_form.html.twig` to display the list of products for a Category in a nicer way. This Twig 'partial' is use both for the **new** Category form and for the **edit** category form.

Our Twig form did contain:

```

{{ form_start(form) }}
{{ form_widget(form) }}

<button>{{ button_label|default('Save') }}</button>
{{ form_end(form) }}

```

Since we want to customise how form elements are displayed, we need to replace `{{ form_widget(form) }}` with our own form elements and HTML.

As explained in an earlier chapter on customising Symfony generated forms, there are 3 parts to a Symfony form output by `{{ form_widget(form) }}`:

```

{{ form_start(form) }}
{{ form_widget(form) }}
{{ form_end(form) }}

```

We don't want the default form elements, then we can display them separately with `form_row`, e.g. we have 2 properties for Entity class `Category`, the name and the collection of related products:

```

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

```

We wish to keep the default rendering of the `name` property, so the start of our customised form will be:

```

{{ form_start(form) }}

{{ form_row(form.name) }}

```

Now we have to decide how to render the `products` array. Let's do something very similar to our show form, and loop through creating list items for each:

```

<div>
    Products for this Category:
    <ul>
        {% for product in form.vars.value.products %}
            <li>
                <a href="{{ url('product_show', {'id':product.id}) }}">

```

```
    {{ product.id }} :: {{ product.description }}
```

```
</a>
```

```
</li>
```

```
{% else %}
```

```
<li>
```

```
    (no products for this category)
```

```
</li>
```

```
{% endfor %}
```

```
</ul>
```

```
</div>
```

As you can see, we can access the array `products` of our Category object with expression:

```
form.vars.value.products
```

So we can write a `for`-loop around this array.

Note - we still need to render the `products` form widget, otherwise Symfony will end the form HTML with all properties not yet rendered. So we can **hide** the default rendering for a selection element by wrapping an HTML comment around the default HTML `select` form element. We also need to display the button `Save` button, since we are rendering the form in pieces:

```
<button class="btn">{{ button_label|default('Save') }}</button>
```

```
<!--
```

```
    {{ form_widget(form.products) }}
```

```
-->
```

We end with `{{ form_end(form) }}`, so the full listing for our new/edit form template `/templates/category/_form.html.twig` is:

```
    {{ form_start(form) }}
```

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

```
<div>
```

```
    Products for this Category:
```

```
<ul>
```

```
    {% for product in form.vars.value.products %}
```

```
        <li>
```

```
            <a href="{{ url('product_show', {'id':product.id}) }}">
```

```
                {{ product.id }} :: {{ product.description }}
```

```
            </a>
```

```
        </li>
```

```
{% else %}
```

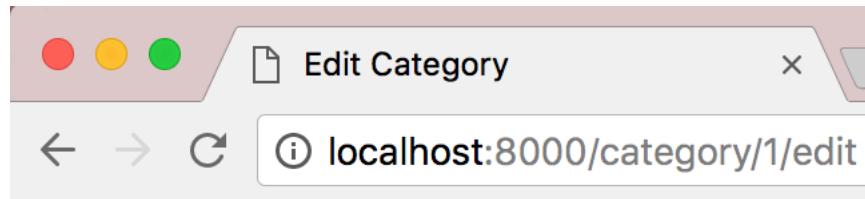
```
<li>
    (no products for this category)
</li>
{% endfor %}
</ul>
</div>

<button class="btn">{{ button_label|default('Save') }}</button>

<!--
{{ form_widget(form.products) }}
-->

{{ form_end(form) }}
```

Figure 26.3 shows a screenshot of our customised Edit form.



# Edit Category

Name

Products for this Category:

- [1 :: hammer](#)
- [3 :: bag of nails](#)

[Edit](#)

Figure 26.3: Screenshot of customised edit form.

## 26.7 Creating related objects as Fixtures (project associations04)

A good way to get a feel for how the Doctrine ORM relates objects, **not** object IDs, is through fixtures. So we can create a `Category` object, and also create a `Product` object, whose `category` property is a reference to the `Category` object. E.g. here are 3 categories and one obnject (hammer) linked to the small items category:

```
namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;
use App\Entity\Category;
use App\Entity\Product;

class CategoryFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        // ----- categories -----
        $catDefault = new Category();
        $catDefault->setName('default');

        $catSmall = new Category();
        $catSmall->setName('small items');

        $catLarge = new Category();
        $catLarge->setName('large items');

        $manager->persist($catDefault);
        $manager->persist($catSmall);
        $manager->persist($catLarge);

        // ----- product -----
        $p1 = new Product();
        $p1->setDescription('hammer');
        $p1->setPrice(9.99);
        $p1->setImage('hammer.png');
        $p1->setCategory($catSmall);

        $manager->persist($p1);
```

```

$manager->flush();
}

```

## 26.8 Using Joins in custom Repository classes

Where Doctrine really shows its worth is when we want to work with tables joined by related properties.

Below a custom Repository method has been created that lists all houses, whose related status object has the title ‘for sale’:

```

class HouseRepository extends ServiceEntityRepository
{
    public function __construct(RegistryInterface $registry)
    {
        parent::__construct($registry, House::class);
    }

    public function findAllForSale()
    {
        return $this->createQueryBuilder('house')
            ->leftJoin('house.status', 'status')

            ->andWhere('status.title = :title')
            ->setParameter('title', 'for sale')
            ->getQuery()
            ->execute();
    }
}

```

By putting complex queries into custom methods in the Repository class, the code in our controllers stays very simple, e.g. below we see an array of all houses ‘for sale’ is being passed to the default home page controller method, to list on the website home page:

```

class DefaultController extends Controller
{
    /**
     * @Route("/", name="home_page")
     */
    public function index(HouseRepository $houseRepository)

```

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

Once again, we see the power of the Symfony paramconvertor, in that to get a reference to a HouseRepository object, we just add a method parameter `HouseRepository $houseRepository`, and as if by magic, we can just start using the repository object!



# 27

Logged-in user stored as item author

## 27.1 Getting User object for currently logged-in user

The Symfony security docs tell us how to get a reference to the currently logged-in user:

```
$user = $this->getUser();
```

or using the hinting and the param-converter:

```
use Symfony\Component\Security\Core\Security;
```

```
...
```

```
public function someMethod(Security $security)
{
    $user = $security->getUser();
}
```

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

Some useful information sources on this topic include:

- [How to Work with Doctrine Relations](#)
- [Forms EntityType Field](#)

## 27.2 Simple example: Users and their county (`associations05`)

First, create, or duplicate a basic user-authenticated secure Symfony website, e.g. project 9 with Twig:

- <https://github.com/dr-matt-smith/php-symfony4-book-codes-security-09-twig-security>

Next, create a `NewsItem` entity, with fields:

- title
- content
- author (which is a relationship associates to the `User` Entity)

Use the `make` interactive CLI tool for this. Create the Entity with text `title` and `content` properties as usual:

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

created: src/Entity/NewsItem.php
created: src/Repository/NewsItemRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):
> title

Field type (enter ? to see all types) [string]:
>

Field length [255]:
>

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/NewsItem.php

Add another property? Enter the property name (or press <return> to stop adding fields):
> content

... etc.
```

---

## CHAPTER 27. LOGGED-IN USER STORED AS ITEM AUTHOR

Then add a property `author` linked as a `relation` Field Type to Entity `User` via a many-to-one relationship:

```
Add another property? Enter the property name (or press <return> to stop adding fields):
```

```
> author
```

```
Field type (enter ? to see all types) [string]:
```

```
> relation
```

```
What class should this entity be related to?:
```

```
> User
```

```
What type of relationship is this?
```

---

Type	Description
ManyToOne	Each NewsItem relates to (has) one User. Each User can relate to (can have) many NewsItem objects
OneToMany	Each NewsItem can relate to (can have) many User objects. Each User relates to (has) one NewsItem
ManyToMany	Each NewsItem can relate to (can have) many User objects. Each User can also relate to (can also have) many NewsItem objects
OneToOne	Each NewsItem relates to (has) exactly one User. Each User also relates to (has) exactly one NewsItem.

---

```
Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
```

```
> ManyToOne
```

```
Is the NewsItem.author property allowed to be null (nullable)? (yes/no) [yes]:
```

```
> yes
```

```
Do you want to add a new property to User so that you can access/update NewsItem objects from it -
```

```
> yes
```

```
A new property will also be added to the User class so that you can access the related NewsItem ob-
```

```
New field name inside User [newsItems]:
```

```
>

updated: src/Entity/NewsItem.php
updated: src/Entity/User.php
```

Now:

- migrate your new Entity to the database
- generate CRUD for Entity NewsItem

### 27.3 Add `toString` method to User

Since our CRUD will wish to list `User` records to associate with a new `NewsItem`, it will expect a `User` object to have a `__toString()` method, for how these users should be shown to the person creating the new `NewsItem`. So add the following `__toString()` method to Entity class `User`:

```
public function __toString()
{
    return (string)$this->username;
}
```

Test the system, run the webserver and visit the `/news/item` pages to test your CRUD.

### 27.4 Use currently logged-in user as author

Let's automatically use the currently logged-in user as the author for a `NewsItem`:

```
$user = $this->getUser();
```

So we need to edit the CRUD code for the `new()` method in `src/Controller/NewsItemController.php` to set the author to the currently logged-in user:

```
public function new(Request $request): Response
{
    $user = $this->getUser();

    $newsItem = new NewsItem();
    $newsItem->setAuthor($user);
```

SO the full listing for method `new()` is now:

```
/**
 * @Route("/new", name="news_item_new", methods={"GET", "POST"})
 */
```

```
public function new(Request $request): Response
{
    $user = $this->getUser();

    $newsItem = new NewsItem();
    $newsItem->setAuthor($user);

    $form = $this->createForm(NewsItemType::class, $newsItem);
    $form->handleRequest($request);

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

        return $this->redirectToRoute('news_item_index');
    }

    return $this->render('news_item/new.html.twig', [
        'news_item' => $newsItem,
        'form' => $form->createView(),
    ]);
}
```

When you visit the CRUD for for a new `NewsItem` you'll see the author is automatically populated with teh currently logged in user.

As an exercise, try to make this a read-only attribute, so that the author cannot be changed to a different user ...

## 27.5 Protect CRUD so must be logged in

We'll get an error at present, if we create a `NewsItem` record when no logged in. So we need to secure the CRUD controller, requiring a user to be logged in to be allowed to create a new item.

By adding the `@IsGranted` annotation before the class declaration, this security requirement is enforced for all routes in this controller:

```
namespace App\Controller;

use App\Entity\NewsItem;
use App\Form\NewsItemType;
```

```
use App\Repository\NewsItemRepository;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;

/**
 * @Route("/news/item")
 * @IsGranted("ROLE_ADMIN")
 */
class NewsItemController extends AbstractController
{
    ...
    ... as before
}
```

# **Part IX**

# **PHPDocumentor (2)**



# 28

## PHPDocumentor

### 28.1 Why document code?

Other people will use your code. And you'll forget things after a few weeks / months / years... So we should create documents describing our code.

There are several good reasons to document your code:

1. It makes you **think** about the code

You might even improve the code having thought about it

2. Writing about code **before** writing the code may lead to better code design
3. It makes you remember **your** code may be used / read by other people
4. It means you don't have to **remember** what things do or why
5. Automated tools can help check for missing documentation
6. It's a handy way to scan your code for TODO comments .

### 28.2 Self-documenting code

Perhaps the simplest way to document code is to insert special comments in the code itself - so the source code files are used to generate the documentation pages about themselves.

‘Docblocks’ are special comments that **precede** the element they describe - they are used by most automated documentation tools.

The PHPDocumentor 2 is a tool to generate HTML documentation pages from these comments

- and if you use an IDE like PHPStorm to speed up writing these special comments...

There are also other PHP code documentation systems out there, including:

- the [Sami](#) Friends of PHP documentation systems
- [Api Gen](#)
- and of course [Doxygen](#)

### 28.3 PHPDocumentor 2

There are several automated tools for supporting PHP code documentation, one of the most popular is PHPDocumentor 2, which is the one described in this chapter.

Learn more about PHPDocumentor 2 at their website:

- [phpdoc.org](#)

### 28.4 Installing PHPDocumentor 2 - the PHAR

The Composer install is a bit big, so it is recommended to just add the PHAR (PHP Archive) either to your project, or globally (somewhere in your system path).

Download the PHAR from their website and either copy into your Symfony project folder, or to some standard folder that is in your CLI execution path.

- [phpdoc.org/phpDocumentor.phar](#)

### 28.5 Installing PHPDocumentor 2 - via Composer

Install via Composer with the following:

```
$ composer req --dev phpdocumentor/phpdocumentor
```

### 28.6 DocBlock comments

The PHPDocumentor is driven by analysing ‘DocBlock’ comments in your code. A DocBlock comment looks as follows:

```
/**  
 * This is a DocBlock.  
 */  
public function indexController()  
{  
}
```

They are multi-line comments that start with a double asterisk `/**`.

## 28.7 Generating the documentation

The PHPDocumentor needs to know at least 2 things:

- where is the PHP source code containing the documentation comments
- where do you want the documentation files to be output

These are specified using the `-d` (PHP source directory), and `-t` (output director) as follows.

So, for example, so analyse **all** files in directory `/src` and output to `/docs` write:

```
$ php phpdoc -d src -t docs
```

To limit the code analysed to just `/src/Controller`, `/src/Util` and `/src/Entity` we would give 3 `-d` arguemnts as follows:

```
php phpdoc.phar -d src/Controller -d src/Entity -d src/Util -t docs
```

## 28.8 Using an XML configuration file `phpdoc.dist.xml`

The simplest way to record your PHPDocumentor configuration options is with an XML file `phpdoc.dist.xml`.

Here is a simple config file:

```
<phpdoc>  
  <parser>  
    <target>./docs</target>  
  </parser>  
  <transformer>  
    <target>./docs</target>  
  </transformer>  
  <files>  
    <directory>./src</directory>  
  </files>
```

```
<transformations>
    <template name="responsive-twig"/>
</transformations>
</phpdoc>
```

This will output the HTML documentation pages in the `responsive-twig` theme, into directory `./docs`, for all PHP classes found in `./src`.

To limit the code analysed to just `/src/Controller`, `/src/Util` and `/src/Entity` we could use the following XML file:

```
<phpdoc>
    <parser>
        <target>./docs</target>
    </parser>
    <transformer>
        <target>./docs</target>
    </transformer>
    <files>
        <directory>./src/Controller</directory>
        <directory>./src/Util</directory>
        <directory>./src/Entity</directory>
    </files>
    <transformations>
        <template name="responsive-twig"/>
    </transformations>
</phpdoc>
```

NOTE: If there is also a file `phpdoc.xml`, any settings in this will override those in `phpdoc.dist.xml`. So, for example, an individual might have some particular settings they prefer defined in their `phpdoc.xml` file, but then could use the team's or organisation's default `phpdoc.dist.xml` for all other settings...

## 28.9 WARNING - PHPStorm default comments

Note that the default comments for a new PHP class provided by PHPStorm will foul-up your documentatin generation:

```
<?php
/**
 * Created by PhpStorm.
 * User: matt
 * Date: 20/03/2018
```

```
* Time: 07:42
*/
```

So delete these default file header comments if you are using the PHPDocumentor.

## 28.10 TODO - special treatment

PHPDocumentor can hoover up special markers, such as TODO, and report them for you. In fact TODO are so handy they get special treatment.

You can mark todo's with:

```
* TODO: fix that bug for stack overflow
```

or with the @todo annotation:

```
* @todo fix that bug for stack overflow
```



## **Part X**

# **Symfony Testing with Codeception**



# 29

## Unit testing in Symfony with Codeception

### 29.1 Codeception Open Source BDD project

Codeception is a BDD (Behaviour Driver Design) open source project, to support acceptance (end-to-end, and unit testing for PHP project. The community supports its close integration with Symfony:

- Github
  - <https://github.com/codeception/codeception>
- Project home page
  - <https://codeception.com/>
- Codeception Symfomy docs
  - <https://codeception.com/for/symfony>

### 29.2 Adding Codeception to an existing project (project `codeception01`)

For these examples we'll start with an existing project;

- security09

- <https://github.com/dr-matt-smith/php-symfony4-book-codes-security-09-twig-security>
- this project has a public home page, student (ROLE\_USER) page, admin (ROLE\_ADMIN) pages, user roles, login pages, CRUD pages

So do the following:

1. clone this existing Symfony project
  - <https://github.com/dr-matt-smith/php-symfony4-book-codes-security-09-twig-security>
2. since we are now in a **TESTING** mode, immediately change the .env file adding the suffix **test** to the database name - we don't test on a project's **production** database !

```
DB_USER=root
DB_PASSWORD=passpass
DB_HOST=127.0.0.1
DB_PORT=3306
DB_NAME=security5test
DATABASE_URL=mysql://${DB_USER}:${DB_PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}
```

- then create the new database with doctrine:database:create

3. Add Codeception via composer

```
$ composer req codeception/codeception --dev
```

```
Using version ^2.5 for codeception/codeception
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 118 installs, 0 updates, 0 removals
```

```
- Installing .... packages ...
```

```
Writing lock file
Generating autoload files
Symfony operations: 2 recipes (d7cfe8af216b9f6fdaf106b4fa7854e0)
- Configuring phpunit/phpunit (>=4.7): From github.com/symfony/recipes:master
```

- you'll get a **WARNING** that you are not installing an 'official' Symfony package, but one contributed by the open source community:
  - WARNING codeception/codeception (>=2.3): From github.com/symfony/recipes-contrib: The recipe for this package comes from the "**contrib**" repository, which is open to c

Review the recipe at <https://github.com/symfony/recipes-contrib/tree/master/codeception/configuring-codeception>

```
Do you want to execute this recipe?  
[y] Yes  
[n] No  
[a] Yes for all packages, only for the current installation session  
[p] Yes permanently, never ask again for this project  
(defaults to n):
```

4. answer 'y' (YES!):

```
- Configuring codeception/codeception (>=2.3): From github.com/symfony/recipes-contrib:master  
ocramius/package-versions: Generating version class...  
ocramius/package-versions: ...done generating version class  
Executing script cache:clear [OK]  
Executing script assets:install public [OK]
```

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

Adding `phpunit/phpunit` as a dependency is discouraged in favor of Symfony's [PHPUnit Bridge](#).

- \* Instead:
  1. Remove it now: `composer remove --dev phpunit/phpunit`
  2. Use Symfony's bridge: `composer require --dev phpunit`
- we see the 'bootstrapping' process for Codeception, so you don't need to bootstrap it yourself (which may be required if using Codeception for a non-Symfony project):

```
Bootstrapping Codeception  
File codeception.yaml created      <- global configuration  
tests/unit created                 <- unit tests  
tests/unit.suite.yaml written     <- unit tests suite configuration  
tests/functional created          <- functional tests  
tests/functional.suite.yaml written <- functional tests suite configuration  
tests/acceptance created          <- acceptance tests  
tests/acceptance.suite.yaml written <- acceptance tests suite configuration  
Codeception is installed for acceptance, functional, and unit testing  
Next steps:
  1. Edit tests/acceptance.suite.yaml to set url of your application. Change PhpBrowser to W...
  2. Edit tests/functional.suite.yaml to enable a Doctrine module if needs.
  3. Create your first acceptance tests using vendor/bin/codecept g:cest acceptance First
  4. Write first test in tests/acceptance/FirstCest.php
```

5. Run tests using: `vendor/bin/codecept run`
5. Notice the message from Symfony about `phpunit/phpunit` being **discouraged**, saying we should use the Symfony PHPUnit **Bridge**:

Adding `phpunit/phpunit` as a dependency is discouraged in favor of Symfony's PHPUnit Br

\* Instead:

1. Remove it now: `composer remove --dev phpunit/phpunit`
2. Use Symfony's bridge: `composer require --dev phpunit`

- let's do what is recommended, so first run the `composer remove ...` command:

```
$ composer remove --dev phpunit/phpunit
```

```
phpunit/phpunit is not required in your composer.json and has not been removed
Loading composer repositories with package information
Updating dependencies (including require-dev)
Restricting packages listed in "symfony/symfony" to "4.2.*"
Nothing to install or update
Generating autoload files
ocramius/package-versions: Generating version class...
ocramius/package-versions: ...done generating version class
Executing script cache:clear [OK]
Executing script assets:install public [OK]
```

- then run the `composer require ...` command to add the Symfony bridge:

```
$ composer require --dev phpunit
```

```
Using version ^1.0 for symfony/test-pack
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Restricting packages listed in "symfony/symfony" to "4.2.*"
Package operations: 3 installs, 0 updates, 0 removals
- Installing symfony/phpunit-bridge (v4.2.4): Loading from cache
- Installing symfony/panther (v0.3.0): Loading from cache
- Installing symfony/test-pack (v1.0.5): Loading from cache
...
```

## 29.3 What Codeception has added to our project

You should now have added to your project the following:

- file `codeception.yaml` in main project folder file:

```
namespace: App\Tests
paths:
    tests: tests
    output: tests/_output
    data: tests/_data
    support: tests/_support
    envs: tests/_envs
actor_suffix: Tester
extensions:
    enabled:
        - Codeception\Extension\RunFailed
params:
    - .env
```

- NOTE: the default URL for testing is: `localhost:8000` - you may need to change this if using a web server other than the Symfony one ...
- new folder `/tests` - you'll create your tests in here
- new folder `vendor/bin/` containing executable: `codecept`
  - you run command line Codeception commands with:

```
vendor/bin/codecept <command>
```



# 30

Check Codeception is working

## 30.1 Run Codeception (with no tests!)

Let's check Codeception is working (even though we haven't created any tests yet):

```
$ vendor/bin/codecept run
```

```
Codeception PHP Testing Framework v2.5.4
Powered by PHPUnit 7.5.6 by Sebastian Bergmann and contributors.
Running with seed:
```

```
App\tests.acceptance Tests (0)
-----
```

```
App\tests.functional Tests (0)
-----
```

```
App\tests.unit Tests (0)
-----
```

```
Time: 1.69 seconds, Memory: 38.25 MB
```

```
No tests executed!
```

## 30.2 Test with a simple Unit test

Let's create a simple Unit test. Codeception is built on top of PHPUnit, so for Unit Testing our classes are very similar to those we'd write for a non-Symfony PHP project:

1. Use the `vendor/bin/codecept` executable to generate a skeleton Unit Test for us:

```
$ vendor/bin/codecept g:test unit FirstUnitTest
Test was created in /cept1/tests/unit/FirstUnitTest.php
```

2. You should now have a new class `tests/unit/FirstUnitTest.php`:

```
namespace App\Tests;

class FirstUnitTest extends \Codeception\Test\Unit
{
    /**
     * @var \App\Tests\UnitTester
     */
    protected $tester;

    protected function _before()
    {
    }

    protected function _after()
    {
    }

    // tests
    public function testSomeFeature()
    {

    }
}
```

3. Let's replace this class content to test the simple assertion that  $1 + 1 = 2$ :

```
namespace App\Tests;
```

```
use Codeception\Test\Unit;

class FirstUnitTest extends Unit
{
    public function testOnePlusOneEqualsTwo()
    {
        // Arrange
        $num1 = 1;
        $num2 = 1;
        $expectedResult = 2;

        // Act
        $result = $num1 + $num2;

        // Assert
        $this->assertEquals($expectedResult, $result);
    }
}
```

4. Run Codeception at the command line, and hopefully our test passes:

```
$ vendor/bin/codecept run
Codeception PHP Testing Framework v2.5.4
Powered by PHPUnit 7.5.6 by Sebastian Bergmann and contributors.
Running with seed:
```

```
App\Tests.acceptance Tests (0)
-----
```

```
App\Tests.functional Tests (0)
-----
```

```
App\Tests.unit Tests (1)
-----
```

```
Testing App\Tests.unit
TICK FirstUnitTest: One plus one equals two (0.01s)
-----
```

```
Time: 198 ms, Memory: 18.00 MB
-----
```

```
OK (1 test, 1 assertion)
```

Note the following:

- unit test classes are located in directory `/tests/unit`
- test classes end with the suffix `Test`, e.g. `SimpleTest`
- simple test classes extend the superclass `\Codeception\Test\Unit`
  - if we add a `use` statement `Codeception\Test\Unit` then we can simply extend `Unit`
- simple test classes are in namespace `App\Tests`
  - the names and namespaces of test classes testing a class in `/src` will reflect the namespace of the class being tested
  - i.e. If we write a class to test `/src/Util/Calculator.php` it will be `/tests/Util/CalculatorTest.php`, and it will be in namespace `App\Util\Test`
  - so our testing class architecture directly matches our source code architecture

### 30.3 Fixing error message about missing `ext-mbstring`:

If you get a message about “`ext mbstring`” required - when trying to work in Windows with Simple PHP Unit or make, then you need to enable this extension in your `php.ini` file:

- Just as you did for `pdo_mysql`, remove the semi-colon in front of the statement in the `php.ini` file:
- e.g. change `;extension=mbstring` to: `extension=mbstring`

### 30.4 Testing other classes (project `codeception02`)

#### our testing structure mirrors the code we are testing

Let’s create a very simple class `Calculator.php` in `/src/Util`<sup>1</sup>, and then write a class to test our class. Our simple class will be a very simple calculator:

- method `add(...)` accepts 2 numbers and returns the result of adding them
- method `subtract()` accepts 2 numbers and returns the result of subtracting the second from the first

So our `/src/Util/Calculator.php` class is as follows:

```
namespace App\Util;
```

```
class Calculator
```

---

<sup>1</sup>Short for ‘Utility’ - i.e. useful stuff!

```
{  
    public function add($n1, $n2)  
    {  
        return $n1 + $n2;  
    }  
  
    public function subtract($n1, $n2)  
    {  
        return $n1 - $n2;  
    }  
}
```

## 30.5 The class to test our calculator

We now need to write a test class to test our calculator class. Since our source code class is `/src/Util/Calculator.php` then our testing class will be `/tests/Util/Calculator.php`.

Let's generate a new Unit Test class to test our `Calculator` class:

```
$ vendor/bin/codecept g:test unit CalculatorTest
```

```
Test was created in /cept2/tests/unit/CalculatorTest.php
```

Since the namespace of our source code class was `App\Util` then the namespace of our testing class will be `App\Util\Test`. Let's test making an instance-object of our class `Calculator`, and we will make 2 assertions:

- the reference to the new object is not NULL
- invoking the `add(...)` method with arguments of (1,1) and returns the correct answer (2!)

Here's the listing for our edited class `CalculatorTest`:

```
namespace App\Util\Tests;  
  
use Codeception\Test\Unit;  
use App\Util\Calculator;  
  
class CalculatorTest extends Unit  
{  
    public function testCanCreateObject()  
    {  
        // Arrange  
        $calculator = new Calculator();
```

```
// Act

// Assert
$this->assertNotNull($calculator);
}

public function testAddOneAndOne()
{
    // Arrange
    $calculator = new Calculator();
    $num1 = 1;
    $num2 = 1;
    $expectedResult = 2;

    // Act
    $result = $calculator->add($num1, $num2);

    // Assert
    $this->assertEquals($expectedResult, $result);
}
}
```

Note:

- we had to add `use` statements for the class we are testing (`App\Util\Calculator`) and the PHP Unit TestCase class we are extending (`\Codeception\Test\Unit`)

Run the tests - if all goes well we should see 3 out of 3 tests passing:

```
$ vendor/bin/codecept run
Codeception PHP Testing Framework v2.5.4
Powered by PHPUnit 7.5.6 by Sebastian Bergmann and contributors.
Running with seed:
```

---

```
App\Tests.acceptance Tests (0)
```

---

```
App\Tests.functional Tests (0)
```

---

```
App\Tests.unit Tests (3)
```

---

```
-----  
Testing App\Tests\unit  
TICK CalculatorTest: Can create object (0.01s)  
TICK CalculatorTest: Add one and one (0.00s)  
TICK FirstUnitTest: One plus one equals two (0.00s)  
-----
```

Time: 166 ms, Memory: 18.00 MB

OK (3 tests, 3 assertions)



# 31

## Acceptance Tests

### 31.1 Test for home page text at / (project codeception03)

The project we started with (`security09`) has a simple public home page template:

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

{% block title %}home page {% endblock %}

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

So let's generate an **acceptance** test to simulate a user visiting / and seeing text `home page`:

1. generate a new acceptance test, note these classes end with `Cest`, and you use the `g:cest` parameter, which automatically give us access to `AcceptanceTester $I`, meaning we can write PHP code that is close to English-pseudocode, e.g. `$I->see('home page')`

```
$ vendor/bin/codecept g:cest acceptance HomePageCest
```

```
Test was created in /cept3/tests/acceptance/HomePageCest.php
```

1. We are given a skeleton Acceptance testing class:

```
namespace App\tests;
```

```
use App\Tests\AcceptanceTester;

class HomePageCest
{
    public function _before(AcceptanceTester $I)
    {
    }

    // tests
    public function tryToTest(AcceptanceTester $I)
    {
    }
}
```

2. Replace the contents of this class with one to test a home page visit:

```
class HomePageCest
{
    public function homePageContent(AcceptanceTester $I)
    {
        $I->amOnPage('/');
        $I->see('home page');
    }
}
```

There are 2 steps to our home page acceptance test;

1. Make the browser open URL /
2. Assert that somewhere in the text contents of the HTTP Response from the server is the text home page

## 31.2 Run the test (fail - server not running)

Run the tests - we'll see a failure of our Acceptance test, since the server isn't running:

```
$ vendor/bin/codecept run
```

```
...
```

```
App\Tests.acceptance Tests (1)
```

```
Testing App\Tests\acceptance
E HomePageCest: Home page content (0.02s)

...
There was 1 error:

-----
1) HomePageCest: Home page content
Test tests/acceptance/HomePageCest.php:homePageContent
[GuzzleHttp\Exception\ConnectException] cURL error 7: Failed to connect to localhost port 8000: C
```

The error details tell us that the connection was refused to localhost port 8000.

### 31.3 Run the test (pass, when server running)

Now run the Symfony server in a CLI terminal window with `php bin/console server:run`, and in a second CLI window run the Codeception tests again. This time it should pass:

```
Testing App\Tests\acceptance
TICK HomePageCest: Home page content (0.36s)
```

### 31.4 From red to green

There is an `about` page in this project:

- our `DefaultController` class defines this route:

```
/**
 * @Route("/about", name="about")
 */
public function about()
{
    $template = 'default/about.html.twig';
    $args = [];

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

- and there is this Twig template

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

{% block title %}about page {% endblock %}

{% block body %}
<h1>about page page</h1>
{% endblock %}
```

However, at present there isn't any HREF link from the home page to the about page. Let's test this:

```
class HomePageCest
{
    public function homePageContent(AcceptanceTester $I)
    {
        $I->amOnPage('/');
        $I->see('home page');
    }

    public function homePageLinkToAbout(AcceptanceTester $I)
    {
        $I->amOnPage('/');
        $I->click('about');
        $I->seeInCurrentUrl('/about');
        $I->see('about');
    }
}
```

If we run Codeception again

```
App\Tests\acceptance Tests (2)
    Testing App\Tests\acceptance
TICK  HomePageCest: Home page content (0.20s)
CROSS HomePageCest: Home page link to about (0.16s)

...
1) HomePageCest: Home page link to about
Test  tests/acceptance/HomePageCest.php:homePageLinkToAbout
Step  Click "about"
Fail  Link or Button by name or CSS or XPath element with 'about' was not found.
```

The details of the failure tells us that no link for text `about` could be found on the home page.

### 31.5 Make green - add link to about page in base Twig template

Add a link to the about page in the base Twig template `templates/base.html.twig`, just below the existing navigation link to the home page:

```
<nav>
    <ul>
        <li>
            <a href="{{ url('homepage') }}>home</a>
        </li>
        <li>
            <a href="{{ url('about') }}>about</a>
        </li>
        <li>
            <a href="{{ url('student_index') }}>student home</a>
        </li>
        <li>
            <a href="{{ url('admin_index') }}>admin home</a>
        </li>
    </ul>
</nav>
```

Now all tests should pass when we run Codeception.

### 31.6 Annotation style data provider to test multiple data

Let's use the Codeception Doctrine-style Annotation data provider, to test:

1. routes to home page and about page
2. our base navigation links, from home page to home page and about page

Create a new `Cest` acceptance test

```
$ vendor/bin/codecept g:cest acceptance NavbarCest
```

Replace the skeleton with the following:

```
namespace App\Tests;

use App\Tests\AcceptanceTester;
use Codeception\Example;
```

```
class NavbarCest
{
    /**
     * @example(url="/", text="home page")
     * @example(url="/about", text="about page")
     */
    public function staticPageContent(AcceptanceTester $I, Example $example)
    {
        $I->amOnPage($example['url']);
        $I->see($example['text']);
    }

    /**
     * @example(url="/", link="home")
     * @example(url="/about", link="about")
     */
    public function staticPageLinks(AcceptanceTester $I, Example $example)
    {
        $I->amOnPage('/');
        $I->click($example['link']);
        $I->seeCurrentUrlEquals($example['url']); // full URL
        $I->seeInCurrentUrl($example['url']); // part of URL
    }
}
```

We see that we are using the `Codeception\Example` class, which allows us to write `@example` annotations (note lower case e):

```
* @example(url="/", text="home page")
* @example(url="/about", text="about page")
```

Method `staticPageContent()` makes use of our custom example values `url` and `text` (we can name them what we want, and have as many as we wish). The example data is provided to the testing method as an associated map array. Each URL is visited, then an assertion is made that we see the provided text on the visited page `$I->see($example['text'])`.

Method `staticPageLinks()` makes use of our custom example values `url` and `link`. We start on the home page, attempt to click a link wrapped around the text of the provided link `$I->click($example['link'])`, and then check the the provided URL is in the browser address bar having clicked the link.

```
* @example(url="/", link="home")
* @example(url="/about", link="about")
```

Notice, for this example, we test both that the provided `url` matches the complete URL `$I->seeCurrentUrlEquals($example['url'])`, and also (a weaker test) is part of the URL string `$I->seeInCurrentUrl($example['url'])`.

## 31.7 Traditional Data Provider syntax

If you prefer the more traditional Data Provided syntax, you can learn about that at the Codeception documentation pages:

- <https://codeception.com/docs/07-AdvancedUsage#DataProvider-Annotations>

## 31.8 Common assertions for Acceptance tests

Here are the more common Codeception assertions for Acceptance tests:

- see / not see text in page
  - `$I->see(<text in page>)`
  - `$I->dontsee(<text in page>)` - text is **NOT** present (e.g. not see ‘invalid login’)
- see a link
  - `$I->seeLink('login');`
- click a link
  - `$I->click('login');`
- part of URL
  - `$I->seeInCurrentUrl('/blog');`
- match of complete URL
  - `$I->seeCurrentUrlEquals('/about');`
- see / don’t see in HTML `<title>`:
  - `$I->seeInTitle('Login')`
  - `$I->dontSeeInTitle('Register')`

More details of what you can do with `$I` can be found in the docs for the `PhpBrowser` module:

- <https://codeception.com/docs/modules/PhpBrowser>



# 32

## Filling out forms

### 32.1 Setup database

Let's generate a `Recipe` Entity class, and its CRUD - with no security.

Generate a `Recipe` Entity class:

- title (string)
- steps (string)
- time (integer)

Also generate its CRUD, and migrate it to the database.

There should now be a new recipe form at URL `/recipe/new`.

### 32.2 Cest to enter a new recipe (project `codeception04`)

Generate a new `Cest` to fill in the form for a new recipe:

```
$ vendor/bin/codecept g:cest acceptance RecipeCest
```

Edit the skeleton as follows:

```
namespace App\tests;
```

```
use App\Tests\AcceptanceTester;

class RecipeCest
{
    public function tryToTest(AcceptanceTester $I)
    {
        $I->amOnPage('/recipe/new');
        $I->fillField('#recipe_title', 'Boston Cheesecake');
        $I->fillField('#recipe_steps', 'buy packet - follow instructions');
        $I->fillField('#recipe_time', 60);
        $I->click('Save');
    }
}
```

As you can see we select an HTML form input by its **id**, so `#recipe_title` is the form field for the `title` property of the new Recipe to be created etc.

When you run the server and run the acceptance tests you'll now see a new recipe in the database, based on your acceptance tests completion of the forms.

# 33

## Codeception Symfony DB testing

### 33.1 Adding Symfony and Doctrine to the settings (project `codeception05`)

Let's configure Codeception to use Doctrine for DB work, and to recognise Symfony controllers.

Note - by stating `cleanup:true` for the Doctrine2 module, we are asking Codeception to create a database **transaction** before each test runs, and then to roll-back the translation after the test runs. So the database should be in the same state **after** each test as it was **before**. Thus, our tests are not linked and do not rely on running in a set sequence.

We need to edit file `/tests/acceptqance.yml.suite` to add the following:

```
- Symfony:  
    app_path: 'src'  
    environment: 'test'  
- Doctrine2:  
    depends: Symfony  
    cleanup: true  
- Asserts
```

So it should now contain:

```
actor: AcceptanceTester  
modules:
```

```
enabled:  
  - PhpBrowser:  
    url: http://localhost:8000  
  - \App\Tests\Helper\Acceptance  
  - Symfony:  
    app_path: 'src'  
    environment: 'test'  
  - Doctrine2:  
    depends: Symfony  
    cleanup: true  
  - Asserts
```

## 33.2 Test Users in DB from Fixtures

Generate a new Acceptance test UserCest:

```
codecept g:cest acceptance UserCest
```

Edit this new file /tests/acceptance/UserCest.php to contain the following:

```
<?php  
namespace App\Tests;  
use App\Tests\AcceptanceTester;  
  
class UserCest  
{  
  
    public function testUsersInDb(AcceptanceTester $I)  
    {  
        $I->seeInRepository('App\Entity\User', [  
            'username' => 'user'  
        ]);  
        $I->seeInRepository('App\Entity\User', [  
            'username' => 'admin'  
        ]);  
        $I->seeInRepository('App\Entity\User', [  
            'username' => 'matt'  
        ]);  
    }  
}
```

When you run the test should pass (assuming these users are in the DB from the Fixtures setup).

### 33.3 Check DB reset after each test

Add 2 more test methods, one to create a new user, and test it exists, and then one to check that that new user was removed when its test finished running:

```
public function testAddToDatabase(AcceptanceTester $I)
{
    $I->haveInRepository('App\Entity\User', [
        'username' => 'userTEMP',
        'password' => 'sdf',
        'roles' => ['ROLE_USER']
    ]);

    $I->seeInRepository('App\Entity\User', [
        'username' => 'userTEMP'
    ]);
}

public function testTEMPNoLongerInDatabase(AcceptanceTester $I)
{
    $I->dontSeeInRepository('App\Entity\User', [
        'username' => 'userTEMP'
    ]);
}
```

### 33.4 Add DB counts to our form-filling Acceptance test

Let's add code to count number of recipes in the DB before and after filling in the form to create a new recipe.

First, let's ensure there is one recipe in the data through a fixture:

```
namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;
```

```
use App\Entity\Recipe;

class RecipeFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        $recipe = new Recipe();
        $recipe->setTitle('Boston Cheesecake');
        $recipe->setSteps('open packge - follow instructions');
        $recipe->setTime(120);

        $manager->persist($recipe);

        $manager->flush();
    }
}
```

And load the fixtures via the Symfony console Doctrine command.

Now we can add to our Recipe form-filling acceptance test to count 1 recipe initially, then 2 after the new one has been created by filling in the form:

```
public function tryToTest(AcceptanceTester $I)
{
    // --- expect 1 in DB from fixtures
    $expectedCount = 1;
    $users = $I->grabEntitiesFromRepository('App\Entity\Recipe');
    $numRecipes = count($users);

    // assert
    $I->assertEquals($expectedCount, $numRecipes);

    // --- create new recipe via FORM
    // title suffix RANDOM number: 'Boston Cheesecake<randNum>'
    $randomNumber = rand(1,100);
    $recipeTitle = 'Boston Cheesecake' . $randomNumber;

    $I->amOnPage('/recipe/new');
    $I->fillField('#recipe_title', $recipeTitle);
    $I->fillField('#recipe_steps', 'buy packet - follow instructions');
    $I->fillField('#recipe_time', 60);
    $I->click('Save');
```

```
// ---- check added to repository
$I->seeInRepository('App\Entity\Recipe', [
    'title' => $recipeTitle
]);

// --- now should be 2 in DB
$expectedCount = 2;
$users = $I->grabEntitiesFromRepository('App\Entity\Recipe');
$numRecipes = count($users);

// assert
$I->assertEquals($expectedCount, $numRecipes);
}
```



## **Part XI**

# **Symfony Testing**



# 34

## Unit testing in Symfony

### 34.1 Testing in Symfony

Symfony is built by an open source community. There is a lot of information about how to test Symfony in the official documentation pages:

- [Symfony testing](#)
- [Testing with user authentication tokens](#)
- [How to Simulate HTTP Authentication in a Functional Test](#)

### 34.2 Installing Simple-PHPUnit (project `test01`)

Symfony has a special ‘bridge’ to work with PHPUnit. Add this to your project as follows:

```
$ composer req --dev simple-phpunit
```

You should now see a `/tests` directory created. Let’s create a simple test ( $1 + 1 = 2!$ ) to check everything is working okay.

Create a new class `/tests/SimpleTest.php` containing the following:

```
<?php  
namespace App\Test;
```

```
use PHPUnit\Framework\TestCase;

class SimpleTest extends TestCase
{
    public function testOnePlusOneEqualsTwo()
    {
        // Arrange
        $num1 = 1;
        $num2 = 1;
        $expectedResult = 2;

        // Act
        $result = $num1 + $num2;

        // Assert
        $this->assertEquals($expectedResult, $result);
    }
}
```

Note the following:

- test classes are located in directory `/tests`
- test classes end with the suffix `Test`, e.g. `SimpleTest`
- simple test classes extend the superclass `\PHPUnit\Framework\TestCase`
  - if we add a `uses` statement `use PHPUnit\Framework\TestCase` then we can simple extend `TestCase`
- simple test classes are in namespace `App\Test`
  - the names and namespaces of test classes testing a class in `/src` will reflect the namespace of the class being tested
    - i.e. If we write a class to test `/src/Controller/DefaultController.php` it will be `/tests/Controller/DefaultControllerTest.php`, and it will be in namespace `App\Controller\Test`
    - so our testing class architecture directly matches our source code architecture

### 34.3 Completing the installation

The first time you run Simple-PHPUnit it will probably need to install some more files.

There is an executable file in `/vendor/bin` named `simple-phpunit`. To run it just type `vendor/bin/simple-phpunit` (or for Windows, to run the BATch file, type `vendor\bin\simple-phpunit` - with backslashes since this is a Windows file path):

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

NOTE: Error message about missing `ext-mbstring`:

- if you get a message about “`ext mbstring`” required - when trying to work in Windows with Simple PHP Unit or make:
  - simple solution - in your `php.ini` file
    - Just as you did for `pdo_mysql`, remove the semi-colon in front of the statement in the `php.ini` file:
      - e.g. change `;extension=mbstring` to: `extension=mbstring`

## 34.4 Running Simple-PHPUnit

Let's run the tests (using the default configuration settings, in `phpunit.dist.xml`):

```
$ vendor/bin/simple-phpunit
PHPUnit 5.7.27 by Sebastian Bergmann and contributors.

Testing Project Test Suite
.

1 / 1 (100%)

Time: 93 ms, Memory: 4.00MB
OK (1 test, 1 assertion)
```

Dots are good. For each passed test you'll see a full stop. Then after all tests have run, you'll see a summary:

```
1 / 1 (100%)
```

This tells us how many passed, out of how many, and what the pass percentage was. In our case, 1 out of 1 passed = 100%.

## 34.5 Testing other classes (project test02)

our testing structure mirrors the code we are testing

Let's create a very simple class `Calculator.php` in `/src/Util`<sup>1</sup>, and then write a class to test our class. Our simple class will be a very simple calculator:

- method `add(...)` accepts 2 numbers and returns the result of adding them
- method `subtract()` accepts 2 numbers and returns the result of subtracting the second from the first

so our `Calculator` class is as follows:

```
<?php  
namespace App\Util;  
  
class Calculator  
{  
    public function add($n1, $n2)  
    {  
        return $n1 + $n2;  
    }  
  
    public function subtract($n1, $n2)  
    {  
        return $n1 - $n2;  
    }  
}
```

## 34.6 The class to test our calculator

We now need to write a test class to test our calculator class. Since our source code class is `/src/Util/Calculator.php` then our testing class will be `/tests/Util/Calculator.php`. And since the namespace of our source code class was `App\Util` then the namespace of our testing class will be `App\Util\Test`. Let's test making an instance-object of our class `Calculator`, and we will make 2 assertions:

- the reference to the new object is not NULL
- invoking the `add(...)` method with arguments of (1,1) and returns the correct answer (2!)

Here's the listing for our class `CalculatorTest`:

---

<sup>1</sup>Short for ‘Utility’ - i.e. useful stuff!

```
namespace App\Util\Test;

use App\Util\Calculator;
use PHPUnit\Framework\TestCase;

class CalculatorTest extends TestCase
{
    public function testCanCreateObject()
    {
        // Arrange
        $calculator = new Calculator();

        // Act

        // Assert
        $this->assertNotNull($calculator);
    }

    public function testAddOneAndOne()
    {
        // Arrange
        $calculator = new Calculator();
        $num1 = 1;
        $num2 = 1;
        $expectedResult = 2;

        // Act
        $result = $calculator->add($num1, $num2);

        // Assert
        $this->assertEquals($expectedResult, $result);
    }
}
```

Note:

- we had to add `use` statements for the class we are testing (`App\Util\Calculator`) and the PHP Unit `TestCase` class we are extending (`use PHPUnit\Framework\TestCase`)

Run the tests - if all goes well we should see 3 out of 3 tests passing:

```
$ vendor/bin/simple-phpunit
PHPUnit 5.7.27 by Sebastian Bergmann and contributors.
```

Testing Project Test Suite

...

3 / 3 (100%)

Time: 64 ms, Memory: 4.00MB  
OK (3 tests, 3 assertions)

## 34.7 Using a data provider to test with multiple datasets (project test03)

Rather than writing lots of methods to test different additions, let's use a **data provider** (via an annotation comment), to provide a single method with many sets of input and expected output values:

Here is our testing method:

```
/**
 * @dataProvider additionProvider
 */
public function testAdditionsWithProvider($num1, $num2, $expectedResult)
{
    // Arrange
    $calculator = new Calculator();

    // Act
    $result = $calculator->add($num1, $num2);

    // Assert
    $this->assertEquals($expectedResult, $result);
}
```

and here is the data provider (an array of arrays, with the right number of values for the parameters of `testAdditionsWithProvider(...)`):

```
public function additionProvider()
{
    return [
        [1, 1, 2],
        [2, 2, 4],
        [0, 1, 1],
    ];
}
```

Take special note of the annotation comment immediately before method `testAdditionsWithProvider(...)`:

```
/**  
 * @dataProvider additionProvider  
 */
```

The special comment starts with `/**`, and declares an annotation `@dataProvider`, followed by the name (identifier) of the method. Note especially that there are no parentheses () after the method name.

When we run Simple-PHPUnit now we see lots of tests being executed, repeatedly invoking `testAdditionsWithProvider(...)` with different arguments from the provider:

```
$ vendor/bin/simple-phpunit  
PHPUnit 5.7.27 by Sebastian Bergmann and contributors.  
  
Testing Project Test Suite  
.....  
6 / 6 (100%)  
  
Time: 65 ms, Memory: 4.00MB  
  
OK (6 tests, 6 assertions)
```

## 34.8 Configuring testing reports (project test04)

In addition to instant reporting at the command line, PHPUnit offers several different methods of recording test output text-based files.

PHPUnit (when run with Symfony's Simple-PHPUnit) reads configuration settings from file `phpunit.dist.xml`. Most of the contents of this file (created as part of the installation of the Simple-PHPUnit package) can be left as their defaults. But we can add a range of logs by adding the following 'logging' element in this file.

Many projects follow a convention where testing output files are stored in a directory named `build`. We'll follow that convention below - but of course change the name and location of the test logs to anywhere you want.

Add the following into file `phpunit.dist.xml`:

```
<logging>  
    <log type="junit" target=".build/logfile.xml"/>  
    <log type="testdox-html" target=".build/testdox.html"/>  
    <log type="testdox-text" target=".build/testdox.txt"/>  
    <log type="tap" target=".build/logfile.tap"/>  
</logging>
```

Figure 34.1 shows a screenshot of the contents of the created `/build` directory after Simple-PHPUnit has been run.

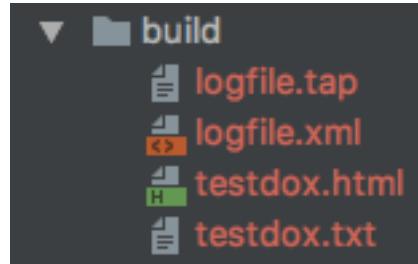


Figure 34.1: Contents of directory `/build`.

The `.txt` file version of `testdox` is perhaps the simplest output - showing `[x]` next to a passed method and `[ ]` for a test that didn't pass. The text output turns the test method names into more English-like sentences:

```
App\Test\Simple
[x] One plus one equals two
```

```
App\Util\Test\Calculator
[x] Can create object
[x] Add one and one
[x] Additions with provider
```

Another easy to understand logging format is the TAP (Test-Anywhere Protocol). Although officially deprecated by PHPUnit it still seems to work. What is nice about the TAP format is that repeated invocations of test methods iterating through a data-provider are enumerated, with the values. So we can see how many times, and their successes, a method was invoked with test data. This file is named (by our XML configuration above) `logfile.tap`:

```
TAP version 13
ok 1 - App\Test\SimpleTest::testOnePlusOneEqualsTwo
ok 2 - App\Util\Test\CalculatorTest::testCanCreateObject
ok 3 - App\Util\Test\CalculatorTest::testAddOneAndOne
ok 4 - App\Util\Test\CalculatorTest::testAdditionsWithProvider with data set #0 (1, 1, 2)
ok 5 - App\Util\Test\CalculatorTest::testAdditionsWithProvider with data set #1 (2, 2, 4)
ok 6 - App\Util\Test\CalculatorTest::testAdditionsWithProvider with data set #2 (0, 1, 1)
1..6
```

## 34.9 Testing for exceptions (project test07)

If our code throws an **Exception** while a test is being executed, and it was not caught, then we'll get an **Error** when we run our test.

For example, let's add a `divide(...)` method to our utility `Calculator` class:

```
public function divide($n, $divisor)
{
    if(empty($divisor)){
        throw new \InvalidArgumentException("Divisor must be a number");
    }

    return $n / $divisor;
}
```

In the code above we are throwing an `\InvalidArgumentException` when our `$divisor` argument is empty (0, null etc.).

Let's write a valid test ( $1/1 = 1$ ) in class `CalculatorTest`:

```
public function testDivideOneAndOne()
{
    // Arrange
    $calculator = new Calculator();
    $num1 = 1;
    $num2 = 1;
    $expectedResult = 1;

    // Act
    $result = $calculator->divide($num1, $num2);

    // Assert
    $this->assertEquals($expectedResult, $result);
}
```

This should pass.

Now let's try to write a test for 1 divided by zero. Not knowing how to deal with exceptions we might write something with a `fail(...)` instead of an `assert...`:

```
public function testDivideOneAndZero()
{
    // Arrange
    $calculator = new Calculator();
```

```
$num1 = 1;
$num2 = 0;
$expectedResult = 1;

// Act
$result = $calculator->divide($num1, $num2);

// Assert - FAIL - should not get here!
$this->fail('should not have got here - divide by zero not permitted');
}
```

But when we run simple-phpunit we'll get an error since the (uncaught) Exceptions is thrown before our fail(...) statement is reached:

```
$ vendor/bin/simple-phpunit
PHPUnit 5.7.27 by Sebastian Bergmann and contributors.

Warning:      Deprecated TAP test listener used

Testing Project Test Suite
.....E                                         10 / 10 (100%)

Time: 1.21 seconds, Memory: 10.00MB

There was 1 error:

1) App\Util\Test\CalculatorTest::testDivideOneAndZero
InvalidArgumentException: Divisor must be a number

.../src/Util/Calculator.php:21
/Users/matt/Library/Mobile Documents/com~apple~CloudDocs/11_Books/symfony/php-symfony4-book-

ERRORS!
Tests: 10, Assertions: 9, Errors: 1.
```

And our logs will confirm the failure:

```
App\Tests\Controller\DefaultController
[x] Homepage response code okay
[x] Homepage content contains hello world

App\Test\Simple
[x] One plus one equals two
```

```
App\Util\Test\Calculator
[x] Can create object
[x] Add one and one
[x] Additions with provider
[x] Divide one and one
[ ] Divide one and zero
```

### 34.10 PHPUnit `expectException(...)`

PHPUnit allows us to declare that we expect an exception - but we must declare this **before** we invoke the method that will throw the exception.

Here is our improved method, with `expectException(...)` and a better `fail(...)` statement, that tells us which exception was expected and not thrown:

```
public function testDivideOneAndZero()
{
    // Arrange
    $calculator = new Calculator();
    $num1 = 1;
    $num2 = 0;
    $expectedResult = 1;

    // Expect exception - BEFORE you Act!
    $this->expectException(\InvalidArgumentException::class);

    // Act
    $result = $calculator->divide($num1, $num2);

    // Assert - FAIL - should not get here!
    $this->fail("Expected exception {\InvalidArgumentException::class} not thrown");
}
```

Now all our tests pass:

```
$ vendor/bin/simple-phpunit
PHPUnit 5.7.27 by Sebastian Bergmann and contributors.

Warning:      Deprecated TAP test listener used

Testing Project Test Suite
```

### 34.11 PHPUnit annotation comment `@expectedException`

PHPUnit allows us to use an annotation comment to state that we expect an exception to be thrown during the execution of a particular test. This is a nice way to keep our test logic simple.

Since annotation comments are declared immediately **before** the method, some programmers (I do!) prefer the annotation way of declaring that we expect a test method to result in an exception being thrown:

```
/**  
 * @expectedException \InvalidArgumentException  
 */  
  
public function testDivideOneAndZeroAnnotation()  
{  
    // Arrange  
    $calculator = new Calculator();  
    $num1 = 1;  
    $num2 = 0;  
  
    // Act  
    $result = $calculator->divide($num1, $num2);  
  
    // Assert - FAIL - should not get here!  
    $this->fail("Expected exception {\InvalidArgumentException::class} not thrown");  
}
```

NOTE: You must ensure the exception class is fully namespaced in the annotation comment (no `::class` shortcuts!).

### 34.12 Testing for custom Exception classes

While the built-in PHP Exceptions are fine for simple projects, it is very useful to create custom exception classes for each project you create. Working with, and testing for, objects of custom Exception classes is very simple in Symfony:

1. Create your custom Exception class in `/src/Exception`, in the namespace `App\Exception`. For example you might create a custom Exception class for an invalid Currency in a money exchange system as follows:

```
// file: /src/Exception/UnknownCurrencyException.php

namespace App\Exception;

use Exception;

class UnknownCurrencyException extends Exception
{
    public function __construct($message = null)
    {
        if(empty($message)) {
            $message = 'Unknown currency';
        }
        parent::__construct($message);
    }

}
```

2. Ensure your source code throws an instance of your custom Exception. For example:

```
use App\Exception\UnknownCurrencyException;

...

public function euroOnlyExchange($currency)
{
    $currency = strtolower($currency);
    if('euro' != $currency){
        throw new UnknownCurrencyException();
    }
}
```

3. In your tests you must check for the expected custom Exception class. E.g. using the annotation approach:

```
/**
 * @expectedException App\Exception\UnknownCurrencyException
 */
public function testInvalidCurrencyException()
{
    ... code here to trigger exception to be thrown ...

    // Assert - FAIL - should not get here!
    $this->fail("Expected exception {\Exception} not thrown");
}
```

```
}
```

**NOTE:** You have to provide the full namespace in the annotation comment, i.e. `App\Exception\UnknownCurrencyException`. Having a `use` statement above will not work properly

### 34.13 Checking Types with assertions

Sometimes we need to check the `type` of a variable. We can do this using the `assertInternalType(...)` method.

For example:

```
$result = 1 + 2;

// check result is an integer
$this->assertInternalType('int', $result);
```

Learn more in the PHPUnit documentation:

- <https://phpunit.de/manual/6.5/en/appendices.assertions.html#appendices.assertions.assertInternalType>

### 34.14 Same vs. Equals

There are 2 similar assertions in PHPUnit:

- `assertSame(...)`: works like the `==` identity operator in PHP
- `assertEquals(...)`: works like the `==` comparison

When we want to know if the values inside (or referred to) by two variables or expressions are equivalent, we use the weaker `==` or `assertEquals(...)`. For example, do two variables refer to object-instances that contain the same property values, but may be different objects in memory.

When we want to know if the values inside (or referred to) by two variables are exactly the same, we use the stronger `==` or `assertSame(...)`. For example, do two variables both refer to the same object in memory.

The use of `assertSame(...)` is useful in unit testing to check the types of values - since the value returned by a function must refer to the same numeric or string (or whatever) literal. So we could write another way to test that a function returns an integer result as follows:

```
$expectedResult = 3;
$result = 1 + 2;
```

```
// check result is an interger  
$this->assertSame($expectedResult, $result);
```



# 35

## Code coverage and xDebug

### 35.1 Code Coverage

It's good to know how **much** of our code we have tested, e.g. how many methods or logic paths (e.g. if-else- branches) we have and have not tested.

Code coverage reports can be text, XML or nice-looking HTML. See Figure 2.2 for a screenshot of an HTML coverage report for a `Util` class with 4 methods. We can see that while `add` and `divide` have been fully (100%) covered by tests, methods `subtract` and `process` are insufficiently covered.

Code Coverage							
		Functions and Methods			Lines		
Total		50.00%	2 / 4	CRAP	72.73%	8 / 11	
Calculator		50.00%	2 / 4	9.30	72.73%	8 / 11	
<code>add</code>		100.00%	1 / 1	1	100.00%	1 / 1	
<code>subtract</code>		0.00%	0 / 1	2	0.00%	0 / 1	
<code>divide</code>		100.00%	1 / 1	2	100.00%	3 / 3	
<code>process</code>		0.00%	0 / 1	4.59	66.67%	4 / 6	

Figure 35.1: Screenshot of HTML coverage report.

This is known as code coverage, and easily achieved by:

1. Adding a line to the PHPUnit configuration file (`php.ini`)
2. Ensuring the **xDebug** PHP debugger is installed and activated

See Appendix R for these steps.

## 35.2 Generating Code Coverage HTML report

Add the following element as a child to the `<logging>` element in file `phpunitinit.xml.dist`:

```
<log type="coverage-html" target=".build/report"/>
```

So the full content of the `<logging>` element is now:

```
<logging>
  <log type="coverage-html" target=".build/report"/>
  <log type="junit" target=".build/logfile.xml"/>
  <log type="testdox-html" target=".build/testdox.html"/>
  <log type="testdox-text" target=".build/testdox.txt"/>
  <log type="tap" target=".build/logfile.tap"/>
</logging>
```

Now when you run `vendor/bin/simple-phpunit` you'll see a new directory `report` inside `/build`. Open the `index.html` file in `/build/report` and you'll see the main page of your coverage report. See Figure 35.2.

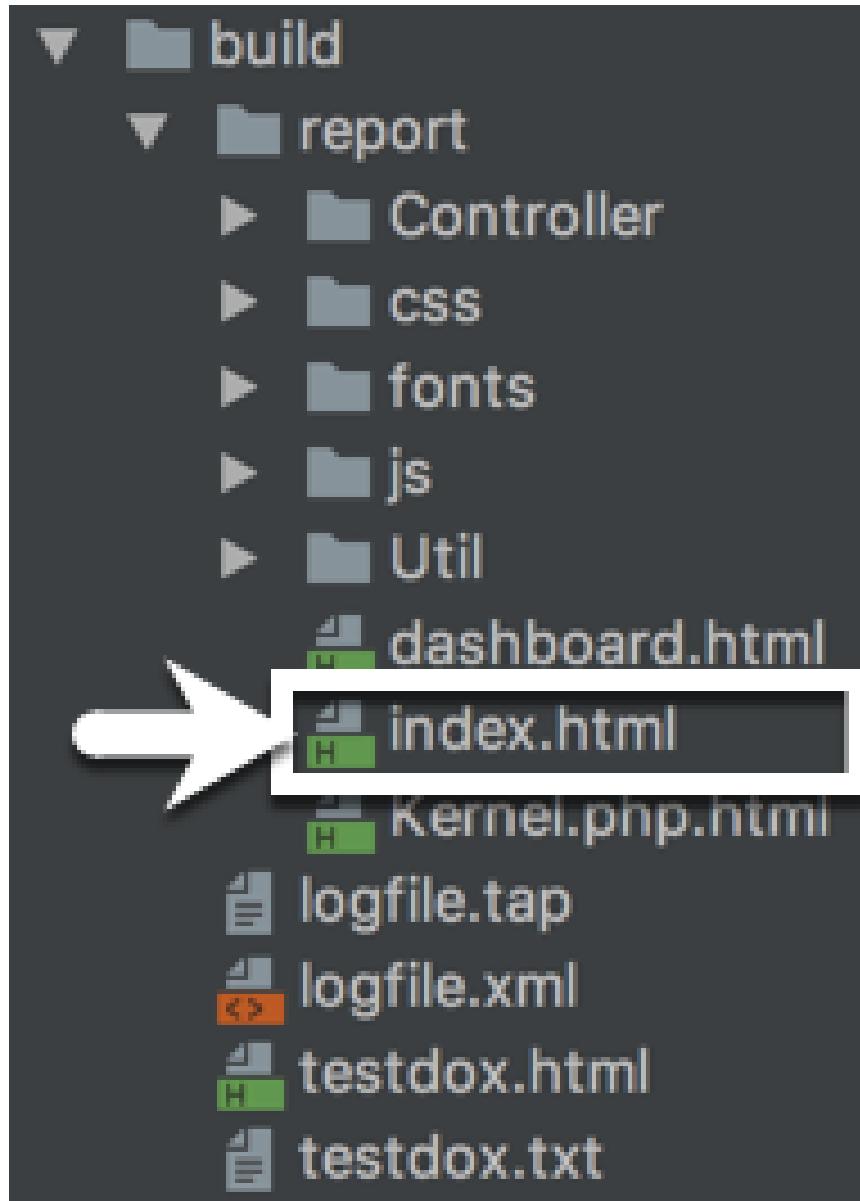


Figure 35.2: Build files showing index.html in /build/report.

### 35.3 Tailoring the ‘whitelist’

PHPUnit decides which sources file to analyse and build coverage reports for by using a ‘whitelist’ - i.e. a list of just those files and/or directories that we are interested in at this point in time. The whitelist is inside the `<filter>` element in PHPUnit configuration file ‘`phpunit.xml.dist`’.

the default whitelist is `./src` - i.e **all** files in our source directory. But, for example, this will include Kernel, which we generally don’t touch. So if you want to go **GREEN** for everything in your coverage report, then you can list only those directories inside `/src` that you are interested in.

For our example above we were working with classes in `/src/Util` and `src/Controller`, so that’s what we can list in our ‘whitelist’. You can always ‘disable’ lines in XML by wrapping an XML command around them `<!-- ... -->`, which we’ve done below to the default `./src/` white list element:

```
<filter>
  <whitelist>
    <!--
      // ignore this element for now ...
      <directory>./src/</directory>
    -->
    <directory>./src/Controller</directory>
    <directory>./src/Util</directory>
  </whitelist>
</filter>
```

# 36

## Web testing

### 36.1 Testing controllers with `WebTestCase` (project `test05`)

Symfony provides a package for simulating web clients so we can (functionally) test the contents of HTTP Responses output by our controllers.

First we need to add 2 packages to the project development environment:

```
composer req --dev browser-kit css-selector
```

Note - these next steps assume your project has Twig, annotations and the Symfony maker packages available, so you may need to add these to your project as well:

```
composer req twig annotations make
```

Let's make a new `DefaultController` class:

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

Let's edit the generated template to include the message Hello World. Edit `/templates/default/index.html.twig`:

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

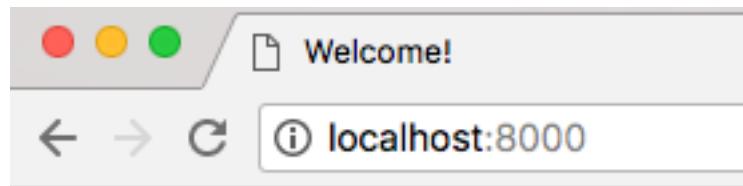
```
{% block body %}  
<h1>Welcome</h1>
```

```
Hello World from the default controller  
{% endblock %}
```

Let's also set the URL to simply / for this route in /src/Controller/DefaultController.php:

```
class DefaultController extends Controller
{
    /**
     * @Route("/", name="default")
     */
    public function index()
    {
        return $this->render('default/index.html.twig', [
            'controller_name' => 'DefaultController',
        ]);
    }
}
```

If we run a web server and visit the home page we should see our 'hello world' message in a browser - see Figure 36.1.



# Welcome

Hello World from the default controller

Figure 36.1: Contents of directory /build.

## 36.2 Automating a test for the home page contents

Let's write a test class for our `DefaultController` class. So we create a new test class `/tests/Controller/DefaultControllerTest.php`. We'll write 2 tests, one to check that we get a 200 OK HTTP success code when we try to request /, and secondly that the content received in the HTTP Reponse contains the text Hello World:

```
namespace App\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;
```

```
use Symfony\Component\HttpFoundation\Response;

class DefaultControllerTest extends WebTestCase
{
    // methods go here
}
```

We see our class must extend `WebTestCase` from package `Symfony\Bundle\FrameworkBundle\Test\`, and also makes use of the Symfony Foundation `Response` class.

Our method to test for a 200 OK Reponse code is as follows:

```
public function testHomepageResponseCodeOkay()
{
    // Arrange
    $url = '/';
    $httpMethod = 'GET';
    $client = static::createClient();

    // Assert
    $client->request($httpMethod, $url);

    // Assert
    $this->assertSame(
        Response::HTTP_OK,
        $client->getResponse()->getStatusCode()
    );
}
```

We see how a web client object `$client` is created and makes a GET request to `/`. We see how we can interrogate the contents of the HTTP Response received using the `getResponse()` method, and within that we can extract the status code, and compare with the class constant `HTTP_OK` (200).

Here is our method to test for a 200 OK Reponse code is as follows:

```
public function testHomepageContentContainsHelloWorld()
{
    // Arrange
    $url = '/';
    $httpMethod = 'GET';
    $client = static::createClient();
    $searchText = 'Hello World';

    // Act
}
```

```
$client->request($httpMethod, $url);

    // Assert
    $this->assertContains(
        $searchText,
        $client->getResponse()->getContent()
    );
}
```

We see how we can use the `assertContains` string method to search for the string `Hello World` in the content of the HTTP Response.

When we run Simple-PHPUnit we can see success both from the full-stops at the CLI, and in our log files, e.g.:

```
App\Tests\Controller\DefaultController
    [x] Homepage response code okay
    [x] Homepage content contains hello world

    ...
```

### 36.3 Normalise content to lowercase (project `test06`)

I lost 30 minutes thinking my web app wasn't working! This was due to the difference between `Hello world` and `Hello World` (`w` vs `W`).

This kind of problem can be avoided if we **normalise** the content from the Response, e.g. making all letters **lower-case**. This only makes sense if you are happy (at this stage) to not worry about the case of text content in your pages (you could always write some specific spelling / grammar checker tests for that ...)

The solution is to use the built-in PHP function `strtolower()`:

```
public function testHomepageContentContainsHelloWorld()
{
    // Arrange
    $url = '/';
    $httpMethod = 'GET';
    $client = static::createClient();
    $searchText = 'Hello World';

    // Act
    $client->request($httpMethod, $url);
```

```
$content = $client->getResponse()->getContent();

// to lower case
$searchTextLowerCase = strtolower($searchText);
$contentLowerCase = strtolower($content);

// Assert
$this->assertContains(
    $searchTextLowerCase,
    $contentLowerCase
);
}
```

## 36.4 Test multiple pages with a data provider

Avoid duplicating code when only the values change, by writing a testing method fed by arrays of test input / expected values from a data provider method:

```
/**
 * @dataProvider basicPagesTextProvider
 */
public function testPublicPagesContainBasicText($url, $expectedLowercaseText)
{
    // Arrange
    $httpMethod = 'GET';
    $client = static::createClient();

    // Act
    $client->request($httpMethod, $url);
    $content = $client->getResponse()->getContent();
    $statusCode = $client->getResponse()->getStatusCode();

    // to lower case
    $contentLowerCase = strtolower($content);

    // Assert - status code 200
    $this->assertSame(Response::HTTP_OK, $statusCode);

    // Assert - expected content
}
```

```
$this->assertContains(
    $exepctedLowercaseText,
    $contentLowerCase
);
}

public function basicPagesTextProvider()
{
    return [
        ['/','home page'],
        ['/about','about'],
    ];
}
```

## 36.5 Testing links (project `test08`)

We can test links with our web crawler as follows:

- get reference to crawler object when you make the initial request

```
$httpMethod = 'GET';
$url = '/about';
$crawler = $client->request($httpMethod, $url);
```

- select a link with:

```
$linkText = 'login';
$link = $crawler->selectLink($linkText)->link();
```

- click the link with:

```
$client->click($link);
```

- then check the content of the new request

```
$content = $client->getResponse()->getContent();

// set $expectedText to what should in page when link has been followed ...
$this->assertContains(
    $exepctedText,
    $content
);
```

For example, if we create a new ‘about’ page Twig template ‘/templates/default/about.html.twig’:

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

```
{% block body %}  
<h1>About page</h1>  
  
<p>  
    About this great website!  
</p>  
  
{% endblock %}
```

and a `DefaultController` method to display this page when the route matches `/about`:

```
/**  
 * @Route("/about", name="about")  
 */  
public function aboutAction()  
{  
    $template = 'default/about.html.twig';  
    $args = [];  
    return $this->render($template, $args);  
}
```

If we add to our base Twig template links to the homepage and the about, in template `/templates/base.html.twig`:

```
<!DOCTYPE html>  
<html>  
    <head>  
        <meta charset="UTF-8">  
        <title>{% block title %}Welcome!{% endblock %}</title>  
        {% block stylesheets %}{% endblock %}  
    </head>  
    <body>  
  
        <nav>  
            <ul>  
                <li>  
                    <a href="{{ url('homepage') }}>home</a>  
                </li>  
                <li>  
                    <a href="{{ url('about') }}>about</a>  
                </li>  
            </ul>  
        </nav>
```

```
</nav>

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

We can now write a test method to:

- request the homepage /
- select and click the `about` link
- test that the content of the new response is the ‘about’ page if it contains ‘about page’

Here is our test method:

```
public function testHomePageLinkToAboutWorks()
{
    // Arrange
    $url = '/';
    $httpMethod = 'GET';
    $client = static::createClient();
    $searchText = 'about page';
    $linkText = 'about';

    // Act
    $crawler = $client->request($httpMethod, $url);
    $link = $crawler->selectLink($linkText)->link();
    $client->click($link);
    $content = $client->getResponse()->getContent();

    // to lower case
    $searchTextLowerCase = strtolower($searchText);
    $contentLowerCase = strtolower($content);

    // Assert
    $this->assertContains($searchTextLowerCase, $contentLowerCase);
}
```

## 36.6 Issue with routes that end with a forward slash /

Often we write (or generate) a controller that adds URL and route name **prefixes** - by writing a route annotation command immediately before the class declaration. For example, look at the first 2 routes of this simple calculator class:

```
/*
 * controller for calculator functions
 *
 * @Route("/calc", name="calc_")
 */
class CalcController extends Controller
{
    /**
     * home page for calc pages
     * @Route("/", name="home")
     */
    public function index()
    {
        $template = 'calc/index.html.twig';
        $args = [];
        return $this->render($template, $args);
    }

    /**
     * process the calc stuff
     *
     * @Route("/process", name="process")
     *
     * @param Request $request
     * @return \Symfony\Component\HttpFoundation\Response
     */
    public function processAction(Request $request)
    {
        ...
    }
}
```

However, a consequence of this is that often the index route for this controller will be defined as having a trailing forward slash /. For example look at this route list:

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

```
calc_home      ANY      ANY      ANY      /calc/
calc_process   ANY      ANY      ANY      /calc/process
```

As we can see the home calculator page URL route is '/calc/'. When running the site with a server and visiting this page with a web browser client, the server will usually simply redirect /calc/ to /calc if the initial request doesn't match any routes. However, when **controller testing** such controllers if there is not a complete match between the URL being tested and the route pattern, the test will fail.

For example, a test in this form will **FAIL** for the route `/calc/` as defined above:

```
public function testExchangePage()
{
    $httpMethod = 'GET';
    $url = '/calc';
    $client = static::createClient();
    $client->request($httpMethod, $url);
    $this->assertSame(Response::HTTP_OK, $client->getResponse()->getStatusCode());
}
```

### 36.6.1 Solution 1: Ensure url pattern in test method exactly matches router url pattern

One solution to this problem is to ensure the URL in our test method exactly matches the URL in the applications router. So for our calculator home page we need to ensure the URL passed to the client ends with this trailing forward slash / character”:

### 36.6.2 Solution 2: Instruct client to ‘follow redirects’

An alternative solution is to instruct our tester's web crawler client to **follow redirects**, so that the request is re-processed if the request with no trailing forward slash / failed:

```
public function testExchangePage()
```

```
$httpMethod = 'GET';
$url = '/calc';
$client = static::createClient();
$client->followRedirects(true); // <<<<<<<<< follow redirects
$client->request($httpMethod, $url);
$this->assertSame(Response::HTTP_OK, $client->getResponse()->getStatusCode());
}
```



# 37

## Testing web forms

### 37.1 Testing forms (project test09)

Testing forms is similar to testing links, in that we need to get a reference to the form (via its submit button), then insert our data, then submit the form, and examine the content of the new response received after the form submission.

Assume we have a Calculator class as follows in `/src/Util/Calculator.php`:

```
namespace App\Util;

class Calculator
{
    public function add($n1, $n2)
    {
        return $n1 + $n2;
    }

    public function subtract($n1, $n2)
    {
        return $n1 - $n2;
    }

    public function divide($n, $divisor)
```

```
{  
    if(empty($divisor)){  
        throw new \InvalidArgumentException("Divisor must be a number");  
    }  
  
    return $n / $divisor;  
}  
  
public function process($n1, $n2, $process)  
{  
    switch($process){  
        case 'subtract':  
            return $this->subtract($n1, $n2);  
            break;  
        case 'divide':  
            return $this->divide($n1, $n2);  
            break;  
        case 'add':  
        default:  
            return $this->add($n1, $n2);  
    }  
}
```

```
}
```

Assume we also have a `CalculatorController` class in `/src/Controller/`:

```
namespace App\Controller;  
  
use App\Util\Calculator;  
use Symfony\Component\Routing\Annotation\Route;  
use Symfony\Bundle\FrameworkBundle\Controller\Controller;  
use Symfony\Component\HttpFoundation\Request;  
  
class CalcController extends Controller  
{  
    ... methods go here ...  
}
```

There is a calculator home page that displays the form Twig template at `/templates/calc/index.html.twig`:

```
/**  
 * @Route("/calc", name="calc_home")  
 */
```

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

and a ‘process’ controller method to received the form data (n1, n2, operator) and process it: There is a calculator home page that displays the form Twig template at /templates/calc/index.html.twig:

```
/**
 * @Route("/calc/process", name="calc_process")
 */
public function processAction(Request $request)
{
    // extract name values from POST data
    $n1 = $request->request->get('num1');
    $n2 = $request->request->get('num2');
    $operator = $request->request->get('operator');

    $calc = new Calculator();
    $answer = $calc->process($n1, $n2, $operator);

    return $this->render(
        'calc/result.html.twig',
        [
            'n1' => $n1,
            'n2' => $n2,
            'operator' => $operator,
            'answer' => $answer
        ]
    );
}
```

The Twig template to display our form looks as follows /templates/calc/index.html.twig:

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

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

<form method="post" action="{{ url('calc_process') }}">
    <p>
        Num 1:
        <input type="text" name="num1" value="1">

```

```
</p>
<p>
    Num 2:
    <input type="text" name="num2" value="1">
</p>
<p>
    Operation:
    <br>
    ADD
    <input type="radio" name="operator" value="add" checked>
    <br>
    SUBTRACT
    <input type="radio" name="operator" value="subtract">
    <br>
    DIVIDE
    <input type="radio" name="operator" value="divide">
</p>

<p>
    <input type="submit" name="calc_submit">
</p>
</form>

{% endblock %}
```

and the Twig template to confirm received values, and display the answer `result.html.twig` contains:

```
<h1>Calc RESULT</h1>
<p>
    Your inputs were:
    <br>
    n1 = {{ n1 }}
    <br>
    n2 = {{ n2 }}
    <br>
    operator = {{ operator }}
<p>
    answer = {{ answer }}
```

## 37.2 Test we can get a reference to the form

Let's test that can see the form page

```
public function testHomepageResponseCodeOkay()
{
    // Arrange
    $url = '/calc';
    $httpMethod = 'GET';
    $client = static::createClient();
    $expectedResult = Response::HTTP_OK;

    // Assert
    $client->request($httpMethod, $url);
    $statusCode = $client->getResponse()->getStatusCode();

    // Assert
    $this->assertSame($expectedResult, $statusCode);
}
```

Let's test that we can get a reference to the form on this page, via its 'submit' button:

```
public function testFormReferenceNotNull()
{
    // Arrange
    $url = '/calc';
    $httpMethod = 'GET';
    $client = static::createClient();
    $crawler = $client->request($httpMethod, $url);
    $buttonName = 'calc_submit';

    // Act
    $buttonCrawlerNode = $crawler->selectButton($buttonName);
    $form = $buttonCrawlerNode->form();

    // Assert
    $this->assertNotNull($form);
}
```

NOTE: We have to give each form button we wish to test either a `name` or `id` attribute. In our example we gave our calculator form the `name` attribute with value `calc_submit`:

```
<input type="submit" name="calc_submit">
```

### 37.3 Submitting the form

Assuming our form has some default values, we can test submitting the form by then checking if the content of the response after clicking the submit button contains test ‘Calc RESULT’:

```
public function testCanSubmitAndSeeResultText()
{
    // Arrange
    $url = '/calc';
    $httpMethod = 'GET';
    $client = static::createClient();
    $crawler = $client->request($httpMethod, $url);
    $expectedContentAfterSubmission = 'Calc RESULT';
    $expectedContentLowerCase = strtolower($expectedContentAfterSubmission);
    $buttonName = 'calc_submit';

    // Act
    $buttonCrawlerNode = $crawler->selectButton($buttonName);
    $form = $buttonCrawlerNode->form();

    // submit the form
    $client->submit($form);

    // get content from next Response & make lower case
    $content = $client->getResponse()->getContent();
    $contentLowerCase = strtolower($content);

    // Assert
    $this->assertContains($expectedContentLowerCase, $contentLowerCase);
}
```

### 37.4 Entering form values then submitting

Once we have a reference to a form (\$form) entering values is completed as array entry:

```
$form['num1'] = 1;
$form['num2'] = 2;
$form['operator'] = 'add';
```

So we can now test that we can enter some values, submit the form, and check the values in the response generated.

Let's submit 1, 2 and add:

```
public function testSubmitOneAndTwoAndValuesConfirmed()
{
    // Arrange
    $url = '/calc';
    $httpMethod = 'GET';
    $client = static::createClient();
    $crawler = $client->request($httpMethod, $url);
    $buttonName = 'calc_submit';

    // Act
    $buttonCrawlerNode = $crawler->selectButton($buttonName);
    $form = $buttonCrawlerNode->form();

    $form['num1'] = 1;
    $form['num2'] = 2;
    $form['operator'] = 'add';

    // submit the form & get content
    $crawler = $client->submit($form);
    $content = $client->getResponse()->getContent();
    $contentLowerCase = strtolower($content);

    // Assert
    $this->assertContains(
        '1',
        $contentLowerCase
    );
    $this->assertContains(
        '2',
        $contentLowerCase
    );
    $this->assertContains(
        'add',
        $contentLowerCase
    );
}
```

The test above tests that after submitting the form we see the values submitted confirmed back to us.

## 37.5 Testing we get the correct result via form submission

Assuming all our `Calculator`, methods have been individually **unit tested**, we can now test that after submitting some values via our web form, we get the correct result returned to the user in the final response.

Let's submit 1, 2 and `add`, and look for 3 in the final response:

```
public function testSubmitOneAndTwoAndResultCorrect()
{
    // Arrange
    $url = '/calc';
    $httpMethod = 'GET';
    $client = static::createClient();
    $num1 = 1;
    $num2 = 2;
    $operator = 'add';
    $expectedResult = 3;
    // must be string for string search
    $expectedResultString = $expectedResult . '';
    $buttonName = 'calc_submit';

    // Act

    // (1) get form page
    $crawler = $client->request($httpMethod, $url);

    // (2) get reference to the form
    $buttonCrawlerNode = $crawler->selectButton($buttonName);
    $form = $buttonCrawlerNode->form();

    // (3) insert form data
    $form['num1'] = $num1;
    $form['num2'] = $num2;
    $form['operator'] = $operator;

    // (4) submit the form
    $crawler = $client->submit($form);
    $content = $client->getResponse()->getContent();

    // Assert
}
```

```
$this->assertContains($expectedResultString, $content);
```

That's it - we can now select forms, enter values, submit the form and interrogate the response after the submitted form has been processed.

## 37.6 Selecting form, entering values and submitting in one step

Using the **fluent** interface,, Symfony allows us to combine the steps of selecting the form, setting form values and submitting the form. E.g.:

```
$client->submit($client->request($httpMethod, $url)->selectButton($buttonName)->form([
    'num1' => $num1,
    'num2' => $num2,
    'operator' => $operator,
]));
```

So we can write a test with fewer steps if we wish:

```
public function testSelectSetValuesSubmitInOneGo()
{
    // Arrange
    $url = '/calc';
    $httpMethod = 'GET';
    $client = static::createClient();
    $num1 = 1;
    $num2 = 2;
    $operator = 'add';
    $expectedResult = 3;
    // must be string for string search
    $expectedResultString = $expectedResult . '';
    $buttonName = 'calc_submit';

    // Act
    $client->submit($client->request($httpMethod, $url)->selectButton($buttonName)->form([
        'num1' => $num1,
        'num2' => $num2,
        'operator' => $operator,
    ]));
    $content = $client->getResponse()->getContent();

    // Assert
}
```

```
$this->assertContains($expectedResultString, $content);  
}
```

## **Part XII**

# **Appendices**



# A

## Software required for Symfony development

### A.1 Don't confuse different software tools

Please do not confuse the following:

- Git and Github
- PHP and PHPStorm

Here is a short description of each:

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

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

## A.2 Software tools

Ensure you have the following setup for developing Symfony software on your local machine

- PHP 7.2.5 or later (free, open source)
- Composer (up-to-date with `composer self-update`)(free, open source - a PHP program!)
- PHPStorm (with educational free account if you're a student!) - or some other editor of your choice
- MySQL Workbench (Community Edition free)
- Git (free, open source)

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

## A.3 Test software by creating a new Symfony 4 project

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

Follow the steps in Appendix ??.

# B

## PHP Windows setup

### B.1 Check if you have PHP installed and working

You need PHP version 7.2.5 or later.

Check your PHP version at the command line with:

```
> php -v
PHP 7.3.1 (cli) (built: May  9 2018 19:49:10)
Copyright (c) 1997-2017 The PHP Group
Zend Engine v3.1.0, Copyright (c) 1998-2017 Zend Technologies
```

If your version is older than 7.2.5, or you get an error about command not understood, then complete the steps below.

#### B.1.1 Download the latest version of PHP

Get the latest (7.4 at the time of writing) PHP Windows ZIP from:

- [php.net](#) click the **Windows Downloads** link

NOTE: If installing on Windows ensure you are **displaying file extensions**, e.g. so you see `php.exe` and `php.ini` not just `php` - Don't rely on Windows to show the right icon while *hiding the full filename...*

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

Do the following:

- unzip the PHP folder into: `C:\php`
- so you should now have a file `php.exe` inside `C:\php`, along with lots of other files
- make a copy the file `C:\php\php.ini-development`, naming the copy `C:\php\php.ini`

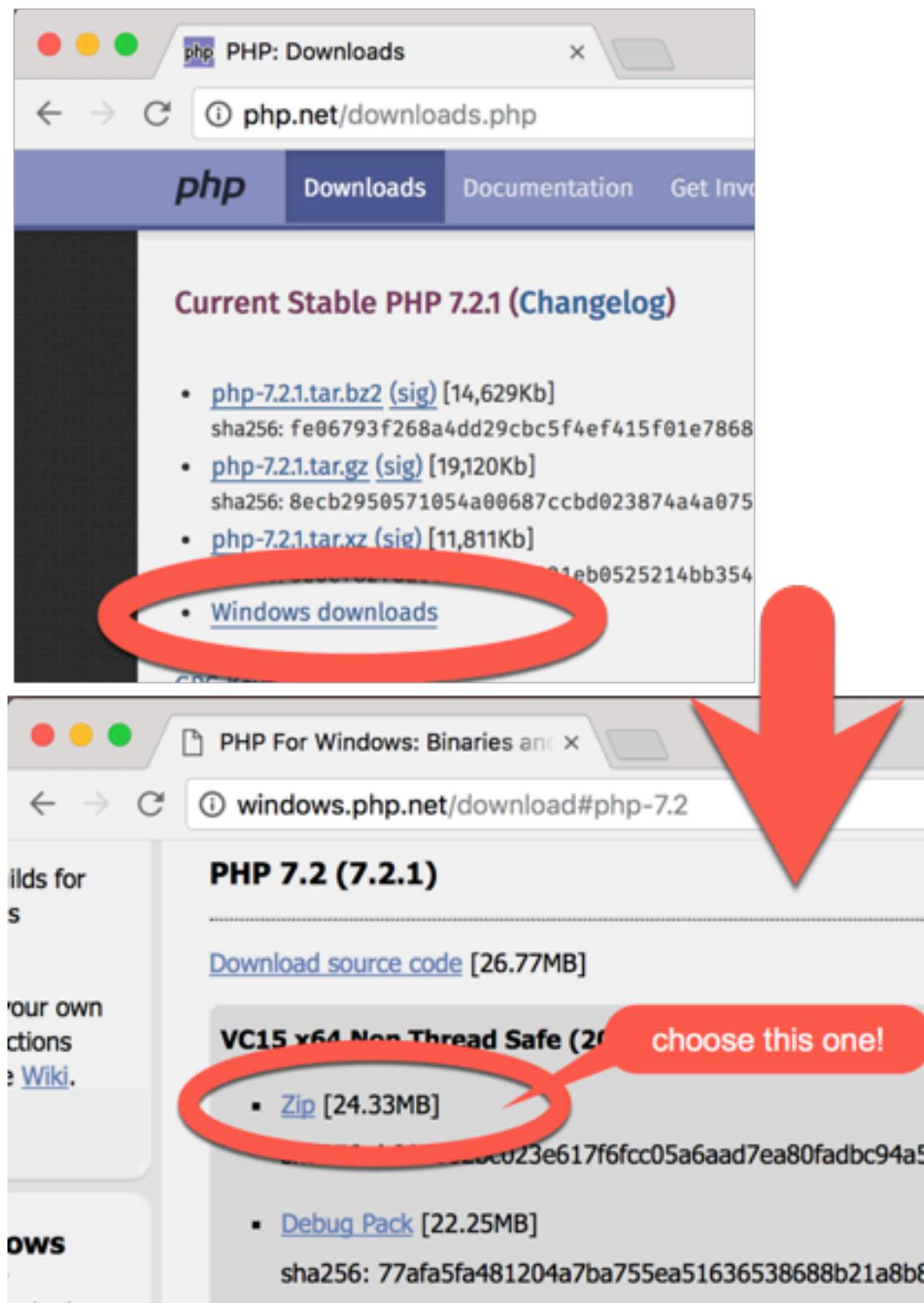


Figure B.1: PHP.net / Windows ZIP download pages.

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

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

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

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

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

## APPENDIX B. PHP WINDOWS SETUP

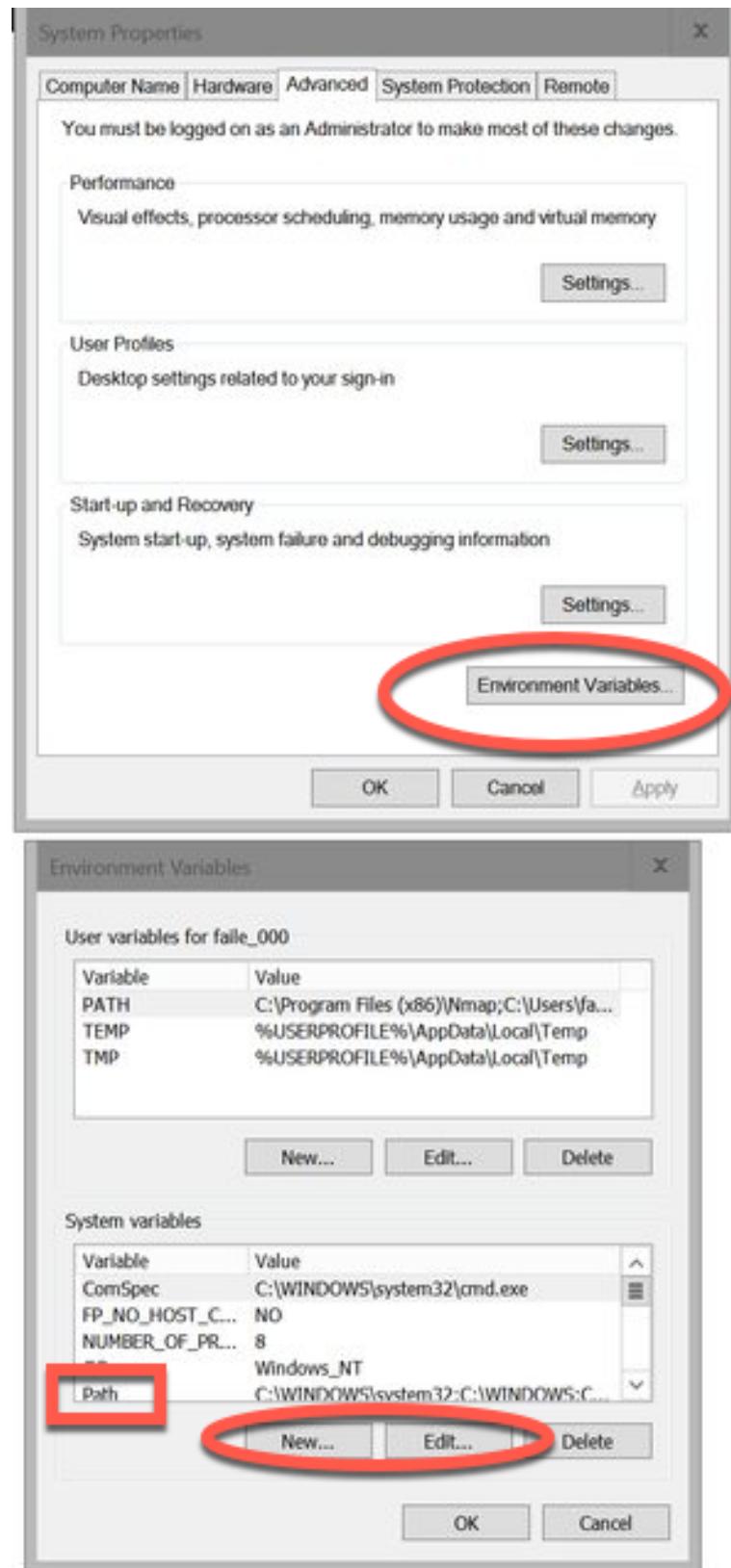


Figure B.2: The Windows Environment Variables editor.  
An Introduction to Symfony 5 ©Matt Smith 2020

### B.3 Check your `php.ini` file

Open a new terminal CLI window (so new settings are loaded) and run `php --ini` to confirm the location of the `php.ini` file that you've just created. Note the following for a Mac - for Windows it should (hopefully) tell you it found the ini file in `c:\php\php.ini`:

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

### B.4 PHP Info & SQL driver test

For database work we need to enable the PDO<sup>1</sup> options for MySQL and SQLite (see later database exercises for how to do this)

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

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

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

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

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

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

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

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

---

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

The figure consists of two screenshots of a web browser displaying the `phpinfo()` page. Both screenshots show the 'PDO' section.

**Top Screenshot:**

PDO support	
PDO drivers	no value

**Bottom Screenshot:**

PDO support	enabled
PDO drivers	mysql, sqlite

A large red arrow points from the top screenshot down to the bottom screenshot, highlighting the change in the PDO drivers configuration.

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

1. In a text editor open file `c:\php\php.ini` and locate the “Dynamic Extensions” section in this file (e.g. use the editor Search feature - or you could just search for `pdo`)
2. Now remove the semi-colon ; comment character at the beginning of the lines for the SQLite and MySQL DLLs to enable them as shown here:

```
;;;;;;;;
; Dynamic Extensions ;
;;;;;;;;
... other lines here ...
extension=php_pdo_mysql <<<<<<< here is the PDO MYSQL driver line
;extension=php_pdo_oci
;extension=php_pdo_odbc
;extension=php_pdo_pgsql
extension=php_pdo_sqlite <<<<<< here is the PDO SQLITE driver line
```

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

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

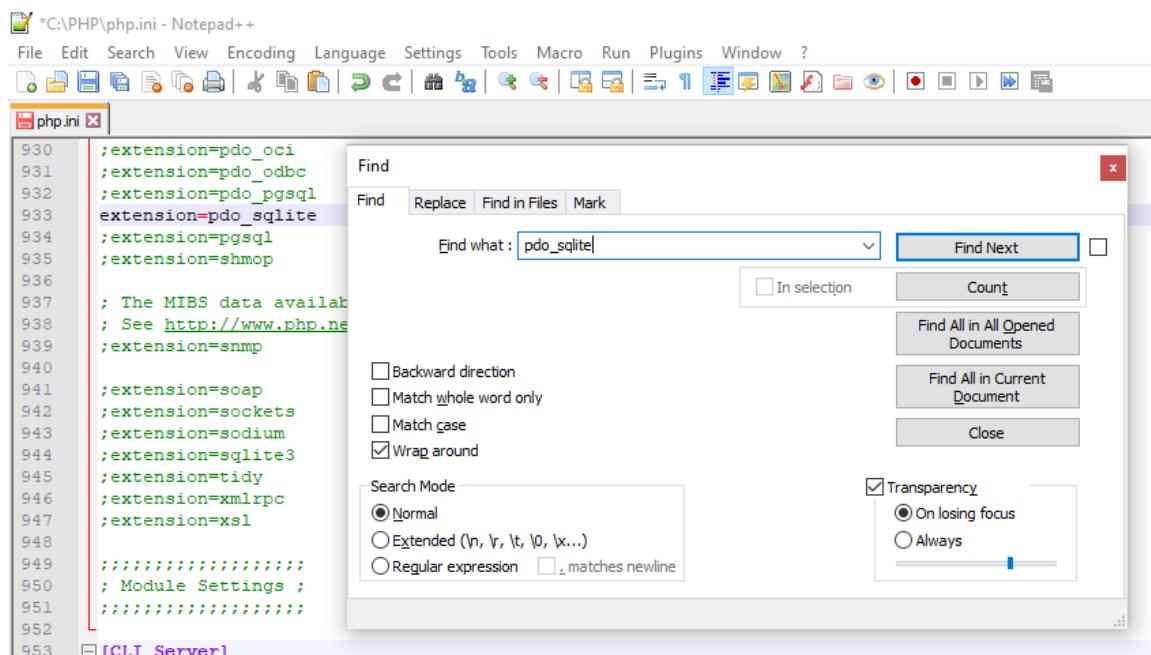


Figure B.4: SQLite being enabled in php.ini in the Notepad++ editor.

# C

## The Composer andd Symfony command line tools

### C.1 Composer

Composer is a **fantastic** PHP tool for managing project dependencies (the libraries and class packages used by OO PHP projects).

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

```
composer self-update
```

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

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

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

```
composer self-update
```

#### C.1.1 Windows Composer install

Get the latest version of Composer from

- [getcomposer.org](http://getcomposer.org)

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

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

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

```
composer self-update
```

### C.1.2 Windows Composer install

Get the latest version of Composer from

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

## C.2 Symfony command line tool

# D

## Software tools

NOTE: All the following should already available on the college computers.

### D.1 PHPStorm editor

Ensure you have your free education Jetbrains licence from:

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

Download and install PHPStorm from:

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

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

- Twig
- Symfony
- Annotations

### D.2 MySQL Workbench

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

Download and install MySQL Workbench from:

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

### D.3 Git

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

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

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

These are common Git commands used in various situations:

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

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

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

>

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

## D.4 Git Windows installation

Visit this page to run the Windows Git installer.

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

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



# E

## The fully-featured Symfony 4 demo

### E.1 Visit Github repo for full Symfony demo

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

### E.2 Git steps for download (clone)

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

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

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

---

<sup>1</sup>For a throw-away exercise like this I just create a directory named `temp` (with `mkdir temp`) and `cd` into that...

```
lab01 $
```

### E.3 Non-git download

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

### E.4 Install dependencies

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

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

### E.5 Run the demo

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

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

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

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

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

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

## E.6 View demo in browser

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

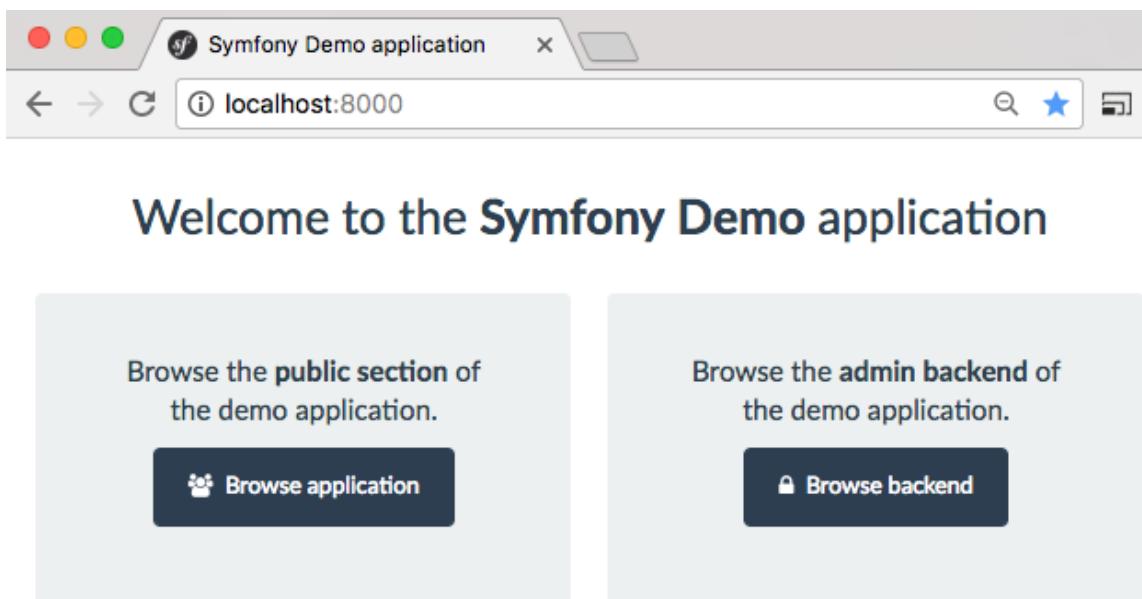


Figure E.1: Default Symfony 4 demo project

## E.7 Explore the code in PHPStorm

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

## E.8 Switch demo from SQLite to MySQL

At present the Symfony demo uses the SQLite driver, working with a database ‘file’ in `/var/data`.

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

Do the following:

1. Run MySQL Workbench

2. Change the **URL** for the projects data in `.env` from:

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

to

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

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

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

```
demo (master) $
```

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

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

```
demo (master) $
```

A migration file has now been created.

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

```
demo (master) $ php bin/console doctrine:migrations:migrate
```

```
Application Migrations
```

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

```
++ migrating 20180127081633
```

```
-> CREATE TABLE symfony_demo_comment (id INT AUTO_INCREMENT NOT NULL, post_id INT NOT NULL)
-> CREATE TABLE symfony_demo_post (id INT AUTO_INCREMENT NOT NULL, author_id INT NOT NULL)
-> CREATE TABLE symfony_demo_post_tag (post_id INT NOT NULL, tag_id INT NOT NULL)
-> CREATE TABLE symfony_demo_tag (id INT AUTO_INCREMENT NOT NULL, name VARCHAR(255) NOT NULL)
-> CREATE TABLE symfony_demo_user (id INT AUTO_INCREMENT NOT NULL, full_name VARCHAR(255) NOT NULL)
```

---

## APPENDIX E. THE FULLY-FEATURED SYMFONY 4 DEMO

```
-> ALTER TABLE symfony_demo_comment ADD CONSTRAINT FK_53AD8F834B89032C FOREIGN KEY (post_id) REFERENCES symfony_demo_post (id)
-> ALTER TABLE symfony_demo_comment ADD CONSTRAINT FK_53AD8F83F675F31B FOREIGN KEY (author_id) REFERENCES symfony_demo_user (id)
-> ALTER TABLE symfony_demo_post ADD CONSTRAINT FK_58A92E65F675F31B FOREIGN KEY (author_id) REFERENCES symfony_demo_user (id)
-> ALTER TABLE symfony_demo_post_tag ADD CONSTRAINT FK_6ABC1CC44B89032C FOREIGN KEY (post_id) REFERENCES symfony_demo_post (id)
-> ALTER TABLE symfony_demo_post_tag ADD CONSTRAINT FK_6ABC1CC4BAD26311 FOREIGN KEY (tag_id) REFERENCES symfony_demo_tag (id)

++ migrated (0.44s)

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

demo (master) $
```

## E.9 Running the tests in the SF4 demo

The project comes with configuration for `simple-phpunit`. Run this once to download the dependencies:

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

lab01/demo $
```

## E.10 Run the tests

Run the tests, by typing `vendor\bin\simple-phpunit`<sup>2</sup>

---

<sup>2</sup>The backlash-forward slash thing is annoying. In a nutshell, for file paths for Windows machines, use backslashes, for everything else use forward slashes. So it's all forward slashes with Linux/Mac machines :-)

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

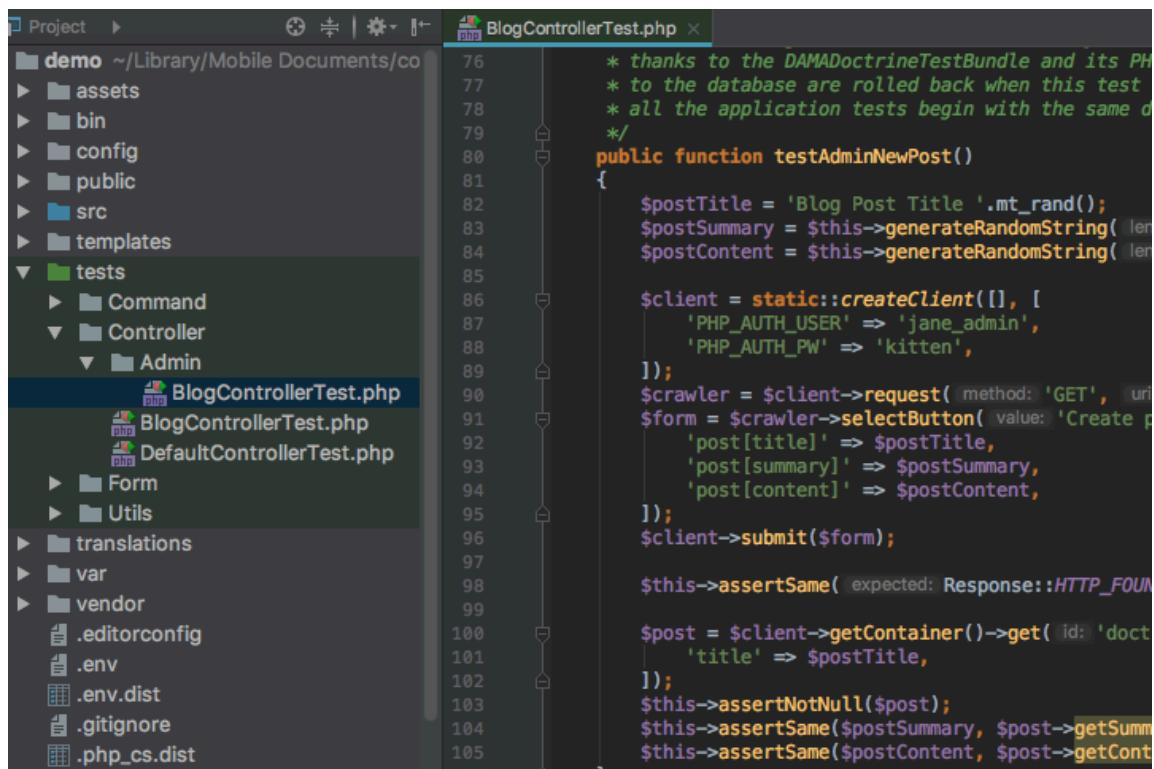
Testing Project Test Suite
.........................
49 / 49 (100%)

Time: 27.65 seconds, Memory: 42.00MB

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

## E.11 Explore directory `/tests`

Look in the `/tests` directory to see how those tests work. For example Figure E.2 shows a screenshot of the admin new post test in PHPStorm.



The screenshot shows the PHPStorm interface with the project structure on the left and the code editor on the right. The project structure includes directories for assets, bin, config, public, src, templates, tests (which contains Command, Controller, and Admin), translations, var, vendor, and various configuration files like .editorconfig, .env, .env.dist, .gitignore, and .php\_cs.dist. The code editor displays the file `BlogControllerTest.php`, which contains a test function `testAdminNewPost()`. The code uses PHPUnit assertions and Symfony's `Client` to perform a POST request to the admin new post endpoint, assert the response status, and check the database for the new post.

```
* thanks to the DAMDoctrineTestBundle and its PH
* to the database are rolled back when this test
* all the application tests begin with the same d
*/
public function testAdminNewPost()
{
    $postTitle = 'Blog Post Title ' . mt_rand();
    $postSummary = $this->generateRandomString( len
    $postContent = $this->generateRandomString( len

    $client = static::createClient([], [
        'PHP_AUTH_USER' => 'jane_admin',
        'PHP_AUTH_PW' => 'kitten',
    ]);
    $crawler = $client->request( method: 'GET', url
    $form = $crawler->selectButton( value: 'Create p
        'post[title]' => $postTitle,
        'post[summary]' => $postSummary,
        'post[content]' => $postContent,
    ]);
    $client->submit($form);

    $this->assertSame( expected: Response::HTTP_FOUN
    $post = $client->getContainer()->get( id: 'doct
        'title' => $postTitle,
    );
    $this->assertNotNull($post);
    $this->assertSame($postSummary, $post->getSumm
    $this->assertSame($postContent, $post->getContent());
}
```

Figure E.2: The admin new post test in PHPStorm

## E.12 Learn more

Learn more about PHPUnit testing and Symfony by visiting:

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



# F

## Solving problems with Symfony

### F.1 No home page loading

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

If Symfony thinks you are in **production** (live public website) then when an error occurs it will throw a 500 server error (which a real production site would catch and display some nicely sanitised message for website visitors).

Since we are in **development** we want to see the **details** of any errors. We set the environment in the `.env` file. For development mode you should see the following in this file:

```
APP_ENV=dev
```

In development you should then get a much more detailed description of the error (including the class / line / template causing the problem etc.).

Also, if you know where your server error logs are stored, you can see the errors written to the log file. Symfony lots are usually create in `/var/log`.

## F.2 “Route not Found” error after adding new controller method

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

## F.3 Issues with timezone

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

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

## F.4 Issues with Symfony 3 and PHPUnit.phar

Symfony 3.2 has issues with PHPUnit (it’s PHPUnit’s fault!). You can solve the problem with the Symphony PHPUnit bridge - which you install via Composer:

```
composer require --dev symfony/phpunit-bridge
```

You then execute your PHPUnit test with the `simple-phpunit` command in `/vendor/bin` as follows:

```
./vendor/bin/simple-phpunit
```

Source:

- [Symfony Blog December 2016](#)

## F.5 PHPUnit installed via Composer

To install PHPUnit with Composer run the following Composer update CLI command:

```
composer require --dev phpunit/phpunit ^6.1
```

To run tests in directory `/tests` execute the following CLI command:

```
./vendor/bin/phpunit tests
```

Source:

- [Stack overflow](#)

As always you can add a shortcut script to your `composer.json` file to save typing, e.g.:

```
"scripts": {  
    "run": "php bin/console server:run",  
    "test": "./vendor/bin/phpunit tests",  
  
    ...  
}
```



# G

Publish via Fortrabbit (PHP as a service)

## G.1 SSH key

Ensure your computer has an SSH key setup, since you'll need this for secure communication with Fortrabbit.

### G.1.1 Windows SSH key setup

A guide to generate SSH keys on Windows can be found at:

- <http://guides.beanstalkapp.com/version-control/git-on-windows.html>

### G.1.2 Mac SSH key setup

A simple guide to generating SSH keys for the Mac can be found at:

- <https://secure.vexxhost.com/billing/knowledgebase/171/How-can-I-generate-SSH-keys-on-Mac-OS-X.html>

### G.1.3 Linux SSH key setup

A guide to generating SSH keys for the Mac can be found at:

- <https://www.ssh.com/ssh/keygen/>

#### G.1.4 Fortrabbit

Do the following:

1. On your Fortrabbit account page click to add a new SSH key. See Figure G.1.

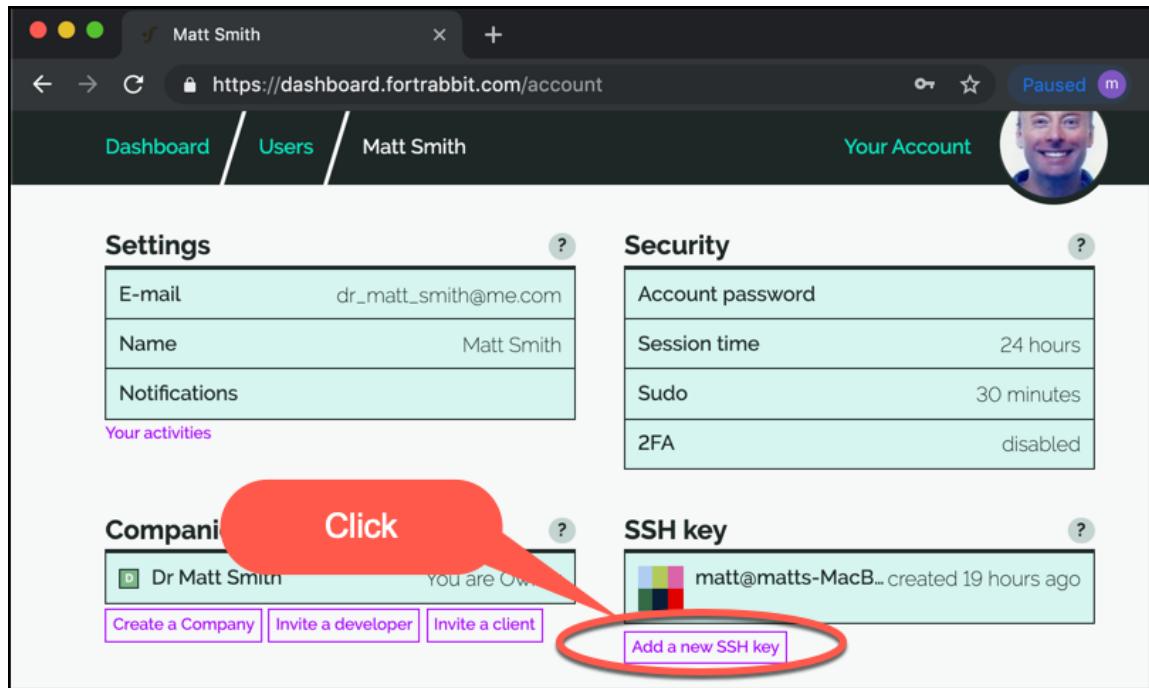


Figure G.1: Screenshot - click to add new SSH key.

2. Paste in SSH key. See Figure G.2.

- or import from your Github account ...

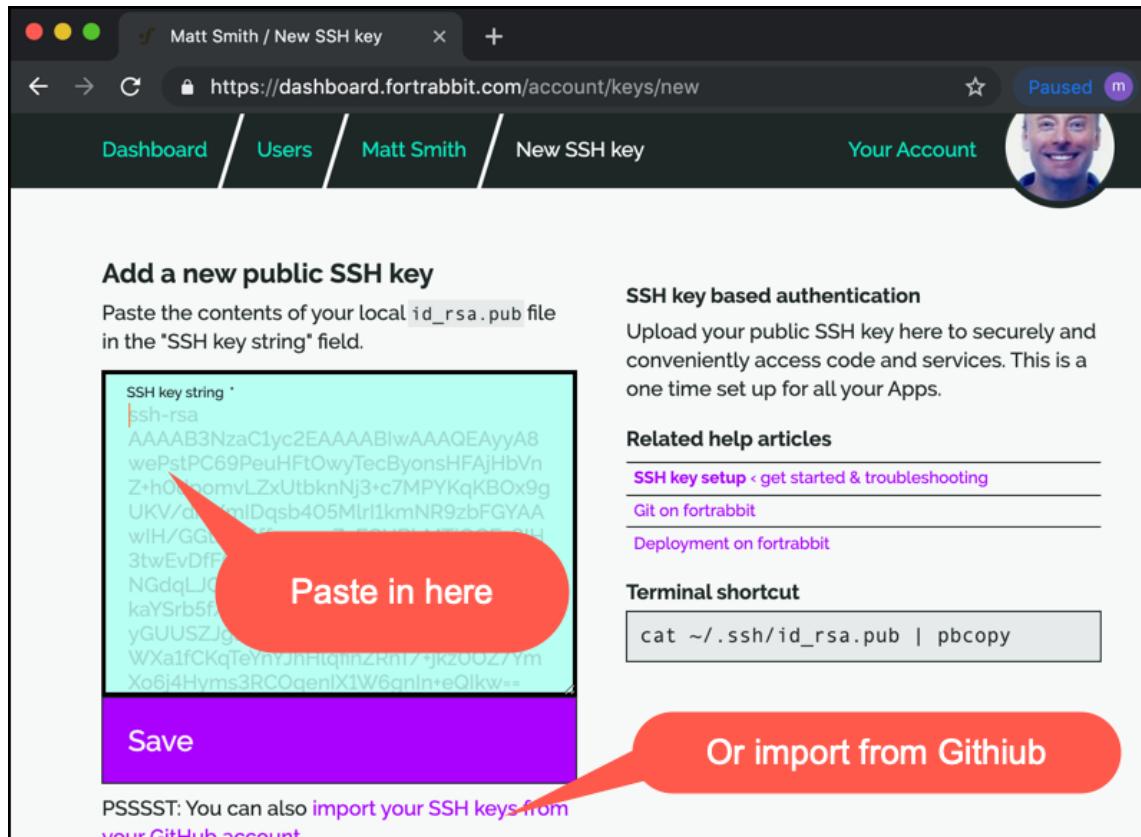


Figure G.2: Screenshot - paste in SSH key.

## G.2 Creating a new web App

Do the following:

1. Go to your account Dashboard, and click to create a new web App. See Figure G.3.

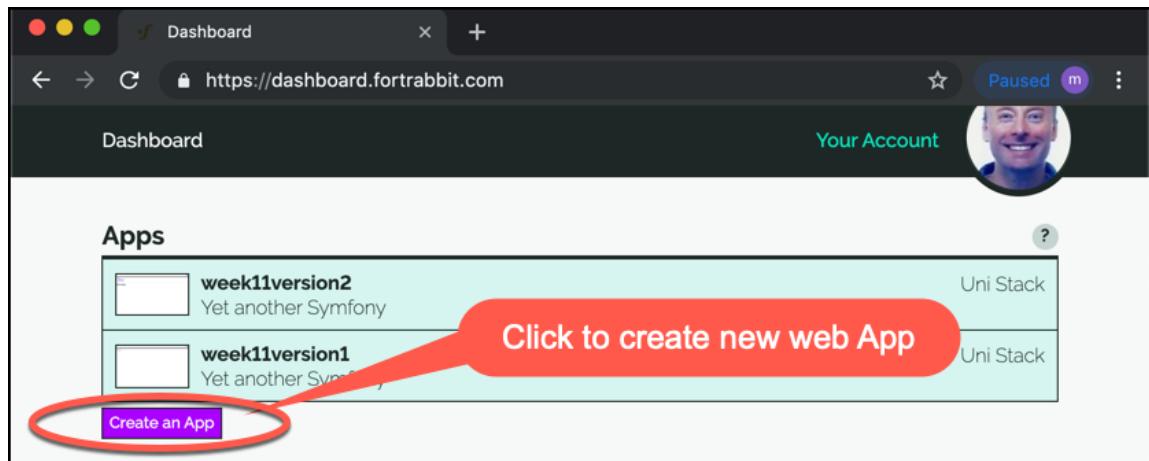


Figure G.3: Screenshot - click to create new app.

2. Enter an app name, e.g. `myproject`. See Figure G.4.

Figure G.4: Screenshot - enter new app name.

3. Choose Symfony project framework from the Choose a sofware page. See Figure G.5.
4. Choose EU (Ireland) Data Center. See Figure G.6.
5. Choose €5 Light Universal Stack. See Figure G.7.

Your new Web App should now have been created!

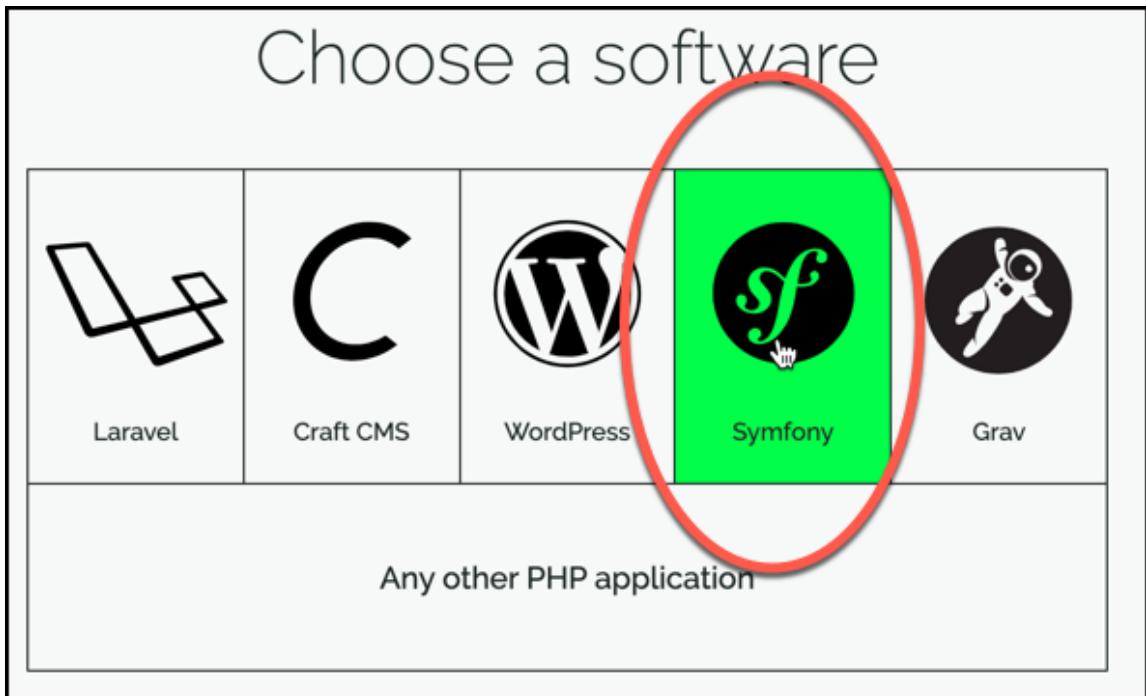


Figure G.5: Screenshot - choose Symfony project type.

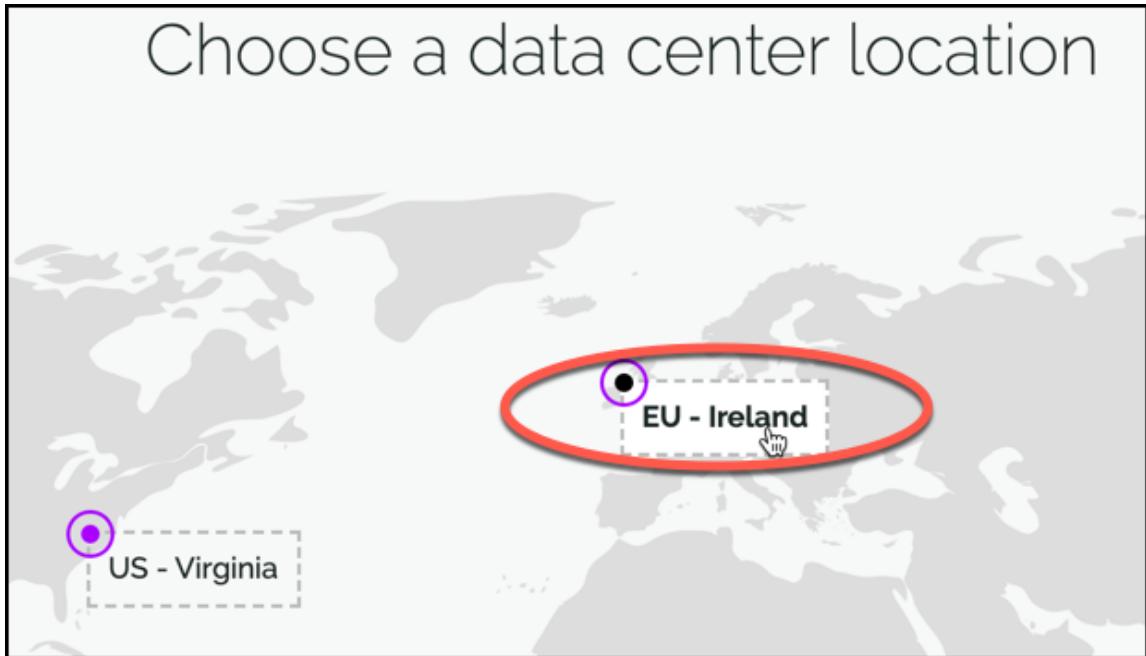


Figure G.6: Screenshot - choose EU (Ireland) Data Center.

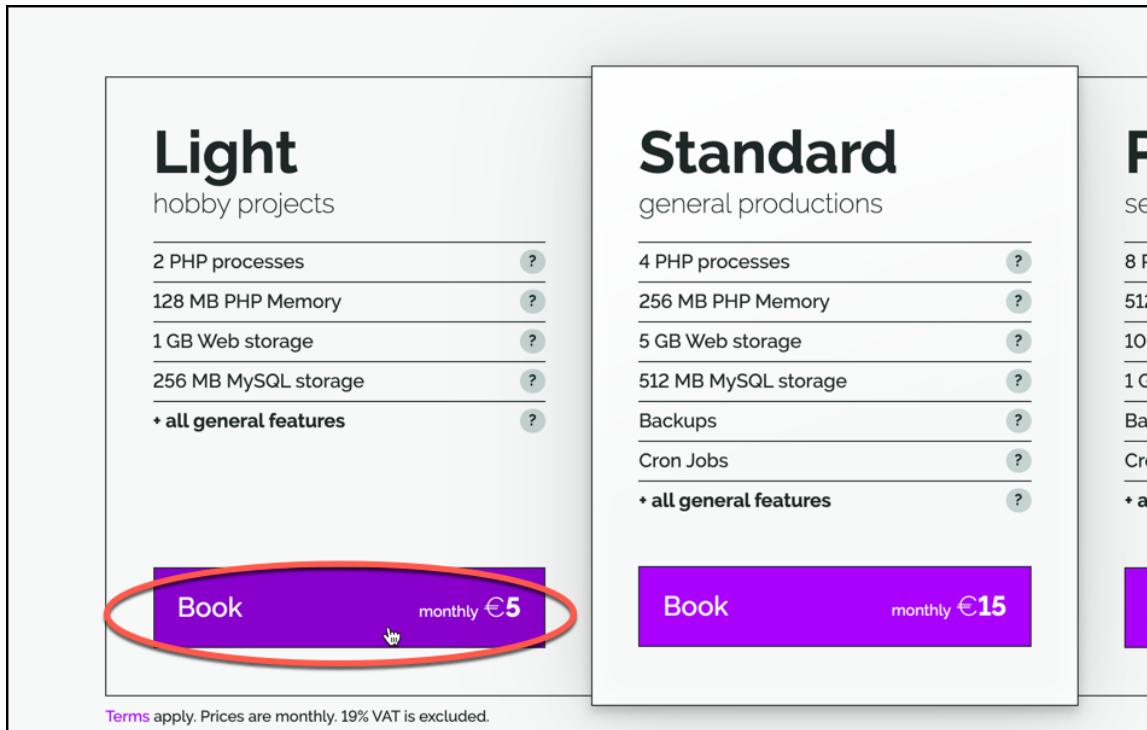


Figure G.7: Screenshot - choose cheapest (Light Universal) Stack .

### G.3 Cloning and populating your Git repo

Fortrabbit has now created a unique `git` repo on its servers. Do the following to make your deploy your existing Symfony project with this repo:

1. Click the `How to deploy git` link. See Figure G.8.
2. You'll now see a page with Git commands, customised to your repo/App name - so you can copy-and-paste them into a CLI Terminal. See Figure G.9.
3. Git `clone` the Fortrabbit repo to your local machine, then `cd` into it (you'll need to use your SSH password ...). See Figure G.10.
4. Copy into your local folder the files for your Symfony project.
  - NOTE: do the following for a clean database setup on your local machine (with migrations ready to use on remote deployment database...)
  - Change the database name in `.env` to a new database
  - delete any exiting `src/Migrations` directory
  - create the database schema with `php bin/console doctrine:database:create`
  - create migrations class with `php bin/console doctrine:migrations:diff`



Figure G.8: Screenshot - click how to deploy git link.

### Simple Git deployment workflow

```
terminal
# 1. Clone the (empty) app to register the remote origin master
$ git clone myproject@deploy.eu2.frbit.com:myproject.git

# 2. Go in the folder
$ cd myproject

# 3. Do stuff
$ echo 'php echo "PHPower to the PHPeople";' &gt;index.php

# 4. Initialize Git locally
$ git add index.php
$ git commit -am 'Initial commit'

# 5. Set upstream and 1st push
$ git push -u origin master
# long output
# After that it already works

# 6. Every deploy from now on
$ git push</pre
```

Figure G.9: Screenshot - Git customized command page.

```
[matts-MacBook-Pro-2:rabbit01 matt$ git clone myproject@deploy.eu2.frbit.com:myproject.git
Cloning into 'myproject'...
[Enter passphrase for key '/Users/matt/.ssh/id_rsa':
warning: You appear to have cloned an empty repository.
[matts-MacBook-Pro-2:rabbit01 matt$ cd myproject/
[matts-MacBook-Pro-2:myproject matt$ ls -al
total 0
drwxr-xr-x  3 matt  staff   96  2 Apr 21:47 .
drwxr-xr-x  3 matt  staff   96  2 Apr 21:47 ..
drwxr-xr-x 10 matt  staff  320  2 Apr 21:47 .git
matts-MacBook-Pro-2:myproject matt$ ]
```

Figure G.10: Screenshot - clone repo to local machine.

- execute the migrations to create the table schema with `php bin/console doctrine:migrations:migrate`
- load the fixtures fixtures into the local database with `php bin/console doctrine:fixtures:load`
- test the fixtures with an **SQL** command, e.g. to list users from table `user` you could execute: `php bin/console doctrine:query:sql "select * from user"`

## G.4 Fixing the Fixtures issue

For first-time database setup of a deployed project you'll usually wish to create the fixtures in the database. However, the default setting of the **environment** for a Fortrabbit project is **production**. See Figure G.11.

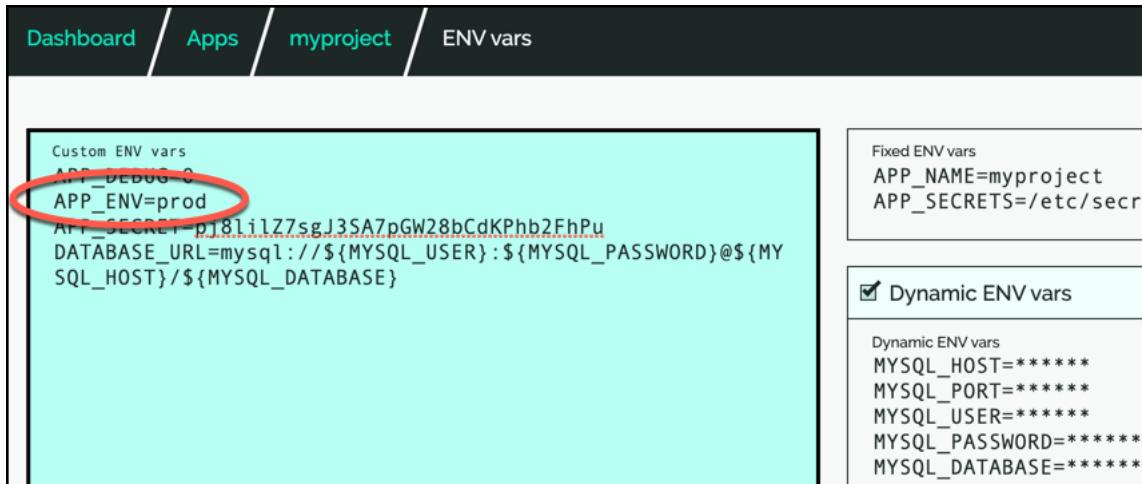


Figure G.11: Screenshot - Fortrabbit PROD environment setting.

However, the settings for when the Doctrine Fixtures ‘bundle’ should be included are only for development and test environments. So if we want to be able to run Fixture insertion into the

database, then we have to add a statement that Fixtures should be available for all environments.

Add this line to file `/config/bundles.php`, before the end of the array ]:

```
Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle::class => ['all' => true]
```

So your listing for `/config/bundles.php` should now looks something like this:

```
<?php

return [
    Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
    Doctrine\Bundle\DoctrineCacheBundle\DoctrineCacheBundle::class => ['all' => true],
    ... etc. lots more packages listed ...
    Symfony\Bundle\SecurityBundle\SecurityBundle::class => ['all' => true],
    Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle::class => ['all' => true]
];
```

## G.5 Fixing the Apache .htaccess issue

Fortrabbit servers PHP projects using the Apache Open Source web server. For the URL patterns to be correct parsed it needs special URL-rewrite rules in a file `.htaccess` in the directory `//public`. To create this file you can simply use Composer to require the dedicated `symfony/apache-pack` package:

```
composer req symfony/apache-pack
```

NOTE: You may be asked to say `yes` to install this package since it's been created by the community (and isn't an offical package at the time of writing ...)

## G.6 Adding, committing and pushing the project files to the repo

We are now ready to upload our production-ready project files to the Fortrabbit repo.

1. Add the new/changed files for staging with `git add ..`

```
matt$ git status
```

```
On branch master
```

```
No commits yet
```

Untracked files:

(use "git add <file>..." to include in what will be committed)

```
.env
.gitignore
README.md
bin/
composer.json
composer.lock
config/
public/
src/
symfony.lock
templates/
```

nothing added to commit but untracked files present (use "git add" to track)

matt\$ git add .

2. Commit with first commit message with `git commit -m "<msg>"`:

```
matt$ git commit -m "first commit"
[master (root-commit) 77946cc] first commit
59 files changed, 6819 insertions(+)
create mode 100755 .env
create mode 100755 .gitignore
create mode 100755 README.md
... etc. for lots of files
create mode 100755 templates/security/success.html.twig
create mode 100755 templates/student/index.html.twig
```

3. Push the committed files to the Fortrabbit repo:

```
matt$ git push -u origin master
Enter passphrase for key '/Users/matt/.ssh/id_rsa':
Counting objects: 80, done.
Delta compression using up to 12 threads.
Compressing objects: 100% (75/75), done.
Writing objects: 100% (80/80), 38.49 KiB | 3.21 MiB/s, done.
Total 80 (delta 7), reused 0 (delta 0)
```

Commit received, starting build of branch master

4. Automatically (since we have a new commit to the `master` branch) a new build on the For-

---

## APPENDIX G. PUBLISH VIA FORTRABBIT (PHP AS A SERVICE)

trabbit server will be triggered:

```
----- *f -----  
B U I L D  
  
Checksum:  
77946cc9c25fc7259e93080e89de48fc635c8a1e  
  
Composer:  
---  
Loading composer repositories with package information  
Installing dependencies (including require-dev) from lock file  
Package operations: 80 installs, 0 updates, 0 removals  
- Installing ocreamius/package-versions (1.4.0): Downloading (100%)  
- Installing symfony/flex (v1.2.0): Downloading (100%)  
  
Prefetching 78 packages  
- Downloading (100%)  
  
- Installing symfony/polyfill-mbstring (v1.10.0): Loading from cache  
... etc. Composer will install lots of files ...  
- Installing symfony/process (v4.2.3): Loading from cache  
- Installing symfony/web-server-bundle (v4.2.3): Loading from cache  
Generating autoload files  
  
ocreamius/package-versions: Generating version class...  
ocreamius/package-versions: ...done generating version class  
  
Executing script cache:clear [OK]  
Executing script assets:install public [OK]  
  
---  
6s 268ms  
  
R E L E A S E  
  
Size:  
6.7 MB  
  
Uploading:  
207ms
```

```
Build & release done in 7s 51ms, now queued for final distribution.
```

```
----- *f -----
```

Note, from now on, we can simply use `git push` for any later committed changes to our source code.

## G.7 SHH CLI Terminal to migrate and install DB fixtures

The final step is to migrate the database (using the migrations from our source code), and load any fixtures we require. We need to do this via an SSH secure terminal connection to Fortrabbit.

Do the following:

1. Use SSH to connect to the Fortrabit virtual Linux machine. See Figure G.12.

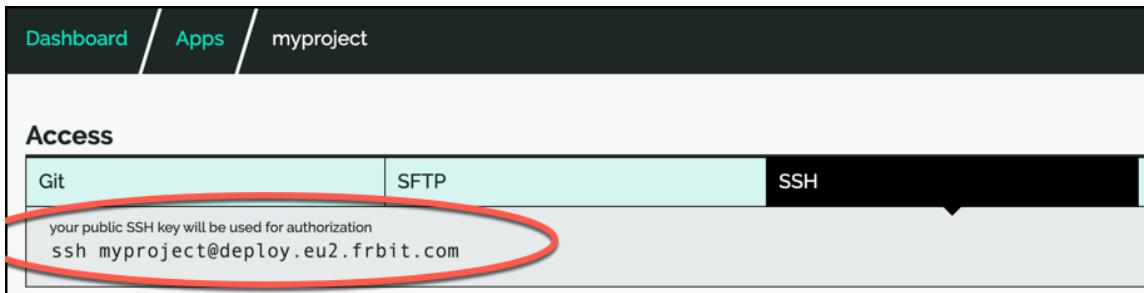


Figure G.12: Screenshot - SSH connect to remote Fortrabit system.

2. Migrate the database schema with ‘doctrine:migrations:migrate’ (and say ‘yes’ when asked). See Figure G.13.
3. Load the fixtures with `doctrine:fixtures:load` (and say ‘yes’ when asked). See Figure G.14.
4. Test the DB contents by listing users via SQL with `doctrine:query:sql "select * from user"`. See Figure G.15.

```
matts-MacBook-Pro-2:myproject matt$ ssh myproject@deploy.eu2.frbit.com
Enter passphrase for key '/Users/matt/.ssh/id_rsa': _____
_____.f_____
[myproject:~$ php bin/console doctrine:migrations:migrate
Application Migrations
[WARNING! You are about to execute a database migration that could result in schema changes and data loss. Are you sure you wish to continue? (y/n)y
Migrating up to 20190402101552 from 0
++ migrating 20190402101552
--> CREATE TABLE user (id INT AUTO_INCREMENT NOT NULL, username VARCHAR(180) NOT NULL, roles JSON NOT NULL, password VARCHAR(255) NOT NULL, UNIQUE INDEX UNIQ_8D93D649F85E0677 (username), PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci ENGINE = InnoDB
++ migrated (took 35.3ms, used 12M memory)
_____
++ finished in 37.8ms
++ used 12M memory
++ 1 migrations executed
++ 1 sql queries
```

Figure G.13: Screenshot - migrate the DB schema.

```
[myproject:~$ php bin/console doctrine:fixtures:load
Careful, database "myproject" will be purged. Do you want to continue? (yes/no) [no]:
[ > y
> purging database
> loading App\DataFixtures\AppFixtures
> loading App\DataFixtures\UserFixtures
```

Figure G.14: Screenshot - load the fixtures into the DB.

```
myproject:~$ php bin/console doctrine:query:sql "select * from user"
array(3) {
    [0]=>
    array(4) {
        ["id"]=>
        string(1) "1"
        ["username"]=>
        string(4) "user"
        ["roles"]=>
        string(13) "["ROLE_USER"]"
        ["password"]=>
        string(60) "$2y$13$4B2R/C1z1gmB0kYPWAg0FeXnOcyiKrE4I6q7UdIacI/JeNt0dXaZ2"
    }
    [1]=>
    array(4) {
        ["id"]=>
        string(1) "2"
        ["username"]=>
        string(5) "admin"
        ["roles"]=>
        string(14) "["ROLE_ADMIN"]"
        ["password"]=>
        string(60) "$2y$13$/u/ImIB4jbSZEfgtZhGy3.8nJq95dGGnBjulPCph10vA8P/Dgk9Zm"
    }
    [2]=>
    array(4) {
        ["id"]=>
        string(1) "3"
        ["username"]=>
        string(4) "matt"
        ["roles"]=>
        string(20) "["ROLE_SUPER_ADMIN"]"
        ["password"]=>
        string(60) "$2y$13$DgmE3aM98CSmR69VK0ClWe.YOY/AsxFUfZ6Ub7eAJyQXjrSVZI7J."
    }
}
```

Figure G.15: Screenshot - testing fixtures by selecting all users via SQL query.

## G.8 Symfony project should now be fully deployed

Your Symfony project should now be fully deployed and working, with DB fixtures and secure logins etc. See Figure G.16.

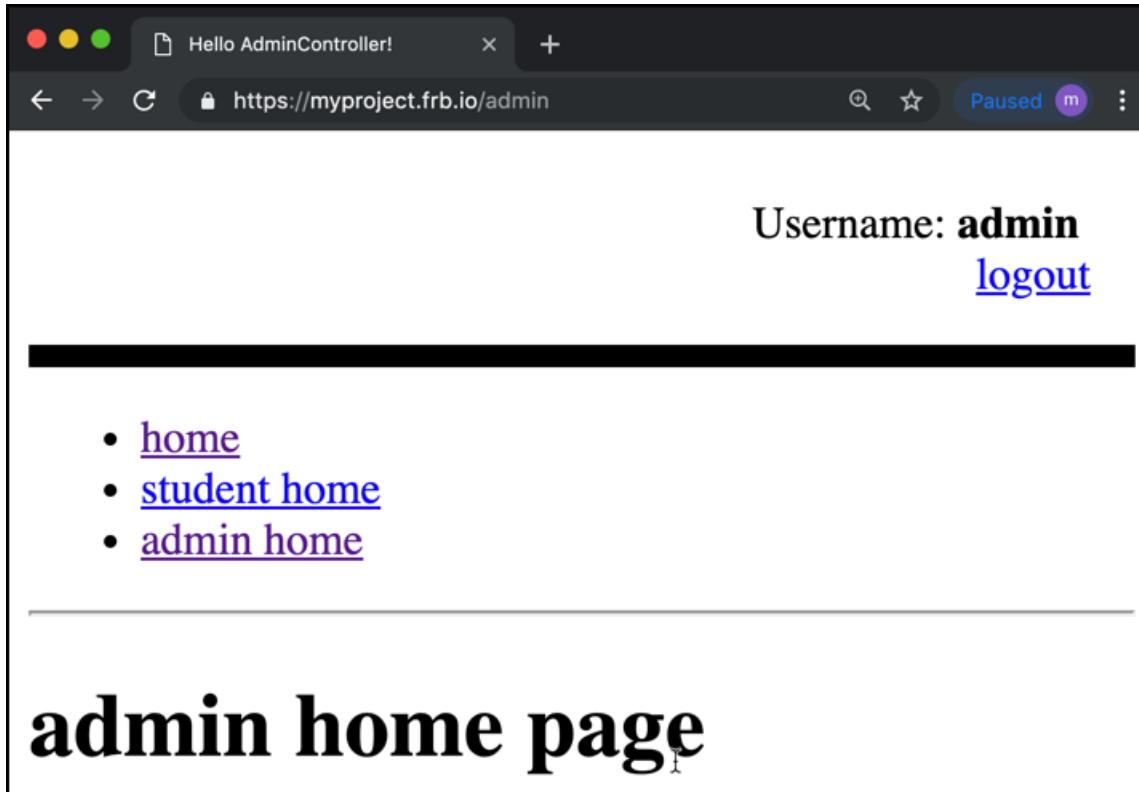


Figure G.16: Screenshot - deployed working Symfony project



# H

Quick setup for new ‘blog’ project

## H.1 Create a new project, e.g. ‘blog’

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

```
$ symfony new blog
```

Or use Composer:

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

Read more at:

- [Symfony create project reference](#)

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

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

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

## H.3 Add run shortcut to your `Composer.json` scripts

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

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

## H.4 Change directories and run the app

Change to new project directory and run the app

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

Now visit: [http://localhost:8000/app\\_dev.php](http://localhost:8000/app_dev.php) in your browser to see the welcome page

## H.5 Remove default content

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

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

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

# I

Steps to download code and get website up and  
running

## I.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

### I.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`

### I.1.2 Getting code from a Git repository

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`

## I.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`

## I.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/`

# J

## About `parameters.yml` and `config.yml`

### J.1 Project (and deployment) specific settings in (`/app/config/parameters.yml`)

Usually the project-specific settings are declared in this file:

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

These parameters are referred to in the more generic `/app/config/config.yml`.

For example the host of a MySQL database for the project would be defined by the following variable in `parameters.yml`:

```
parameters:  
    database_host: 127.0.0.1
```

Note that this file (`parameters.yml`) is included 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 accidentally publicly expose your database credentials on an open source Github page :-)

If there isn't already a `parameters.yml` file, then you can copy the `parameters.yml.dist` file and edit it as appropriate.

## J.2 More general project configuration (`/app/config/config.yml`)

The file `/app/config/config.yml` is actually the one used by Symfony when it looks up project settings. So the `config.yml` file uses references to the variables declared in the `/app/config/parameters.yml` file. For example the following lines in `config.yml` make a reference to the variable `database_path` that is declared in `parameters.yml`:

```
doctrine:  
    dbal:  
        driver:   pdo_mysql  
        host:     "%database_host%"
```

For many projects we need to make **no changes** to the contents of `config.yml`. Although, since Symfony is setup with defaults for a MySQL database, if we are using SQLite, for example then we do need to change the configuration settings, as well as declaring appropriate variables in `parameters.yml`. This is discussed in Appendix , describing how to set up a Symfony project to work with SQLite.

# K

## Setting up for MySQL Database

### K.1 Declaring the parameters for the database (`parameters.yml`)

Usually the project-specific settings are declared in this file:

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

These parameters are referred to in the more generic `/app/config/config.yml` - which for MySQL projects we don't need to touch.

The simplest way to connect your Symfony application to a MySQL database is by setting the following variables in `parameters.yml` (located in `(/app/config/)`):

```
# This file is auto-generated during the composer install
parameters:
    database_host: 127.0.0.1
    database_port: null
    database_name: symfony_book
    database_user: root
    database_password: null
```

Note, you can learn more about `parameters.yml` and `config.yml` in Appendix J.

You can replace `127.0.0.1` with `localhost` if you wish. If your code cannot connect to the database check the 'port' that your MySQL server is running at (usually 3306 but may be different, for example my Mac MAMP server uses 8889 for MySQL for some reason). So my parameters look

like this:

```
parameters:  
    database_host:      127.0.0.1  
    database_port:      8889  
    database_name:      symfony_book  
    database_user:      symfony  
    database_password:  pass
```

We can now use the Symfony CLI to **generate** the new database for us. You've guessed it, we type:

```
$ php bin/console doctrine:database:create
```

You should now see a new database in your DB manager. Figure K.1 shows our new `symfony_book` database created for us.

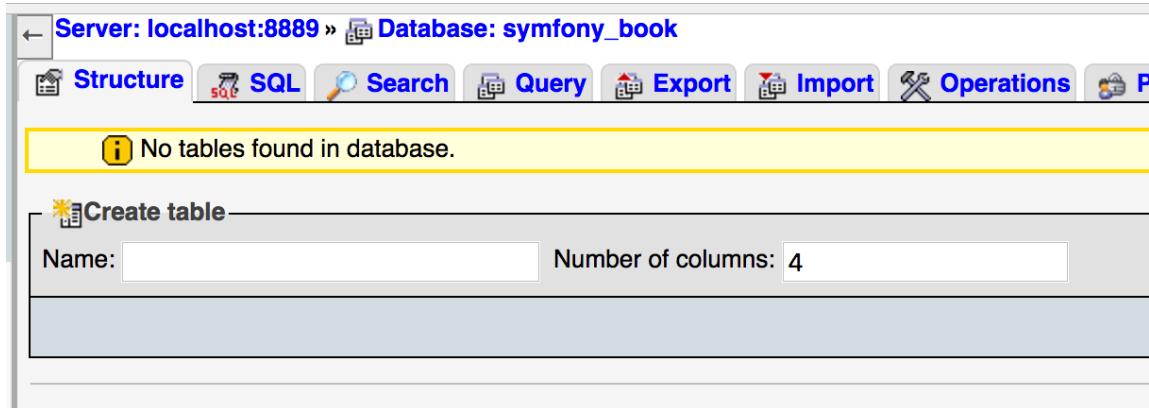


Figure K.1: CLI created database in PHPMyAdmin.

**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 ...

# L

## Setting up for SQLite Database

### L.1 NOTE regarding FIXTURES

If you are using the Doctrine Fixtures Bundle, install that first **before** changing parameters and config for SQLite. The fixtures bundle assumes MySQL, and will overwrite some of the parameters during installation.

If that does happen, you'll just have to repeat the steps in this Appendix to set things back to SQLite after fixtures installation.

### L.2 SQLite suitable for most small-medium websites

For small/medium projects, and learning frameworks like Symfony, it's often simplest to just use a file-based SQLite database.

Learn more about SQLite at the project's website, and their discussion of when SQLite is a good choice, and when a networked DBMS like MySQL is more appropriate:

- [SQLite website](#)
- [Appropriate Uses For SQLite](#)

### L.3 Create directory where SQLite database will be stored

Setting one up with Symfony is **very** easy. These steps assume you are gong to use an SQLite database file named `data.sqlite` located in directory `/var/data`.

Our first step to configuring a Symfony project to work with SQLite is to ensure the directory exists where the SQLite file is to be created. The usual location for Symfony projects is `/var/data`. So create directory `data` in `/var` if it doesn't already exist in your project.

### L.4 Declaring the parameters for the database (`parameters.yml`)

In `/app/parameters.yml` replace the default `database_host/name/user/password` parameters with a single parameter `database_path` as follows:

```
```yaml
parameters:
    database_path: ../var/data/data.sqlite
    mailer_transport: smtp
    mailer_host: 127.0.0.1
    etc.
````
```

### L.5 Setting project configuraetion to work with the SQLite database driver and path (`/app/config/config.yml`)

In `/app/config.yml` change the `doctrine` settings **from** these MySQL defaults:

```
```yaml
# Doctrine Configuration
doctrine:
    dbal:
        driver:   pdo_mysql
        host:     "%database_host%"
        port:     "%database_port%"
        dbname:  "%database_name%"
        user:    "%database_user%"
        password: "%database_password%"
        charset:  UTF8
````
```

**to** these SQLite settings:

```
```yaml
# Doctrine Configuration
doctrine:
    dbal:
        driver:    pdo_sqlite
        path:      "%kernel.root_dir%/%database_path%"
```

```

That's it! You can now tell Symfony to create your database with CLI command:

```
php bin/console doctrine:database:create
```

You'll now have an SQLite database file at `/var/data/data.sqlite`. You can even use the PHPStorm to open and read the DB for you. See Figures L.1 and L.2.

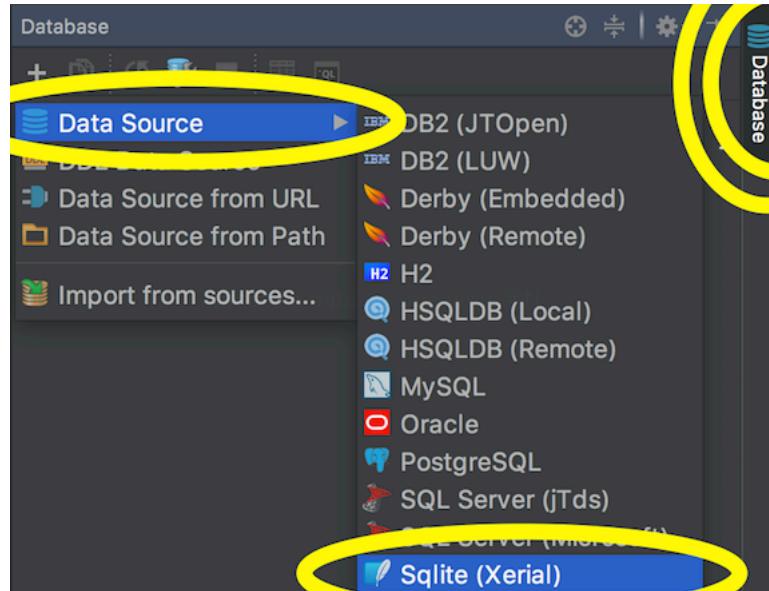


Figure L.1: Open SQLite view in PHPMyAdmin.

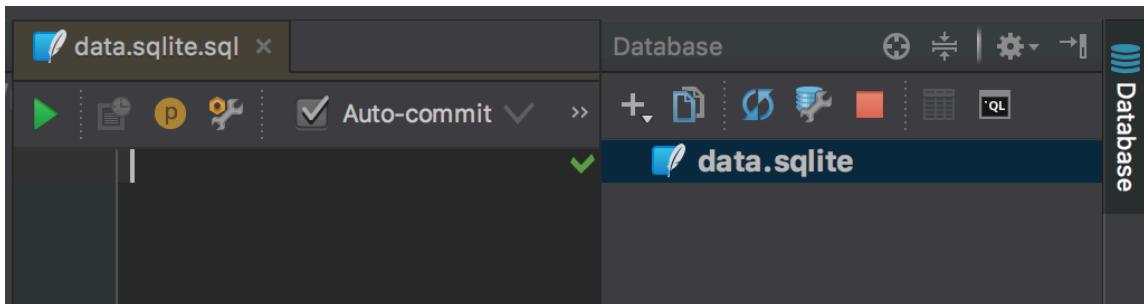


Figure L.2: Viewing /var/data.sqlite in PHPStorm.



# M

## Setting up Adminer for DB GUI interaction

### M.1 Adminer - small and simple DB GUI

Adminer is a lightweight PHP, web-based GUI for DB interaction. It supports both MySQL and SQLite.

Figure M.1 shows **Adminer** listing 2 user records created from Symfony Doctrine fixtures.

The screenshot shows the Adminer web interface. At the top, there's a navigation bar with links: 'SQLite 3' > 'Server' > '/Users/matt/Desktop/kill/symfony/product1/var/data/data.sqlite' > 'Select: app\_users'. Below this is a light blue header bar with the text 'Select: app\_users'. Underneath is a white content area. At the top of the content area, there are several buttons: 'Select data', 'Show structure', 'Alter table', and 'New item'. Below these are four input fields: 'Select' (checkbox), 'Search' (text input), 'Sort' (checkbox), and 'Limit' (checkbox). The 'Limit' checkbox is checked, with '50' in the input field and '100' in the 'Text length' field next to it. A 'Select' button is also present. Below this is a SQL query: 'SELECT \*, rowid FROM "app\_users" LIMIT 50 (0.000 s) Edit'. At the bottom is a table with two rows of data:

|   | id | username | password  | email                           | is_active |
|---|----|----------|---|---------------------------------|-----------|
| <input type="checkbox"/> <a href="#">Modify</a> | 1  | admin    | \$2y\$13\$tnz0hyFSj3a0IcFOujDjWO5yuOJ8I2/EjhVVonfjGOXT4vnigWnpi | <a href="#">admin@admin.com</a> | 1         |
| <input type="checkbox"/> <a href="#">edit</a>   | 2  | matt     | \$2y\$13\$mZcJwp.S5P47TS.SSayeQOuPFqhZAeTUTqotY2MQPUtAn6oeh0BYW | <a href="#">matt@matt.com</a>   | 1         |

Figure M.1: Using CLI to load database fixtures.

## M.2 Getting Adminer

Download Adminer from the project website. I recommend you get the English only version - it's smaller...

- [Adminer.org website](#)

## M.3 Setting up

Extract the file to a suitable location. For example you could create an '/adminer' directory in your current project:

```
.../project/adminer/adminer.php
```

To keep things simple, and also to remove the login requirement for SQLite access, create file `index.php` in your Adminer directory, containing the following:

```
<?php
// index.php

function adminer_object()
{
    class AdminerSoftware extends Adminer
    {
        function login($login, $password) {
            return true;
        }
    }
    return new AdminerSoftware;
}

include __DIR__ . "/adminer.php";
```

## M.4 Running Adminer

Since we have an `index.php` page, we just need to run a web server pointing its root to our Adminer directory. Perhaps the simplest way to do this is with the built-in PHP server, e.g.:

```
php -S localhost:3306 -t ./adminer
```

To save typing, you could add a script alias to your `composer.json` file:

```
"adminer":"php -S localhost:3306 -t adminer",
```

## APPENDIX M. SETTING UP ADMINER FOR DB GUI INTERACTION

When run, choose the appropriate DBMS from the dropdown menu (e.g. **SQLite 3**), and enter the required credentials. For SQLite all we need to enter is the path to the location of the SQLite database file, e.g.:

```
/Users/matt/Desktop/kill/symfony/product1/var/data/data.sqlite
```



# N

## 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
```



# O

## 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 these 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 entities 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.
```

## APPENDIX O. TRANSCRIPT OF INTERACTIVE ENTITY GENERATION

---

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 :).

\$

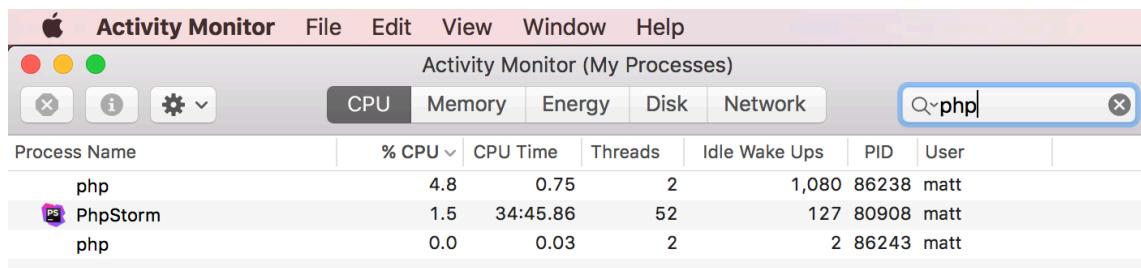
# P

## 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!



The screenshot shows the Mac OS X Activity Monitor application. The title bar reads "Activity Monitor". A search bar at the top right contains the text "Q~php". The main window displays a table of processes. The columns are: Process Name, % CPU, CPU Time, Threads, Idle Wake Ups, PID, and User. There are three entries in the table:

| Process Name | % CPU | CPU Time | Threads | Idle Wake Ups | PID   | User |
|--------------|-------|----------|---------|---------------|-------|------|
| php          | 4.8   | 0.75     | 2       | 1,080         | 86238 | matt |
| PhpStorm     | 1.5   | 34:45.86 | 52      | 127           | 80908 | matt |
| php          | 0.0   | 0.03     | 2       | 2             | 86243 | matt |

Figure P.1: Mac running php process.





## Docker and Symfony projects

### Q.1 Setup

Start with your Symfony project directory

### Q.2 Dockerfile

Create a file `Dockerfile`, containing the steps to build a Docker Image of your virtual computer system.

This `Dockerfile` assumes your Symofny project code is in directory “admin-prototype”:

```
FROM php:7.1.7-apache

COPY admin-prototype /var/www

## Expose apache.
EXPOSE 80

## Copy this repo into place. - if /www/site is referred to in Apache conf file ...
#ADD admin-prototype/web /var/www/site

## Update the default apache site with the config we created.
```

```
ADD apache-config.conf /etc/apache2/sites-enabled/000-default.conf

##### fix Symfony var Cache issue #####
# source: http://symfony.com/doc/current/setup/file_permissions.html
CMD HTTPDUSER=$(ps axo user,comm | grep -E '[a]pache|[h]ttpd|[_]www|[w]ww-data|[n]ginx' | grep -v 'grep')
CMD setfacl -dR -m u:"$HTTPDUSER":rwX -m u:$whoami:rwX /var/www/var
CMD setfacl -R -m u:"$HTTPDUSER":rwX -m u:$whoami:rwX /var/www/var

## Run symfony server
CMD php /var/www/bin/console server:run 0.0.0.0:80&
```

### Q.3 Build your Docker image

Build your Docker image:

```
$ docker build -t my-application .
```

### Q.4 Run a Container process based on your image (exposing HTTP port 80)

Now run your Docker **Image** as a Docker **Container** process. The `-p 80:80` option is to expose port 80 in the container as port 80 on your main computer system, so you can visit the web site via your web browser at `http://localhost`.

```
docker run -it -p 80:80 my-application
```

### Q.5 Alternative Dockerfile for a basic PHP application, using Apache

This Dockerfile assumes the PHP project files are in directory “game1”:

```
FROM php:7.1.7-apache

COPY game1 /var/www

## Expose apache.
EXPOSE 80

## Copy this repo into place. - if /www/site is referred to in Apache conf file ...
```

```
#ADD game1/web /var/www/site

## Update the default apache site with the config we created.
ADD apache-config.conf /etc/apache2/sites-enabled/000-default.conf

## By default start up apache in the foreground, override with /bin/bash for interative.
CMD /usr/sbin/apache2ctl -D FOREGROUND
```

## Q.6 Create config file for Apache

Create a file `apache-config.conf`, containing the following:

```
<VirtualHost *:80>
    ServerAdmin me@mydomain.com
    DocumentRoot /var/www/web

    <Directory /var/www/web/>
        Options Indexes FollowSymLinks MultiViews
        AllowOverride All
        Order deny,allow
        Allow from all
    </Directory>

    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined

</VirtualHost>
```

## Q.7 Other Docker reference stuff

### Q.7.1 Docker Images

List images on disk

```
docker images
```

See the details of an image:

```
docker history php:7.1-cli
```

(hint: `php:7.1-cli` = repository name : tag)

### Q.7.2 Containers

List currently running containers (processes):

```
docker ps
```

Run an image as a container process:

```
docker run -it repository:tag /bin/bash
```

Note:

- the “-it” means go into an INTERACTIVE TERMINAL
- the “/bin/bash” is the command to run - i.e. run our BASH shell

Kill a process:

```
docker kill NAME
```

E.g. if container name was `wonderful_wozniak` then you'd type:

```
docker kill wonderful_wozniak
```

### Q.7.3 New Image from current (changed) state of a running Container

To Save an updated filesystem in Container to a new Image do the following:

```
docker commit -m "comments" containerName
```

E.g. if container name was `nifty_hodgkin` and you'd installed, say, git and composer, then write:

```
$ docker commit -m "installed git and composer" nifty_hodgkin
sha256:7e555cc0df651a1b68593733a35cdac341175bed294084eb73b7fb23ebdc5bbd
$
```

Note that the SHA is output, the ID of the new Image.

You can then add a **tag** to the new image.

First look at the images, and note its short Image Id:

| \$ docker images |        |              |            |        |
|------------------|--------|--------------|------------|--------|
| REPOSITORY       | TAG    | IMAGE ID     | CREATED    | SIZE   |
| <none>           | <none> | 7e555cc0df65 | 5 days ago | 433 MB |
| phpd             | latest | d0ee7be93033 | 5 days ago | 372 MB |

Now give a TAG to our image, e.g. `php_composer_git`:

```
$ docker tag 7e555cc0df65 php_composer_git
```

Now we see our nicely tagged Image:

| \$ docker images |        |              |            |        |
|------------------|--------|--------------|------------|--------|
| REPOSITORY       | TAG    | IMAGE ID     | CREATED    | SIZE   |
| php_composer_git | latest | 7e555cc0df65 | 5 days ago | 433 MB |
| phpd             | latest | d0ee7be93033 | 5 days ago | 372 MB |

#### Q.7.4 Exposing HTTP ports for Containers running web application servers

We can use the option `-p PORT:PORT` to expose a port from the Container to our main computer system.

E.g. To expose Container port 80 as port 80 on our computer we add `-p 80:80`, as part of our `docker run` command:

```
$ docker run -it -p 80:80 php_composer_git /bin/bash
```

We can **Inspect** the details of a running Container with the `docker inspect` command:

```
docker inspect wonderful_wozniak
```

### Q.8 Useful reference sites

Some useful sites for Docker and PHP include:

- (Good overview)[<http://odewahn.github.io/docker-jumpstart/docker-images.html>]
- (Web Server Docker - with note about Mac IP)[<https://writing.pupius.co.uk/apache-and-php-on-docker-44faef716150>]
- (Nice intro for PHP)[<https://semaphoreci.com/community/tutorials/dockerizing-a-php-application>]

From the offical Docker documentation pages:

- (Introduction)[<https://docs.docker.com/get-started/#conclusion>]
- (Download Docker)[<https://www.docker.com/community-edition#/download>]



# R

## xDebug for Windows

### R.1 Steps for Windows

To setup xDebug for Windows you need:

1. to download the appropriate DLL for your PHP system into C:\php\ext (or elsewhere if you installed PHP somewhere else on your system)
2. add/uncomment the following line at the end of your `php.ini` file:

```
zend_extensions = C:\php\ext\php_xdebug-2-6-0-7.1-vc14-x86_64.dll
```

NOTE: The location / name of this file will depend on your PHP installation (see Wizard steps below)

### R.2 Steps for Linux/Mac

You can quickly confirm xDebug status with the following CLI command:

```
$ php -ini |grep 'xdebug support'  
xdebug support => enabled
```

If you see ‘enabled’ then no further work is needed. Otherwise, the simplest way to get xDebug working is to use the wizard ...

### R.3 Use the xDebug wizard!

Perhaps the easiest way to setup xDebug is to follow the steps recommended by their ‘wizard’ at:

- xDebug Windows wizard: <https://xdebug.org/wizard.php>

Figure ?? shows a screenshot of the xDebug wizard web page output.

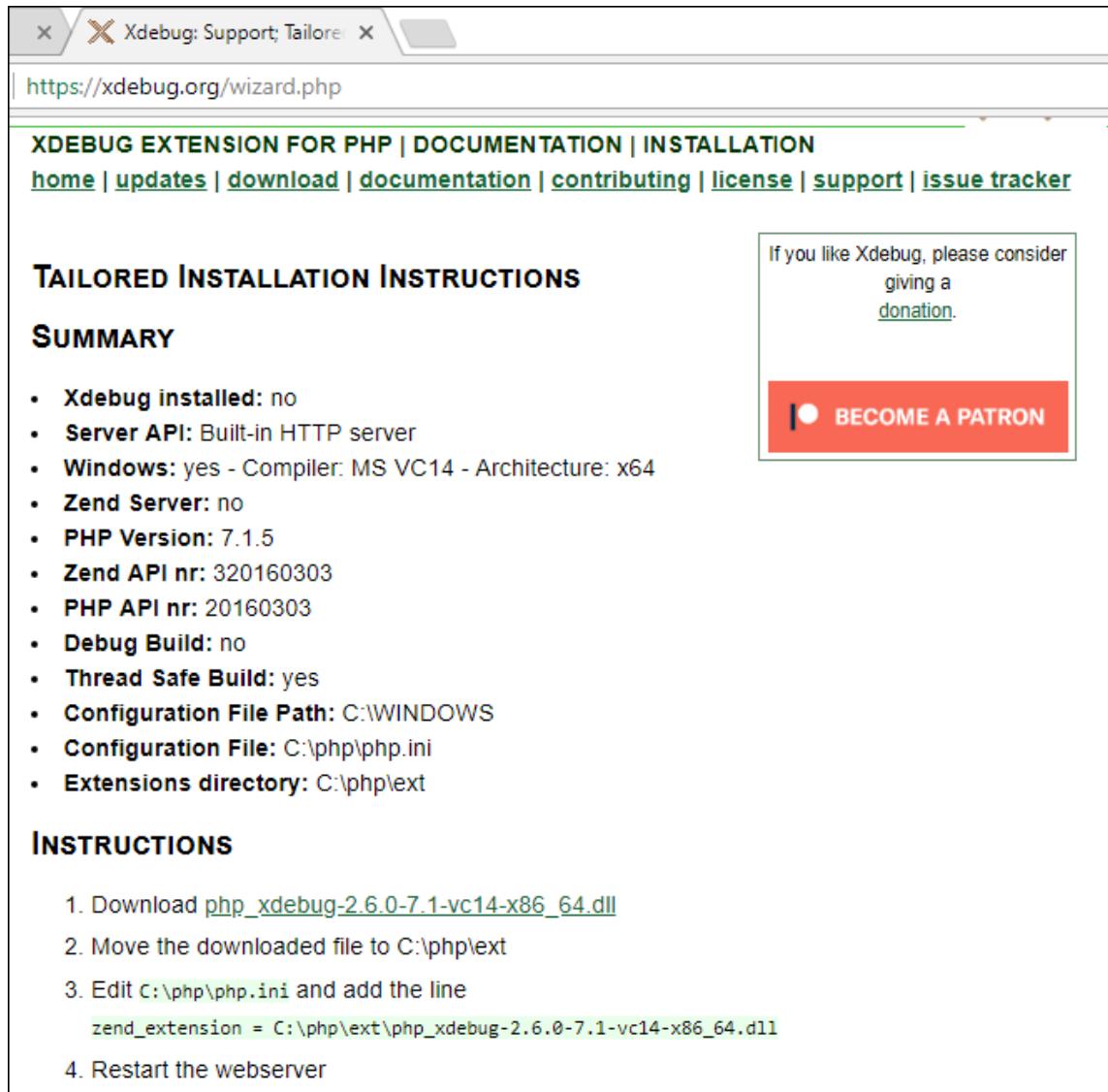


Figure R.1: Screenshot xDebug wizard output.

## R.4 PHP Function `phpinfo()`

The `phpinfo()` output is a summary (as an HTML page) of your PHP setup. Figure R.2 shows a screenshot of a browser showing a PHP info page.

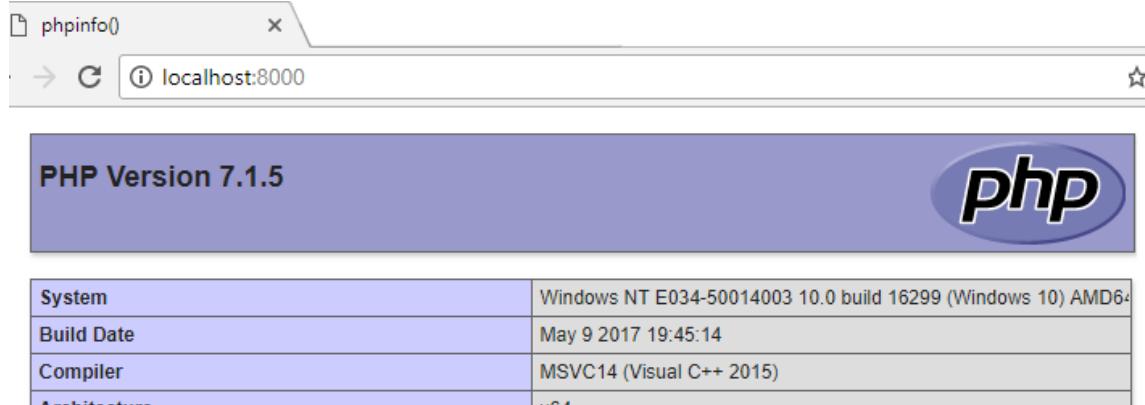


Figure R.2: Screenshot `phpinfo` in browser.

To use the ‘wizard’ you need to generate, copy and then paste the text output of `print phpinfo()` into the web page form.

To get the output from `phpinfo()` you can do one of these:

- at the CLI type ``php -r 'print phpinfo();' > info.html``  
and then get the contents of file `info.html`
- create a temporary directory, containing PHP file `index.php` that contains

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

run your webserver and visit the directory. Then copy and paste the contents of your browser window

- in Symfony you could create a temporary controller method that outputs a Reponse containing the outut of `phpinfo()`, e.g.

```
/**  
 * @Route("/info")  
 */  
public function infoAction()  
{  
    return new Response( phpinfo() );  
}
```

## R.5 More information

For more information follow the steps at:

- [xDebug Windows wizard](#)
- [xDebug project home page](#)

## List of References