



Symfony

The Cookbook

Version: 3.0

generated on July 28, 2016

The Cookbook (3.0)

This work is licensed under the “Attribution-Share Alike 3.0 Unported” license (<http://creativecommons.org/licenses/by-sa/3.0/>).

You are free **to share** (to copy, distribute and transmit the work), and **to remix** (to adapt the work) under the following conditions:

- **Attribution:** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Share Alike:** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license. For any reuse or distribution, you must make clear to others the license terms of this work.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor SensioLabs shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

If you find typos or errors, feel free to report them by creating a ticket on the Symfony ticketing system (<http://github.com/symfony/symfony-docs/issues>). Based on tickets and users feedback, this book is continuously updated.

Contents at a Glance

How to Use Assetic for Asset Management	7
Combining, Compiling and Minimizing Web Assets with PHP Libraries.....	14
How to Minify CSS/JS Files (Using UglifyJS and UglifyCSS)	17
How to Minify JavaScripts and Stylesheets with YUI Compressor.....	21
How to Use Assetic for Image Optimization with Twig Functions	23
How to Apply an Assetic Filter to a specific File Extension	26
How to Install 3rd Party Bundles	28
Best Practices for Reusable Bundles	31
How to Use Bundle Inheritance to Override Parts of a Bundle	37
How to Override any Part of a Bundle	39
How to Remove the AcmeDemoBundle	42
How to Load Service Configuration inside a Bundle	45
How to Create Friendly Configuration for a Bundle	48
How to Simplify Configuration of multiple Bundles	54
How to Use Varnish to Speed up my Website	56
Caching Pages that Contain CSRF Protected Forms	60
Installing Composer	61
How to Master and Create new Environments	62
Building your own Framework with the MicroKernelTrait.....	67
How to Override Symfony's default Directory Structure	72
Using Parameters within a Dependency Injection Class	75
Understanding how the Front Controller, Kernel and Environments Work together.....	77
How to Set external Parameters in the Service Container	80
How to Use the Apache Router	82
Configuring a Web Server	83
How to Organize Configuration Files	89
How to Create a Console Command	93
How to Use the Console.....	97
How to Style a Console Command	98
How to Call a Command from a Controller	105
How to Generate URLs from the Console	107
How to Enable Logging in Console Commands	109
How to Define Commands as Services	113
How to Customize Error Pages.....	115
How to Define Controllers as Services	120
How to Upload Files	125

How to Optimize your Development Environment for Debugging	131
How to Deploy a Symfony Application	133
Deploying to Microsoft Azure Website Cloud	137
Deploying to Heroku Cloud	150
Deploying to Platform.sh.....	155
Deploying to forttrabbit	159
How to use Doctrine Extensions: Timestampable, Sluggable, Translatable, etc.	163
How to Register Event Listeners and Subscribers	164
How to Use Doctrine DBAL	167
How to Generate Entities from an Existing Database.....	169
How to Work with multiple Entity Managers and Connections	172
How to Register custom DQL Functions.....	175
How to Define Relationships with Abstract Classes and Interfaces.....	176
How to Provide Model Classes for several Doctrine Implementations	179
How to Implement a Simple Registration Form.....	182
How to Use PdoSessionHandler to Store Sessions in the Database	188
How to Use MongoDBSessionHandler to Store Sessions in a MongoDB Database.....	191
Console Commands.....	193
How to Send an Email.....	194
How to Use Gmail to Send Emails	197
How to Use the Cloud to Send Emails	199
How to Work with Emails during Development.....	201
How to Spool Emails.....	204
How to Test that an Email is Sent in a Functional Test.....	206
How to Create Event Listeners and Subscribers.....	208
How to Set Up Before and After Filters	212
How to Extend a Class without Using Inheritance.....	216
How to Customize a Method Behavior without Using Inheritance	219
How to use Expressions in Security, Routing, Services, and Validation	221
How to Customize Form Rendering	224
How to Use Data Transformers	237
How to Dynamically Modify Forms Using Form Events	245
How to Embed a Collection of Forms	255
How to Create a Custom Form Field Type.....	267
How to Create a Form Type Extension	272
How to Reduce Code Duplication with "inherit_data"	277
How to Unit Test your Forms.....	280
How to Configure empty Data for a Form Class.....	285
How to Use the submit() Function to Handle Form Submissions.....	287
How to Use the virtual Form Field Option	289
Using Bower with Symfony	290
How to Install or Upgrade to the Latest, Unreleased Symfony Version	293
How to Use Monolog to Write Logs	295
How to Configure Monolog to Email Errors	300
How to Configure Monolog to Display Console Messages.....	302
How to Configure Monolog to Exclude 404 Errors from the Log	304
How to Log Messages to different Files.....	305

How to Create a custom Data Collector.....	307
How to Use Matchers to Enable the Profiler Conditionally.....	311
Switching the Profiler Storage.....	313
How to Access Profiling Data Programmatically.....	314
The PSR-7 Bridge.....	316
How to Configure Symfony to Work behind a Load Balancer or a Reverse Proxy.....	318
How to Register a new Request Format and Mime Type.....	320
How to Force Routes to always Use HTTPS or HTTP.....	321
How to Allow a "/" Character in a Route Parameter.....	322
How to Configure a Redirect without a custom Controller.....	323
How to Use HTTP Methods beyond GET and POST in Routes.....	325
How to Use Service Container Parameters in your Routes.....	327
How to Create a custom Route Loader.....	329
Redirect URLs with a Trailing Slash.....	333
How to Pass Extra Information from a Route to a Controller.....	335
Looking up Routes from a Database: Symfony CMF DynamicRouter.....	336
How to Build a Traditional Login Form.....	337
Authenticating against an LDAP server.....	342
How to Load Security Users from the Database (the Entity Provider).....	346
How to Create a Custom Authentication System with Guard.....	353
How to Add "Remember Me" Login Functionality.....	360
How to Impersonate a User.....	364
How to Customize your Form Login.....	367
How to Create a custom User Provider.....	370
How to Create a Custom Form Password Authenticator.....	375
How to Authenticate Users with API Keys.....	379
How to Create a custom Authentication Provider.....	388
Using pre Authenticated Security Firewalls.....	397
How to Change the default Target Path Behavior.....	399
Using CSRF Protection in the Login Form.....	401
How to Choose the Password Encoder Algorithm Dynamically.....	403
How to Use multiple User Providers.....	405
How to Use Multiple Guard Authenticators.....	407
How to Restrict Firewalls to a Specific Request.....	409
How to Restrict Firewalls to a Specific Host.....	411
How to Create and Enable Custom User Checkers.....	412
How to Use Voters to Check User Permissions.....	414
How to Use Access Control Lists (ACLs).....	419
How to Use advanced ACL Concepts.....	423
How to Force HTTPS or HTTP for different URLs.....	427
How to Secure any Service or Method in your Application.....	428
How Does the Security access_control Work?.....	431
How to Use the Serializer.....	435
How to Define Non Shared Services.....	438
How to Work with Compiler Passes in Bundles.....	439
Session Proxy Examples.....	440
Making the Locale "Sticky" during a User's Session.....	442

Configuring the Directory where Session Files are Saved	445
Bridge a legacy Application with Symfony Sessions	447
Limit Session Metadata Writes	448
Avoid Starting Sessions for Anonymous Users.....	449
How to Inject Variables into all Templates (i.e. global Variables)	450
How to Use and Register Namespaced Twig Paths	452
How to Use PHP instead of Twig for Templates.....	454
How to Write a custom Twig Extension	459
How to Render a Template without a custom Controller.....	461
How to Simulate HTTP Authentication in a Functional Test	463
How to Simulate Authentication with a Token in a Functional Test.....	464
How to Test the Interaction of several Clients	466
How to Use the Profiler in a Functional Test.....	467
How to Test Code that Interacts with the Database.....	469
How to Test Doctrine Repositories	472
How to Customize the Bootstrap Process before Running Tests.....	474
Upgrading a Patch Version (e.g. 2.6.0 to 2.6.1)	475
Upgrading a Minor Version (e.g. 2.5.3 to 2.6.1)	476
Upgrading a Major Version (e.g. 2.7.0 to 3.0.0).....	478
Upgrading a Third-Party Bundle for a Major Symfony Version	482
How to Create a custom Validation Constraint	486
How to Handle Different Error Levels.....	490
How to Dynamically Configure Validation Groups	492
How to Use PHP's built-in Web Server	494
How to Create a SOAP Web Service in a Symfony Controller	497
How to Create and Store a Symfony Project in Git	500
How to Create and Store a Symfony Project in Subversion.....	503
Using Symfony with Homestead/Vagrant.....	507



Chapter 1

How to Use Assetic for Asset Management

Installing and Enabling Assetic

Starting from Symfony 2.8, Assetic is no longer included by default in the Symfony Standard Edition. Before using any of its features, install the AsseticBundle executing this console command in your project:

Listing 1-1 1 `$ composer require symfony/assetic-bundle`

Then, enable the bundle in the `AppKernel.php` file of your Symfony application:

Listing 1-2

```
1  // app/AppKernel.php
2
3  // ...
4  class AppKernel extends Kernel
5  {
6      // ...
7
8      public function registerBundles()
9      {
10         $bundles = array(
11             // ...
12             new Symfony\Bundle\AsseticBundle\AsseticBundle(),
13         );
14
15         // ...
16     }
17 }
```

Finally, add the following minimal configuration to enable Assetic support in your application:

Listing 1-3

```
1  # app/config/config.yml
2  assetic:
3      debug:          '%kernel.debug%'
4      use_controller: '%kernel.debug%'
5      filters:
6          cssrewrite: ~
7
8  # ...
```

Introducing Assetic

Assetic combines two major ideas: assets and filters. The assets are files such as CSS, JavaScript and image files. The filters are things that can be applied to these files before they are served to the browser. This allows a separation between the asset files stored in the application and the files actually presented to the user.

Without Assetic, you just serve the files that are stored in the application directly:

Listing 1-4 1 `<script src="{{ asset('js/script.js') }}"></script>`

But *with* Assetic, you can manipulate these assets however you want (or load them from anywhere) before serving them. This means you can:

- Minify and combine all of your CSS and JS files
- Run all (or just some) of your CSS or JS files through some sort of compiler, such as LESS, SASS or CoffeeScript
- Run image optimizations on your images

Assets

Using Assetic provides many advantages over directly serving the files. The files do not need to be stored where they are served from and can be drawn from various sources such as from within a bundle.

You can use Assetic to process CSS stylesheets, JavaScript files and images. The philosophy behind adding either is basically the same, but with a slightly different syntax.

Including JavaScript Files

To include JavaScript files, use the **javascripts** tag in any template:

Listing 1-5 1 `{% javascripts '@AppBundle/Resources/public/js/*' %}`
2 `<script src="{{ asset_url }}"></script>`
3 `{% endjavascripts %}`



If your application templates use the default block names from the Symfony Standard Edition, the **javascripts** tag will most commonly live in the **javascripts** block:

Listing 1-6 1 `{# ... #}`
2 `{% block javascripts %}`
3 `{% javascripts '@AppBundle/Resources/public/js/*' %}`
4 `<script src="{{ asset_url }}"></script>`
5 `{% endjavascripts %}`
6 `{% endblock %}`
7 `{# ... #}`



You can also include CSS stylesheets: see Including CSS Stylesheets.

In this example, all files in the **Resources/public/js/** directory of the AppBundle will be loaded and served from a different location. The actual rendered tag might simply look like:

Listing 1-7 1 `<script src="/app_dev.php/js/abcd123.js"></script>`

This is a key point: once you let Assetic handle your assets, the files are served from a different location. This *will* cause problems with CSS files that reference images by their relative path. See [Fixing CSS Paths](#) with the `cssrewrite` Filter.

Including CSS Stylesheets

To bring in CSS stylesheets, you can use the same technique explained above, except with the `stylesheets` tag:

Listing 1-8

```
1 {% stylesheets 'bundles/app/css/*' filter='cssrewrite' %}
2 <link rel="stylesheet" href="{{ asset_url }}" />
3 {% endstylesheets %}
```



If your application templates use the default block names from the Symfony Standard Edition, the `stylesheets` tag will most commonly live in the `stylesheets` block:

Listing 1-9

```
1 {# ... #}
2 {% block stylesheets %}
3     {% stylesheets 'bundles/app/css/*' filter='cssrewrite' %}
4         <link rel="stylesheet" href="{{ asset_url }}" />
5     {% endstylesheets %}
6 {% endblock %}
7 {# ... #}
```

But because Assetic changes the paths to your assets, this *will* break any background images (or other paths) that uses relative paths, unless you use the `cssrewrite` filter.



Notice that in the original example that included JavaScript files, you referred to the files using a path like `@AppBundle/Resources/public/file.js`, but that in this example, you referred to the CSS files using their actual, publicly-accessible path: `bundles/app/css`. You can use either, except that there is a known issue that causes the `cssrewrite` filter to fail when using the `@AppBundle` syntax for CSS stylesheets.

Including Images

To include an image you can use the `image` tag.

Listing 1-10

```
1 {% image '@AppBundle/Resources/public/images/example.jpg' %}
2 
3 {% endimage %}
```

You can also use Assetic for image optimization. More information in *How to Use Assetic for Image Optimization with Twig Functions*.



Instead of using Assetic to include images, you may consider using the *LiipImagineBundle*¹ community bundle, which allows to compress and manipulate images (rotate, resize, watermark, etc.) before serving them.

1. <https://github.com/liip/LiipImagineBundle>

Fixing CSS Paths with the `cssrewrite` Filter

Since Assetic generates new URLs for your assets, any relative paths inside your CSS files will break. To fix this, make sure to use the `cssrewrite` filter with your `stylesheets` tag. This parses your CSS files and corrects the paths internally to reflect the new location.

You can see an example in the previous section.



When using the `cssrewrite` filter, don't refer to your CSS files using the `@AppBundle` syntax. See the note in the above section for details.

Combining Assets

One feature of Assetic is that it will combine many files into one. This helps to reduce the number of HTTP requests, which is great for front-end performance. It also allows you to maintain the files more easily by splitting them into manageable parts. This can help with re-usability as you can easily split project-specific files from those which can be used in other applications, but still serve them as a single file:

Listing 1-11

```
1 {% javascripts
2     '@AppBundle/Resources/public/js/*'
3     '@AcmeBarBundle/Resources/public/js/form.js'
4     '@AcmeBarBundle/Resources/public/js/calendar.js' %}
5 <script src="{{ asset_url }}"></script>
6 {% endjavascripts %}
```

In the **dev** environment, each file is still served individually, so that you can debug problems more easily. However, in the **prod** environment (or more specifically, when the **debug** flag is **false**), this will be rendered as a single **script** tag, which contains the contents of all of the JavaScript files.



If you're new to Assetic and try to use your application in the **prod** environment (by using the **app.php** controller), you'll likely see that all of your CSS and JS breaks. Don't worry! This is on purpose. For details on using Assetic in the **prod** environment, see [Dumping Asset Files](#).

And combining files doesn't only apply to *your* files. You can also use Assetic to combine third party assets, such as jQuery, with your own into a single file:

Listing 1-12

```
1 {% javascripts
2     '@AppBundle/Resources/public/js/thirdparty/jquery.js'
3     '@AppBundle/Resources/public/js/*' %}
4 <script src="{{ asset_url }}"></script>
5 {% endjavascripts %}
```

Using Named Assets

AsseticBundle configuration directives allow you to define named asset sets. You can do so by defining the input files, filters and output files in your configuration under the **assetic** section. Read more in the [assetic config reference](#).

Listing 1-13

```
1 # app/config/config.yml
2 assetic:
3     assets:
4         jquery_and_ui:
5             inputs:
```

```

6         - '@AppBundle/Resources/public/js/thirdparty/jquery.js'
7         - '@AppBundle/Resources/public/js/thirdparty/jquery.ui.js'

```

After you have defined the named assets, you can reference them in your templates with the `@named_asset` notation:

Listing 1-14

```

1  {% javascripts
2      '@jquery_and_ui'
3      '@AppBundle/Resources/public/js/*' %}
4      <script src="{{ asset_url }}"></script>
5  {% endjavascripts %}

```

Filters

Once they're managed by Assetic, you can apply filters to your assets before they are served. This includes filters that compress the output of your assets for smaller file sizes (and better frontend optimization). Other filters can compile CoffeeScript files to JavaScript and process SASS into CSS. In fact, Assetic has a long list of available filters.

Many of the filters do not do the work directly, but use existing third-party libraries to do the heavy-lifting. This means that you'll often need to install a third-party library to use a filter. The great advantage of using Assetic to invoke these libraries (as opposed to using them directly) is that instead of having to run them manually after you work on the files, Assetic will take care of this for you and remove this step altogether from your development and deployment processes.

To use a filter, you first need to specify it in the Assetic configuration. Adding a filter here doesn't mean it's being used - it just means that it's available to use (you'll use the filter below).

For example to use the UglifyJS JavaScript minifier the following configuration should be defined:

Listing 1-15

```

1  # app/config/config.yml
2  assetic:
3      filters:
4          uglifyjs2:
5              bin: /usr/local/bin/uglifyjs

```

Now, to actually *use* the filter on a group of JavaScript files, add it into your template:

Listing 1-16

```

1  {% javascripts '@AppBundle/Resources/public/js/*' filter='uglifyjs2' %}
2      <script src="{{ asset_url }}"></script>
3  {% endjavascripts %}

```

A more detailed guide about configuring and using Assetic filters as well as details of Assetic's debug mode can be found in *How to Minify CSS/JS Files (Using UglifyJS and UglifyCSS)*.

Controlling the URL Used

If you wish to, you can control the URLs that Assetic produces. This is done from the template and is relative to the public document root:

Listing 1-17

```

1  {% javascripts '@AppBundle/Resources/public/js/*' output='js/compiled/main.js' %}
2      <script src="{{ asset_url }}"></script>
3  {% endjavascripts %}

```



Symfony also contains a method for cache *busting*, where the final URL generated by Assetic contains a query parameter that can be incremented via configuration on each deployment. For more information, see the version configuration option.

Dumping Asset Files

In the **dev** environment, Assetic generates paths to CSS and JavaScript files that don't physically exist on your computer. But they render nonetheless because an internal Symfony controller opens the files and serves back the content (after running any filters).

This kind of dynamic serving of processed assets is great because it means that you can immediately see the new state of any asset files you change. It's also bad, because it can be quite slow. If you're using a lot of filters, it might be downright frustrating.

Fortunately, Assetic provides a way to dump your assets to real files, instead of being generated dynamically.

Dumping Asset Files in the **prod** Environment

In the **prod** environment, your JS and CSS files are represented by a single tag each. In other words, instead of seeing each JavaScript file you're including in your source, you'll likely just see something like this:

Listing 1-18 1 `<script src="/js/abcd123.js"></script>`

Moreover, that file does **not** actually exist, nor is it dynamically rendered by Symfony (as the asset files are in the **dev** environment). This is on purpose - letting Symfony generate these files dynamically in a production environment is just too slow.

Instead, each time you use your application in the **prod** environment (and therefore, each time you deploy), you should run the following command:

Listing 1-19 1 `$ php bin/console assetic:dump --env=prod --no-debug`

This will physically generate and write each file that you need (e.g. `/js/abcd123.js`). If you update any of your assets, you'll need to run this again to regenerate the file.

Dumping Asset Files in the **dev** Environment

By default, each asset path generated in the **dev** environment is handled dynamically by Symfony. This has no disadvantage (you can see your changes immediately), except that assets can load noticeably slow. If you feel like your assets are loading too slowly, follow this guide.

First, tell Symfony to stop trying to process these files dynamically. Make the following change in your `config_dev.yml` file:

Listing 1-20 1 `# app/config/config_dev.yml`
2 `assetic:`
3 `use_controller: false`

Next, since Symfony is no longer generating these assets for you, you'll need to dump them manually. To do so, run the following command:

Listing 1-21 1 `$ php bin/console assetic:dump`

This physically writes all of the asset files you need for your **dev** environment. The big disadvantage is that you need to run this each time you update an asset. Fortunately, by using the **assetic:watch** command, assets will be regenerated automatically *as they change*:

Listing 1-22 1 \$ php bin/console assetic:watch

The **assetic:watch** command was introduced in AsseticBundle 2.4. In prior versions, you had to use the **--watch** option of the **assetic:dump** command for the same behavior.

Since running this command in the **dev** environment may generate a bunch of files, it's usually a good idea to point your generated asset files to some isolated directory (e.g. **/js/compiled**), to keep things organized:

Listing 1-23 1 {% javascripts '@AppBundle/Resources/public/js/*' output='js/compiled/main.js' %}
2 <script src="{{ asset_url }}"></script>
3 {% endjavascripts %}



Chapter 2

Combining, Compiling and Minimizing Web Assets with PHP Libraries



Starting from Symfony 2.8, Assetic is no longer included by default in the Symfony Standard Edition. Refer to *this article* to learn how to install and enable Assetic in your Symfony application.

The official Symfony Best Practices recommend to use Assetic to *manage web assets*, unless you are comfortable with JavaScript-based front-end tools.

Even if those JavaScript-based solutions are the most suitable ones from a technical point of view, using pure PHP alternative libraries can be useful in some scenarios:

- If you can't install or use `npm` and the other JavaScript solutions;
- If you prefer to limit the amount of different technologies used in your applications;
- If you want to simplify application deployment.

In this article, you'll learn how to combine and minimize CSS and JavaScript files and how to compile Sass files using PHP-only libraries with Assetic.

Installing the Third-Party Compression Libraries

Assetic includes a lot of ready-to-use filters, but it doesn't include their associated libraries. Therefore, before enabling the filters used in this article, you must install two libraries. Open a command console, browse to your project directory and execute the following commands:

Listing 2-1

```
1 $ composer require leafo/scssphp
2 $ composer require patchwork/jsqueeze
```

Organizing your Web Asset Files

This example will include a setup using the Bootstrap CSS framework, jQuery, FontAwesome and some regular CSS and JavaScript application files (called `main.css` and `main.js`). The recommended directory structure for this set-up looks like this:

Listing 2-2

```
1 web/assets/
2   ├── css
3       ├── main.css
4       └── code-highlight.css
5   ├── js
6       ├── bootstrap.js
7       ├── jquery.js
8       └── main.js
9   └── scss
10      ├── bootstrap
11          ├── _alerts.scss
12          ├── ...
13          ├── _variables.scss
14          ├── _wells.scss
15          └── mixins
16              ├── _alerts.scss
17              ├── ...
18              └── _vendor-prefixes.scss
19      ├── bootstrap.scss
20      ├── font-awesome
21          ├── _animated.scss
22          ├── ...
23          ├── _variables.scss
24          └── font-awesome.scss
```

Combining and Minimizing CSS Files and Compiling SCSS Files

First, configure a new `scssphp` Assetic filter:

Listing 2-3

```
1 # app/config/config.yml
2 assetic:
3     filters:
4         scssphp:
5             formatter: 'Leafo\ScssPhp\Formatter\Compressed'
6             # ...
```

The value of the `formatter` option is the fully qualified class name of the formatter used by the filter to produce the compiled CSS file. Using the compressed formatter will minimize the resulting file, regardless of whether the original files are regular CSS files or SCSS files.

Next, update your Twig template to add the `{% stylesheets %}` tag defined by Assetic:

Listing 2-4

```
1 {# app/Resources/views/base.html.twig #}
2 <!DOCTYPE html>
3 <html>
4     <head>
5         <!-- ... -->
6
7         {% stylesheets filter="scssphp" output="css/app.css"
8             "assets/scss/bootstrap.scss"
9             "assets/scss/font-awesome.scss"
10            "assets/css/*.css"
11         %}
12         <link rel="stylesheet" href="{{ asset_url }}" />
13     {% endstylesheets %}
```

This simple configuration compiles, combines and minifies the SCSS files into a regular CSS file that's put in **web/css/app.css**. This is the only CSS file which will be served to your visitors.

Combining and Minimizing JavaScript Files

First, configure a new **jsqueeze** Assetic filter as follows:

Listing 2-5

```
1 # app/config/config.yml
2 assetic:
3     filters:
4         jsqueeze: ~
5         # ...
```

Next, update the code of your Twig template to add the **{% javascripts %}** tag defined by Assetic:

Listing 2-6

```
1 <!-- ... -->
2
3 {% javascripts filter="?jsqueeze" output="js/app.js"
4     "assets/js/jquery.js"
5     "assets/js/bootstrap.js"
6     "assets/js/main.js"
7 %}
8 <script src="{{ asset_url }}"></script>
9 {% endjavascripts %}
10
11 </body>
12 </html>
```

This simple configuration combines all the JavaScript files, minimizes the contents and saves the output in the **web/js/app.js** file, which is the one that is served to your visitors.

The leading **?** character in the **jsqueeze** filter name tells Assetic to only apply the filter when *not* in **debug** mode. In practice, this means that you'll see unminified files while developing and minimized files in the **prod** environment.



Chapter 3

How to Minify CSS/JS Files (Using UglifyJS and UglifyCSS)



Starting from Symfony 2.8, Assetic is no longer included by default in the Symfony Standard Edition. Refer to *this article* to learn how to install and enable Assetic in your Symfony application.

*UglifyJS*¹ is a JavaScript parser/compressor/beautifier toolkit. It can be used to combine and minify JavaScript assets so that they require less HTTP requests and make your site load faster. *UglifyCSS*² is a CSS compressor/beautifier that is very similar to UglifyJS.

In this cookbook, the installation, configuration and usage of UglifyJS is shown in detail. UglifyCSS works pretty much the same way and is only talked about briefly.

Install UglifyJS

UglifyJS is available as a *Node.js*³ module. First, you need to *install Node.js*⁴ and then, decide the installation method: global or local.

Global Installation

The global installation method makes all your projects use the very same UglifyJS version, which simplifies its maintenance. Open your command console and execute the following command (you may need to run it as a root user):

Listing 3-1 1 \$ npm install -g uglify-js

Now you can execute the global **uglifyjs** command anywhere on your system:

-
1. <https://github.com/mishoo/UglifyJS>
 2. <https://github.com/fmarcia/UglifyCSS>
 3. <https://nodejs.org/>
 4. <https://nodejs.org/>

Listing 3-2 1 \$ uglifyjs --help

Local Installation

It's also possible to install UglifyJS inside your project only, which is useful when your project requires a specific UglifyJS version. To do this, install it without the **-g** option and specify the path where to put the module:

Listing 3-3 1 \$ cd /path/to/your/symfony/project
2 \$ npm install uglify-js --prefix app/Resources

It is recommended that you install UglifyJS in your **app/Resources** folder and add the **node_modules** folder to version control. Alternatively, you can create an npm *package.json*⁵ file and specify your dependencies there.

Now you can execute the **uglifyjs** command that lives in the **node_modules** directory:

Listing 3-4 1 \$ *./app/Resources/node_modules/.bin/uglifyjs* --help

Configure the uglifyjs2 Filter

Now we need to configure Symfony to use the **uglifyjs2** filter when processing your JavaScripts:

Listing 3-5 1 *# app/config/config.yml*
2 **assetic:**
3 **filters:**
4 **uglifyjs2:**
5 *# the path to the uglifyjs executable*
6 **bin:** /usr/local/bin/uglifyjs



The path where UglifyJS is installed may vary depending on your system. To find out where npm stores the **bin** folder, execute the following command:

Listing 3-6 1 \$ npm bin -g

It should output a folder on your system, inside which you should find the UglifyJS executable.

If you installed UglifyJS locally, you can find the **bin** folder inside the **node_modules** folder. It's called **.bin** in this case.

You now have access to the **uglifyjs2** filter in your application.

Configure the node Binary

Assetic tries to find the node binary automatically. If it cannot be found, you can configure its location using the **node** key:

Listing 3-7 1 *# app/config/config.yml*
2 **assetic:**

5. <http://browsenpm.org/package.json>

```

3     # the path to the node executable
4     node: /usr/bin/nodejs
5     filters:
6         uglifyjs2:
7             # the path to the uglifyjs executable
8             bin: /usr/local/bin/uglifyjs

```

Minify your Assets

In order to apply UglifyJS on your assets, add the **filter** option in the asset tags of your templates to tell Assetic to use the **uglifyjs2** filter:

Listing 3-8

```

1  {% javascripts '@AppBundle/Resources/public/js/*' filter='uglifyjs2' %}
2  <script src="{{ asset_url }}"></script>
3  {% endjavascripts %}

```



The above example assumes that you have a bundle called AppBundle and your JavaScript files are in the **Resources/public/js** directory under your bundle. However you can include your JavaScript files no matter where they are.

With the addition of the **uglifyjs2** filter to the asset tags above, you should now see minified JavaScripts coming over the wire much faster.

Disable Minification in Debug Mode

Minified JavaScripts are very difficult to read, let alone debug. Because of this, Assetic lets you disable a certain filter when your application is in debug (e.g. **app_dev.php**) mode. You can do this by prefixing the filter name in your template with a question mark: **?**. This tells Assetic to only apply this filter when debug mode is off (e.g. **app.php**):

Listing 3-9

```

1  {% javascripts '@AppBundle/Resources/public/js/*' filter='?uglifyjs2' %}
2  <script src="{{ asset_url }}"></script>
3  {% endjavascripts %}

```

To try this out, switch to your **prod** environment (**app.php**). But before you do, don't forget to clear your cache and dump your assetic assets.



Instead of adding the filters to the asset tags, you can also configure which filters to apply for each file in your application configuration file. See [Filtering Based on a File Extension](#) for more details.

Install, Configure and Use UglifyCSS

The usage of UglifyCSS works the same way as UglifyJS. First, make sure the node package is installed:

Listing 3-10

```

1  # global installation
2  $ npm install -g uglifycss
3
4  # local installation
5  $ cd /path/to/your/symfony/project
6  $ npm install uglifycss --prefix app/Resources

```

Next, add the configuration for this filter:

Listing 3-11

```
1 # app/config/config.yml
2 assetic:
3     filters:
4         uglifycss:
5             bin: /usr/local/bin/uglifycss
```

To use the filter for your CSS files, add the filter to the Assetic **stylesheets** helper:

Listing 3-12

```
1 {% stylesheets 'bundles/App/css/*' filter='uglifycss' filter='cssrewrite' %}
2     <link rel="stylesheet" href="{{ asset_url }}" />
3 {% endstylesheets %}
```

Just like with the **uglifyjs2** filter, if you prefix the filter name with **?** (i.e. **?uglifycss**), the minification will only happen when you're not in debug mode.



Chapter 4

How to Minify JavaScripts and Stylesheets with YUI Compressor



The YUI Compressor is *no longer maintained by Yahoo*¹. That's why you are **strongly advised to avoid using YUI utilities** unless strictly necessary. Read *How to Minify CSS/JS Files (Using UglifyJS and UglifyCSS)* for a modern and up-to-date alternative.



Starting from Symfony 2.8, Assetic is no longer included by default in the Symfony Standard Edition. Refer to *this article* to learn how to install and enable Assetic in your Symfony application.

Yahoo! provides an excellent utility for minifying JavaScripts and stylesheets so they travel over the wire faster, the *YUI Compressor*². Thanks to Assetic, you can take advantage of this tool very easily.

Download the YUI Compressor JAR

The YUI Compressor is written in Java and distributed as a JAR. *Download the JAR*³ from the Yahoo! website and save it to `app/Resources/java/yuicompressor.jar`.

Configure the YUI Filters

Now you need to configure two Assetic filters in your application, one for minifying JavaScripts with the YUI Compressor and one for minifying stylesheets:

Listing 4-1

```
1 # app/config/config.yml
2 assetic:
```

-
1. <http://yuiblog.com/blog/2013/01/24/yui-compressor-has-a-new-owner/>
 2. <http://yui.github.io/yuicompressor/>
 3. <https://github.com/yui/yuicompressor/releases>

```

3  # java: '/usr/bin/java'
4  filters:
5    yui_css:
6      jar: '%kernel.root_dir%/Resources/java/yuicompressor.jar'
7    yui_js:
8      jar: '%kernel.root_dir%/Resources/java/yuicompressor.jar'

```



Windows users need to remember to update config to proper Java location. In Windows7 x64 bit by default it's **C:\Program Files (x86)\Java\jre6\bin\java.exe**.

You now have access to two new Assetic filters in your application: **yui_css** and **yui_js**. These will use the YUI Compressor to minify stylesheets and JavaScripts, respectively.

Minify your Assets

You have YUI Compressor configured now, but nothing is going to happen until you apply one of these filters to an asset. Since your assets are a part of the view layer, this work is done in your templates:

Listing 4-2

```

1  {% javascripts '@AppBundle/Resources/public/js/*' filter='yui_js' %}
2    <script src="{{ asset_url }}"></script>
3  {% endjavascripts %}

```



The above example assumes that you have a bundle called AppBundle and your JavaScript files are in the **Resources/public/js** directory under your bundle. This isn't important however - you can include your JavaScript files no matter where they are.

With the addition of the **yui_js** filter to the asset tags above, you should now see minified JavaScripts coming over the wire much faster. The same process can be repeated to minify your stylesheets.

Listing 4-3

```

1  {% stylesheets '@AppBundle/Resources/public/css/*' filter='yui_css' %}
2    <link rel="stylesheet" type="text/css" media="screen" href="{{ asset_url }}" />
3  {% endstylesheets %}

```

Disable Minification in Debug Mode

Minified JavaScripts and stylesheets are very difficult to read, let alone debug. Because of this, Assetic lets you disable a certain filter when your application is in debug mode. You can do this by prefixing the filter name in your template with a question mark: **?**. This tells Assetic to only apply this filter when debug mode is off.

Listing 4-4

```

1  {% javascripts '@AppBundle/Resources/public/js/*' filter='?yui_js' %}
2    <script src="{{ asset_url }}"></script>
3  {% endjavascripts %}

```



Instead of adding the filter to the asset tags, you can also globally enable it by adding the **apply_to** attribute to the filter configuration, for example in the **yui_js** filter **apply_to: "\.js\$"**. To only have the filter applied in production, add this to the **config_prod** file rather than the common config file. For details on applying filters by file extension, see [Filtering Based on a File Extension](#).



Chapter 5

How to Use Assetic for Image Optimization with Twig Functions



Starting from Symfony 2.8, Assetic is no longer included by default in the Symfony Standard Edition. Refer to *this article* to learn how to install and enable Assetic in your Symfony application.

Among its many filters, Assetic has four filters which can be used for on-the-fly image optimization. This allows you to get the benefits of smaller file sizes without having to use an image editor to process each image. The results are cached and can be dumped for production so there is no performance hit for your end users.

Using Jpegoptim

*Jpegoptim*¹ is a utility for optimizing JPEG files. To use it with Assetic, make sure to have it already installed on your system and then, configure its location using the `bin` option of the `jpegoptim` filter:

Listing 5-1

```
1 # app/config/config.yml
2 assetic:
3     filters:
4         jpegoptim:
5             bin: path/to/jpegoptim
```

It can now be used from a template:

Listing 5-2

```
1 {% image '@AppBundle/Resources/public/images/example.jpg'
2     filter='jpegoptim' output='/images/example.jpg' %}
3 
4 {% endimage %}
```

1. <http://www.kokkonen.net/tjko/projects.html>

Removing all EXIF Data

By default, the `jpegoptim` filter removes some meta information stored in the image. To remove all EXIF data and comments, set the `strip_all` option to `true`:

Listing 5-3

```
1 # app/config/config.yml
2 assetic:
3   filters:
4     jpegoptim:
5       bin: path/to/jpegoptim
6       strip_all: true
```

Lowering Maximum Quality

By default, the `jpegoptim` filter doesn't alter the quality level of the JPEG image. Use the `max` option to configure the maximum quality setting (in a scale of `0` to `100`). The reduction in the image file size will of course be at the expense of its quality:

Listing 5-4

```
1 # app/config/config.yml
2 assetic:
3   filters:
4     jpegoptim:
5       bin: path/to/jpegoptim
6       max: 70
```

Shorter Syntax: Twig Function

If you're using Twig, it's possible to achieve all of this with a shorter syntax by enabling and using a special Twig function. Start by adding the following configuration:

Listing 5-5

```
1 # app/config/config.yml
2 assetic:
3   filters:
4     jpegoptim:
5       bin: path/to/jpegoptim
6   twig:
7     functions:
8       jpegoptim: ~
```

The Twig template can now be changed to the following:

Listing 5-6

```
1 
```

You can also specify the output directory for images in the Assetic configuration file:

Listing 5-7

```
1 # app/config/config.yml
2 assetic:
3   filters:
4     jpegoptim:
5       bin: path/to/jpegoptim
6   twig:
7     functions:
8       jpegoptim: { output: images/*.jpg }
```



For uploaded images, you can compress and manipulate them using the *LiipImagineBundle*² community bundle.



Chapter 6

How to Apply an Assetic Filter to a specific File Extension



Starting from Symfony 2.8, Assetic is no longer included by default in the Symfony Standard Edition. Refer to *this article* to learn how to install and enable Assetic in your Symfony application.

Assetic filters can be applied to individual files, groups of files or even, as you'll see here, files that have a specific extension. To show you how to handle each option, suppose that you want to use Assetic's CoffeeScript filter, which compiles CoffeeScript files into JavaScript.

The main configuration is just the paths to `coffee`, `node` and `node_modules`. An example configuration might look like this:

Listing 6-1

```
1 # app/config/config.yml
2 assetic:
3     filters:
4         coffee:
5             bin:      /usr/bin/coffee
6             node:     /usr/bin/node
7             node_paths: [/usr/lib/node_modules/]
```

Filter a single File

You can now serve up a single CoffeeScript file as JavaScript from within your templates:

Listing 6-2

```
1 {% javascripts '@AppBundle/Resources/public/js/example.coffee' filter='coffee' %}
2 <script src="{{ asset_url }}"></script>
3 {% endjavascripts %}
```

This is all that's needed to compile this CoffeeScript file and serve it as the compiled JavaScript.

Filter multiple Files

You can also combine multiple CoffeeScript files into a single output file:

Listing 6-3

```
1 {% javascripts '@AppBundle/Resources/public/js/example.coffee'
2             '@AppBundle/Resources/public/js/another.coffee'
3     filter='coffee' %}
4     <script src="{{ asset_url }}"></script>
5 {% endjavascripts %}
```

Both files will now be served up as a single file compiled into regular JavaScript.

Filtering Based on a File Extension

One of the great advantages of using Assetic is reducing the number of asset files to lower HTTP requests. In order to make full use of this, it would be good to combine *all* your JavaScript and CoffeeScript files together since they will ultimately all be served as JavaScript. Unfortunately just adding the JavaScript files to the files to be combined as above will not work as the regular JavaScript files will not survive the CoffeeScript compilation.

This problem can be avoided by using the **apply_to** option, which allows you to specify which filter should always be applied to particular file extensions. In this case you can specify that the **coffee** filter is applied to all **.coffee** files:

Listing 6-4

```
1 # app/config/config.yml
2 assetic:
3     filters:
4         coffee:
5             bin:      /usr/bin/coffee
6             node:     /usr/bin/node
7             node_paths: [/usr/lib/node_modules/]
8             apply_to: /\.coffee$'
```

With this option, you no longer need to specify the **coffee** filter in the template. You can also list regular JavaScript files, all of which will be combined and rendered as a single JavaScript file (with only the **.coffee** files being run through the CoffeeScript filter):

Listing 6-5

```
1 {% javascripts '@AppBundle/Resources/public/js/example.coffee'
2             '@AppBundle/Resources/public/js/another.coffee'
3             '@AppBundle/Resources/public/js/regular.js' %}
4     <script src="{{ asset_url }}"></script>
5 {% endjavascripts %}
```



Chapter 7

How to Install 3rd Party Bundles

Most bundles provide their own installation instructions. However, the basic steps for installing a bundle are the same:

- A) Add Composer Dependencies
- B) Enable the Bundle
- C) Configure the Bundle

A) Add Composer Dependencies

Dependencies are managed with Composer, so if Composer is new to you, learn some basics in *their documentation*¹. This involves two steps:

1) Find out the Name of the Bundle on Packagist

The README for a bundle (e.g. *FOSUserBundle*²) usually tells you its name (e.g. **friendsofsymfony/user-bundle**). If it doesn't, you can search for the bundle on the *Packagist.org*³ site.



Looking for bundles? Try searching at *KnjBundles.com*⁴: the unofficial archive of Symfony Bundles.

2) Install the Bundle via Composer

Now that you know the package name, you can install it via Composer:

Listing 7-1 1 `$ composer require friendsofsymfony/user-bundle`

1. <https://getcomposer.org/doc/00-intro.md>
2. <https://github.com/FriendsOfSymfony/FOSUserBundle>
3. <https://packagist.org>
4. <http://knjbundles.com/>

This will choose the best version for your project, add it to **composer.json** and download its code into the **vendor/** directory. If you need a specific version, include it as the second argument of the *composer require*⁵ command:

Listing 7-2 1 \$ composer require friendsofsymfony/user-bundle "~2.0"

B) Enable the Bundle

At this point, the bundle is installed in your Symfony project (in **vendor/friendsofsymfony/**) and the autoloader recognizes its classes. The only thing you need to do now is register the bundle in **AppKernel**:

Listing 7-3

```
1 // app/AppKernel.php
2
3 // ...
4 class AppKernel extends Kernel
5 {
6     // ...
7
8     public function registerBundles()
9     {
10         $bundles = array(
11             // ...
12             new FOS\UserBundle\FOSUserBundle(),
13         );
14
15         // ...
16     }
17 }
```

In a few rare cases, you may want a bundle to be *only* enabled in the development *environment*. For example, the DoctrineFixturesBundle helps to load dummy data - something you probably only want to do while developing. To only load this bundle in the **dev** and **test** environments, register the bundle in this way:

Listing 7-4

```
1 // app/AppKernel.php
2
3 // ...
4 class AppKernel extends Kernel
5 {
6     // ...
7
8     public function registerBundles()
9     {
10         $bundles = array(
11             // ...
12         );
13
14         if (in_array($this->getEnvironment(), array('dev', 'test'))) {
15             $bundles[] = new Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle();
16         }
17
18         // ...
19     }
20 }
```

5. <https://getcomposer.org/doc/03-cli.md#require>

C) Configure the Bundle

It's pretty common for a bundle to need some additional setup or configuration in `app/config/config.yml`. The bundle's documentation will tell you about the configuration, but you can also get a reference of the bundle's configuration via the `config:dump-reference` command:

Listing 7-5 1 `$ bin/console config:dump-reference AsseticBundle`

Instead of the full bundle name, you can also pass the short name used as the root of the bundle's configuration:

Listing 7-6 1 `$ bin/console config:dump-reference assetic`

The output will look like this:

Listing 7-7

```
1  assetic:
2      debug:                '%kernel.debug%'
3      use_controller:
4          enabled:          '%kernel.debug%'
5          profiler:         false
6      read_from:            '%kernel.root_dir%../web'
7      write_to:             '%assetic.read_from%'
8      java:                 /usr/bin/java
9      node:                 /usr/local/bin/node
10     node_paths:           []
11     # ...
```

Other Setup

At this point, check the **README** file of your brand new bundle to see what to do next. Have fun!



Chapter 8

Best Practices for Reusable Bundles

There are two types of bundles:

- Application-specific bundles: only used to build your application;
- Reusable bundles: meant to be shared across many projects.

This article is all about how to structure your **reusable bundles** so that they're easy to configure and extend. Many of these recommendations do not apply to application bundles because you'll want to keep those as simple as possible. For application bundles, just follow the practices shown throughout the book and cookbook.

The best practices for application-specific bundles are discussed in [The Symfony Framework Best Practices](#).

Bundle Name

A bundle is also a PHP namespace. The namespace must follow the *PSR-0*¹ or *PSR-4*² interoperability standards for PHP namespaces and class names: it starts with a vendor segment, followed by zero or more category segments, and it ends with the namespace short name, which must end with a **Bundle** suffix.

A namespace becomes a bundle as soon as you add a bundle class to it. The bundle class name must follow these simple rules:

- Use only alphanumeric characters and underscores;
- Use a CamelCased name;
- Use a descriptive and short name (no more than two words);
- Prefix the name with the concatenation of the vendor (and optionally the category namespaces);
- Suffix the name with `Bundle`.

Here are some valid bundle namespaces and class names:

1. <http://www.php-fig.org/psr/psr-0/>
2. <http://www.php-fig.org/psr/psr-4/>

Namespace	Bundle Class Name
Acme\Bundle\BlogBundle	AcmeBlogBundle
Acme\BlogBundle	AcmeBlogBundle

By convention, the `getName()` method of the bundle class should return the class name.



If you share your bundle publicly, you must use the bundle class name as the name of the repository (**AcmeBlogBundle** and not **BlogBundle** for instance).



Symfony core Bundles do not prefix the Bundle class with **Symfony** and always add a **Bundle** sub-namespace; for example: *FrameworkBundle*³.

Each bundle has an alias, which is the lower-cased short version of the bundle name using underscores (**acme_blog** for **AcmeBlogBundle**). This alias is used to enforce uniqueness within a project and for defining bundle's configuration options (see below for some usage examples).

Directory Structure

The basic directory structure of an AcmeBlogBundle must read as follows:

Listing 8-1

```

1 <your-bundle>/
2 |— AcmeBlogBundle.php
3 |— Controller/
4 |— README.md
5 |— LICENSE
6 |— Resources/
7 |   |— config/
8 |   |— doc/
9 |   |   |— index.rst
10 |   |— translations/
11 |   |— views/
12 |   |— public/
13 |— Tests/
```

The following files are mandatory, because they ensure a structure convention that automated tools can rely on:

- **AcmeBlogBundle.php**: This is the class that transforms a plain directory into a Symfony bundle (change this to your bundle's name);
- **README.md**: This file contains the basic description of the bundle and it usually shows some basic examples and links to its full documentation (it can use any of the markup formats supported by GitHub, such as **README.rst**);
- **LICENSE**: The full contents of the license used by the code. Most third-party bundles are published under the MIT license, but you can *choose any license*⁴;
- **Resources/doc/index.rst**: The root file for the Bundle documentation.

The depth of sub-directories should be kept to the minimum for most used classes and files (two levels maximum).

3. <http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/FrameworkBundle.html>

4. <http://choosealicense.com/>

The bundle directory is read-only. If you need to write temporary files, store them under the **cache/** or **log/** directory of the host application. Tools can generate files in the bundle directory structure, but only if the generated files are going to be part of the repository.

The following classes and files have specific emplacements (some are mandatory and others are just conventions followed by most developers):

Type	Directory	Mandatory?
Commands	Command/	Yes
Controllers	Controller/	No
Service Container Extensions	DependencyInjection/	Yes
Event Listeners	EventListener/	No
Model classes [1]	Model/	No
Configuration	Resources/config/	No
Web Resources (CSS, JS, images)	Resources/public/	Yes
Translation files	Resources/translations/	Yes
Templates	Resources/views/	Yes
Unit and Functional Tests	Tests/	No

[1] See *How to Provide Model Classes for several Doctrine Implementations* for how to handle the mapping with a compiler pass.

Classes

The bundle directory structure is used as the namespace hierarchy. For instance, a **ContentController** controller is stored in **Acme/BlogBundle/Controller/ContentController.php** and the fully qualified class name is **Acme\BlogBundle\Controller\ContentController**.

All classes and files must follow the *Symfony coding standards*.

Some classes should be seen as facades and should be as short as possible, like Commands, Helpers, Listeners and Controllers.

Classes that connect to the event dispatcher should be suffixed with **Listener**.

Exception classes should be stored in an **Exception** sub-namespace.

Vendors

A bundle must not embed third-party PHP libraries. It should rely on the standard Symfony autoloading instead.

A bundle should not embed third-party libraries written in JavaScript, CSS or any other language.

Tests

A bundle should come with a test suite written with PHPUnit and stored under the **Tests/** directory. Tests should follow the following principles:

- The test suite must be executable with a simple `phpunit` command run from a sample application;
- The functional tests should only be used to test the response output and some profiling information if you have some;
- The tests should cover at least 95% of the code base.



A test suite must not contain `AllTests.php` scripts, but must rely on the existence of a `phpunit.xml.dist` file.

Documentation

All classes and functions must come with full PHPDoc.

Extensive documentation should also be provided in the *reStructuredText* format, under the `Resources/doc/` directory; the `Resources/doc/index.rst` file is the only mandatory file and must be the entry point for the documentation.

Installation Instructions

In order to ease the installation of third-party bundles, consider using the following standardized instructions in your `README.md` file.

Listing 8-2

```

1  Installation
2  =====
3
4  Step 1: Download the Bundle
5  -----
6
7  Open a command console, enter your project directory and execute the
8  following command to download the latest stable version of this bundle:
9
10 ``bash
11 $ composer require <package-name> "~1"
12 ``
13
14 This command requires you to have Composer installed globally, as explained
15 in the [installation chapter](https://getcomposer.org/doc/00-intro.md)
16 of the Composer documentation.
17
18 Step 2: Enable the Bundle
19 -----
20
21 Then, enable the bundle by adding it to the list of registered bundles
22 in the `app/AppKernel.php` file of your project:
23
24 ``php
25 <?php
26 // app/AppKernel.php
27
28 // ...
29 class AppKernel extends Kernel
30 {
31     public function registerBundles()
32     {
33         $bundles = array(
34             // ...
35
36             new <vendor>\<bundle-name>\<bundle-long-name>(),
37         );
38
39         // ...
40     }
41 
```

```

42     // ...
43 }
44 ...

```

The example above assumes that you are installing the latest stable version of the bundle, where you don't have to provide the package version number (e.g. `composer require friendsofsymfony/user-bundle`). If the installation instructions refer to some past bundle version or to some unstable version, include the version constraint (e.g. `composer require friendsofsymfony/user-bundle "~2.0@dev"`).

Optionally, you can add more installation steps (*Step 3*, *Step 4*, etc.) to explain other required installation tasks, such as registering routes or dumping assets.

Routing

If the bundle provides routes, they must be prefixed with the bundle alias. For example, if your bundle is called `AcmeBlogBundle`, all its routes must be prefixed with `acme_blog_`.

Templates

If a bundle provides templates, they must use Twig. A bundle must not provide a main layout, except if it provides a full working application.

Translation Files

If a bundle provides message translations, they must be defined in the XLIFF format; the domain should be named after the bundle name (`acme_blog`).

A bundle must not override existing messages from another bundle.

Configuration

To provide more flexibility, a bundle can provide configurable settings by using the Symfony built-in mechanisms.

For simple configuration settings, rely on the default `parameters` entry of the Symfony configuration. Symfony parameters are simple key/value pairs; a value being any valid PHP value. Each parameter name should start with the bundle alias, though this is just a best-practice suggestion. The rest of the parameter name will use a period (`.`) to separate different parts (e.g. `acme_blog.author.email`).

The end user can provide values in any configuration file:

Listing 8-3

```

1 # app/config/config.yml
2 parameters:
3     acme_blog.author.email: 'fabien@example.com'

```

Retrieve the configuration parameters in your code from the container:

Listing 8-4

```

$container->getParameter('acme_blog.author.email');

```

Even if this mechanism is simple enough, you should consider using the more advanced *semantic bundle configuration*.

Versioning

Bundles must be versioned following the *Semantic Versioning Standard*⁵.

Services

If the bundle defines services, they must be prefixed with the bundle alias. For example, `AcmeBlogBundle` services must be prefixed with `acme_blog`.

In addition, services not meant to be used by the application directly, should be defined as private.

You can learn much more about service loading in bundles reading this article: [How to Load Service Configuration inside a Bundle](#).

Composer Metadata

The `composer.json` file should include at least the following metadata:

name

Consists of the vendor and the short bundle name. If you are releasing the bundle on your own instead of on behalf of a company, use your personal name (e.g. `johnsmith/blog-bundle`). The bundle short name excludes the vendor name and separates each word with an hyphen. For example: `AcmeBlogBundle` is transformed into `blog-bundle` and `AcmeSocialConnectBundle` is transformed into `social-connect-bundle`.

description

A brief explanation of the purpose of the bundle.

type

Use the `symfony-bundle` value.

license

MIT is the preferred license for Symfony bundles, but you can use any other license.

autoload

This information is used by Symfony to load the classes of the bundle. The *PSR-4*⁶ autoload standard is recommended for modern bundles, but *PSR-0*⁷ standard is also supported.

In order to make it easier for developers to find your bundle, register it on *Packagist*⁸, the official repository for Composer packages.

Learn more from the Cookbook

- *[How to Load Service Configuration inside a Bundle](#)*

5. <http://semver.org/>

6. <http://www.php-fig.org/psr/psr-4/>

7. <http://www.php-fig.org/psr/psr-0/>

8. <https://packagist.org/>



Chapter 9

How to Use Bundle Inheritance to Override Parts of a Bundle

When working with third-party bundles, you'll probably come across a situation where you want to override a file in that third-party bundle with a file in one of your own bundles. Symfony gives you a very convenient way to override things like controllers, templates, and other files in a bundle's **Resources/** directory.

For example, suppose that you're installing the *FOSUserBundle*¹, but you want to override its base `layout.html.twig` template, as well as one of its controllers. Suppose also that you have your own *UserBundle* where you want the overridden files to live. Start by registering the *FOSUserBundle* as the "parent" of your bundle:

Listing 9-1

```
1  // src/UserBundle/UserBundle.php
2  namespace UserBundle;
3
4  use Symfony\Component\HttpKernel\Bundle\Bundle;
5
6  class UserBundle extends Bundle
7  {
8      public function getParent()
9      {
10         return 'FOSUserBundle';
11     }
12 }
```

By making this simple change, you can now override several parts of the *FOSUserBundle* simply by creating a file with the same name.



Despite the method name, there is no parent/child relationship between the bundles, it is just a way to extend and override an existing bundle.

1. <https://github.com/friendsofsymfony/fosuserbundle>

Overriding Controllers

Suppose you want to add some functionality to the `registerAction` of a `RegistrationController` that lives inside `FOSUserBundle`. To do so, just create your own `RegistrationController.php` file, override the bundle's original method, and change its functionality:

Listing 9-2

```
1 // src/UserBundle/Controller/RegistrationController.php
2 namespace UserBundle\Controller;
3
4 use FOS\UserBundle\Controller\RegistrationController as BaseController;
5
6 class RegistrationController extends BaseController
7 {
8     public function registerAction()
9     {
10         $response = parent::registerAction();
11
12         // ... do custom stuff
13         return $response;
14     }
15 }
```



Depending on how severely you need to change the behavior, you might call `parent::registerAction()` or completely replace its logic with your own.



Overriding controllers in this way only works if the bundle refers to the controller using the standard `FOSUserBundle:Registration:register` syntax in routes and templates. This is the best practice.

Overriding Resources: Templates, Routing, etc

Most resources can also be overridden, simply by creating a file in the same location as your parent bundle.

For example, it's very common to need to override the `FOSUserBundle`'s `layout.html.twig` template so that it uses your application's base layout. Since the file lives at `Resources/views/layout.html.twig` in the `FOSUserBundle`, you can create your own file in the same location of `UserBundle`. Symfony will ignore the file that lives inside the `FOSUserBundle` entirely, and use your file instead.

The same goes for routing files and some other resources.



The overriding of resources only works when you refer to resources with the `@FOSUserBundle/Resources/config/routing/security.xml` method. If you refer to resources without using the `@BundleName` shortcut, they can't be overridden in this way.



Translation and validation files do not work in the same way as described above. Read "Translations" if you want to learn how to override translations and see "Validation Metadata" for tricks to override the validation.



Chapter 10

How to Override any Part of a Bundle

This document is a quick reference for how to override different parts of third-party bundles.

Templates

For information on overriding templates, see

- [Overriding Bundle Templates.](#)
- *[How to Use Bundle Inheritance to Override Parts of a Bundle](#)*

Routing

Routing is never automatically imported in Symfony. If you want to include the routes from any bundle, then they must be manually imported from somewhere in your application (e.g. `app/config/routing.yml`).

The easiest way to "override" a bundle's routing is to never import it at all. Instead of importing a third-party bundle's routing, simply copy that routing file into your application, modify it, and import it instead.

Controllers

Assuming the third-party bundle involved uses non-service controllers (which is almost always the case), you can easily override controllers via bundle inheritance. For more information, see *[How to Use Bundle Inheritance to Override Parts of a Bundle](#)*. If the controller is a service, see the next section on how to override it.

Services & Configuration

In order to override/extend a service, there are two options. First, you can set the parameter holding the service's class name to your own class by setting it in `app/config/config.yml`. This of course is only possible if the class name is defined as a parameter in the service config of the bundle containing the service. For example, to override the class used for Symfony's **translator** service, you would override the **translator.class** parameter. Knowing exactly which parameter to override may take some research. For the translator, the parameter is defined and used in the `Resources/config/translation.xml` file in the core FrameworkBundle:

Listing 10-1

```
1 # app/config/config.yml
2 parameters:
3     translator.class: Acme\HelloBundle\Translation\Translator
```

Secondly, if the class is not available as a parameter, you want to make sure the class is always overridden when your bundle is used or if you need to modify something beyond just the class name, you should use a compiler pass:

Listing 10-2

```
1 // src/Acme/DemoBundle/DependencyInjection/Compiler/OverrideServiceCompilerPass.php
2 namespace Acme\DemoBundle\DependencyInjection\Compiler;
3
4 use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
5 use Symfony\Component\DependencyInjection\ContainerBuilder;
6
7 class OverrideServiceCompilerPass implements CompilerPassInterface
8 {
9     public function process(ContainerBuilder $container)
10     {
11         $definition = $container->getDefinition('original-service-id');
12         $definition->setClass('Acme\DemoBundle\YourService');
13     }
14 }
```

In this example you fetch the service definition of the original service, and set its class name to your own class.

See *How to Work with Compiler Passes in Bundles* for information on how to use compiler passes. If you want to do something beyond just overriding the class, like adding a method call, you can only use the compiler pass method.

Entities & Entity Mapping

Due to the way Doctrine works, it is not possible to override entity mapping of a bundle. However, if a bundle provides a mapped superclass (such as the **User** entity in the FOSUserBundle) one can override attributes and associations. Learn more about this feature and its limitations in *the Doctrine documentation*¹.

Forms

Form types are referred to by their fully-qualified class name:

Listing 10-3

```
$builder->add('name', CustomType::class);
```

1. <http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/inheritance-mapping.html#overrides>

This means that you cannot override this by creating a sub-class of **CustomType** and registering it as a service and tagging it with **form.type** (you *could* do this in earlier version).

Instead, you should use a "form type extension" to modify the existing form type. For more information, see *How to Create a Form Type Extension*.

Validation Metadata

Symfony loads all validation configuration files from every bundle and combines them into one validation metadata tree. This means you are able to add new constraints to a property, but you cannot override them.

To override this, the 3rd party bundle needs to have configuration for validation groups. For instance, the FOSUserBundle has this configuration. To create your own validation, add the constraints to a new validation group:

Listing 10-4

```
1 # src/Acme/UserBundle/Resources/config/validation.yml
2 FOS\UserBundle\Model\User:
3   properties:
4     plainPassword:
5       - NotBlank:
6         groups: [AcmeValidation]
7       - Length:
8         min: 6
9         minMessage: fos_user.password.short
10        groups: [AcmeValidation]
```

Now, update the FOSUserBundle configuration, so it uses your validation groups instead of the original ones.

Translations

Translations are not related to bundles, but to domains. That means that you can override the translations from any translation file, as long as it is in the correct domain.



The last translation file always wins. That means that you need to make sure that the bundle containing *your* translations is loaded after any bundle whose translations you're overriding. This is done in **AppKernel**.

Translation files are also not aware of *bundle inheritance*. If you want to override translations from the parent bundle, be sure that the parent bundle is loaded before the child bundle in the **AppKernel** class.

The file that always wins is the one that is placed in **app/Resources/translations**, as those files are always loaded last.



Chapter 11

How to Remove the AcmeDemoBundle

The Symfony Standard Edition comes with a complete demo that lives inside a bundle called AcmeDemoBundle. It is a great boilerplate to refer to while starting a project, but you'll probably want to eventually remove it.



This article uses the AcmeDemoBundle as an example, but you can use these steps to remove any bundle.

1. Unregister the Bundle in the AppKernel

To disconnect the bundle from the framework, you should remove the bundle from the `AppKernel::registerBundles()` method. The bundle is normally found in the `$bundles` array but the AcmeDemoBundle is only registered in the development environment and you can find it inside the if statement below:

Listing 11-1

```
1  // app/AppKernel.php
2
3  // ...
4  class AppKernel extends Kernel
5  {
6      public function registerBundles()
7      {
8          $bundles = array(...);
9
10         if (in_array($this->getEnvironment(), array('dev', 'test'))) {
11             // comment or remove this line:
12             // $bundles[] = new Acme\DemoBundle\AcmeDemoBundle();
13             // ...
14         }
15     }
16 }
```

2. Remove Bundle Configuration

Now that Symfony doesn't know about the bundle, you need to remove any configuration and routing configuration inside the `app/config` directory that refers to the bundle.

2.1 Remove Bundle Routing

The routing for the `AcmeDemoBundle` can be found in `app/config/routing_dev.yml`. Remove the `_acme_demo` entry at the bottom of this file.

2.2 Remove Bundle Configuration

Some bundles contain configuration in one of the `app/config/config*.yml` files. Be sure to remove the related configuration from these files. You can quickly spot bundle configuration by looking for an `acme_demo` (or whatever the name of the bundle is, e.g. `fos_user` for the `FOSUserBundle`) string in the configuration files.

The `AcmeDemoBundle` doesn't have configuration. However, the bundle is used in the configuration for the `app/config/security.yml` file. You can use it as a boilerplate for your own security, but you **can** also remove everything: it doesn't matter to Symfony if you remove it or not.

3. Remove the Bundle from the Filesystem

Now you have removed every reference to the bundle in your application, you should remove the bundle from the filesystem. The bundle is located in the `src/Acme/DemoBundle` directory. You should remove this directory and you can remove the `Acme` directory as well.



If you don't know the location of a bundle, you can use the `getPath()`¹ method to get the path of the bundle:

Listing 11-2

```
dump($this->container->get('kernel')->getBundle('AcmeDemoBundle')->getPath());
die();
```

3.1 Remove Bundle Assets

Remove the assets of the bundle in the `web/` directory (e.g. `web/bundles/acmedemo` for the `AcmeDemoBundle`).

4. Remove Integration in other Bundles



This doesn't apply to the `AcmeDemoBundle` - no other bundles depend on it, so you can skip this step.

Some bundles rely on other bundles, if you remove one of the two, the other will probably not work. Be sure that no other bundles, third party or self-made, rely on the bundle you are about to remove.

1. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Bundle/BundleInterface.html#method_getPath



If one bundle relies on another, in most cases it means that it uses some services from the bundle. Searching for the bundle alias string may help you spot them (e.g. `acme_demo` for bundles depending on `AcmeDemoBundle`).



If a third party bundle relies on another bundle, you can find that bundle mentioned in the `composer.json` file included in the bundle directory.



Chapter 12

How to Load Service Configuration inside a Bundle

In Symfony, you'll find yourself using many services. These services can be registered in the `app/config/` directory of your application. But when you want to decouple the bundle for use in other projects, you want to include the service configuration in the bundle itself. This article will teach you how to do that.

Creating an Extension Class

In order to load service configuration, you have to create a Dependency Injection (DI) Extension for your bundle. This class has some conventions in order to be detected automatically. But you'll later see how you can change it to your own preferences. By default, the Extension has to comply with the following conventions:

- It has to live in the `DependencyInjection` namespace of the bundle;
- The name is equal to the bundle name with the `Bundle` suffix replaced by `Extension` (e.g. the Extension class of the `AppBundle` would be called `AppBundleExtension` and the one for `AcmeHelloBundle` would be called `AcmeHelloExtension`).

The Extension class should implement the *ExtensionInterface*¹, but usually you would simply extend the *Extension*² class:

Listing 12-1

```
1 // src/Acme/HelloBundle/DependencyInjection/AcmeHelloExtension.php
2 namespace Acme\HelloBundle\DependencyInjection;
3
4 use Symfony\Component\HttpKernel\DependencyInjection\Extension;
5 use Symfony\Component\DependencyInjection\ContainerBuilder;
6
7 class AcmeHelloExtension extends Extension
8 {
```

1. <http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Extension/ExtensionInterface.html>

2. <http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Extension/Extension.html>

```

9     public function load(array $configs, ContainerBuilder $container)
10    {
11        // ... you'll load the files here later
12    }
13 }

```

Manually Registering an Extension Class

When not following the conventions, you will have to manually register your extension. To do this, you should override the *Bundle::getContainerExtension()*³ method to return the instance of the extension:

Listing 12-2

```

1 // ...
2 use Acme\HelloBundle\DependencyInjection\UnconventionalExtensionClass;
3
4 class AcmeHelloBundle extends Bundle
5 {
6     public function getContainerExtension()
7     {
8         return new UnconventionalExtensionClass();
9     }
10 }

```

Since the new Extension class name doesn't follow the naming conventions, you should also override *Extension::getAlias()*⁴ to return the correct DI alias. The DI alias is the name used to refer to the bundle in the container (e.g. in the `app/config/config.yml` file). By default, this is done by removing the `Extension` suffix and converting the class name to underscores (e.g. `AcmeHelloExtension`'s DI alias is `acme_hello`).

Using the `load()` Method

In the `load()` method, all services and parameters related to this extension will be loaded. This method doesn't get the actual container instance, but a copy. This container only has the parameters from the actual container. After loading the services and parameters, the copy will be merged into the actual container, to ensure all services and parameters are also added to the actual container.

In the `load()` method, you can use PHP code to register service definitions, but it is more common if you put these definitions in a configuration file (using the Yaml, XML or PHP format). Luckily, you can use the file loaders in the extension!

For instance, assume you have a file called `services.xml` in the `Resources/config` directory of your bundle, your load method looks like:

Listing 12-3

```

1 use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;
2 use Symfony\Component\Config\FileLocator;
3
4 // ...
5 public function load(array $configs, ContainerBuilder $container)
6 {
7     $loader = new XmlFileLoader(
8         $container,
9         new FileLocator(__DIR__.'/../Resources/config')
10    );
11    $loader->load('services.xml');
12 }

```

3. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Bundle/Bundle.html#method_build

4. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Extension/Extension.html#method_getAlias

Other available loaders are the `YamlFileLoader`, `PhpFileLoader` and `IniFileLoader`.



The `IniFileLoader` can only be used to load parameters and it can only load them as strings.



If you removed the default file with service definitions (i.e. `app/config/services.yml`), make sure to also remove it from the `imports` key in `app/config/config.yml`.

Using Configuration to Change the Services

The `Extension` is also the class that handles the configuration for that particular bundle (e.g. the configuration in `app/config/config.yml`). To read more about it, see the *"How to Create Friendly Configuration for a Bundle"* article.

Adding Classes to Compile

Symfony creates a big `classes.php` file in the cache directory to aggregate the contents of the PHP classes that are used in every request. This reduces the I/O operations and increases the application performance.

Your bundles can also add their own classes into this file thanks to the `addClassesToCompile()` method. Define the classes to compile as an array of their fully qualified class names:

Listing 12-4

```
1  // ...
2  public function load(array $configs, ContainerBuilder $container)
3  {
4      // ...
5
6      $this->addClassesToCompile(array(
7          'AppBundle\\Manager\\UserManager',
8          'AppBundle\\Utils\\Slugger',
9          // ...
10     ));
11 }
```



If some class extends from other classes, all its parents are automatically included in the list of classes to compile.

Beware that this technique **can't be used in some cases**:

- When classes contain annotations, such as controllers with `@Route` annotations and entities with `@ORM` or `@Assert` annotations, because the file location retrieved from PHP reflection changes;
- When classes use the `__DIR__` and `__FILE__` constants, because their values will change when loading these classes from the `classes.php` file.



Chapter 13

How to Create Friendly Configuration for a Bundle

If you open your application configuration file (usually `app/config/config.yml`), you'll see a number of different configuration sections, such as `framework`, `twig` and `doctrine`. Each of these configures a specific bundle, allowing you to define options at a high level and then let the bundle make all the low-level, complex changes based on your settings.

For example, the following configuration tells the FrameworkBundle to enable the form integration, which involves the definition of quite a few services as well as integration of other related components:

Listing 13-1

```
1 framework:
2     form: true
```



Using Parameters to Configure your Bundle

If you don't have plans to share your bundle between projects, it doesn't make sense to use this more advanced way of configuration. Since you use the bundle only in one project, you can just change the service configuration each time.

If you *do* want to be able to configure something from within `config.yml`, you can always create a parameter there and use that parameter somewhere else.

Using the Bundle Extension

The basic idea is that instead of having the user override individual parameters, you let the user configure just a few, specifically created, options. As the bundle developer, you then parse through that configuration and load correct services and parameters inside an "Extension" class.

As an example, imagine you are creating a social bundle, which provides integration with Twitter and such. To be able to reuse your bundle, you have to make the `client_id` and `client_secret` variables configurable. Your bundle configuration would look like:

Listing 13-2

```

1 # app/config/config.yml
2 acme_social:
3     twitter:
4         client_id: 123
5         client_secret: your_secret

```

Read more about the extension in *How to Load Service Configuration inside a Bundle*.



If a bundle provides an Extension class, then you should *not* generally override any service container parameters from that bundle. The idea is that if an Extension class is present, every setting that should be configurable should be present in the configuration made available by that class. In other words, the extension class defines all the public configuration settings for which backward compatibility will be maintained.

For parameter handling within a dependency injection container see *Using Parameters within a Dependency Injection Class*.

Processing the \$configs Array

First things first, you have to create an extension class as explained in *How to Load Service Configuration inside a Bundle*.

Whenever a user includes the `acme_social` key (which is the DI alias) in a configuration file, the configuration under it is added to an array of configurations and passed to the `load()` method of your extension (Symfony automatically converts XML and YAML to an array).

For the configuration example in the previous section, the array passed to your `load()` method will look like this:

Listing 13-3

```

1 array(
2     array(
3         'twitter' => array(
4             'client_id' => 123,
5             'client_secret' => 'your_secret',
6         ),
7     ),
8 )

```

Notice that this is an *array of arrays*, not just a single flat array of the configuration values. This is intentional, as it allows Symfony to parse several configuration resources. For example, if `acme_social` appears in another configuration file - say `config_dev.yml` - with different values beneath it, the incoming array might look like this:

Listing 13-4

```

1 array(
2     // values from config.yml
3     array(
4         'twitter' => array(
5             'client_id' => 123,
6             'client_secret' => 'your_secret',
7         ),
8     ),
9     // values from config_dev.yml
10    array(
11        'twitter' => array(
12            'client_id' => 456,
13        ),
14    ),
15 )

```

The order of the two arrays depends on which one is set first.

But don't worry! Symfony's Config component will help you merge these values, provide defaults and give the user validation errors on bad configuration. Here's how it works. Create a **Configuration** class in the **DependencyInjection** directory and build a tree that defines the structure of your bundle's configuration.

The **Configuration** class to handle the sample configuration looks like:

Listing 13-5

```
1 // src/Acme/SocialBundle/DependencyInjection/Configuration.php
2 namespace Acme\SocialBundle\DependencyInjection;
3
4 use Symfony\Component\Config\Definition\Builder\TreeBuilder;
5 use Symfony\Component\Config\Definition\ConfigurationInterface;
6
7 class Configuration implements ConfigurationInterface
8 {
9     public function getConfigTreeBuilder()
10     {
11         $treeBuilder = new TreeBuilder();
12         $rootNode = $treeBuilder->root('acme_social');
13
14         $rootNode
15             ->children()
16                 ->arrayNode('twitter')
17                     ->children()
18                         ->integerNode('client_id')->end()
19                         ->scalarNode('client_secret')->end()
20                     ->end()
21                 ->end() // twitter
22             ->end()
23         ;
24
25         return $treeBuilder;
26     }
27 }
```

The Configuration class can be much more complicated than shown here, supporting "prototype" nodes, advanced validation, XML-specific normalization and advanced merging. You can read more about this in the Config component documentation. You can also see it in action by checking out some core Configuration classes, such as the one from the FrameworkBundle Configuration¹ or the TwigBundle Configuration².

This class can now be used in your **load()** method to merge configurations and force validation (e.g. if an additional option was passed, an exception will be thrown):

Listing 13-6

```
1 public function load(array $configs, ContainerBuilder $container)
2 {
3     $configuration = new Configuration();
4
5     $config = $this->processConfiguration($configuration, $configs);
6     // ...
7 }
```

The **processConfiguration()** method uses the configuration tree you've defined in the **Configuration** class to validate, normalize and merge all the configuration arrays together.

1. <https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/FrameworkBundle/DependencyInjection/Configuration.php>
2. <https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/TwigBundle/DependencyInjection/Configuration.php>



Instead of calling `processConfiguration()` in your extension each time you provide some configuration options, you might want to use the *ConfigurableExtension*³ to do this automatically for you:

Listing 13-7

```
1 // src/Acme/HelloBundle/DependencyInjection/AcmeHelloExtension.php
2 namespace Acme\HelloBundle\DependencyInjection;
3
4 use Symfony\Component\DependencyInjection\ContainerBuilder;
5 use Symfony\Component\HttpKernel\DependencyInjection\ConfigurableExtension;
6
7 class AcmeHelloExtension extends ConfigurableExtension
8 {
9     // note that this method is called loadInternal and not load
10    protected function loadInternal(array $mergedConfig, ContainerBuilder $container)
11    {
12        // ...
13    }
14 }
```

This class uses the `getConfiguration()` method to get the Configuration instance. You should override it if your Configuration class is not called **Configuration** or if it is not placed in the same namespace as the extension.



Processing the Configuration yourself

Using the Config component is fully optional. The `load()` method gets an array of configuration values. You can simply parse these arrays yourself (e.g. by overriding configurations and using *isset*⁴ to check for the existence of a value). Be aware that it'll be very hard to support XML.

Listing 13-8

```
1 public function load(array $configs, ContainerBuilder $container)
2 {
3     $config = array();
4     // let resources override the previous set value
5     foreach ($configs as $subConfig) {
6         $config = array_merge($config, $subConfig);
7     }
8
9     // ... now use the flat $config array
10 }
```

Modifying the Configuration of Another Bundle

If you have multiple bundles that depend on each other, it may be useful to allow one **Extension** class to modify the configuration passed to another bundle's **Extension** class, as if the end-developer has actually placed that configuration in their `app/config/config.yml` file. This can be achieved using a prepend extension. For more details, see *How to Simplify Configuration of multiple Bundles*.

Dump the Configuration

The `config:dump-reference` command dumps the default configuration of a bundle in the console using the Yaml format.

3. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/DependencyInjection/ConfigurableExtension.html>

4. <http://php.net/manual/en/function.isset.php>

As long as your bundle's configuration is located in the standard location (`YourBundle\DependencyInjection\Configuration`) and does not have a constructor it will work automatically. If you have something different, your `Extension` class must override the `Extension::getConfiguration()`⁵ method and return an instance of your `Configuration`.

Supporting XML

Symfony allows people to provide the configuration in three different formats: Yaml, XML and PHP. Both Yaml and PHP use the same syntax and are supported by default when using the Config component. Supporting XML requires you to do some more things. But when sharing your bundle with others, it is recommended that you follow these steps.

Make your Config Tree ready for XML

The Config component provides some methods by default to allow it to correctly process XML configuration. See "Normalization" of the component documentation. However, you can do some optional things as well, this will improve the experience of using XML configuration:

Choosing an XML Namespace

In XML, the *XML namespace*⁶ is used to determine which elements belong to the configuration of a specific bundle. The namespace is returned from the `Extension::getNamespace()`⁷ method. By convention, the namespace is a URL (it doesn't have to be a valid URL nor does it need to exist). By default, the namespace for a bundle is `http://example.org/dic/schema/DI_ALIAS`, where `DI_ALIAS` is the DI alias of the extension. You might want to change this to a more professional URL:

Listing 13-9

```
1 // src/Acme/HelloBundle/DependencyInjection/AcmeHelloExtension.php
2
3 // ...
4 class AcmeHelloExtension extends Extension
5 {
6     // ...
7
8     public function getNamespace()
9     {
10         return 'http://acme_company.com/schema/dic/hello';
11     }
12 }
```

Providing an XML Schema

XML has a very useful feature called *XML schema*⁸. This allows you to describe all possible elements and attributes and their values in an XML Schema Definition (an `xsd` file). This XSD file is used by IDEs for auto completion and it is used by the Config component to validate the elements.

In order to use the schema, the XML configuration file must provide an `xsi:schemaLocation` attribute pointing to the XSD file for a certain XML namespace. This location always starts with the XML namespace. This XML namespace is then replaced with the XSD validation base path returned from `Extension::getXsdValidationBasePath()`⁹ method. This namespace is then followed by the rest of the path from the base path to the file itself.

5. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/DependencyInjection/Extension.html#method_getConfiguration

6. https://en.wikipedia.org/wiki/XML_namespace

7. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Extension/Extension.html#method_getNamespace

8. https://en.wikipedia.org/wiki/XML_schema

9. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Extension/ExtensionInterface.html#method_getXsdValidationBasePath

By convention, the XSD file lives in the **Resources/config/schema**, but you can place it anywhere you like. You should return this path as the base path:

```
Listing 13-10 1 // src/Acme/HelloBundle/DependencyInjection/AcmeHelloExtension.php
2
3 // ...
4 class AcmeHelloExtension extends Extension
5 {
6     // ...
7
8     public function getXsdValidationBasePath()
9     {
10         return __DIR__.'../Resources/config/schema';
11     }
12 }
```

Assuming the XSD file is called **hello-1.0.xsd**, the schema location will be **http://acme_company.com/schema/dic/hello/hello-1.0.xsd**:

```
Listing 13-11 1 <!-- app/config/config.xml -->
2 <?xml version="1.0" ?>
3
4 <container xmlns="http://symfony.com/schema/dic/services"
5     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6     xmlns:acme-hello="http://acme_company.com/schema/dic/hello"
7     xsi:schemaLocation="http://acme_company.com/schema/dic/hello
8         http://acme_company.com/schema/dic/hello/hello-1.0.xsd">
9
10     <acme-hello:config>
11         <!-- ... -->
12     </acme-hello:config>
13
14     <!-- ... -->
15 </container>
```



Chapter 14

How to Simplify Configuration of multiple Bundles

When building reusable and extensible applications, developers are often faced with a choice: either create a single large bundle or multiple smaller bundles. Creating a single bundle has the drawback that it's impossible for users to choose to remove functionality they are not using. Creating multiple bundles has the drawback that configuration becomes more tedious and settings often need to be repeated for various bundles.

Using the below approach, it is possible to remove the disadvantage of the multiple bundle approach by enabling a single Extension to prepend the settings for any bundle. It can use the settings defined in the `app/config/config.yml` to prepend settings just as if they had been written explicitly by the user in the application configuration.

For example, this could be used to configure the entity manager name to use in multiple bundles. Or it can be used to enable an optional feature that depends on another bundle being loaded as well.

To give an Extension the power to do this, it needs to implement *PrependExtensionInterface*¹:

Listing 14-1

```
1 // src/Acme/HelloBundle/DependencyInjection/AcmeHelloExtension.php
2 namespace Acme\HelloBundle\DependencyInjection;
3
4 use Symfony\Component\HttpKernel\DependencyInjection\Extension;
5 use Symfony\Component\DependencyInjection\Extension\PrependExtensionInterface;
6 use Symfony\Component\DependencyInjection\ContainerBuilder;
7
8 class AcmeHelloExtension extends Extension implements PrependExtensionInterface
9 {
10     // ...
11
12     public function prepend(ContainerBuilder $container)
13     {
14         // ...
15     }
16 }
```

1. <http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Extension/PrependExtensionInterface.html>

Inside the *prepend()*² method, developers have full access to the *ContainerBuilder*³ instance just before the *load()*⁴ method is called on each of the registered bundle Extensions. In order to prepend settings to a bundle extension developers can use the *prependExtensionConfig()*⁵ method on the *ContainerBuilder*⁶ instance. As this method only prepends settings, any other settings done explicitly inside the `app/config/config.yml` would override these prepended settings.

The following example illustrates how to prepend a configuration setting in multiple bundles as well as disable a flag in multiple bundles in case a specific other bundle is not registered:

Listing 14-2

```

1 public function prepend(ContainerBuilder $container)
2 {
3     // get all bundles
4     $bundles = $container->getParameter('kernel.bundles');
5     // determine if AcmeGoodbyeBundle is registered
6     if (!isset($bundles['AcmeGoodbyeBundle'])) {
7         // disable AcmeGoodbyeBundle in bundles
8         $config = array('use_acme_goodbye' => false);
9         foreach ($container->getExtensions() as $name => $extension) {
10             switch ($name) {
11                 case 'acme_something':
12                 case 'acme_other':
13                     // set use_acme_goodbye to false in the config of
14                     // acme_something and acme_other note that if the user manually
15                     // configured use_acme_goodbye to true in the app/config/config.yml
16                     // then the setting would in the end be true and not false
17                     $container->prependExtensionConfig($name, $config);
18                     break;
19             }
20         }
21     }
22
23     // process the configuration of AcmeHelloExtension
24     $configs = $container->getExtensionConfig($this->getAlias());
25     // use the Configuration class to generate a config array with
26     // the settings "acme_hello"
27     $config = $this->processConfiguration(new Configuration(), $configs);
28
29     // check if entity_manager_name is set in the "acme_hello" configuration
30     if (isset($config['entity_manager_name'])) {
31         // prepend the acme_something settings with the entity_manager_name
32         $config = array('entity_manager_name' => $config['entity_manager_name']);
33         $container->prependExtensionConfig('acme_something', $config);
34     }
35 }
```

The above would be the equivalent of writing the following into the `app/config/config.yml` in case `AcmeGoodbyeBundle` is not registered and the `entity_manager_name` setting for `acme_hello` is set to `non_default`:

Listing 14-3

```

1 # app/config/config.yml
2 acme_something:
3     # ...
4     use_acme_goodbye: false
5     entity_manager_name: non_default
6
7 acme_other:
8     # ...
9     use_acme_goodbye: false
```

-
2. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Extension/PrependExtensionInterface.html#method_prepend
 3. <http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/ContainerBuilder.html>
 4. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Extension/ExtensionInterface.html#method_load
 5. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/ContainerBuilder.html#method_prependExtensionConfig
 6. <http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/ContainerBuilder.html>



Chapter 15

How to Use Varnish to Speed up my Website

Because Symfony's cache uses the standard HTTP cache headers, the Symfony Reverse Proxy can easily be replaced with any other reverse proxy. *Varnish*¹ is a powerful, open-source, HTTP accelerator capable of serving cached content fast and including support for Edge Side Includes.

Make Symfony Trust the Reverse Proxy

Varnish automatically forwards the IP as **X-Forwarded-For** and leaves the **X-Forwarded-Proto** header in the request. If you do not configure Varnish as trusted proxy, Symfony will see all requests as coming through insecure HTTP connections from the Varnish host instead of the real client.

Remember to configure `framework.trusted_proxies` in the Symfony configuration so that Varnish is seen as a trusted proxy and the X-Forwarded headers are used.

Varnish, in its default configuration, sends the **X-Forwarded-For** header but does not filter out the **Forwarded** header. If you have access to the Varnish configuration file, you can configure Varnish to remove the **Forwarded** header:

Listing 15-1

```
1 sub vcl_recv {
2     remove req.http.Forwarded;
3 }
```

If you do not have access to your Varnish configuration, you can instead configure Symfony to distrust the **Forwarded** header as detailed in the cookbook.

Routing and X-FORWARDED Headers

To ensure that the Symfony Router generates URLs correctly with Varnish, an **X-Forwarded-Port** header must be present for Symfony to use the correct port number.

This port number corresponds to the port your setup is using to receive external connections (**80** is the default value for HTTP connections). If the application also accepts HTTPS connections, there could be

1. <https://www.varnish-cache.org>

another proxy (as Varnish does not do HTTPS itself) on the default HTTPS port 443 that handles the SSL termination and forwards the requests as HTTP requests to Varnish with an **X-Forwarded-Proto** header. In this case, you need to add the following configuration snippet:

Listing 15-2

```
1 sub vcl_recv {
2     if (req.http.X-Forwarded-Proto == "https" ) {
3         set req.http.X-Forwarded-Port = "443";
4     } else {
5         set req.http.X-Forwarded-Port = "80";
6     }
7 }
```

Cookies and Caching

By default, a sane caching proxy does not cache anything when a request is sent with cookies or a basic authentication header. This is because the content of the page is supposed to depend on the cookie value or authentication header.

If you know for sure that the backend never uses sessions or basic authentication, have Varnish remove the corresponding header from requests to prevent clients from bypassing the cache. In practice, you will need sessions at least for some parts of the site, e.g. when using forms with CSRF Protection. In this situation, make sure to *only start a session when actually needed* and clear the session when it is no longer needed. Alternatively, you can look into *Caching Pages that Contain CSRF Protected Forms*.

Cookies created in JavaScript and used only in the frontend, e.g. when using Google Analytics, are nonetheless sent to the server. These cookies are not relevant for the backend and should not affect the caching decision. Configure your Varnish cache to *clean the cookies header*². You want to keep the session cookie, if there is one, and get rid of all other cookies so that pages are cached if there is no active session. Unless you changed the default configuration of PHP, your session cookie has the name **PHPSESSID**:

Listing 15-3

```
1 sub vcl_recv {
2     // Remove all cookies except the session ID.
3     if (req.http.Cookie) {
4         set req.http.Cookie = ";" + req.http.Cookie;
5         set req.http.Cookie = regsuball(req.http.Cookie, "; +", ";");
6         set req.http.Cookie = regsuball(req.http.Cookie, "(PHPSESSID)=", "; \1=");
7         set req.http.Cookie = regsuball(req.http.Cookie, "[^ ]*[;]*", "");
8         set req.http.Cookie = regsuball(req.http.Cookie, "^[; ]+|$", "");
9
10        if (req.http.Cookie == "") {
11            // If there are no more cookies, remove the header to get page cached.
12            unset req.http.Cookie;
13        }
14    }
15 }
```



If content is not different for every user, but depends on the roles of a user, a solution is to separate the cache per group. This pattern is implemented and explained by the *FOSHttpCacheBundle*³ under the name *User Context*⁴.

2. <https://www.varnish-cache.org/trac/wiki/VCLExampleRemovingSomeCookies>

3. <http://foshttpcachebundle.readthedocs.org/>

4. <http://foshttpcachebundle.readthedocs.org/en/latest/features/user-context.html>

Ensure Consistent Caching Behavior

Varnish uses the cache headers sent by your application to determine how to cache content. However, versions prior to Varnish 4 did not respect **Cache-Control: no-cache, no-store** and **private**. To ensure consistent behavior, use the following configuration if you are still using Varnish 3:

Listing 15-4

```
1 sub vcl_fetch {
2     /* By default, Varnish3 ignores Cache-Control: no-cache and private
3        https://www.varnish-cache.org/docs/3.0/tutorial/increasing_your_hitrate.html#cache-control
4     */
5     if (beresp.http.Cache-Control ~ "private" ||
6         beresp.http.Cache-Control ~ "no-cache" ||
7         beresp.http.Cache-Control ~ "no-store") {
8     } {
9         return (hit_for_pass);
10    }
11 }
```



You can see the default behavior of Varnish in the form of a VCL file: *default.vcl*⁵ for Varnish 3, *builtin.vcl*⁶ for Varnish 4.

Enable Edge Side Includes (ESI)

As explained in the Edge Side Includes section, Symfony detects whether it talks to a reverse proxy that understands ESI or not. When you use the Symfony reverse proxy, you don't need to do anything. But to make Varnish instead of Symfony resolve the ESI tags, you need some configuration in Varnish. Symfony uses the **Surrogate-Capability** header from the *Edge Architecture*⁷ described by Akamai.



Varnish only supports the **src** attribute for ESI tags (**onerror** and **alt** attributes are ignored).

First, configure Varnish so that it advertises its ESI support by adding a **Surrogate-Capability** header to requests forwarded to the backend application:

Listing 15-5

```
1 sub vcl_recv {
2     // Add a Surrogate-Capability header to announce ESI support.
3     set req.http.Surrogate-Capability = "abc=ESI/1.0";
4 }
```



The **abc** part of the header isn't important unless you have multiple "surrogates" that need to advertise their capabilities. See *Surrogate-Capability Header*⁸ for details.

Then, optimize Varnish so that it only parses the response contents when there is at least one ESI tag by checking the **Surrogate-Control** header that Symfony adds automatically:

Listing 15-6

```
1 sub vcl_backend_response {
2     // Check for ESI acknowledgement and remove Surrogate-Control header
```

5. <https://github.com/varnish/Varnish-Cache/blob/3.0/bin/varnishd/default.vcl>

6. <https://github.com/varnish/Varnish-Cache/blob/4.1/bin/varnishd/builtin.vcl>

7. <http://www.w3.org/TR/edge-arch>

8. <http://www.w3.org/TR/edge-arch>

```
3  if (beresp.http.Surrogate-Control ~ "ESI/1.0") {  
4      unset beresp.http.Surrogate-Control;  
5      set beresp.do_esi = true;  
6  }  
7  }
```



If you followed the advice about ensuring a consistent caching behavior, those VCL functions already exist. Just append the code to the end of the function, they won't interfere with each other.

Cache Invalidation

If you want to cache content that changes frequently and still serve the most recent version to users, you need to invalidate that content. While *cache invalidation*⁹ allows you to purge content from your proxy before it has expired, it adds complexity to your caching setup.



The open source *FOSHtpCacheBundle*¹⁰ takes the pain out of cache invalidation by helping you to organize your caching and invalidation setup.

The documentation of the *FOSHtpCacheBundle*¹¹ explains how to configure Varnish and other reverse proxies for cache invalidation.

9. <http://tools.ietf.org/html/rfc2616#section-13.10>

10. <http://foshttpcachebundle.readthedocs.org/>

11. <http://foshttpcachebundle.readthedocs.org/>



Chapter 16

Caching Pages that Contain CSRF Protected Forms

CSRF tokens are meant to be different for every user. This is why you need to be cautious if you try to cache pages with forms including them.

For more information about how CSRF protection works in Symfony, please check [CSRF Protection](#).

Why Caching Pages with a CSRF token is Problematic

Typically, each user is assigned a unique CSRF token, which is stored in the session for validation. This means that if you *do* cache a page with a form containing a CSRF token, you'll cache the CSRF token of the *first* user only. When a user submits the form, the token won't match the token stored in the session and all users (except for the first) will fail CSRF validation when submitting the form.

In fact, many reverse proxies (like Varnish) will refuse to cache a page with a CSRF token. This is because a cookie is sent in order to preserve the PHP session open and Varnish's default behavior is to not cache HTTP requests with cookies.

How to Cache Most of the Page and still be able to Use CSRF Protection

To cache a page that contains a CSRF token, you can use more advanced caching techniques like ESI fragments, where you cache the full page and embedding the form inside an ESI tag with no cache at all.

Another option would be to load the form via an uncached AJAX request, but cache the rest of the HTML response.

Or you can even load just the CSRF token with an AJAX request and replace the form field value with it.



Chapter 17

Installing Composer

*Composer*¹ is the package manager used by modern PHP applications. Use Composer to manage dependencies in your Symfony applications and to install Symfony Components in your PHP projects. It's recommended to install Composer globally in your system as explained in the following sections.

Install Composer on Linux and Mac OS X

1. Run the installer as described in *the official Composer documentation*²;
2. Execute the following command to move the **composer.phar** to a directory that is in your path:

Listing 17-1 1 \$ sudo mv composer.phar /usr/local/bin/composer

Install Composer on Windows

Download the installer from *getcomposer.org*³, execute it and follow the instructions.

Learn more

Read the *Composer documentation*⁴ to learn more about its usage and features.

1. <https://getcomposer.org/>

2. <https://getcomposer.org/download>

3. <https://getcomposer.org/Composer-Setup.exe>

4. <https://getcomposer.org/doc/00-intro.md>



Chapter 18

How to Master and Create new Environments

Every application is the combination of code and a set of configuration that dictates how that code should function. The configuration may define the database being used, if something should be cached or how verbose logging should be.

In Symfony, the idea of "environments" is the idea that the same codebase can be run using multiple different configurations. For example, the **dev** environment should use configuration that makes development easy and friendly, while the **prod** environment should use a set of configuration optimized for speed.

Different Environments, different Configuration Files

A typical Symfony application begins with three environments: **dev**, **prod**, and **test**. As mentioned, each environment simply represents a way to execute the same codebase with different configuration. It should be no surprise then that each environment loads its own individual configuration file. If you're using the YAML configuration format, the following files are used:

- for the dev environment: `app/config/config_dev.yml`
- for the prod environment: `app/config/config_prod.yml`
- for the test environment: `app/config/config_test.yml`

This works via a simple standard that's used by default inside the **AppKernel** class:

Listing 18-1

```
1  // app/AppKernel.php
2
3  // ...
4
5  class AppKernel extends Kernel
6  {
7      // ...
8
9      public function registerContainerConfiguration(LoaderInterface $loader)
10     {
11         $loader->load($this->getRootDir().'/config/config_'.$this->getEnvironment().'.yml');
12     }
13 }
```

As you can see, when Symfony is loaded, it uses the given environment to determine which configuration file to load. This accomplishes the goal of multiple environments in an elegant, powerful and transparent way.

Of course, in reality, each environment differs only somewhat from others. Generally, all environments will share a large base of common configuration. Opening the **config_dev.yml** configuration file, you can see how this is accomplished easily and transparently:

Listing 18-2

```
1 imports:
2   - { resource: config.yml }
3
4 # ...
```

To share common configuration, each environment's configuration file simply first imports from a central configuration file (**config.yml**). The remainder of the file can then deviate from the default configuration by overriding individual parameters. For example, by default, the **web_profiler** toolbar is disabled. However, in the **dev** environment, the toolbar is activated by modifying the value of the **toolbar** option in the **config_dev.yml** configuration file:

Listing 18-3

```
1 # app/config/config_dev.yml
2 imports:
3   - { resource: config.yml }
4
5 web_profiler:
6   toolbar: true
7   # ...
```

Executing an Application in different Environments

To execute the application in each environment, load up the application using either the **app.php** (for the **prod** environment) or the **app_dev.php** (for the **dev** environment) front controller:

Listing 18-4

```
1 http://localhost/app.php      -> *prod* environment
2 http://localhost/app_dev.php -> *dev* environment
```



The given URLs assume that your web server is configured to use the **web/** directory of the application as its root. Read more in *Installing Symfony*.

If you open up one of these files, you'll quickly see that the environment used by each is explicitly set:

Listing 18-5

```
1 // web/app.php
2 // ...
3
4 $kernel = new AppKernel('prod', false);
5
6 // ...
```

The **prod** key specifies that this application will run in the **prod** environment. A Symfony application can be executed in any environment by using this code and changing the environment string.



The **test** environment is used when writing functional tests and is not accessible in the browser directly via a front controller. In other words, unlike the other environments, there is no **app_test.php** front controller file.



Debug Mode

Important, but unrelated to the topic of *environments* is the **false** argument as the second argument to the **AppKernel** constructor. This specifies if the application should run in "debug mode". Regardless of the environment, a Symfony application can be run with debug mode set to **true** or **false**. This affects many things in the application, such as displaying stacktraces on error pages or if cache files are dynamically rebuilt on each request. Though not a requirement, debug mode is generally set to **true** for the **dev** and **test** environments and **false** for the **prod** environment.

Internally, the value of the debug mode becomes the **kernel.debug** parameter used inside the *service container*. If you look inside the application configuration file, you'll see the parameter used, for example, to turn logging on or off when using the Doctrine DBAL:

Listing 18-6

```

1 doctrine:
2     dbal:
3         logging: '%kernel.debug%'
4         # ...
```

Selecting the Environment for Console Commands

By default, Symfony commands are executed in the **dev** environment and with the debug mode enabled. Use the **--env** and **--no-debug** options to modify this behavior:

Listing 18-7

```

1 # 'dev' environment and debug enabled
2 $ php bin/console command_name
3
4 # 'prod' environment (debug is always disabled for 'prod')
5 $ php bin/console command_name --env=prod
6
7 # 'test' environment and debug disabled
8 $ php bin/console command_name --env=test --no-debug
```

In addition to the **--env** and **--debug** options, the behavior of Symfony commands can also be controlled with environment variables. The Symfony console application checks the existence and value of these environment variables before executing any command:

SYMFONY_ENV

Sets the execution environment of the command to the value of this variable (dev, prod, test, etc.);

SYMFONY_DEBUG

If 0, debug mode is disabled. Otherwise, debug mode is enabled.

These environment variables are very useful for production servers because they allow you to ensure that commands always run in the **prod** environment without having to add any command option.

Creating a new Environment

By default, a Symfony application has three environments that handle most cases. Of course, since an environment is nothing more than a string that corresponds to a set of configuration, creating a new environment is quite easy.

Suppose, for example, that before deployment, you need to benchmark your application. One way to benchmark the application is to use near-production settings, but with Symfony's **web_profiler** enabled. This allows Symfony to record information about your application while benchmarking.

The best way to accomplish this is via a new environment called, for example, **benchmark**. Start by creating a new configuration file:

Listing 18-8

```

1 # app/config/config_benchmark.yml
2 imports:
3   - { resource: config_prod.yml }
4
5 framework:
6   profiler: { only_exceptions: false }

```



Due to the way in which parameters are resolved, you cannot use them to build paths in imports dynamically. This means that something like the following doesn't work:

Listing 18-9

```

1 # app/config/config.yml
2 imports:
3   - { resource: '%kernel.root_dir%/parameters.yml' }

```

And with this simple addition, the application now supports a new environment called **benchmark**.

This new configuration file imports the configuration from the **prod** environment and modifies it. This guarantees that the new environment is identical to the **prod** environment, except for any changes explicitly made here.

Because you'll want this environment to be accessible via a browser, you should also create a front controller for it. Copy the **web/app.php** file to **web/app_benchmark.php** and edit the environment to be **benchmark**:

Listing 18-10

```

1 // web/app_benchmark.php
2 // ...
3
4 // change just this line
5 $kernel = new AppKernel('benchmark', false);
6
7 // ...

```

The new environment is now accessible via:

Listing 18-11 http://localhost/app_benchmark.php



Some environments, like the **dev** environment, are never meant to be accessed on any deployed server by the public. This is because certain environments, for debugging purposes, may give too much information about the application or underlying infrastructure. To be sure these environments aren't accessible, the front controller is usually protected from external IP addresses via the following code at the top of the controller:

Listing 18-12

```

if (!in_array(@$_SERVER['REMOTE_ADDR'], array('127.0.0.1', '::1'))) {
    die('You are not allowed to access this file. Check '.__FILE__.' for more information.');
```

Environments and the Cache Directory

Symfony takes advantage of caching in many ways: the application configuration, routing configuration, Twig templates and more are cached to PHP objects stored in files on the filesystem.

By default, these cached files are largely stored in the **var/cache** directory. However, each environment caches its own set of files:

Listing 18-13

```

1 your-project/
2 └─ var/

```

```

3 | | cache/
4 | | └─ dev/  # cache directory for the *dev* environment
5 | | └─ prod/ # cache directory for the *prod* environment
6 | | └─ ...

```

Sometimes, when debugging, it may be helpful to inspect a cached file to understand how something is working. When doing so, remember to look in the directory of the environment you're using (most commonly **dev** while developing and debugging). While it can vary, the **var/cache/dev** directory includes the following:

appDevDebugProjectContainer.php

The cached "service container" that represents the cached application configuration.

appDevUrlGenerator.php

The PHP class generated from the routing configuration and used when generating URLs.

appDevUrlMatcher.php

The PHP class used for route matching - look here to see the compiled regular expression logic used to match incoming URLs to different routes.

twig/

This directory contains all the cached Twig templates.



You can easily change the directory location and name. For more information read the article *How to Override Symfony's default Directory Structure*.

Going further

Read the article on *How to Set external Parameters in the Service Container*.



Chapter 19

Building your own Framework with the MicroKernelTrait

A traditional Symfony app contains a sensible directory structure, various configuration files and an **AppKernel** with several bundles already-registered. This is a fully-featured app that's ready to go.

But did you know, you can create a fully-functional Symfony application in as little as one file? This is possible thanks to the new *MicroKernelTrait*¹. This allows you to start with a tiny application, and then add features and structure as you need to.

A Single-File Symfony Application

Start with a completely empty directory. Get **symfony/symfony** as a dependency via Composer:

Listing 19-1 `1 $ composer require symfony/symfony`

Next, create an **index.php** file that creates a kernel class and executes it:

Listing 19-2

```
1 use Symfony\Bundle\FrameworkBundle\Kernel\MicroKernelTrait;
2 use Symfony\Component\Config\Loader\LoaderInterface;
3 use Symfony\Component\DependencyInjection\ContainerBuilder;
4 use Symfony\Component\HttpFoundation\JsonResponse;
5 use Symfony\Component\HttpFoundation\Request;
6 use Symfony\Component\HttpKernel\Kernel;
7 use Symfony\Component\Routing\RouteCollectionBuilder;
8
9 // require Composer's autoloader
10 require __DIR__.'/vendor/autoload.php';
11
12 class AppKernel extends Kernel
13 {
14     use MicroKernelTrait;
15
16     public function registerBundles()
```

1. <http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Kernel/MicroKernelTrait.html>

```

17     {
18         return array(
19             new Symfony\Bundle\FrameworkBundle\FrameworkBundle()
20         );
21     }
22
23     protected function configureContainer(ContainerBuilder $c, LoaderInterface $loader)
24     {
25         // PHP equivalent of config.yml
26         $c->loadFromExtension('framework', array(
27             'secret' => 'SOME_SECRET'
28         ));
29     }
30
31     protected function configureRoutes(RouteCollectionBuilder $routes)
32     {
33         // kernel is a service that points to this class
34         // optional 3rd argument is the route name
35         $routes->add('/random/{limit}', 'kernel:randomAction');
36     }
37
38     public function randomAction($limit)
39     {
40         return new JsonResponse(array(
41             'number' => rand(0, $limit)
42         ));
43     }
44 }
45
46 $kernel = new AppKernel('dev', true);
47 $request = Request::createFromGlobals();
48 $response = $kernel->handle($request);
49 $response->send();
50 $kernel->terminate($request, $response);

```

That's it! To test it, you can start the built-in web server:

Listing 19-3 1 \$ php -S localhost:8000

Then see the JSON response in your browser:

> <http://localhost:8000/random/10>

The Methods of a "Micro" Kernel

When you use the `MicroKernelTrait`, your kernel needs to have exactly three methods that define your bundles, your services and your routes:

registerBundles()

This is the same `registerBundles()` that you see in a normal kernel.

configureContainer(ContainerBuilder \$c, LoaderInterface \$loader)

This methods builds and configures the container. In practice, you will use `loadFromExtension` to configure different bundles (this is the equivalent of what you see in a normal `config.yml` file). You can also register services directly in PHP or load external configuration files (shown below).

configureRoutes(RouteCollectionBuilder \$routes)

Your job in this method is to add routes to the application. The `RouteCollectionBuilder` has methods that make adding routes in PHP more fun. You can also load external routing files (shown below).

Advanced Example: Twig, Annotations and the Web Debug Toolbar

The purpose of the `MicroKernelTrait` is *not* to have a single-file application. Instead, its goal to give you the power to choose your bundles and structure.

First, you'll probably want to put your PHP classes in an `src/` directory. Configure your `composer.json` file to load from there:

Listing 19-4

```
1 {
2     "require": {
3         "...": "...",
4     },
5     "autoload": {
6         "psr-4": {
7             "": "src/"
8         }
9     }
10 }
```

Now, suppose you want to use Twig and load routes via annotations. For annotation routing, you need `SensioFrameworkExtraBundle`. This comes with a normal Symfony project. But in this case, you need to download it:

Listing 19-5

```
1 $ composer require sensio/framework-extra-bundle
```

Instead of putting *everything* in `index.php`, create a new `app/AppKernel.php` to hold the kernel. Now it looks like this:

Listing 19-6

```
1 // app/AppKernel.php
2
3 use Symfony\Bundle\FrameworkBundle\Kernel\MicroKernelTrait;
4 use Symfony\Component\Config\Loader\LoaderInterface;
5 use Symfony\Component\DependencyInjection\ContainerBuilder;
6 use Symfony\Component\HttpKernel\Kernel;
7 use Symfony\Component\Routing\RouteCollectionBuilder;
8 use Doctrine\Common\Annotations\AnnotationRegistry;
9
10 // require Composer's autoloader
11 $loader = require __DIR__.'/../vendor/autoload.php';
12 // auto-load annotations
13 AnnotationRegistry::registerLoader(array($loader, 'loadClass'));
14
15 class AppKernel extends Kernel
16 {
17     use MicroKernelTrait;
18
19     public function registerBundles()
20     {
21         $bundles = array(
22             new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
23             new Symfony\Bundle\TwigBundle\TwigBundle(),
24             new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle()
25         );
26
27         if ($this->getEnvironment() == 'dev') {
28             $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
29         }
30
31         return $bundles;
32     }
33
34     protected function configureContainer(ContainerBuilder $c, LoaderInterface $loader)
35     {
36         $loader->load(__DIR__.'/config/config.yml');
37
38         // configure WebProfilerBundle only if the bundle is enabled
```

```

39         if (isset($this->bundles['WebProfilerBundle'])) {
40             $c->loadFromExtension('web_profiler', array(
41                 'toolbar' => true,
42                 'intercept_redirects' => false,
43             ));
44         }
45     }
46
47     protected function configureRoutes(RouteCollectionBuilder $routes)
48     {
49         // import the WebProfilerRoutes, only if the bundle is enabled
50         if (isset($this->bundles['WebProfilerBundle'])) {
51             $routes->import('@WebProfilerBundle/Resources/config/routing/wdt.xml', '/_wdt');
52             $routes->import('@WebProfilerBundle/Resources/config/routing/profiler.xml', '/_profiler');
53         }
54
55         // load the annotation routes
56         $routes->import(__DIR__.'../src/App/Controller/', '/', 'annotation');
57     }
58 }

```

Unlike the previous kernel, this loads an external `app/config/config.yml` file, because the configuration started to get bigger:

Listing 19-7

```

1  # app/config/config.yml
2  framework:
3      secret: SOME_SECRET
4      templating:
5          engines: ['twig']
6      profiler: { only_exceptions: false }

```

This also loads annotation routes from an `src/App/Controller/` directory, which has one file in it:

Listing 19-8

```

1  // src/App/Controller/MicroController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6
7  class MicroController extends Controller
8  {
9      /**
10       * @Route("/random/{limit}")
11       */
12       public function randomAction($limit)
13       {
14           $number = rand(0, $limit);
15
16           return $this->render('micro/random.html.twig', array(
17               'number' => $number
18           ));
19       }
20 }

```

Template files should live in the `Resources/views` directory of whatever directory your *kernel* lives in. Since `AppKernel` lives in `app/`, this template lives at `app/Resources/views/micro/random.html.twig`.

Finally, you need a front controller to boot and run the application. Create a `web/index.php`:

Listing 19-9

```

1  // web/index.php
2
3  use Symfony\Component\HttpFoundation\Request;
4
5  require __DIR__.'../app/AppKernel.php';
6

```

```

7  $kernel = new AppKernel('dev', true);
8  $request = Request::createFromGlobals();
9  $response = $kernel->handle($request);
10 $response->send();
11 $kernel->terminate($request, $response);

```

That's it! This `/random/10` URL will work, Twig will render, and you'll even get the web debug toolbar to show up at the bottom. The final structure looks like this:

Listing 19-10

```

1  your-project/
2  └─ app/
3     └─ AppKernel.php
4     └─ cache/
5     └─ config/
6     └─ logs/
7     └─ Resources
8        └─ views
9           └─ base.html.twig
10          └─ micro
11             └─ random.html.twig
12 └─ src/
13    └─ App
14       └─ Controller
15          └─ MicroController.php
16 └─ vendor/
17    └─ ...
18 └─ web/
19    └─ index.php
20 └─ composer.json
21 └─ composer.lock

```

Hey, that looks a lot like a *traditional* Symfony application! You're right: the `MicroKernelTrait` is still Symfony: but you can control your structure and features quite easily.



Chapter 20

How to Override Symfony's default Directory Structure

Symfony automatically ships with a default directory structure. You can easily override this directory structure to create your own. The default directory structure is:

Listing 20-1

```
1 your-project/
2 | app/
3 | | config/
4 | | ...
5 | bin/
6 | | ...
7 | src/
8 | | ...
9 | tests/
10 | | ...
11 | var/
12 | | cache/
13 | | logs/
14 | | ...
15 | vendor/
16 | | ...
17 | web/
18 | | app.php
19 | | ...
```

Override the cache Directory

You can change the default cache directory by overriding the `getCacheDir` method in the `AppKernel` class of your application:

Listing 20-2

```
1 // app/AppKernel.php
2
3 // ...
4 class AppKernel extends Kernel
5 {
```



```

6      // ...
7
8      public function getCacheDir()
9      {
10         return dirname(__DIR__).'/var/'.$this->environment.'/cache';
11     }
12 }

```

In this code, `$this->environment` is the current environment (i.e. `dev`). In this case you have changed the location of the cache directory to `var/{environment}/cache`.



You should keep the **cache** directory different for each environment, otherwise some unexpected behavior may happen. Each environment generates its own cached configuration files, and so each needs its own directory to store those cache files.

Override the Logs Directory

Overriding the **logs** directory is the same as overriding the **cache** directory. The only difference is that you need to override the `getLogDir` method:

Listing 20-3

```

1  // app/AppKernel.php
2
3  // ...
4  class AppKernel extends Kernel
5  {
6      // ...
7
8      public function getLogDir()
9      {
10         return dirname(__DIR__).'/var/'.$this->environment.'/logs';
11     }
12 }

```

Here you have changed the location of the directory to `var/{environment}/logs`.

Override the web Directory

If you need to rename or move your **web** directory, the only thing you need to guarantee is that the path to the **var** directory is still correct in your `app.php` and `app_dev.php` front controllers. If you simply renamed the directory, you're fine. But if you moved it in some way, you may need to modify these paths inside those files:

Listing 20-4

```
require_once __DIR__.'/../path/to/app/autoload.php';
```

You also need to change the `extra.symfony-web-dir` option in the `composer.json` file:

Listing 20-5

```

1  {
2      "...": "...",
3      "extra": {
4          "...": "...",
5          "symfony-web-dir": "my_new_web_dir"
6      }
7  }

```



Some shared hosts have a **public_html** web directory root. Renaming your web directory from **web** to **public_html** is one way to make your Symfony project work on your shared host. Another way is to deploy your application to a directory outside of your web root, delete your **public_html** directory, and then replace it with a symbolic link to the **web** in your project.



If you use the AsseticBundle, you need to configure the **read_from** option to point to the correct **web** directory:

Listing 20-6

```
1 # app/config/config.yml
2
3 # ...
4 assetic:
5     # ...
6     read_from: '%kernel.root_dir%../../public_html'
```

Now you just need to clear the cache and dump the assets again and your application should work:

Listing 20-7

```
1 $ php bin/console cache:clear --env=prod
2 $ php bin/console assetic:dump --env=prod --no-debug
```

Override the **vendor** Directory

To override the **vendor** directory, you need to introduce changes in the **app/autoload.php** and **composer.json** files.

The change in the **composer.json** will look like this:

Listing 20-8

```
1 {
2     "config": {
3         "bin-dir": "bin",
4         "vendor-dir": "/some/dir/vendor"
5     },
6 }
```

Then, update the path to the **autoload.php** file in **app/autoload.php**:

Listing 20-9

```
// app/autoload.php
// ...
$loader = require '/some/dir/vendor/autoload.php';
```



This modification can be of interest if you are working in a virtual environment and cannot use NFS - for example, if you're running a Symfony application using Vagrant/VirtualBox in a guest operating system.



Chapter 21

Using Parameters within a Dependency Injection Class

You have seen how to use configuration parameters within Symfony service containers. There are special cases such as when you want, for instance, to use the `%kernel.debug%` parameter to make the services in your bundle enter debug mode. For this case there is more work to do in order to make the system understand the parameter value. By default, your parameter `%kernel.debug%` will be treated as a simple string. Consider the following example:

Listing 21-1

```
1 // inside Configuration class
2 $rootNode
3     ->children()
4         ->booleanNode('logging')->defaultValue('%kernel.debug%')->end()
5         // ...
6     ->end()
7 ;
8
9 // inside the Extension class
10 $config = $this->processConfiguration($configuration, $configs);
11 var_dump($config['logging']);
```

Now, examine the results to see this closely:

Listing 21-2

```
1 my_bundle:
2     logging: true
3     # true, as expected
4
5 my_bundle:
6     logging: '%kernel.debug%'
7     # true/false (depends on 2nd parameter of AppKernel),
8     # as expected, because %kernel.debug% inside configuration
9     # gets evaluated before being passed to the extension
10
11 my_bundle: ~
12 # passes the string "%kernel.debug%".
13 # Which is always considered as true.
14 # The Configurator does not know anything about
15 # "%kernel.debug%" being a parameter.
```

In order to support this use case, the **Configuration** class has to be injected with this parameter via the extension as follows:

```
Listing 21-3 1 namespace AppBundle\DependencyInjection;
2
3 use Symfony\Component\Config\Definition\Builder\TreeBuilder;
4 use Symfony\Component\Config\Definition\ConfigurationInterface;
5
6 class Configuration implements ConfigurationInterface
7 {
8     private $debug;
9
10    public function __construct($debug)
11    {
12        $this->debug = (bool) $debug;
13    }
14
15    public function getConfigTreeBuilder()
16    {
17        $treeBuilder = new TreeBuilder();
18        $rootNode = $treeBuilder->root('my_bundle');
19
20        $rootNode
21            ->children()
22                // ...
23                ->booleanNode('logging')->defaultValue($this->debug)->end()
24                // ...
25            ->end()
26        ;
27
28        return $treeBuilder;
29    }
30 }
```

And set it in the constructor of **Configuration** via the **Extension** class:

```
Listing 21-4 1 namespace AppBundle\DependencyInjection;
2
3 use Symfony\Component\DependencyInjection\ContainerBuilder;
4 use Symfony\Component\HttpKernel\DependencyInjection\Extension;
5
6 class AppExtension extends Extension
7 {
8     // ...
9
10    public function getConfiguration(array $config, ContainerBuilder $container)
11    {
12        return new Configuration($container->getParameter('kernel.debug'));
13    }
14 }
```



Setting the Default in the Extension

There are some instances of `%kernel.debug%` usage within a **Configurator** class in TwigBundle and AsseticBundle. However this is because the default parameter value is set by the Extension class. For example in AsseticBundle, you can find:

```
Listing 21-5 $container->setParameter('assetic.debug', $config['debug']);
```

The string `%kernel.debug%` passed here as an argument handles the interpreting job to the container which in turn does the evaluation. Both ways accomplish similar goals. AsseticBundle will not use `%kernel.debug%` but rather the new `%assetic.debug%` parameter.



Chapter 22

Understanding how the Front Controller, Kernel and Environments Work together

The section *How to Master and Create new Environments* explained the basics on how Symfony uses environments to run your application with different configuration settings. This section will explain a bit more in-depth what happens when your application is bootstrapped. To hook into this process, you need to understand three parts that work together:

- The Front Controller
- The Kernel Class
- The Environments



Usually, you will not need to define your own front controller or **AppKernel** class as the *Symfony Standard Edition*¹ provides sensible default implementations.

This documentation section is provided to explain what is going on behind the scenes.

The Front Controller

The *front controller*² is a well-known design pattern; it is a section of code that *all* requests served by an application run through.

In the *Symfony Standard Edition*³, this role is taken by the *app.php*⁴ and *app_dev.php*⁵ files in the **web/** directory. These are the very first PHP scripts executed when a request is processed.

The main purpose of the front controller is to create an instance of the **AppKernel** (more on that in a second), make it handle the request and return the resulting response to the browser.

1. <https://github.com/symfony/symfony-standard>

2. https://en.wikipedia.org/wiki/Front_Controller_pattern

3. <https://github.com/symfony/symfony-standard>

4. <https://github.com/symfony/symfony-standard/blob/master/web/app.php>

5. https://github.com/symfony/symfony-standard/blob/master/web/app_dev.php

Because every request is routed through it, the front controller can be used to perform global initialization prior to setting up the kernel or to *decorate*⁶ the kernel with additional features. Examples include:

- Configuring the autoloader or adding additional autoloading mechanisms;
- Adding HTTP level caching by wrapping the kernel with an instance of AppCache;
- Enabling (or skipping) the *ClassCache*;
- Enabling the *Debug Component*.

The front controller can be chosen by requesting URLs like:

Listing 22-1 1 `http://localhost/app_dev.php/some/path/...`

As you can see, this URL contains the PHP script to be used as the front controller. You can use that to easily switch the front controller or use a custom one by placing it in the `web/` directory (e.g. `app_cache.php`).

When using Apache and the *RewriteRule* shipped with the *Symfony Standard Edition*⁷, you can omit the filename from the URL and the *RewriteRule* will use `app.php` as the default one.



Pretty much every other web server should be able to achieve a behavior similar to that of the *RewriteRule* described above. Check your server documentation for details or see *Configuring a Web Server*.



Make sure you appropriately secure your front controllers against unauthorized access. For example, you don't want to make a debugging environment available to arbitrary users in your production environment.

Technically, the `bin/console`⁸ script used when running Symfony on the command line is also a front controller, only that is not used for web, but for command line requests.

The Kernel Class

The *Kernel*⁹ is the core of Symfony. It is responsible for setting up all the bundles that make up your application and providing them with the application's configuration. It then creates the service container before serving requests in its *handle()*¹⁰ method.

There are two methods declared in the *KernelInterface*¹¹ that are left unimplemented in *Kernel*¹² and thus serve as *template methods*¹³:

*registerBundles()*¹⁴

It must return an array of all bundles needed to run the application.

*registerContainerConfiguration()*¹⁵

It loads the application configuration.

6. https://en.wikipedia.org/wiki/Decorator_pattern

7. <https://github.com/symfony/symfony-standard/blob/master/web/.htaccess>

8. <https://github.com/symfony/symfony-standard/blob/master/bin/console>

9. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Kernel.html>

10. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/HttpKernelInterface.html#method_handle

11. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/KernelInterface.html>

12. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Kernel.html>

13. https://en.wikipedia.org/wiki/Template_method_pattern

14. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/KernelInterface.html#method_registerBundles

15. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/KernelInterface.html#method_registerContainerConfiguration

To fill these (small) blanks, your application needs to subclass the Kernel and implement these methods. The resulting class is conventionally called the **AppKernel**.

Again, the Symfony Standard Edition provides an *AppKernel*¹⁶ in the **app/** directory. This class uses the name of the environment - which is passed to the Kernel's *constructor*¹⁷ method and is available via *getEnvironment()*¹⁸ - to decide which bundles to create. The logic for that is in *registerBundles()*, a method meant to be extended by you when you start adding bundles to your application.

You are, of course, free to create your own, alternative or additional **AppKernel** variants. All you need is to adapt your (or add a new) front controller to make use of the new kernel.



The name and location of the **AppKernel** is not fixed. When putting multiple Kernels into a single application, it might therefore make sense to add additional sub-directories, for example **app/admin/AdminKernel.php** and **app/api/ApiKernel.php**. All that matters is that your front controller is able to create an instance of the appropriate kernel.

Having different **AppKernels** might be useful to enable different front controllers (on potentially different servers) to run parts of your application independently (for example, the admin UI, the front-end UI and database migrations).



There's a lot more the **AppKernel** can be used for, for example *overriding the default directory structure*. But odds are high that you don't need to change things like this on the fly by having several **AppKernel** implementations.

The Environments

As just mentioned, the **AppKernel** has to implement another method - *registerContainerConfiguration()*¹⁹. This method is responsible for loading the application's configuration from the right *environment*.

Environments have been covered extensively *in the previous chapter*, and you probably remember that the Symfony Standard Edition comes with three of them - **dev**, **prod** and **test**.

More technically, these names are nothing more than strings passed from the front controller to the **AppKernel**'s constructor. This name can then be used in the *registerContainerConfiguration()*²⁰ method to decide which configuration files to load.

The Symfony Standard Edition's *AppKernel*²¹ class implements this method by simply loading the **app/config/config_****environment*.yml* file. You are, of course, free to implement this method differently if you need a more sophisticated way of loading your configuration.

16. <https://github.com/symfony/symfony-standard/blob/master/app/AppKernel.php>

17. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Kernel.html#method___construct

18. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Kernel.html#method_getEnvironment

19. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/KernelInterface.html#method_registerContainerConfiguration

20. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/KernelInterface.html#method_registerContainerConfiguration

21. <https://github.com/symfony/symfony-standard/blob/master/app/AppKernel.php>



Chapter 23

How to Set external Parameters in the Service Container

In the chapter *How to Master and Create new Environments*, you learned how to manage your application configuration. At times, it may benefit your application to store certain credentials outside of your project code. Database configuration is one such example. The flexibility of the Symfony service container allows you to easily do this.

Environment Variables

Symfony will grab any environment variable prefixed with `SYMFONY__` and set it as a parameter in the service container. Some transformations are applied to the resulting parameter name:

- `SYMFONY__` prefix is removed;
- Parameter name is lowercased;
- Double underscores are replaced with a period, as a period is not a valid character in an environment variable name.

For example, if you're using Apache, environment variables can be set using the following `VirtualHost` configuration:

Listing 23-1

```
1 <VirtualHost *:80>
2     ServerName      Symfony
3     DocumentRoot    "/path/to/symfony_2_app/web"
4     DirectoryIndex  index.php index.html
5     SetEnv          SYMFONY__DATABASE__USER user
6     SetEnv          SYMFONY__DATABASE__PASSWORD secret
7
8     <Directory "/path/to/symfony_2_app/web">
9         AllowOverride All
10        Allow from All
11    </Directory>
12 </VirtualHost>
```




The example above is for an Apache configuration, using the *SetEnv*¹ directive. However, this will work for any web server which supports the setting of environment variables.

Also, in order for your console to work (which does not use Apache), you must export these as shell variables. On a Unix system, you can run the following:

Listing 23-2

```
1 $ export SYMFONY__DATABASE__USER=user
2 $ export SYMFONY__DATABASE__PASSWORD=secret
```

Now that you have declared an environment variable, it will be present in the PHP `$_SERVER` global variable. Symfony then automatically sets all `$_SERVER` variables prefixed with `SYMFONY__` as parameters in the service container.

You can now reference these parameters wherever you need them.

Listing 23-3

```
1 doctrine:
2   dbal:
3     driver      pdo_mysql
4     dbname:     symfony_project
5     user:       '%database.user%'
6     password:   '%database.password%'
```

Constants

The container also has support for setting PHP constants as parameters. See Constants as Parameters for more details.

Miscellaneous Configuration

The `imports` directive can be used to pull in parameters stored elsewhere. Importing a PHP file gives you the flexibility to add whatever is needed in the container. The following imports a file named `parameters.php`.

Listing 23-4

```
1 # app/config/config.yml
2 imports:
3   - { resource: parameters.php }
```



A resource file can be one of many types. PHP, XML, YAML, INI, and closure resources are all supported by the `imports` directive.

In `parameters.php`, tell the service container the parameters that you wish to set. This is useful when important configuration is in a non-standard format. The example below includes a Drupal database configuration in the Symfony service container.

Listing 23-5

```
1 // app/config/parameters.php
2 include_once('/path/to/drupal/sites/default/settings.php');
3 $container->setParameter('drupal.database.url', $db_url);
```

1. <http://httpd.apache.org/docs/current/env.html>



Chapter 24

How to Use the Apache Router



Using the Apache Router is no longer considered a good practice. The small increase obtained in the application routing performance is not worth the hassle of continuously updating the routes configuration.

The Apache Router will be removed in Symfony 3 and it's highly recommended to not use it in your applications.

Symfony, while fast out of the box, also provides various ways to increase that speed with a little bit of tweaking. One of these ways is by letting Apache handle routes directly, rather than using Symfony for this task.



Apache router was deprecated in Symfony 2.5 and removed in Symfony 3.0. Since the PHP implementation of the Router was improved, performance gains were no longer significant (while it's very hard to replicate the same behavior).



Chapter 25

Configuring a Web Server

The preferred way to develop your Symfony application is to use *PHP's internal web server*. However, when using an older PHP version or when running the application in the production environment, you'll need to use a fully-featured web server. This article describes several ways to use Symfony with Apache or Nginx.

When using Apache, you can configure PHP as an Apache module or with FastCGI using PHP FPM. FastCGI also is the preferred way to use PHP with Nginx.



The Web Directory

The web directory is the home of all of your application's public and static files, including images, stylesheets and JavaScript files. It is also where the front controllers (`app.php` and `app_dev.php`) live.

The web directory serves as the document root when configuring your web server. In the examples below, the `web/` directory will be the document root. This directory is `/var/www/project/web/`.

If your hosting provider requires you to change the `web/` directory to another location (e.g. `public_html/`) make sure you override the location of the `web/` directory.

Apache with mod_php/PHP-CGI

The **minimum configuration** to get your application running under Apache is:

Listing 25-1

```
1 <VirtualHost *:80>
2     ServerName domain.tld
3     ServerAlias www.domain.tld
4
5     DocumentRoot /var/www/project/web
6     <Directory /var/www/project/web>
7         AllowOverride All
8         Order Allow,Deny
9         Allow from All
10    </Directory>
11
12    # uncomment the following lines if you install assets as symlinks
```

```

13  # or run into problems when compiling LESS/Sass/CoffeScript assets
14  # <Directory /var/www/project>
15  #     Options FollowSymlinks
16  # </Directory>
17
18  ErrorLog /var/log/apache2/project_error.log
19  CustomLog /var/log/apache2/project_access.log combined
20 </VirtualHost>

```



If your system supports the `APACHE_LOG_DIR` variable, you may want to use `${APACHE_LOG_DIR}/` instead of hardcoding `/var/log/apache2/`.

Use the following **optimized configuration** to disable `.htaccess` support and increase web server performance:

Listing 25-2

```

1 <VirtualHost *:80>
2     ServerName domain.tld
3     ServerAlias www.domain.tld
4
5     DocumentRoot /var/www/project/web
6     <Directory /var/www/project/web>
7         AllowOverride None
8         Order Allow,Deny
9         Allow from All
10
11         <IfModule mod_rewrite.c>
12             Options -MultiViews
13             RewriteEngine On
14             RewriteCond %{REQUEST_FILENAME} !-f
15             RewriteRule ^(.*)$ app.php [QSA,L]
16         </IfModule>
17     </Directory>
18
19     # uncomment the following lines if you install assets as symlinks
20     # or run into problems when compiling LESS/Sass/CoffeScript assets
21     # <Directory /var/www/project>
22     #     Options FollowSymlinks
23     # </Directory>
24
25     # optionally disable the RewriteEngine for the asset directories
26     # which will allow apache to simply reply with a 404 when files are
27     # not found instead of passing the request into the full symfony stack
28     <Directory /var/www/project/web/bundles>
29         <IfModule mod_rewrite.c>
30             RewriteEngine Off
31         </IfModule>
32     </Directory>
33     ErrorLog /var/log/apache2/project_error.log
34     CustomLog /var/log/apache2/project_access.log combined
35 </VirtualHost>

```



If you are using **php-cgi**, Apache does not pass HTTP basic username and password to PHP by default. To work around this limitation, you should use the following configuration snippet:

Listing 25-3 `1 RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]`

Using mod_php/PHP-CGI with Apache 2.4

In Apache 2.4, `Order Allow,Deny` has been replaced by `Require all granted`. Hence, you need to modify your `Directory` permission settings as follows:

Listing 25-4

```
1 <Directory /var/www/project/web>
2     Require all granted
3     # ...
4 </Directory>
```

For advanced Apache configuration options, read the official *Apache documentation*¹.

Apache with PHP-FPM

To make use of PHP5-FPM with Apache, you first have to ensure that you have the FastCGI process manager **php-fpm** binary and Apache's FastCGI module installed (for example, on a Debian based system you have to install the **libapache2-mod-fastcgi** and **php5-fpm** packages).

PHP-FPM uses so-called *pools* to handle incoming FastCGI requests. You can configure an arbitrary number of pools in the FPM configuration. In a pool you configure either a TCP socket (IP and port) or a Unix domain socket to listen on. Each pool can also be run under a different UID and GID:

Listing 25-5

```
1 ; a pool called www
2 [www]
3 user = www-data
4 group = www-data
5
6 ; use a unix domain socket
7 listen = /var/run/php5-fpm.sock
8
9 ; or listen on a TCP socket
10 listen = 127.0.0.1:9000
```

Using mod_proxy_fcgi with Apache 2.4

If you are running Apache 2.4, you can easily use **mod_proxy_fcgi** to pass incoming requests to PHP-FPM. Configure PHP-FPM to listen on a TCP socket (**mod_proxy** currently *does not support Unix sockets*²), enable **mod_proxy** and **mod_proxy_fcgi** in your Apache configuration and use the **SetHandler** directive to pass requests for PHP files to PHP FPM:

Listing 25-6

```
1 <VirtualHost *:80>
2     ServerName domain.tld
3     ServerAlias www.domain.tld
4
5     # Uncomment the following line to force Apache to pass the Authorization
6     # header to PHP: required for "basic_auth" under PHP-FPM and FastCGI
7     #
8     # SetEnvIfNoCase ^Authorization$ "(.+)" HTTP_AUTHORIZATION=$1
9
10    # For Apache 2.4.9 or higher
11    # Using SetHandler avoids issues with using ProxyPassMatch in combination
12    # with mod_rewrite or mod_autoindex
13    <FilesMatch \.php$>
14        SetHandler proxy:fcgi://127.0.0.1:9000
15    </FilesMatch>
16
17    # If you use Apache version below 2.4.9 you must consider update or use this instead
18    # ProxyPassMatch ^/(.*\.php(/.*)?)$ fcgi://127.0.0.1:9000/var/www/project/web/$1
19
20    # If you run your Symfony application on a subpath of your document root, the
21    # regular expression must be changed accordingly:
22    # ProxyPassMatch ^/path-to-app/(.*\.php(/.*)?)$ fcgi://127.0.0.1:9000/var/www/project/web/$1
```

1. <http://httpd.apache.org/docs/>

2. https://bz.apache.org/bugzilla/show_bug.cgi?id=54101

```

23
24 DocumentRoot /var/www/project/web
25 <Directory /var/www/project/web>
26     # enable the .htaccess rewrites
27     AllowOverride All
28     Require all granted
29 </Directory>
30
31 # uncomment the following lines if you install assets as symlinks
32 # or run into problems when compiling LESS/Sass/CoffeeScript assets
33 # <Directory /var/www/project>
34 #     Options FollowSymlinks
35 # </Directory>
36
37 ErrorLog /var/log/apache2/project_error.log
38 CustomLog /var/log/apache2/project_access.log combined
39 </VirtualHost>

```

PHP-FPM with Apache 2.2

On Apache 2.2 or lower, you cannot use `mod_proxy_fcgi`. You have to use the *FastCgiExternalServer*³ directive instead. Therefore, your Apache configuration should look something like this:

Listing 25-7

```

1 <VirtualHost *:80>
2     ServerName domain.tld
3     ServerAlias www.domain.tld
4
5     AddHandler php5-fcgi .php
6     Action php5-fcgi /php5-fcgi
7     Alias /php5-fcgi /usr/lib/cgi-bin/php5-fcgi
8     FastCgiExternalServer /usr/lib/cgi-bin/php5-fcgi -host 127.0.0.1:9000 -pass-header Authorization
9
10    DocumentRoot /var/www/project/web
11    <Directory /var/www/project/web>
12        # enable the .htaccess rewrites
13        AllowOverride All
14        Order Allow,Deny
15        Allow from all
16    </Directory>
17
18    # uncomment the following lines if you install assets as symlinks
19    # or run into problems when compiling LESS/Sass/CoffeeScript assets
20    # <Directory /var/www/project>
21    #     Options FollowSymlinks
22    # </Directory>
23
24    ErrorLog /var/log/apache2/project_error.log
25    CustomLog /var/log/apache2/project_access.log combined
26 </VirtualHost>

```

If you prefer to use a Unix socket, you have to use the `-socket` option instead:

Listing 25-8

```

1 FastCgiExternalServer /usr/lib/cgi-bin/php5-fcgi -socket /var/run/php5-fpm.sock -pass-header Authorization

```

Nginx

The **minimum configuration** to get your application running under Nginx is:

Listing 25-9

```

1 server {
2     server_name domain.tld www.domain.tld;

```

3. http://www.fastcgi.com/mod_fastcgi/docs/mod_fastcgi.html#FastCgiExternalServer

```

3     root /var/www/project/web;
4
5     location / {
6         # try to serve file directly, fallback to app.php
7         try_files $uri /app.php$is_args$args;
8     }
9     # DEV
10    # This rule should only be placed on your development environment
11    # In production, don't include this and don't deploy app_dev.php or config.php
12    location ~ ^/(app_dev|config)\.php(/|$) {
13        fastcgi_pass unix:/var/run/php5-fpm.sock;
14        fastcgi_split_path_info ^(.+\.php)(/.*)$;
15        include fastcgi_params;
16        # When you are using symlinks to link the document root to the
17        # current version of your application, you should pass the real
18        # application path instead of the path to the symlink to PHP
19        # FPM.
20        # Otherwise, PHP's OPcache may not properly detect changes to
21        # your PHP files (see https://github.com/zendtech/ZendOptimizerPlus/issues/126
22        # for more information).
23        fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
24        fastcgi_param DOCUMENT_ROOT $realpath_root;
25    }
26    # PROD
27    location ~ ^/app\.php(/|$) {
28        fastcgi_pass unix:/var/run/php5-fpm.sock;
29        fastcgi_split_path_info ^(.+\.php)(/.*)$;
30        include fastcgi_params;
31        # When you are using symlinks to link the document root to the
32        # current version of your application, you should pass the real
33        # application path instead of the path to the symlink to PHP
34        # FPM.
35        # Otherwise, PHP's OPcache may not properly detect changes to
36        # your PHP files (see https://github.com/zendtech/ZendOptimizerPlus/issues/126
37        # for more information).
38        fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
39        fastcgi_param DOCUMENT_ROOT $realpath_root;
40        # Prevents URIs that include the front controller. This will 404:
41        # http://domain.tld/app.php/some-path
42        # Remove the internal directive to allow URIs like this
43        internal;
44    }
45
46    # return 404 for all other php files not matching the front controller
47    # this prevents access to other php files you don't want to be accessible.
48    location ~ \.php$ {
49        return 404;
50    }
51
52    error_log /var/log/nginx/project_error.log;
53    access_log /var/log/nginx/project_access.log;
54 }

```



Depending on your PHP-FPM config, the `fastcgi_pass` can also be `fastcgi_pass 127.0.0.1:9000`.



This executes **only** `app.php`, `app_dev.php` and `config.php` in the web directory. All other files ending in ".php" will be denied.

If you have other PHP files in your web directory that need to be executed, be sure to include them in the `location` block above.



After you deploy to production, make sure that you **cannot** access the `app_dev.php` or `config.php` scripts (i.e. `http://example.com/app_dev.php` and `http://example.com/config.php`). If you *can* access these, be sure to remove the DEV section from the above configuration.

For advanced Nginx configuration options, read the official *Nginx documentation*⁴.

4. <http://wiki.nginx.org/Symfony>



Chapter 26

How to Organize Configuration Files

The default Symfony Standard Edition defines three *execution environments* called **dev**, **prod** and **test**. An environment simply represents a way to execute the same codebase with different configurations.

In order to select the configuration file to load for each environment, Symfony executes the `registerContainerConfiguration()` method of the `AppKernel` class:

Listing 26-1

```
1 // app/AppKernel.php
2 use Symfony\Component\HttpKernel\Kernel;
3 use Symfony\Component\Config\Loader\LoaderInterface;
4
5 class AppKernel extends Kernel
6 {
7     // ...
8
9     public function registerContainerConfiguration(LoaderInterface $loader)
10    {
11        $loader->load($this->getRootDir().'/config/config_'.$this->getEnvironment().'.yaml');
12    }
13 }
```

This method loads the `app/config/config_dev.yaml` file for the **dev** environment and so on. In turn, this file loads the common configuration file located at `app/config/config.yaml`. Therefore, the configuration files of the default Symfony Standard Edition follow this structure:

Listing 26-2

```
1 your-project/
2 └─ app/
3     └─ ...
4         └─ config/
5             └─ config.yaml
6             └─ config_dev.yaml
7             └─ config_prod.yaml
8             └─ config_test.yaml
9             └─ parameters.yaml
10            └─ parameters.yaml.dist
11            └─ routing.yaml
12            └─ routing_dev.yaml
13            └─ security.yaml
14 └─ ...
```

This default structure was chosen for its simplicity — one file per environment. But as any other Symfony feature, you can customize it to better suit your needs. The following sections explain different ways to organize your configuration files. In order to simplify the examples, only the **dev** and **prod** environments are taken into account.

Different Directories per Environment

Instead of suffixing the files with `_dev` and `_prod`, this technique groups all the related configuration files under a directory with the same name as the environment:

Listing 26-3

```
1 your-project/
2   └─ app/
3       └─ ...
4           └─ config/
5               └─ common/
6                   └─ config.yml
7                   └─ parameters.yml
8                   └─ routing.yml
9                   └─ security.yml
10              └─ dev/
11                  └─ config.yml
12                  └─ parameters.yml
13                  └─ routing.yml
14                  └─ security.yml
15              └─ prod/
16                  └─ config.yml
17                  └─ parameters.yml
18                  └─ routing.yml
19                  └─ security.yml
20   └─ ...
```

To make this work, change the code of the `registerContainerConfiguration()`¹ method:

Listing 26-4

```
1 // app/AppKernel.php
2 use Symfony\Component\HttpKernel\Kernel;
3 use Symfony\Component\Config\Loader\LoaderInterface;
4
5 class AppKernel extends Kernel
6 {
7     // ...
8
9     public function registerContainerConfiguration(LoaderInterface $loader)
10     {
11         $loader->load($this->getRootDir().'/config/'.$this->getEnvironment().'/config.yml');
12     }
13 }
```

Then, make sure that each `config.yml` file loads the rest of the configuration files, including the common files. For instance, this would be the imports needed for the `app/config/dev/config.yml` file:

Listing 26-5

```
1 # app/config/dev/config.yml
2 imports:
3     - { resource: '../common/config.yml' }
4     - { resource: 'parameters.yml' }
5     - { resource: 'security.yml' }
6
7 # ...
```

1. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/KernelInterface.html#method_registerContainerConfiguration



Due to the way in which parameters are resolved, you cannot use them to build paths in imports dynamically. This means that something like the following doesn't work:

Listing 26-6

```
1 # app/config/config.yml
2 imports:
3   - { resource: '%kernel.root_dir%/parameters.yml' }
```

Semantic Configuration Files

A different organization strategy may be needed for complex applications with large configuration files. For instance, you could create one file per bundle and several files to define all application services:

Listing 26-7

```
1 your-project/
2   └─ app/
3       └─ ...
4           └─ config/
5               └─ bundles/
6                   └─ bundle1.yml
7                   └─ bundle2.yml
8                   └─ ...
9                   └─ bundleN.yml
10              └─ environments/
11                  └─ common.yml
12                  └─ dev.yml
13                  └─ prod.yml
14              └─ routing/
15                  └─ common.yml
16                  └─ dev.yml
17                  └─ prod.yml
18              └─ services/
19                  └─ frontend.yml
20                  └─ backend.yml
21                  └─ ...
22                  └─ security.yml
23  └─ ...
```

Again, change the code of the `registerContainerConfiguration()` method to make Symfony aware of the new file organization:

Listing 26-8

```
1 // app/AppKernel.php
2 use Symfony\Component\HttpKernel\Kernel;
3 use Symfony\Component\Config\Loader\LoaderInterface;
4
5 class AppKernel extends Kernel
6 {
7     // ...
8
9     public function registerContainerConfiguration(LoaderInterface $loader)
10     {
11         $loader->load($this->getRootDir().'/config/environments/'.$this->getEnvironment().'.yml');
12     }
13 }
```

Following the same technique explained in the previous section, make sure to import the appropriate configuration files from each main file (`common.yml`, `dev.yml` and `prod.yml`).

Advanced Techniques

Symfony loads configuration files using the *Config component*, which provides some advanced features.

Mix and Match Configuration Formats

Configuration files can import files defined with any other built-in configuration format (`.yaml`, `.xml`, `.php`, `.ini`):

Listing 26-9

```
1 # app/config/config.yaml
2 imports:
3   - { resource: 'parameters.yaml' }
4   - { resource: 'services.xml' }
5   - { resource: 'security.yaml' }
6   - { resource: 'legacy.php' }
7
8 # ...
```



The `IniFileLoader` parses the file contents using the `parse_ini_file`² function. Therefore, you can only set parameters to string values. Use one of the other loaders if you want to use other data types (e.g. boolean, integer, etc.).

If you use any other configuration format, you have to define your own loader class extending it from `FileLoader`³. When the configuration values are dynamic, you can use the PHP configuration file to execute your own logic. In addition, you can define your own services to load configurations from databases or web services.

Global Configuration Files

Some system administrators may prefer to store sensitive parameters in files outside the project directory. Imagine that the database credentials for your website are stored in the `/etc/sites/mysite.com/parameters.yaml` file. Loading this file is as simple as indicating the full file path when importing it from any other configuration file:

Listing 26-10

```
1 # app/config/config.yaml
2 imports:
3   - { resource: 'parameters.yaml' }
4   - { resource: '/etc/sites/mysite.com/parameters.yaml' }
5
6 # ...
```

Most of the time, local developers won't have the same files that exist on the production servers. For that reason, the Config component provides the `ignore_errors` option to silently discard errors when the loaded file doesn't exist:

Listing 26-11

```
1 # app/config/config.yaml
2 imports:
3   - { resource: 'parameters.yaml' }
4   - { resource: '/etc/sites/mysite.com/parameters.yaml', ignore_errors: true }
5
6 # ...
```

As you've seen, there are lots of ways to organize your configuration files. You can choose one of these or even create your own custom way of organizing the files. Don't feel limited by the Standard Edition that comes with Symfony. For even more customization, see "*How to Override Symfony's default Directory Structure*".

2. <http://php.net/manual/en/function.parse-ini-file.php>

3. <http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/Loader/FileLoader.html>



Chapter 27

How to Create a Console Command

The Console page of the Components section (*The Console Component*) covers how to create a console command. This cookbook article covers the differences when creating console commands within the Symfony Framework.

Automatically Registering Commands

To make the console commands available automatically with Symfony, create a **Command** directory inside your bundle and create a PHP file suffixed with **Command.php** for each command that you want to provide. For example, if you want to extend the AppBundle to greet you from the command line, create **GreetCommand.php** and add the following to it:

```
Listing 27-1 1 // src/AppBundle/Command/GreetCommand.php
2 namespace AppBundle\Command;
3
4 use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;
5 use Symfony\Component\Console\Input\InputArgument;
6 use Symfony\Component\Console\Input\InputInterface;
7 use Symfony\Component\Console\Input\InputOption;
8 use Symfony\Component\Console\Output\OutputInterface;
9
10 class GreetCommand extends ContainerAwareCommand
11 {
12     protected function configure()
13     {
14         $this
15             ->setName('demo:greet')
16             ->setDescription('Greet someone')
17             ->addArgument(
18                 'name',
19                 InputArgument::OPTIONAL,
20                 'Who do you want to greet?'
21             )
22             ->addOption(
23                 'yell',
24                 null,
25                 InputOption::VALUE_NONE,
26                 'If set, the task will yell in uppercase letters'
27             )
28     }
29 }
```

```

28     ;
29 }
30
31 protected function execute(InputInterface $input, OutputInterface $output)
32 {
33     $name = $input->getArgument('name');
34     if ($name) {
35         $text = 'Hello '.$name;
36     } else {
37         $text = 'Hello';
38     }
39
40     if ($input->getOption('yell')) {
41         $text = strtoupper($text);
42     }
43
44     $output->writeln($text);
45 }
46 }

```

This command will now automatically be available to run:

Listing 27-2 1 \$ php bin/console demo:greet Fabien

Register Commands in the Service Container

Just like controllers, commands can be declared as services. See the *dedicated cookbook entry* for details.

Getting Services from the Service Container

By using *ContainerAwareCommand*¹ as the base class for the command (instead of the more basic *Command*²), you have access to the service container. In other words, you have access to any configured service:

Listing 27-3

```

1 protected function execute(InputInterface $input, OutputInterface $output)
2 {
3     $name = $input->getArgument('name');
4     $logger = $this->getContainer()->get('logger');
5
6     $logger->info('Executing command for '.$name);
7     // ...
8 }

```

Invoking other Commands

See *Calling an Existing Command* if you need to implement a command that runs other dependent commands.

1. <http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Command/ContainerAwareCommand.html>
2. <http://api.symfony.com/3.0/Symfony/Component/Console/Command/Command.html>

Testing Commands

When testing commands used as part of the full-stack framework, *Symfony\Bundle\FrameworkBundle\Console\Application*³ should be used instead of *Symfony\Component\Console\Application*⁴:

Listing 27-4

```
1 use Symfony\Component\Console\Tester\CommandTester;
2 use Symfony\Bundle\FrameworkBundle\Console\Application;
3 use AppBundle\Command\GreetCommand;
4
5 class ListCommandTest extends \PHPUnit_Framework_TestCase
6 {
7     public function testExecute()
8     {
9         // mock the Kernel or create one depending on your needs
10        $application = new Application($kernel);
11        $application->add(new GreetCommand());
12
13        $command = $application->find('demo:greet');
14        $commandTester = new CommandTester($command);
15        $commandTester->execute(
16            array(
17                'name' => 'Fabien',
18                '--yell' => true,
19            )
20        );
21
22        $this->assertRegExp('/.../', $commandTester->getDisplay());
23
24        // ...
25    }
26 }
```



In the specific case above, the **name** parameter and the **--yell** option are not mandatory for the command to work, but are shown so you can see how to customize them when calling the command.

To be able to use the fully set up service container for your console tests you can extend your test from *KernelTestCase*⁵:

Listing 27-5

```
1 use Symfony\Component\Console\Tester\CommandTester;
2 use Symfony\Bundle\FrameworkBundle\Console\Application;
3 use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
4 use AppBundle\Command\GreetCommand;
5
6 class ListCommandTest extends KernelTestCase
7 {
8     public function testExecute()
9     {
10        $kernel = $this->createKernel();
11        $kernel->boot();
12
13        $application = new Application($kernel);
14        $application->add(new GreetCommand());
15
16        $command = $application->find('demo:greet');
17        $commandTester = new CommandTester($command);
18        $commandTester->execute(
19            array(
20                'name' => 'Fabien',
21                '--yell' => true,
22            )
23        );
24    }
25 }
```

3. <http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Console/Application.html>

4. <http://api.symfony.com/3.0/Symfony/Component/Console/Application.html>

5. <http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Test/KernelTestCase.html>

```
22         )
23     );
24
25     $this->assertRegExp('/.../', $commandTester->getDisplay());
26
27     // ...
28 }
29 }
```




Chapter 28

How to Use the Console

The *Using Console Commands, Shortcuts and Built-in Commands* page of the components documentation looks at the global console options. When you use the console as part of the full-stack framework, some additional global options are available as well.

By default, console commands run in the **dev** environment and you may want to change this for some commands. For example, you may want to run some commands in the **prod** environment for performance reasons. Also, the result of some commands will be different depending on the environment. For example, the **cache:clear** command will clear and warm the cache for the specified environment only. To clear and warm the **prod** cache you need to run:

Listing 28-1 1 \$ php bin/console cache:clear --env=prod

or the equivalent:

Listing 28-2 1 \$ php bin/console cache:clear -e prod

In addition to changing the environment, you can also choose to disable debug mode. This can be useful where you want to run commands in the **dev** environment but avoid the performance hit of collecting debug data:

Listing 28-3 1 \$ php bin/console list --no-debug



Chapter 29

How to Style a Console Command

One of the most boring tasks when creating console commands is to deal with the styling of the command's input and output. Displaying titles and tables or asking questions to the user involves a lot of repetitive code.

Consider for example the code used to display the title of the following command:

Listing 29-1

```
1  // src/AppBundle/Command/GreetCommand.php
2  namespace AppBundle\Command;
3
4  use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;
5  use Symfony\Component\Console\Input\InputInterface;
6  use Symfony\Component\Console\Output\OutputInterface;
7
8  class GreetCommand extends ContainerAwareCommand
9  {
10     // ...
11
12     protected function execute(InputInterface $input, OutputInterface $output)
13     {
14         $output->writeln(array(
15             '<info>Lorem Ipsum Dolor Sit Amet</>',
16             '<info>=====</>',
17             '',
18         ));
19
20         // ...
21     }
22 }
```

Displaying a simple title requires three lines of code, to change the font color, underline the contents and leave an additional blank line after the title. Dealing with styles is required for well-designed commands, but it complicates their code unnecessarily.

In order to reduce that boilerplate code, Symfony commands can optionally use the **Symfony Style Guide**. These styles are implemented as a set of helper methods which allow to create *semantic* commands and forget about their styling.

Basic Usage

In your command, instantiate the *SymfonyStyle*¹ class and pass the `$input` and `$output` variables as its arguments. Then, you can start using any of its helpers, such as `title()`, which displays the title of the command:

Listing 29-2

```
1 // src/AppBundle/Command/GreetCommand.php
2 namespace AppBundle\Command;
3
4 use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;
5 use Symfony\Component\Console\Style\SymfonyStyle;
6 use Symfony\Component\Console\Input\InputInterface;
7 use Symfony\Component\Console\Output\OutputInterface;
8
9 class GreetCommand extends ContainerAwareCommand
10 {
11     // ...
12
13     protected function execute(InputInterface $input, OutputInterface $output)
14     {
15         $io = new SymfonyStyle($input, $output);
16         $io->title('Lorem Ipsum Dolor Sit Amet');
17
18         // ...
19     }
20 }
```

Helper Methods

The *SymfonyStyle*² class defines some helper methods that cover the most common interactions performed by console commands.

Titling Methods

*title()*³

It displays the given string as the command title. This method is meant to be used only once in a given command, but nothing prevents you to use it repeatedly:

Listing 29-3

```
$io->title('Lorem ipsum dolor sit amet');
```

*section()*⁴

It displays the given string as the title of some command section. This is only needed in complex commands which want to better separate their contents:

Listing 29-4

```
1 $io->section('Adding a User');
2
3 // ...
4
5 $io->section('Generating the Password');
6
7 // ...
```

-
1. <http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html>
 2. <http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html>
 3. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_title
 4. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_section

Content Methods

`text()`⁵

It displays the given string or array of strings as regular text. This is useful to render help messages and instructions for the user running the command:

```
Listing 29-5 1 // use simple strings for short messages
2 $io->text('Lorem ipsum dolor sit amet');
3
4 // ...
5
6 // consider using arrays when displaying long messages
7 $io->text(array(
8     'Lorem ipsum dolor sit amet',
9     'Consectetur adipiscing elit',
10    'Aenean sit amet arcu vitae sem faucibus porta',
11 ));
```

`listing()`⁶

It displays an unordered list of elements passed as an array:

```
Listing 29-6 1 $io->listing(array(
2     'Element #1 Lorem ipsum dolor sit amet',
3     'Element #2 Lorem ipsum dolor sit amet',
4     'Element #3 Lorem ipsum dolor sit amet',
5 ));
```

`table()`⁷

It displays the given array of headers and rows as a compact table:

```
Listing 29-7 1 $io->table(
2     array('Header 1', 'Header 2'),
3     array(
4         array('Cell 1-1', 'Cell 1-2'),
5         array('Cell 2-1', 'Cell 2-2'),
6         array('Cell 3-1', 'Cell 3-2'),
7     )
8 );
```

`newLine()`⁸

It displays a blank line in the command output. Although it may seem useful, most of the times you won't need it at all. The reason is that every helper already adds their own blank lines, so you don't have to care about the vertical spacing:

```
Listing 29-8 1 // outputs a single blank line
2 $io->newLine();
3
4 // outputs three consecutive blank lines
5 $io->newLine(3);
```

5. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_text

6. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_listing

7. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_table

8. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_newLine

Admonition Methods

*note()*⁹

It displays the given string or array of strings as a highlighted admonition. Use this helper sparingly to avoid cluttering command's output:

Listing 29-9

```
1 // use simple strings for short notes
2 $io->note('Lorem ipsum dolor sit amet');
3
4 // ...
5
6 // consider using arrays when displaying long notes
7 $io->note(array(
8     'Lorem ipsum dolor sit amet',
9     'Consectetur adipiscing elit',
10    'Aenean sit amet arcu vitae sem faucibus porta',
11 ));
```

*caution()*¹⁰

Similar to the **note()** helper, but the contents are more prominently highlighted. The resulting contents resemble an error message, so you should avoid using this helper unless strictly necessary:

Listing 29-10

```
1 // use simple strings for short caution message
2 $io->caution('Lorem ipsum dolor sit amet');
3
4 // ...
5
6 // consider using arrays when displaying long caution messages
7 $io->caution(array(
8     'Lorem ipsum dolor sit amet',
9     'Consectetur adipiscing elit',
10    'Aenean sit amet arcu vitae sem faucibus porta',
11 ));
```

Progress Bar Methods

*progressStart()*¹¹

It displays a progress bar with a number of steps equal to the argument passed to the method (don't pass any value if the length of the progress bar is unknown):

Listing 29-11

```
1 // displays a progress bar of unknown length
2 $io->progressStart();
3
4 // displays a 100-step length progress bar
5 $io->progressStart(100);
```

*progressAdvance()*¹²

It makes the progress bar advance the given number of steps (or **1** step if no argument is passed):

Listing 29-12

```
1 // advances the progress bar 1 step
2 $io->progressAdvance();
3
4 // advances the progress bar 10 steps
5 $io->progressAdvance(10);
```

9. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_note

10. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_caution

11. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_progressStart

12. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_progressAdvance

*progressFinish()*¹³

It finishes the progress bar (filling up all the remaining steps when its length is known):

Listing 29-13 `$io->progressFinish();`

User Input Methods

*ask()*¹⁴

It asks the user to provide some value:

Listing 29-14 `$io->ask('What is your name?');`

You can pass the default value as the second argument so the user can simply hit the <Enter> key to select that value:

Listing 29-15 `$io->ask('Where are you from?', 'United States');`

In case you need to validate the given value, pass a callback validator as the third argument:

Listing 29-16

```
1 $io->ask('Number of workers to start', 1, function ($number) {
2     if (!is_integer($number)) {
3         throw new \RuntimeException('You must type an integer.');
```

*askHidden()*¹⁵

It's very similar to the `ask()` method but the user's input will be hidden and it cannot define a default value. Use it when asking for sensitive information:

Listing 29-17

```
1 $io->askHidden('What is your password?');
2
3 // validates the given answer
4 $io->askHidden('What is your password?', function ($password) {
5     if (empty($password)) {
6         throw new \RuntimeException('Password cannot be empty.');
```

*confirm()*¹⁶

It asks a Yes/No question to the user and it only returns `true` or `false`:

Listing 29-18 `$io->confirm('Restart the web server?');`

You can pass the default value as the second argument so the user can simply hit the <Enter> key to select that value:

Listing 29-19 `$io->confirm('Restart the web server?', true);`

*choice()*¹⁷

It asks a question whose answer is constrained to the given list of valid answers:

Listing 29-20 `$io->choice('Select the queue to analyze', array('queue1', 'queue2', 'queue3'));`

13. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_progressFinish
14. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_ask
15. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_askHidden
16. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_confirm
17. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_choice

You can pass the default value as the third argument so the user can simply hit the <Enter> key to select that value:

Listing 29-21 `$io->choice('Select the queue to analyze', array('queue1', 'queue2', 'queue3'), 'queue1');`

Result Methods

`success()`¹⁸

It displays the given string or array of strings highlighted as a successful message (with a green background and the [OK] label). It's meant to be used once to display the final result of executing the given command, but you can use it repeatedly during the execution of the command:

Listing 29-22

```
1 // use simple strings for short success messages
2 $io->success('Lorem ipsum dolor sit amet');
3
4 // ...
5
6 // consider using arrays when displaying long success messages
7 $io->success(array(
8     'Lorem ipsum dolor sit amet',
9     'Consectetur adipiscing elit',
10 ));
```

`warning()`¹⁹

It displays the given string or array of strings highlighted as a warning message (with a red background and the [WARNING] label). It's meant to be used once to display the final result of executing the given command, but you can use it repeatedly during the execution of the command:

Listing 29-23

```
1 // use simple strings for short warning messages
2 $io->warning('Lorem ipsum dolor sit amet');
3
4 // ...
5
6 // consider using arrays when displaying long warning messages
7 $io->warning(array(
8     'Lorem ipsum dolor sit amet',
9     'Consectetur adipiscing elit',
10 ));
```

`error()`²⁰

It displays the given string or array of strings highlighted as an error message (with a red background and the [ERROR] label). It's meant to be used once to display the final result of executing the given command, but you can use it repeatedly during the execution of the command:

Listing 29-24

```
1 // use simple strings for short error messages
2 $io->error('Lorem ipsum dolor sit amet');
3
4 // ...
5
6 // consider using arrays when displaying long error messages
7 $io->error(array(
8     'Lorem ipsum dolor sit amet',
9     'Consectetur adipiscing elit',
10 ));
```

18. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_success

19. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_warning

20. http://api.symfony.com/3.0/Symfony/Component/Console/Style/SymfonyStyle.html#method_error

Defining your Own Styles

If you don't like the design of the commands that use the Symfony Style, you can define your own set of console styles. Just create a class that implements the *StyleInterface*²¹:

```
Listing 29-25 1 namespace AppBundle\Console;
2
3 use Symfony\Component\Console\Style\StyleInterface;
4
5 class CustomStyle implements StyleInterface
6 {
7     // ...implement the methods of the interface
8 }
```

Then, instantiate this custom class instead of the default **SymfonyStyle** in your commands. Thanks to the **StyleInterface** you won't need to change the code of your commands to change their appearance:

```
Listing 29-26 1 namespace AppBundle\Console;
2
3 use AppBundle\Console\CustomStyle;
4 use Symfony\Component\Console\Input\InputInterface;
5 use Symfony\Component\Console\Output\OutputInterface;
6 use Symfony\Component\Console\Style\SymfonyStyle;
7
8 class GreetCommand extends ContainerAwareCommand
9 {
10     // ...
11
12     protected function execute(InputInterface $input, OutputInterface $output)
13     {
14         // Before
15         // $io = new SymfonyStyle($input, $output);
16
17         // After
18         $io = new CustomStyle($input, $output);
19
20         // ...
21     }
22 }
```

21. <http://api.symfony.com/3.0/Symfony/Component/Console/Style/StyleInterface.html>



Chapter 30

How to Call a Command from a Controller

The *Console component documentation* covers how to create a console command. This cookbook article covers how to use a console command directly from your controller.

You may have the need to execute some function that is only available in a console command. Usually, you should refactor the command and move some logic into a service that can be reused in the controller. However, when the command is part of a third-party library, you wouldn't want to modify or duplicate their code. Instead, you can execute the command directly.



In comparison with a direct call from the console, calling a command from a controller has a slight performance impact because of the request stack overhead.

Imagine you want to send spooled Swift Mailer messages by *using the `swiftmailer:spool:send` command*. Run this command from inside your controller via:

```
Listing 30-1 1 // src/AppBundle/Controller/SpoolController.php
2 namespace AppBundle\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Console\Application;
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\Console\Input\ArrayInput;
7 use Symfony\Component\Console\Output\BufferedOutput;
8 use Symfony\Component\HttpFoundation\Response;
9
10 class SpoolController extends Controller
11 {
12     public function sendSpoolAction($messages = 10)
13     {
14         $kernel = $this->get('kernel');
15         $application = new Application($kernel);
16         $application->setAutoExit(false);
17
18         $input = new ArrayInput(array(
19             'command' => 'swiftmailer:spool:send',
20             '--message-limit' => $messages,
21         ));
22         // You can use NullOutput() if you don't need the output
23         $output = new BufferedOutput();
24         $application->run($input, $output);
25     }
26 }
```

```

26         // return the output, don't use if you used NullOutput()
27         $content = $output->fetch();
28
29         // return new Response(""), if you used NullOutput()
30         return new Response($content);
31     }
32 }

```

Showing Colorized Command Output

By telling the **BufferedOutput** it is decorated via the second parameter, it will return the Ansi color-coded content. The *SensioLabs AnsiToHtml converter*¹ can be used to convert this to colorful HTML.

First, require the package:

Listing 30-2 `1 $ composer require sensiolabs/ansi-to-html`

Now, use it in your controller:

Listing 30-3

```

1  // src/AppBundle/Controller/SpoolController.php
2  namespace AppBundle\Controller;
3
4  use SensioLabs\AnsiConverter\AnsiToHtmlConverter;
5  use Symfony\Component\Console\Output\BufferedOutput;
6  use Symfony\Component\Console\Output\OutputInterface;
7  use Symfony\Component\HttpFoundation\Response;
8  // ...
9
10 class SpoolController extends Controller
11 {
12     public function sendSpoolAction($messages = 10)
13     {
14         // ...
15         $output = new BufferedOutput(
16             OutputInterface::VERBOSITY_NORMAL,
17             true // true for decorated
18         );
19         // ...
20
21         // return the output
22         $converter = new AnsiToHtmlConverter();
23         $content = $output->fetch();
24
25         return new Response($converter->convert($content));
26     }
27 }

```

The **AnsiToHtmlConverter** can also be registered as a *Twig Extension*², and supports optional themes.

1. <https://github.com/sensiolabs/ansi-to-html>

2. <https://github.com/sensiolabs/ansi-to-html#twig-integration>



Chapter 31

How to Generate URLs from the Console

Unfortunately, the command line context does not know about your VirtualHost or domain name. This means that if you generate absolute URLs within a console command you'll probably end up with something like `http://localhost/foo/bar` which is not very useful.

To fix this, you need to configure the "request context", which is a fancy way of saying that you need to configure your environment so that it knows what URL it should use when generating URLs.

There are two ways of configuring the request context: at the application level and per Command.

Configuring the Request Context Globally

To configure the Request Context - which is used by the URL Generator - you can redefine the parameters it uses as default values to change the default host (localhost) and scheme (http). You can also configure the base path if Symfony is not running in the root directory.

Note that this does not impact URLs generated via normal web requests, since those will override the defaults.

Listing 31-1

```
1 # app/config/parameters.yml
2 parameters:
3     router.request_context.host: example.org
4     router.request_context.scheme: https
5     router.request_context.base_url: my/path
```

Configuring the Request Context per Command

To change it only in one command you can simply fetch the Request Context from the **router** service and override its settings:

Listing 31-2

```
1 // src/AppBundle/Command/DemoCommand.php
2
3 // ...
4 class DemoCommand extends ContainerAwareCommand
5 {
```

```
6     protected function execute(InputInterface $input, OutputInterface $output)
7     {
8         $context = $this->getContainer()->get('router')->getContext();
9         $context->setHost('example.com');
10        $context->setScheme('https');
11        $context->setBaseUrl('my/path');
12
13        // ... your code here
14    }
15 }
```



Chapter 32

How to Enable Logging in Console Commands

The Console component doesn't provide any logging capabilities out of the box. Normally, you run console commands manually and observe the output, which is why logging is not provided. However, there are cases when you might need logging. For example, if you are running console commands unattended, such as from cron jobs or deployment scripts, it may be easier to use Symfony's logging capabilities instead of configuring other tools to gather console output and process it. This can be especially handful if you already have some existing setup for aggregating and analyzing Symfony logs.

There are basically two logging cases you would need:

- Manually logging some information from your command;
- Logging uncaught exceptions.

Manually Logging from a Console Command

This one is really simple. When you create a console command within the full-stack framework as described in "*How to Create a Console Command*", your command extends *ContainerAwareCommand*¹. This means that you can simply access the standard logger service through the container and use it to do the logging:

Listing 32-1

```
1  // src/AppBundle/Command/GreetCommand.php
2  namespace AppBundle\Command;
3
4  use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;
5  use Symfony\Component\Console\Input\InputArgument;
6  use Symfony\Component\Console\Input\InputInterface;
7  use Symfony\Component\Console\Input\InputOption;
8  use Symfony\Component\Console\Output\OutputInterface;
9  use Psr\Log\LoggerInterface;
10
11 class GreetCommand extends ContainerAwareCommand
12 {
13     // ...
14
15     protected function execute(InputInterface $input, OutputInterface $output)
16     {
```

1. <http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Command/ContainerAwareCommand.html>

```

17     /** @var $logger LoggerInterface */
18     $logger = $this->getContainer()->get('logger');
19
20     $name = $input->getArgument('name');
21     if ($name) {
22         $text = 'Hello '.$name;
23     } else {
24         $text = 'Hello';
25     }
26
27     if ($input->getOption('yell')) {
28         $text = strtoupper($text);
29         $logger->warning('Yelled: '.$text);
30     } else {
31         $logger->info('Greeted: '.$text);
32     }
33
34     $output->writeln($text);
35 }
36 }

```

Depending on the environment in which you run your command (and your logging setup), you should see the logged entries in `var/logs/dev.log` or `var/logs/prod.log`.

Enabling automatic Exceptions Logging

To get your console application to automatically log uncaught exceptions for all of your commands, you can use *console events*.

First configure a listener for console exception events in the service container:

Listing 32-2

```

1 # app/config/services.yml
2 services:
3     app.listener.command_exception:
4         class: AppBundle\EventListener\ConsoleExceptionListener
5         arguments: ['@logger']
6         tags:
7             - { name: kernel.event_listener, event: console.exception }

```

Then implement the actual listener:

Listing 32-3

```

1 // src/AppBundle/EventListener/ConsoleExceptionListener.php
2 namespace AppBundle\EventListener;
3
4 use Symfony\Component\Console\Event\ConsoleExceptionEvent;
5 use Psr\Log\LoggerInterface;
6
7 class ConsoleExceptionListener
8 {
9     private $logger;
10
11     public function __construct(LoggerInterface $logger)
12     {
13         $this->logger = $logger;
14     }
15
16     public function onConsoleException(ConsoleExceptionEvent $event)
17     {
18         $command = $event->getCommand();
19         $exception = $event->getException();
20
21         $message = sprintf(
22             '%s: %s (uncaught exception) at %s line %s while running console command `%s`',
23             get_class($exception),
24             $exception->getMessage(),

```

```

25         $exception->getFile(),
26         $exception->getLine(),
27         $command->getName()
28     );
29
30     $this->logger->error($message, array('exception' => $exception));
31 }
32 }

```

In the code above, when any command throws an exception, the listener will receive an event. You can simply log it by passing the logger service via the service configuration. Your method receives a *ConsoleExceptionEvent*² object, which has methods to get information about the event and the exception.

Logging non-0 Exit Statuses

The logging capabilities of the console can be further extended by logging non-0 exit statuses. This way you will know if a command had any errors, even if no exceptions were thrown.

First configure a listener for console terminate events in the service container:

Listing 32-4

```

1  # app/config/services.yml
2  services:
3      app.listener.command_error:
4          class: AppBundle\EventListener\ErrorLoggerListener
5          arguments: ['@logger']
6          tags:
7              - { name: kernel.event_listener, event: console.terminate }

```

Then implement the actual listener:

Listing 32-5

```

1  // src/AppBundle/EventListener/ErrorLoggerListener.php
2  namespace AppBundle\EventListener;
3
4  use Symfony\Component\Console\Event\ConsoleTerminateEvent;
5  use Psr\Log\LoggerInterface;
6
7  class ErrorLoggerListener
8  {
9      private $logger;
10
11      public function __construct(LoggerInterface $logger)
12      {
13          $this->logger = $logger;
14      }
15
16      public function onConsoleTerminate(ConsoleTerminateEvent $event)
17      {
18          $statusCode = $event->getExitCode();
19          $command = $event->getCommand();
20
21          if ($statusCode === 0) {
22              return;
23          }
24
25          if ($statusCode > 255) {
26              $statusCode = 255;
27              $event->setExitCode($statusCode);
28          }
29
30          $this->logger->warning(sprintf(
31              'Command `%s` exited with status code %d',

```

2. <http://api.symfony.com/3.0/Symfony/Component/Console/Event/ConsoleExceptionEvent.html>

```
32         $command->getName(),  
33         $statusCode  
34     ));  
35 }  
36 }
```




Chapter 33

How to Define Commands as Services

By default, Symfony will take a look in the **Command** directory of each bundle and automatically register your commands. If a command extends the *ContainerAwareCommand*¹, Symfony will even inject the container. While making life easier, this has some limitations:

- Your command must live in the **Command** directory;
- There's no way to conditionally register your service based on the environment or availability of some dependencies;
- You can't access the container in the `configure()` method (because `setContainer` hasn't been called yet);
- You can't use the same class to create many commands (i.e. each with different configuration).

To solve these problems, you can register your command as a service and tag it with **console.command**:

Listing 33-1

```
1 # app/config/config.yml
2 services:
3     app.command.my_command:
4         class: AppBundle\Command\MyCommand
5         tags:
6             - { name: console.command }
```

Using Dependencies and Parameters to Set Default Values for Options

Imagine you want to provide a default value for the **name** option. You could pass one of the following as the 5th argument of `addOption()`:

- a hardcoded string;
- a container parameter (e.g. something from `parameters.yml`);
- a value computed by a service (e.g. a repository).

By extending **ContainerAwareCommand**, only the first is possible, because you can't access the container inside the `configure()` method. Instead, inject any parameter or service you need into the

1. <http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Command/ContainerAwareCommand.html>

constructor. For example, suppose you store the default value in some `%command.default_name%` parameter:

```
Listing 33-2 1 // src/AppBundle/Command/GreetCommand.php
2 namespace AppBundle\Command;
3
4 use Symfony\Component\Console\Command\Command;
5 use Symfony\Component\Console\Input\InputInterface;
6 use Symfony\Component\Console\Input\InputOption;
7 use Symfony\Component\Console\Output\OutputInterface;
8
9 class GreetCommand extends Command
10 {
11     protected $defaultName;
12
13     public function __construct($defaultName)
14     {
15         $this->defaultName = $defaultName;
16
17         parent::__construct();
18     }
19
20     protected function configure()
21     {
22         // try to avoid work here (e.g. database query)
23         // this method is *always* called - see warning below
24         $defaultName = $this->defaultName;
25
26         $this
27             ->setName('demo:greet')
28             ->setDescription('Greet someone')
29             ->addOption(
30                 'name',
31                 '-n',
32                 InputOption::VALUE_REQUIRED,
33                 'Who do you want to greet?',
34                 $defaultName
35             )
36         ;
37     }
38
39     protected function execute(InputInterface $input, OutputInterface $output)
40     {
41         $name = $input->getOption('name');
42
43         $output->writeln($name);
44     }
45 }
```

Now, just update the arguments of your service configuration like normal to inject the `command.default_name` parameter:

```
Listing 33-3 1 # app/config/config.yml
2 parameters:
3     command.default_name: Javier
4
5 services:
6     app.command.my_command:
7         class: AppBundle\Command\MyCommand
8         arguments: [ "%command.default_name%" ]
9         tags:
10             - { name: console.command }
```

Great, you now have a dynamic default value!



Be careful not to actually do any work in **configure** (e.g. make database queries), as your code will be run, even if you're using the console to execute a different command.

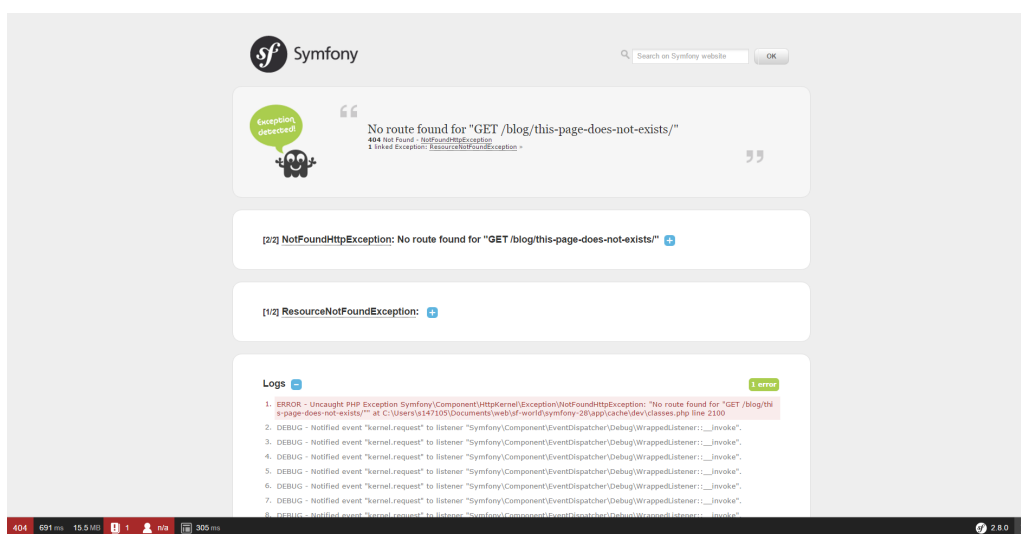


Chapter 34

How to Customize Error Pages

In Symfony applications, all errors are treated as exceptions, no matter if they are just a 404 Not Found error or a fatal error triggered by throwing some exception in your code.

In the *development environment*, Symfony catches all the exceptions and displays a special **exception page** with lots of debug information to help you quickly discover the root problem:



Since these pages contain a lot of sensitive internal information, Symfony won't display them in the production environment. Instead, it'll show a simple and generic **error page**:

Oops! An Error Occurred

The server returned a "404 Not Found".

Something is broken. Please let us know what you were doing when this error occurred. We will fix it as soon as possible. Sorry for any inconvenience caused.

Error pages for the production environment can be customized in different ways depending on your needs:

1. If you just want to change the contents and styles of the error pages to match the rest of your application, override the default error templates;
2. If you also want to tweak the logic used by Symfony to generate error pages, override the default exception controller;
3. If you need total control of exception handling to execute your own logic use the `kernel.exception` event.

Overriding the Default Error Templates

When the error page loads, an internal *ExceptionController*¹ is used to render a Twig template to show the user.

This controller uses the HTTP status code, the request format and the following logic to determine the template filename:

1. Look for a template for the given format and status code (like `error404.json.twig` or `error500.html.twig`);
2. If the previous template doesn't exist, discard the status code and look for a generic template for the given format (like `error.json.twig` or `error.xml.twig`);
3. If none of the previous template exist, fall back to the generic HTML template (`error.html.twig`).

To override these templates, simply rely on the standard Symfony method for overriding templates that live inside a bundle: put them in the `app/Resources/TwigBundle/views/Exception/` directory.

A typical project that returns HTML and JSON pages, might look like this:

Listing 34-1

```
1 app/
2   └─ Resources/
3       └─ TwigBundle/
4           └─ views/
5               └─ Exception/
6                   └─ error404.html.twig
7                   └─ error403.html.twig
8                   └─ error.html.twig      # All other HTML errors (including 500)
9                   └─ error404.json.twig
10                  └─ error403.json.twig
11                  └─ error.json.twig      # All other JSON errors (including 500)
```

1. <http://api.symfony.com/3.0/Symfony/Bundle/TwigBundle/Controller/ExceptionController.html>

Example 404 Error Template

To override the 404 error template for HTML pages, create a new `error404.html.twig` template located at `app/Resources/TwigBundle/views/Exception/`:

Listing 34-2

```
1  {# app/Resources/TwigBundle/views/Exception/error404.html.twig #}
2  {% extends 'base.html.twig' %}
3
4  {% block body %}
5      <h1>Page not found</h1>
6
7      {# example security usage, see below #}
8      {% if is_granted('IS_AUTHENTICATED_FULLY') %}
9          {# ... #}
10     {% endif %}
11
12     <p>
13         The requested page couldn't be located. Checkout for any URL
14         misspelling or <a href="{{ path('homepage') }}">return to the homepage</a>.
15     </p>
16     {% endblock %}
```

In case you need them, the `ExceptionHandler` passes some information to the error template via the `status_code` and `status_text` variables that store the HTTP status code and message respectively.



You can customize the status code by implementing *HttpExceptionInterface*² and its required `getStatusCode()` method. Otherwise, the `status_code` will default to 500.



The exception pages shown in the development environment can be customized in the same way as error pages. Create a new `exception.html.twig` template for the standard HTML exception page or `exception.json.twig` for the JSON exception page.

Testing Error Pages during Development

While you're in the development environment, Symfony shows the big *exception* page instead of your shiny new customized error page. So, how can you see what it looks like and debug it?

Fortunately, the default `ExceptionHandler` allows you to preview your *error* pages during development.

To use this feature, you need to have a definition in your `routing_dev.yml` file like so:

Listing 34-3

```
1  # app/config/routing_dev.yml
2  _errors:
3      resource: "@TwigBundle/Resources/config/routing/errors.xml"
4      prefix:   /_error
```

If you're coming from an older version of Symfony, you might need to add this to your `routing_dev.yml` file. If you're starting from scratch, the *Symfony Standard Edition*³ already contains it for you.

With this route added, you can use URLs like

Listing 34-4

2. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Exception/HttpExceptionInterface.html>
3. <https://github.com/symfony/symfony-standard/>

```
1 http://localhost/app_dev.php/_error/{statusCode}
2 http://localhost/app_dev.php/_error/{statusCode}.{format}
```

to preview the *error* page for a given status code as HTML or for a given status code and format.

Overriding the Default ExceptionController

If you need a little more flexibility beyond just overriding the template, then you can change the controller that renders the error page. For example, you might need to pass some additional variables into your template.

To do this, simply create a new controller anywhere in your application and set the `twig.exception_controller` configuration option to point to it:

Listing 34-5

```
1 # app/config/config.yml
2 twig:
3     exception_controller: AppBundle\Exception:showException
```

The *ExceptionListener*⁴ class used by the TwigBundle as a listener of the `kernel.exception` event creates the request that will be dispatched to your controller. In addition, your controller will be passed two parameters:

exception

A *FlattenException*⁵ instance created from the exception being handled.

logger

A *DebugLoggerInterface*⁶ instance which may be null in some circumstances.

Instead of creating a new exception controller from scratch you can, of course, also extend the default *ExceptionController*⁷. In that case, you might want to override one or both of the `showAction()` and `findTemplate()` methods. The latter one locates the template to be used.



In case of extending the *ExceptionController*⁸ you may configure a service to pass the Twig environment and the **debug** flag to the constructor.

Listing 34-6

```
1 # app/config/services.yml
2 services:
3     app.exception_controller:
4         class: AppBundle\Controller\CustomExceptionController
5         arguments: ['@twig', '%kernel.debug%']
```

And then configure `twig.exception_controller` using the controller as services syntax (e.g. `app.exception_controller:showAction`).



The error page preview also works for your own controllers set up this way.

4. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/EventListener/ExceptionListener.html>
5. <http://api.symfony.com/3.0/Symfony/Component/Debug/Exception/FlattenException.html>
6. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Log/DebugLoggerInterface.html>
7. <http://api.symfony.com/3.0/Symfony/Bundle/TwigBundle/Controller/ExceptionController.html>
8. <http://api.symfony.com/3.0/Symfony/Bundle/TwigBundle/Controller/ExceptionController.html>

Working with the `kernel.exception` Event

When an exception is thrown, the *HttpKernel*⁹ class catches it and dispatches a `kernel.exception` event. This gives you the power to convert the exception into a **Response** in a few different ways.

Working with this event is actually much more powerful than what has been explained before, but also requires a thorough understanding of Symfony internals. Suppose that your code throws specialized exceptions with a particular meaning to your application domain.

Writing your own event listener for the `kernel.exception` event allows you to have a closer look at the exception and take different actions depending on it. Those actions might include logging the exception, redirecting the user to another page or rendering specialized error pages.



If your listener calls `setResponse()` on the *GetResponseForExceptionEvent*¹⁰, event, propagation will be stopped and the response will be sent to the client.

This approach allows you to create centralized and layered error handling: instead of catching (and handling) the same exceptions in various controllers time and again, you can have just one (or several) listeners deal with them.



See *ExceptionListener*¹¹ class code for a real example of an advanced listener of this type. This listener handles various security-related exceptions that are thrown in your application (like *AccessDeniedException*¹²) and takes measures like redirecting the user to the login page, logging them out and other things.

9. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/HttpKernel.html>

10. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html>

11. <http://api.symfony.com/3.0/Symfony/Component/Security/Http/Firewall/ExceptionListener.html>

12. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Exception/AccessDeniedException.html>



Chapter 35

How to Define Controllers as Services



Defining controllers as services is **not officially recommended** by Symfony. They are used by some developers for very specific use cases, such as DDD (*domain-driven design*) and Hexagonal Architecture applications.

In the book, you've learned how easily a controller can be used when it extends the base *Controller*¹ class. While this works fine, controllers can also be specified as services. Even if you don't specify your controllers as services, you might see them being used in some open-source Symfony bundles, so it may be useful to understand both approaches.

These are the main **advantages** of defining controllers as services:

- The entire controller and any service passed to it can be modified via the service container configuration. This is useful when developing reusable bundles;
- Your controllers are more "sandboxed". By looking at the constructor arguments, it's easy to see what types of things this controller may or may not do;
- Since dependencies must be injected manually, it's more obvious when your controller is becoming too big (i.e. if you have many constructor arguments).

These are the main **drawbacks** of defining controllers as services:

- It takes more work to create the controllers because they don't have automatic access to the services or to the base controller shortcuts;
- The constructor of the controllers can rapidly become too complex because you must inject every single dependency needed by them;
- The code of the controllers is more verbose because you can't use the shortcuts of the base controller and you must replace them with some lines of code.

The recommendation from the *best practices* is also valid for controllers defined as services: avoid putting your business logic into the controllers. Instead, inject services that do the bulk of the work.

1. <http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html>

Defining the Controller as a Service

A controller can be defined as a service in the same way as any other class. For example, if you have the following simple controller:

Listing 35-1

```
1 // src/AppBundle/Controller/HelloController.php
2 namespace AppBundle\Controller;
3
4 use Symfony\Component\HttpFoundation\Response;
5
6 class HelloController
7 {
8     public function indexAction($name)
9     {
10         return new Response('<html><body>Hello '.$name.'!</body></html>');
11     }
12 }
```

Then you can define it as a service as follows:

Listing 35-2

```
1 # app/config/services.yml
2 services:
3     app.hello_controller:
4         class: AppBundle\Controller\HelloController
```

Referring to the Service

To refer to a controller that's defined as a service, use the single colon (:) notation. For example, to forward to the `indexAction()` method of the service defined above with the id `app.hello_controller`:

Listing 35-3

```
$this->forward('app.hello_controller:indexAction', array('name' => $name));
```



You cannot drop the **Action** part of the method name when using this syntax.

You can also route to the service by using the same notation when defining the route `_controller` value:

Listing 35-4

```
1 # app/config/routing.yml
2 hello:
3     path: /hello
4     defaults: { _controller: app.hello_controller:indexAction }
```



You can also use annotations to configure routing using a controller defined as a service. Make sure you specify the service ID in the `@Route` annotation. See the *FrameworkExtraBundle documentation*² for details.



If your controller implements the `__invoke()` method, you can simply refer to the service id (`app.hello_controller`).

2. <https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/routing.html#controller-as-service>

Alternatives to base Controller Methods

When using a controller defined as a service, it will most likely not extend the base **Controller** class. Instead of relying on its shortcut methods, you'll interact directly with the services that you need. Fortunately, this is usually pretty easy and the base *Controller class source code*³ is a great source on how to perform many common tasks.

For example, if you want to render a template instead of creating the **Response** object directly, then your code would look like this if you were extending Symfony's base controller:

Listing 35-5

```
1 // src/AppBundle/Controller/HelloController.php
2 namespace AppBundle\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5
6 class HelloController extends Controller
7 {
8     public function indexAction($name)
9     {
10         return $this->render(
11             'AppBundle:Hello:index.html.twig',
12             array('name' => $name)
13         );
14     }
15 }
```

If you look at the source code for the **render** function in Symfony's *base Controller class*⁴, you'll see that this method actually uses the **templating** service:

Listing 35-6

```
public function render($view, array $parameters = array(), Response $response = null)
{
    return $this->container->get('templating')->renderResponse($view, $parameters, $response);
}
```

In a controller that's defined as a service, you can instead inject the **templating** service and use it directly:

Listing 35-7

```
1 // src/AppBundle/Controller/HelloController.php
2 namespace AppBundle\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Templating\EngineInterface;
5 use Symfony\Component\HttpFoundation\Response;
6
7 class HelloController
8 {
9     private $templating;
10
11     public function __construct(EngineInterface $templating)
12     {
13         $this->templating = $templating;
14     }
15
16     public function indexAction($name)
17     {
18         return $this->templating->renderResponse(
19             'AppBundle:Hello:index.html.twig',
20             array('name' => $name)
21         );
22     }
23 }
```

The service definition also needs modifying to specify the constructor argument:

3. <https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php>

4. <https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php>

Listing 35-8

```

1 # app/config/services.yml
2 services:
3     app.hello_controller:
4         class: AppBundle\Controller\HelloController
5         arguments: ['@templating']

```

Rather than fetching the **templating** service from the container, you can inject *only* the exact service(s) that you need directly into the controller.



This does not mean that you cannot extend these controllers from your own base controller. The move away from the standard base controller is because its helper methods rely on having the container available which is not the case for controllers that are defined as services. It may be a good idea to extract common code into a service that's injected rather than place that code into a base controller that you extend. Both approaches are valid, exactly how you want to organize your reusable code is up to you.

Base Controller Methods and Their Service Replacements

This list explains how to replace the convenience methods of the base controller:

***createForm()*⁵ (service: form.factory)**

Listing 35-9

```

1 $formFactory->create($type, $data, $options);

```

***createFormBuilder()*⁶ (service: form.factory)**

Listing 35-10

```

1 $formFactory->createBuilder('form', $data, $options);

```

***createNotFoundException()*⁷**

Listing 35-11

```

1 new NotFoundHttpException($message, $previous);

```

***forward()*⁸ (service: http_kernel)**

Listing 35-12

```

1 use Symfony\Component\HttpKernel\HttpKernelInterface;
2 // ...
3
4 $request = ...;
5 $attributes = array_merge($path, array('_controller' => $controller));
6 $subRequest = $request->duplicate($query, null, $attributes);
7 $httpKernel->handle($subRequest, HttpKernelInterface::SUB_REQUEST);

```

***generateUrl()*⁹ (service: router)**

Listing 35-13

```

1 $router->generate($route, $params, $referenceType);

```



The ***\$referenceType*** argument must be one of the constants defined in the ***UrlGeneratorInterface*¹⁰**.

5. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_createForm
6. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_createFormBuilder
7. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_createNotFoundException
8. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_forward
9. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_generateUrl
10. <http://api.symfony.com/3.0/Symfony/Component/Routing/Generator/UrlGeneratorInterface.html>

***getDoctrine()*¹¹ (service: doctrine)**

Simply inject doctrine instead of fetching it from the container.

***getUser()*¹² (service: security.token_storage)**

```
Listing 35-14 1 $user = null;
                2 $token = $tokenStorage->getToken();
                3 if (null !== $token && is_object($token->getUser())) {
                4     $user = $token->getUser();
                5 }
```

***isGranted()*¹³ (service: security.authorization_checker)**

```
Listing 35-15 1 $authChecker->isGranted($attributes, $object);
```

***redirect()*¹⁴**

```
Listing 35-16 1 use Symfony\Component\HttpFoundation\RedirectResponse;
                2
                3 return new RedirectResponse($url, $status);
```

***render()*¹⁵ (service: templating)**

```
Listing 35-17 1 $templating->renderResponse($view, $parameters, $response);
```

***renderView()*¹⁶ (service: templating)**

```
Listing 35-18 1 $templating->render($view, $parameters);
```

***stream()*¹⁷ (service: templating)**

```
Listing 35-19 1 use Symfony\Component\HttpFoundation\StreamedResponse;
                2
                3 $templating = $this->templating;
                4 $callback = function () use ($templating, $view, $parameters) {
                5     $templating->stream($view, $parameters);
                6 }
                7
                8 return new StreamedResponse($callback);
```



getRequest has been deprecated. Instead, have an argument to your controller action method called **Request \$request**. The order of the parameters is not important, but the typehint must be provided.

11. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_getDoctrine
12. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_getUser
13. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_isGranted
14. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_redirect
15. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_render
16. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_renderView
17. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_stream



Chapter 36

How to Upload Files



Instead of handling file uploading yourself, you may consider using the *VichUploaderBundle*¹ community bundle. This bundle provides all the common operations (such as file renaming, saving and deleting) and it's tightly integrated with Doctrine ORM, MongoDB ODM, PHPCR ODM and Propel.

Imagine that you have a **Product** entity in your application and you want to add a PDF brochure for each product. To do so, add a new property called **brochure** in the **Product** entity:

Listing 36-1

```
1  // src/AppBundle/Entity/Product.php
2  namespace AppBundle\Entity;
3
4  use Doctrine\ORM\Mapping as ORM;
5  use Symfony\Component\Validator\Constraints as Assert;
6
7  class Product
8  {
9      // ...
10
11      /**
12       * @ORM\Column(type="string")
13       *
14       * @Assert\NotBlank(message="Please, upload the product brochure as a PDF file.")
15       * @Assert\File(mimeTypes={ "application/pdf" })
16       */
17      private $brochure;
18
19      public function getBrochure()
20      {
21          return $this->brochure;
22      }
23
24      public function setBrochure($brochure)
25      {
26          $this->brochure = $brochure;
27
28          return $this;
29      }
30  }
```

1. <https://github.com/dustin10/VichUploaderBundle>

Note that the type of the **brochure** column is **string** instead of **binary** or **blob** because it just stores the PDF file name instead of the file contents.

Then, add a new **brochure** field to the form that manages the **Product** entity:

Listing 36-2

```
1 // src/AppBundle/Form/ProductType.php
2 namespace AppBundle\Form;
3
4 use Symfony\Component\Form\AbstractType;
5 use Symfony\Component\Form\FormBuilderInterface;
6 use Symfony\Component\OptionsResolver\OptionsResolver;
7 use Symfony\Component\Form\Extension\Core\Type\FileType;
8
9 class ProductType extends AbstractType
10 {
11     public function buildForm(FormBuilderInterface $builder, array $options)
12     {
13         $builder
14             // ...
15             ->add('brochure', FileType::class, array('label' => 'Brochure (PDF file)'))
16             // ...
17     ;
18     }
19
20     public function configureOptions(OptionsResolver $resolver)
21     {
22         $resolver->setDefaults(array(
23             'data_class' => 'AppBundle\Entity\Product',
24         ));
25     }
26 }
```

Now, update the template that renders the form to display the new **brochure** field (the exact template code to add depends on the method used by your application to *customize form rendering*):

Listing 36-3

```
1 {# app/Resources/views/product/new.html.twig #}
2 <h1>Adding a new product</h1>
3
4 {{ form_start(form) }}
5     {# ... #}
6
7     {{ form_row(form.brochure) }}
8 {{ form_end(form) }}
```

Finally, you need to update the code of the controller that handles the form:

Listing 36-4

```
1 // src/AppBundle/Controller/ProductController.php
2 namespace AppBundle\Controller;
3
4 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use AppBundle\Entity\Product;
8 use AppBundle\Form\ProductType;
9
10 class ProductController extends Controller
11 {
12     /**
13      * @Route("/product/new", name="app_product_new")
14      */
15     public function newAction(Request $request)
16     {
17         $product = new Product();
18         $form = $this->createForm(ProductType::class, $product);
19         $form->handleRequest($request);
20
21         if ($form->isSubmitted() && $form->isValid()) {
22             // $file stores the uploaded PDF file
23             /** @var Symfony\Component\HttpFoundation\File\UploadedFile $file */
```

```

24     $file = $product->getBrochure();
25
26     // Generate a unique name for the file before saving it
27     $fileName = md5(uniqid()).'.'.$file->guessExtension();
28
29     // Move the file to the directory where brochures are stored
30     $file->move(
31         $this->getParameter('brochures_directory'),
32         $fileName
33     );
34
35     // Update the 'brochure' property to store the PDF file name
36     // instead of its contents
37     $product->setBrochure($fileName);
38
39     // ... persist the $product variable or any other work
40
41     return $this->redirect($this->generateUrl('app_product_list'));
42 }
43
44 return $this->render('product/new.html.twig', array(
45     'form' => $form->createView(),
46 ));
47 }
48 }

```

Now, create the `brochures_directory` parameter that was used in the controller to specify the directory in which the brochures should be stored:

Listing 36-5

```

1 # app/config/config.yml
2
3 # ...
4 parameters:
5     brochures_directory: '%kernel.root_dir%/../web/uploads/brochures'

```

There are some important things to consider in the code of the above controller:

1. When the form is uploaded, the `brochure` property contains the whole PDF file contents. Since this property stores just the file name, you must set its new value before persisting the changes of the entity;
2. In Symfony applications, uploaded files are objects of the `UploadedFile`² class. This class provides methods for the most common operations when dealing with uploaded files;
3. A well-known security best practice is to never trust the input provided by users. This also applies to the files uploaded by your visitors. The `UploadedFile` class provides methods to get the original file extension (`getExtension()`³), the original file size (`getClientSize()`⁴) and the original file name (`getClientOriginalName()`⁵). However, they are considered *not safe* because a malicious user could tamper that information. That's why it's always better to generate a unique name and use the `guessExtension()`⁶ method to let Symfony guess the right extension according to the file MIME type;

You can use the following code to link to the PDF brochure of a product:

Listing 36-6

```

1 <a href="{ asset('uploads/brochures/' ~ product.brochure) }" >View brochure (PDF)</a>

```

² <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/File/UploadedFile.html>
³ http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/File/UploadedFile.html#method_getExtension
⁴ http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/File/UploadedFile.html#method_getClientSize
⁵ http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/File/UploadedFile.html#method_getClientOriginalName
⁶ http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/File/UploadedFile.html#method_guessExtension



When creating a form to edit an already persisted item, the file form type still expects a *File*⁷ instance. As the persisted entity now contains only the relative file path, you first have to concatenate the configured upload path with the stored filename and create a new **File** class:

Listing 36-7

```
1 use Symfony\Component\HttpFoundation\File\File;
2 // ...
3
4 $product->setBrochure(
5     new File($this->getParameter('brochures_directory').'/'.$product->getBrochure())
6 );
```

Creating an Uploader Service

To avoid logic in controllers, making them big, you can extract the upload logic to a separate service:

Listing 36-8

```
1 // src/AppBundle/FileUploader.php
2 namespace AppBundle;
3
4 use Symfony\Component\HttpFoundation\File\UploadedFile;
5
6 class FileUploader
7 {
8     private $targetDir;
9
10    public function __construct($targetDir)
11    {
12        $this->targetDir = $targetDir;
13    }
14
15    public function upload(UploadedFile $file)
16    {
17        $fileName = md5(uniqid()).'.'.$file->guessExtension();
18
19        $file->move($this->targetDir, $fileName);
20
21        return $fileName;
22    }
23 }
```

Then, define a service for this class:

Listing 36-9

```
1 # app/config/services.yml
2 services:
3     # ...
4     app.brochure_uploader:
5         class: AppBundle\FileUploader
6         arguments: ['%brochures_directory%']
```

Now you're ready to use this service in the controller:

Listing 36-10

```
1 // src/AppBundle/Controller/ProductController.php
2
3 // ...
4 public function newAction(Request $request)
5 {
6     // ...
7
8     if ($form->isValid()) {
9         $file = $product->getBrochure();
10        $fileName = $this->get('app.brochure_uploader')->upload($file);
11    }
```

7. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/File/File.html>


```

12     $product->setBrochure($fileName);
13
14     // ...
15 }
16
17 // ...
18 }

```

Using a Doctrine Listener

If you are using Doctrine to store the Product entity, you can create a *Doctrine listener* to automatically upload the file when persisting the entity:

Listing 36-11

```

1  // src/AppBundle/EventListener/BrochureUploadListener.php
2  namespace AppBundle\EventListener;
3
4  use Symfony\Component\HttpFoundation\File\UploadedFile;
5  use Doctrine\ORM\Event\LifecycleEventArgs;
6  use Doctrine\ORM\Event\PreUpdateEventArgs;
7  use AppBundle\Entity\Product;
8  use AppBundle\FileUploader;
9
10 class BrochureUploadListener
11 {
12     private $uploader;
13
14     public function __construct(FileUploader $uploader)
15     {
16         $this->uploader = $uploader;
17     }
18
19     public function prePersist(LifecycleEventArgs $args)
20     {
21         $entity = $args->getEntity();
22
23         $this->uploadFile($entity);
24     }
25
26     public function preUpdate(PreUpdateEventArgs $args)
27     {
28         $entity = $args->getEntity();
29
30         $this->uploadFile($entity);
31     }
32
33     private function uploadFile($entity)
34     {
35         // upload only works for Product entities
36         if (!$entity instanceof Product) {
37             return;
38         }
39
40         $file = $entity->getBrochure();
41
42         // only upload new files
43         if (!$file instanceof UploadedFile) {
44             return;
45         }
46
47         $fileName = $this->uploader->upload($file);
48         $entity->setBrochure($fileName);
49     }
50 }

```

Now, register this class as a Doctrine listener:

Listing 36-12

```

1 # app/config/services.yml
2 services:
3     # ...
4     app.doctrine_brochure_listener:
5         class: AppBundle\EventListener\BrochureUploadListener
6         arguments: ['@app.brochure_uploader']
7         tags:
8             - { name: doctrine.event_listener, event: prePersist }
9             - { name: doctrine.event_listener, event: preUpdate }

```

This listeners is now automatically executed when persisting a new Product entity. This way, you can remove everything related to uploading from the controller.



This listener can also create the **File** instance based on the path when fetching entities from the database:

Listing 36-13

```

1 // ...
2 use Symfony\Component\HttpFoundation\File\File;
3
4 // ...
5 class BrochureUploadListener
6 {
7     // ...
8
9     public function postLoad(LifecycleEventArgs $args)
10    {
11        $entity = $args->getEntity();
12
13        $fileName = $entity->getBrochure();
14
15        $entity->setBrochure(new File($this->targetPath.'/'.$fileName));
16    }
17 }

```

After adding these lines, configure the listener to also listen for the **postLoad** event.



Chapter 37

How to Optimize your Development Environment for Debugging

When you work on a Symfony project on your local machine, you should use the **dev** environment (`app_dev.php` front controller). This environment configuration is optimized for two main purposes:

- Give the developer accurate feedback whenever something goes wrong (web debug toolbar, nice exception pages, profiler, ...);
- Be as similar as possible as the production environment to avoid problems when deploying the project.

Disabling the Bootstrap File and Class Caching

And to make the production environment as fast as possible, Symfony creates big PHP files in your cache containing the aggregation of PHP classes your project needs for every request. However, this behavior can confuse your debugger, because the same class can be located in two different places: the original class file and the big file which aggregates lots of classes.

This recipe shows you how you can tweak this caching mechanism to make it friendlier when you need to debug code that involves Symfony classes.

The `app_dev.php` front controller reads as follows by default:

Listing 37-1

```
1 // ...
2
3 $loader = require __DIR__.'/../app/autoload.php';
4 Debug::enable();
5
6 $kernel = new AppKernel('dev', true);
7 $kernel->loadClassCache();
8 $request = Request::createFromGlobals();
9 // ...
```

To make your debugger happier, disable the loading of all PHP class caches by removing the call to `loadClassCache()`:

Listing 37-2

```
1 // ...
2
3 $loader = require_once __DIR__.'../app/autoload.php';
4 Debug::enable();
5
6 $kernel = new AppKernel('dev', true);
7 // $kernel->loadClassCache();
8 $request = Request::createFromGlobals();
```



If you disable the PHP caches, don't forget to revert after your debugging session.

Some IDEs do not like the fact that some classes are stored in different locations. To avoid problems, you can tell your IDE to ignore the PHP cache file.



Chapter 38

How to Deploy a Symfony Application



Deploying can be a complex and varied task depending on the setup and the requirements of your application. This article is not a step-by-step guide, but is a general list of the most common requirements and ideas for deployment.

Symfony Deployment Basics

The typical steps taken while deploying a Symfony application include:

1. Upload your code to the production server;
2. Install your vendor dependencies (typically done via Composer and may be done before uploading);
3. Running database migrations or similar tasks to update any changed data structures;
4. Clearing (and optionally, warming up) your cache.

A deployment may also include other tasks, such as:

- Tagging a particular version of your code as a release in your source control repository;
- Creating a temporary staging area to build your updated setup "offline";
- Running any tests available to ensure code and/or server stability;
- Removal of any unnecessary files from the `web/` directory to keep your production environment clean;
- Clearing of external cache systems (like *Memcached*¹ or *Redis*²).

How to Deploy a Symfony Application

There are several ways you can deploy a Symfony application. Start with a few basic deployment strategies and build up from there.

1. <http://memcached.org/>
2. <http://redis.io/>

Basic File Transfer

The most basic way of deploying an application is copying the files manually via ftp/scp (or similar method). This has its disadvantages as you lack control over the system as the upgrade progresses. This method also requires you to take some manual steps after transferring the files (see Common Post-Deployment Tasks)

Using Source Control

If you're using source control (e.g. Git or SVN), you can simplify by having your live installation also be a copy of your repository. When you're ready to upgrade it is as simple as fetching the latest updates from your source control system.

This makes updating your files *easier*, but you still need to worry about manually taking other steps (see Common Post-Deployment Tasks).

Using Build Scripts and other Tools

There are also tools to help ease the pain of deployment. Some of them have been specifically tailored to the requirements of Symfony.

Capistrano³ with Symfony plugin⁴

Capistrano⁵ is a remote server automation and deployment tool written in Ruby. *Symfony plugin⁶* is a plugin to ease Symfony related tasks, inspired by *Capifony⁷* (which works only with Capistrano 2)

sf2debpkg⁸

Helps you build a native Debian package for your Symfony project.

Magallanes⁹

This Capistrano-like deployment tool is built in PHP, and may be easier for PHP developers to extend for their needs.

Fabric¹⁰

This Python-based library provides a basic suite of operations for executing local or remote shell commands and uploading/downloading files.

Deployer¹¹

This is another native PHP rewrite of Capistrano, with some ready recipes for Symfony.

Bundles

There are some *bundles that add deployment features¹²* directly into your Symfony console.

Basic scripting

You can of course use shell, *Ant¹³* or any other build tool to script the deploying of your project.

3. <http://capistranorb.com/>

4. <https://github.com/capistrano/symfony/>

5. <http://capistranorb.com/>

6. <https://github.com/capistrano/symfony/>

7. <http://capifony.org/>

8. <https://github.com/liip/sf2debpkg>

9. <https://github.com/andres-montanez/Magallanes>

10. <http://www.fabfile.org/>

11. <http://deployer.org/>

12. <http://knpbundles.com/search?q=deploy>

13. <http://blog.sznepka.pl/deploying-symfony2-applications-with-ant>

Platform as a Service Providers

The Symfony Cookbook includes detailed articles for some of the most well-known Platform as a Service (PaaS) providers:

- *Microsoft Azure*
- *Heroku*
- *Platform.sh*

Common Post-Deployment Tasks

After deploying your actual source code, there are a number of common things you'll need to do:

A) Check Requirements

Check if your server meets the requirements by running:

Listing 38-1 1 `$ php bin/symfony_requirements`

B) Configure your `app/config/parameters.yml` File

This file should *not* be deployed, but managed through the automatic utilities provided by Symfony.

C) Install/Update your Vendors

Your vendors can be updated before transferring your source code (i.e. update the **vendor/** directory, then transfer that with your source code) or afterwards on the server. Either way, just update your vendors as you normally do:

Listing 38-2 1 `$ composer install --no-dev --optimize-autoloader`



The **--optimize-autoloader** flag improves Composer's autoloader performance significantly by building a "class map". The **--no-dev** flag ensures that development packages are not installed in the production environment.



If you get a "class not found" error during this step, you may need to run **export SYMFONY_ENV=prod** before running this command so that the **post-install-cmd** scripts run in the **prod** environment.

D) Clear your Symfony Cache

Make sure you clear (and warm-up) your Symfony cache:

Listing 38-3 1 `$ php bin/console cache:clear --env=prod --no-debug`

E) Dump your Assetic Assets

If you're using Assetic, you'll also want to dump your assets:

Listing 38-4 1 `$ php bin/console assetic:dump --env=prod --no-debug`

F) Other Things!

There may be lots of other things that you need to do, depending on your setup:

- Running any database migrations
- Clearing your APC cache
- Running `assets:install` (already taken care of in `composer install`)
- Add/edit CRON jobs
- Pushing assets to a CDN
- ...

Application Lifecycle: Continuous Integration, QA, etc

While this entry covers the technical details of deploying, the full lifecycle of taking code from development up to production may have a lot more steps (think deploying to staging, QA (Quality Assurance), running tests, etc).

The use of staging, testing, QA, continuous integration, database migrations and the capability to roll back in case of failure are all strongly advised. There are simple and more complex tools and one can make the deployment as easy (or sophisticated) as your environment requires.

Don't forget that deploying your application also involves updating any dependency (typically via Composer), migrating your database, clearing your cache and other potential things like pushing assets to a CDN (see Common Post-Deployment Tasks).



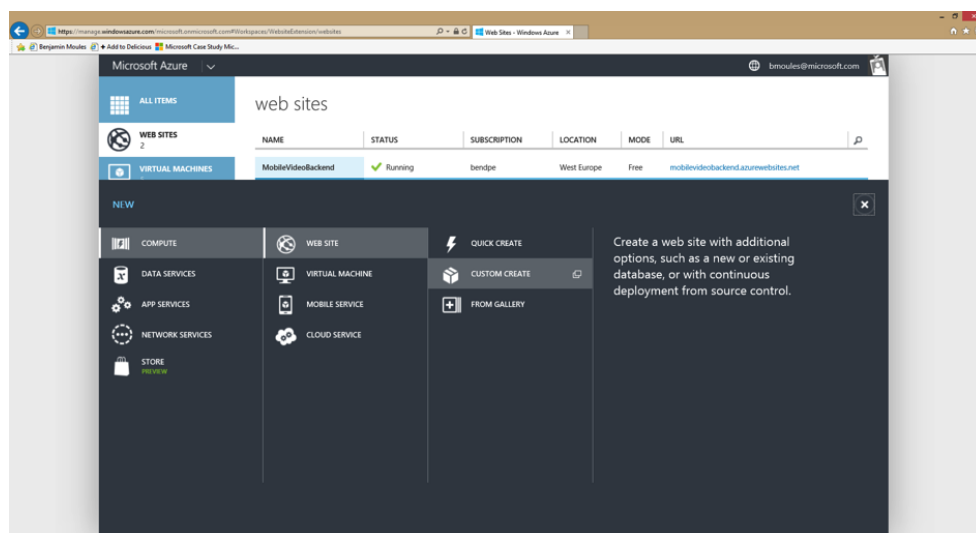
Chapter 39

Deploying to Microsoft Azure Website Cloud

This step by step cookbook describes how to deploy a small Symfony web application to the Microsoft Azure Website cloud platform. It will explain how to set up a new Azure website including configuring the right PHP version and global environment variables. The document also shows how to you can leverage Git and Composer to deploy your Symfony application to the cloud.

Setting up the Azure Website

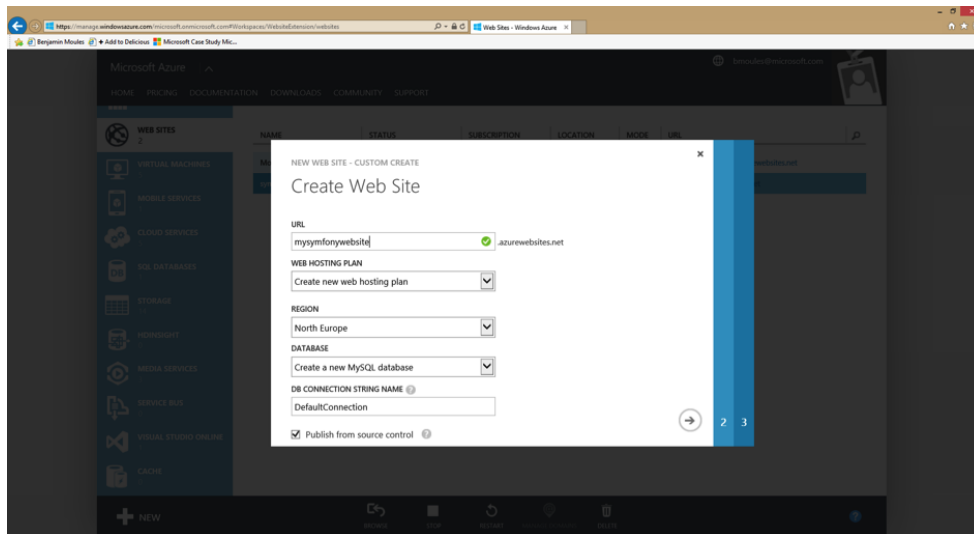
To set up a new Microsoft Azure Website, first *sign up with Azure*¹ or sign in with your credentials. Once you're connected to your *Azure Portal*² interface, scroll down to the bottom and select the **New** panel. On this panel, click **Web Site** and choose **Custom Create**:



1. <https://signup.live.com/signup.aspx>
2. <https://manage.windowsazure.com>

Step 1: Create Web Site

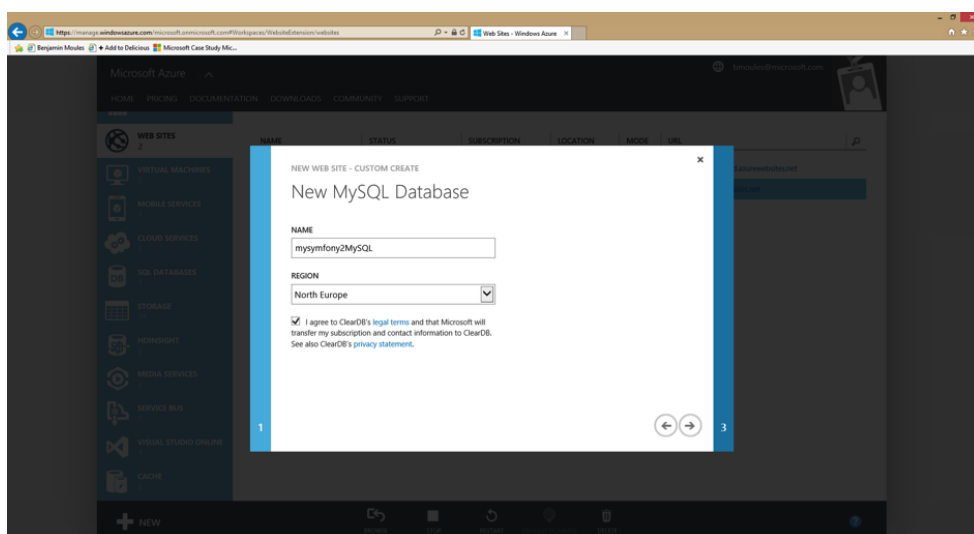
Here, you will be prompted to fill in some basic information.



For the URL, enter the URL that you would like to use for your Symfony application, then pick **Create new web hosting plan** in the region you want. By default, a *free 20 MB SQL database* is selected in the database dropdown list. In this tutorial, the Symfony app will connect to a MySQL database. Pick the **Create a new MySQL database** option in the dropdown list. You can keep the **DefaultConnection** string name. Finally, check the box **Publish from source control** to enable a Git repository and go to the next step.

Step 2: New MySQL Database

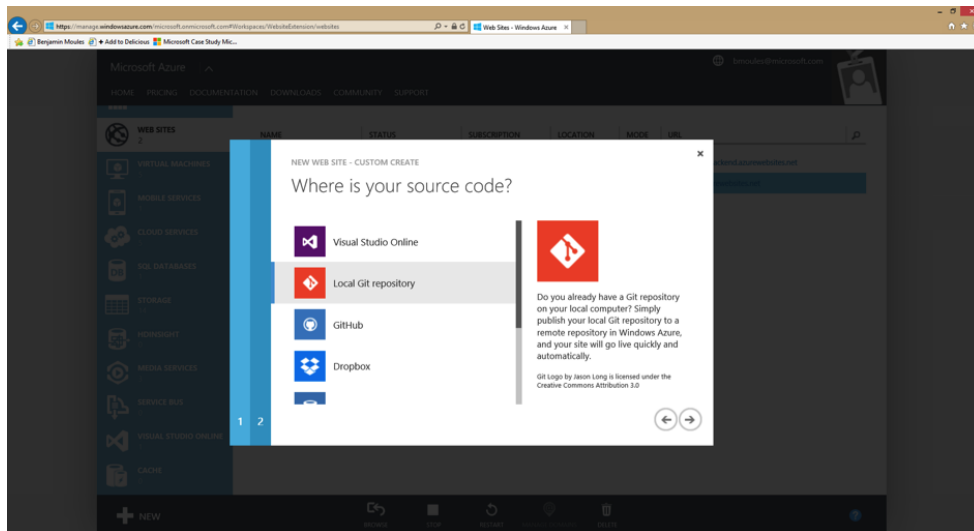
On this step, you will be prompted to set up your MySQL database storage with a database name and a region. The MySQL database storage is provided by Microsoft in partnership with ClearDB. Choose the same region you selected for the hosting plan configuration in the previous step.



Agree to the terms and conditions and click on the right arrow to continue.

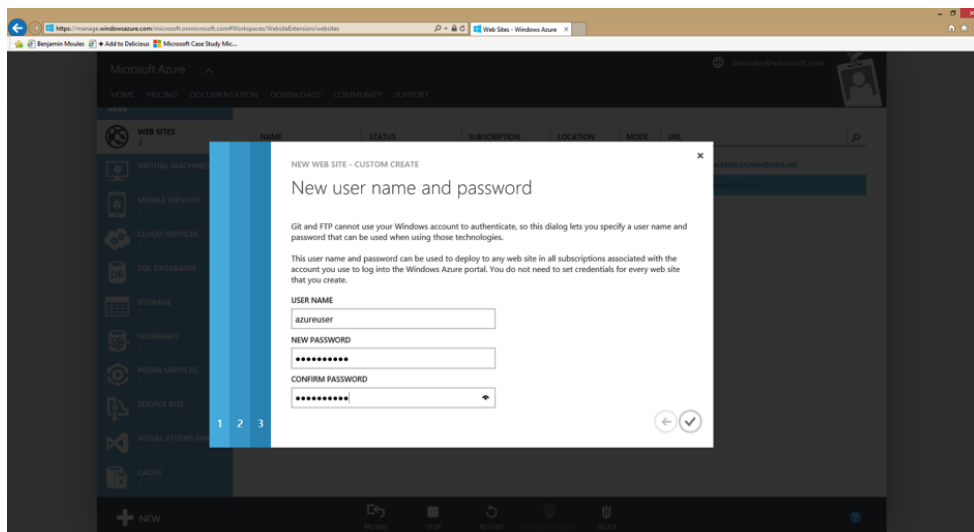
Step 3: Where Is your Source Code

Now, on the third step, select a **Local Git repository** item and click on the right arrow to configure your Azure Website credentials.

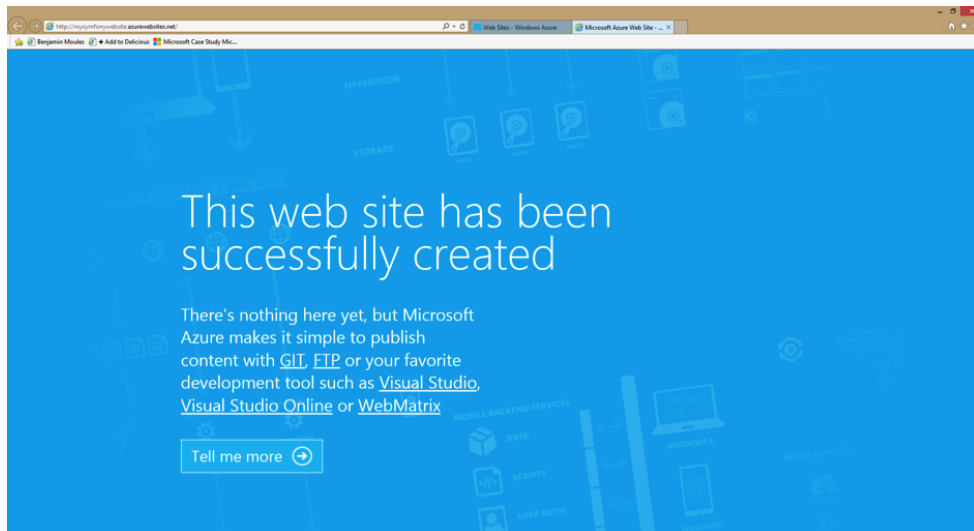


Step 4: New Username and Password

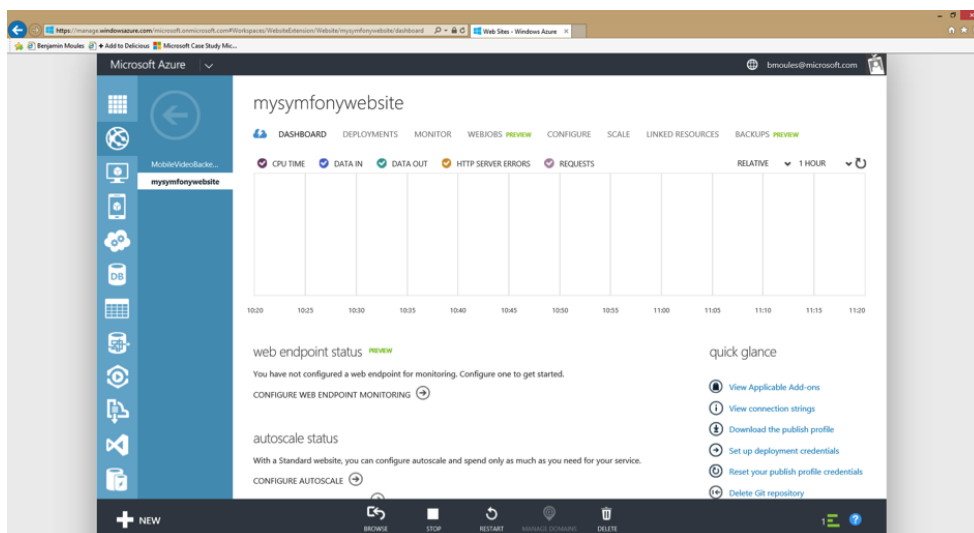
Great! You're now on the final step. Create a username and a secure password: these will become essential identifiers to connect to the FTP server and also to push your application code to the Git repository.



Congratulations! Your Azure Website is now up and running. You can check it by browsing to the Website url you configured in the first step. You should see the following display in your web browser:



The Microsoft Azure portal also provides a complete control panel for the Azure Website.



Your Azure Website is ready! But to run a Symfony site, you need to configure just a few additional things.

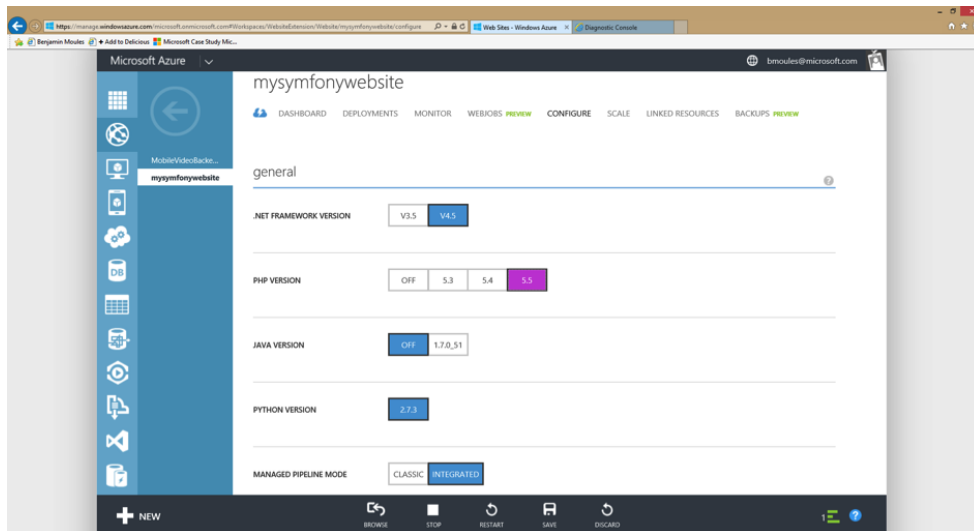
Configuring the Azure Website for Symfony

This section of the tutorial details how to configure the correct version of PHP to run Symfony. It also shows you how to enable some mandatory PHP extensions and how to properly configure PHP for a production environment.

Configuring the latest PHP Runtime

Even though Symfony only requires PHP 5.5.9 to run, it's always recommended to use the most recent PHP version whenever possible. Earlier versions are no longer supported by the PHP core team, but you can update it easily in Azure.

To update your PHP version on Azure, go to the **Configure** tab of the control panel and select the version you want.



Click the **Save** button in the bottom bar to save your changes and restart the web server.



Choosing a more recent PHP version can greatly improve runtime performance. PHP 5.5 ships with a new built-in PHP accelerator called OPcache that replaces APC. On an Azure Website, OPcache is already enabled and there is no need to install and set up APC.

The following screenshot shows the output of a *phpinfo*³ script run from an Azure Website to verify that PHP 5.5 is running with OPcache enabled.

Zend OPcache		
Opcache Caching	Up and Running	
Optimization	Enabled	
Startup	OK	
Shared memory model	win32	
Cache hits	0	
Cache misses	1	
Used memory	10710876	
Free memory	123506752	
Wasted memory	0	
Interned Strings Used memory	302624	
Interned Strings Free memory	8085684	
Cached scripts	1	
Cached keys	1	
Max keys	7963	
OOM restarts	0	
Hash keys restarts	0	
Manual restarts	0	

Directive	Local Value	Master Value
opcache.blacklist_filename	no value	no value
opcache.consistency_checks	0	0
opcache.dups_fix	Off	Off
opcache.enable	On	On
opcache.enable_cli	On	On
opcache.enable_file_override	Off	Off
opcache.error_log	no value	no value
opcache.fast_shutdown	0	0
opcache.file_update_protection	2	2
opcache.force_restart_timeout	180	180
opcache.inherited_hack	On	On
opcache.interned_strings_buffer	8	8
opcache.load_comments	1	1
opcache.log_verbosity_level	1	1

Tweaking php.ini Configuration Settings

Microsoft Azure allows you to override the `php.ini` global configuration settings by creating a custom `.user.ini` file under the project root directory (`site/wwwroot`).

Listing 39-1

```

1 ; .user.ini
2 expose_php = Off
3 memory_limit = 256M
4 upload_max_filesize = 10M

```

3. <http://php.net/manual/en/function.phpinfo.php>

None of these settings *needs* to be overridden. The default PHP configuration is already pretty good, so this is just an example to show how you can easily tweak PHP internal settings by uploading your custom **.ini** file.

You can either manually create this file on your Azure Website FTP server under the **site/wwwroot** directory or deploy it with Git. You can get your FTP server credentials from the Azure Website Control panel under the **Dashboard** tab on the right sidebar. If you want to use Git, simply put your **.user.ini** file at the root of your local repository and push your commits to your Azure Website repository.



This cookbook has a section dedicated to explaining how to configure your Azure Website Git repository and how to push the commits to be deployed. See *Deploying from Git*. You can also learn more about configuring PHP internal settings on the official *PHP MSDN documentation*⁴ page.

Enabling the PHP intl Extension

This is the tricky part of the guide! At the time of writing this cookbook, Microsoft Azure Website provided the **intl** extension, but it's not enabled by default. To enable the **intl** extension, there is no need to upload any DLL files as the **php_intl.dll** file already exists on Azure. In fact, this file just needs to be moved into the custom website extension directory.

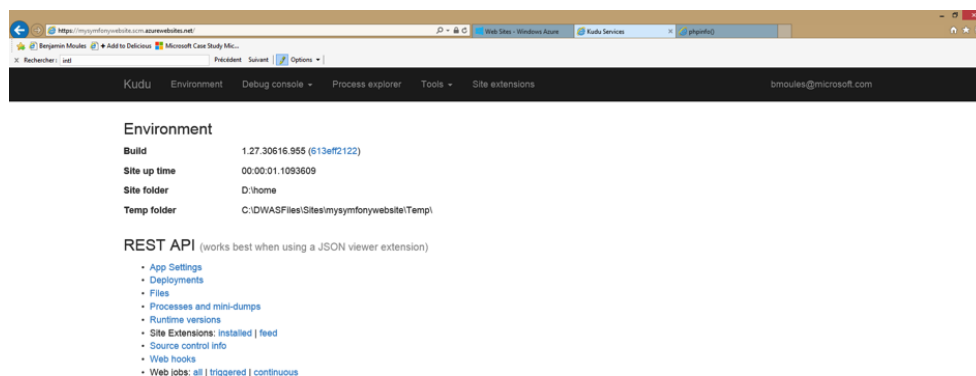


The Microsoft Azure team is currently working on enabling the **intl** PHP extension by default. In the near future, the following steps will no longer be necessary.

To get the **php_intl.dll** file under your **site/wwwroot** directory, simply access the online **Kudu** tool by browsing to the following URL:

Listing 39-2 1 [https://\[your-website-name\].scm.azurewebsites.net](https://[your-website-name].scm.azurewebsites.net)

Kudu is a set of tools to manage your application. It comes with a file explorer, a command line prompt, a log stream and a configuration settings summary page. Of course, this section can only be accessed if you're logged in to your main Azure Website account.



4. <http://blogs.msdn.com/b/silverlining/archive/2012/07/10/configuring-php-in-windows-azure-websites-with-user-ini-files.aspx>

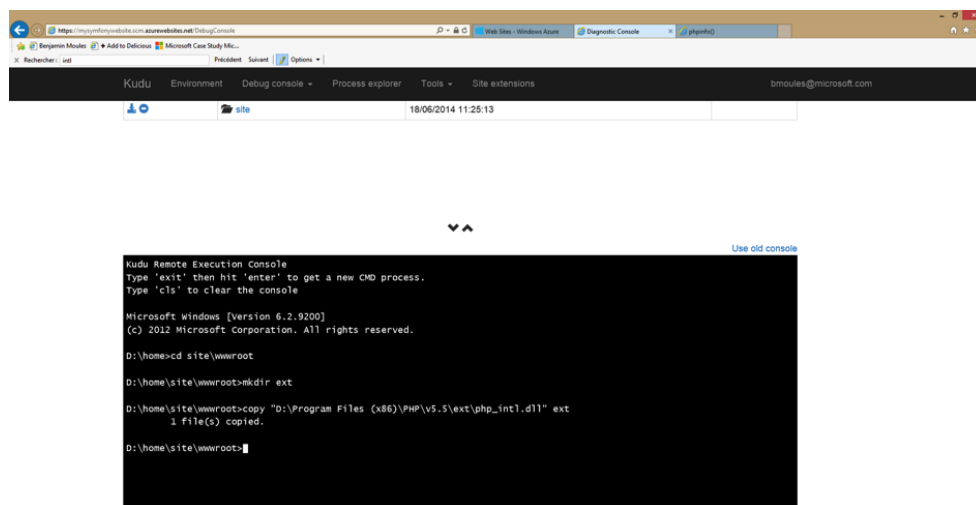
From the Kudu front page, click on the **Debug Console** navigation item in the main menu and choose **CMD**. This should open the **Debug Console** page that shows a file explorer and a console prompt below.

In the console prompt, type the following three commands to copy the original `php_intl.dll` extension file into a custom website `ext/` directory. This new directory must be created under the main directory `site/wwwroot`.

Listing 39-3

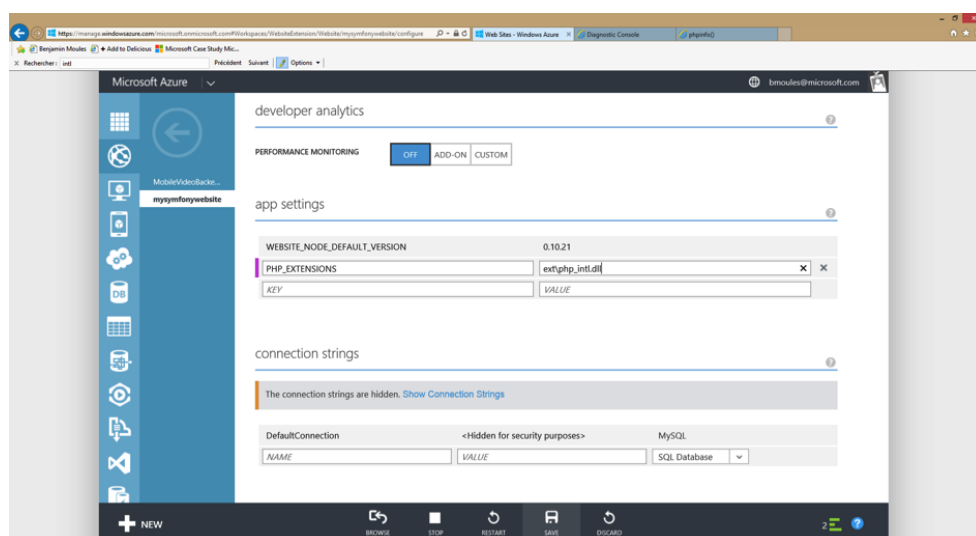
```
1 $ cd site\wwwroot
2 $ mkdir ext
3 $ copy "D:\Program Files (x86)\PHP\v5.5\ext\php_intl.dll" ext
```

The whole process and output should look like this:

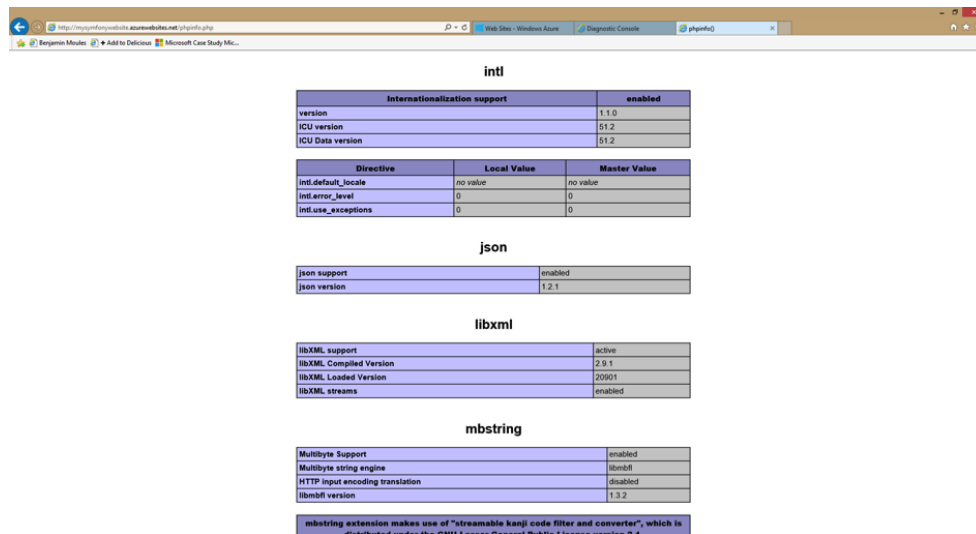


To complete the activation of the `php_intl.dll` extension, you must tell Azure Website to load it from the newly created `ext` directory. This can be done by registering a global `PHP_EXTENSIONS` environment variable from the **Configure** tab of the main Azure Website Control panel.

In the **app settings** section, register the `PHP_EXTENSIONS` environment variable with the value `ext\php_intl.dll` as shown in the screenshot below:



Hit "save" to confirm your changes and restart the web server. The PHP `Intl` extension should now be available in your web server environment. The following screenshot of a *phpinfo*⁵ page verifies the `intl` extension is properly enabled:



intl		
Internationalization support		enabled
version	1.1.0	
ICU version	51.2	
ICU Data version	51.2	
Directive	Local Value	Master Value
intl.default_locale	no value	no value
intl.error_level	0	0
intl.use_exceptions	0	0
json		
json support	enabled	
json version	1.2.1	
libxml		
libXML support	active	
libXML Compiled Version	2.9.1	
libXML Loaded Version	20001	
libXML streams	enabled	
mbstring		
MultiByte Support	enabled	
MultiByte string engine	glibc	
HTTP input encoding translation	disabled	
libmbfl version	1.3.2	
mbstring extension makes use of "streamable kanji code filter and converter", which is distributed under the GNU Lesser General Public License version 2.1.		

Great! The PHP environment setup is now complete. Next, you'll learn how to configure the Git repository and push code to production. You'll also learn how to install and configure the Symfony app after it's deployed.

Deploying from Git

First, make sure Git is correctly installed on your local machine using the following command in your terminal:

Listing 39-4 1 `$ git --version`



Get your Git from the git-scm.com⁶ website and follow the instructions to install and configure it on your local machine.

In the Azure Website Control panel, browse the **Deployment** tab to get the Git repository URL where you should push your code:

5. <http://php.net/manual/en/function.phpinfo.php>

6. <http://git-scm.com/download>


```
tomphpdevbox.cloudapp.net - PuTTY
tom@tomphpdevbox:~/symfony-azure$ git remote add azure https://userwebsite@mysymfonywebsite.scm.azurewebsites.net:443/mysymfonywebsite.git
tom@tomphpdevbox:~/symfony-azure$ git push azure master
Password for 'https://userwebsite@mysymfonywebsite.scm.azurewebsites.net:443':
Counting objects: 134, done.
Compressing objects: 100% (116/116), done.
Writing objects: 100% (134/134), 73.94 KiB | 0 bytes/s, done.
Total 134 (delta 16), reused 0 (delta 0)
remote: Updating branch 'master'.
remote: Updating submodules.
remote: Preparing deployment for commit id '37dc946eb3'.
remote: Generating deployment script.
remote: Generating deployment script for Web Site
remote: Generated deployment script files
remote: Running deployment command...
remote: Handling Basic Web Site deployment.
remote: KuduSync.NET from: 'D:\home\site\repository' to: 'D:\home\site\wwwroot'
remote: Deleting file: 'hostingstart.html'
remote: Copying file: '.gitignore'
remote: Copying file: '.travis.yml'
remote: Copying file: 'composer.json'
remote: Copying file: 'composer.lock'
remote: Copying file: 'LICENSE'
remote: Copying file: 'README.md'
remote: Copying file: 'UPGRADE-2.2.md'
remote: Copying file: 'UPGRADE-2.3.md'
remote: Copying file: 'UPGRADE-2.4.md'
remote: Copying file: 'UPGRADE.md'
remote: Copying file: 'app\.htaccess'
remote: Copying file: 'app\AppCache.php'
remote: Copying file: 'app\AppKernel.php'
remote: Copying file: 'app\autoload.php'
remote: Copying file: 'app\check.php'
remote: Copying file: 'app\console'
remote: Copying file: 'app\phpunit.xml.dist'
remote: Copying file: 'app\SymfonyRequirements.php'
remote: Copying file: 'app\cache\gitkeep'
remote: Copying file: 'app\config\config.yml'
remote: Copying file: 'app\config\config_dev.yml'
remote: Copying file: 'app\config\config_prod.yml'
```

The code of the Symfony application has now been deployed to the Azure Website which you can browse from the file explorer of the Kudu application. You should see the **app/**, **src/** and **web/** directories under your **site/wwwroot** directory on the Azure Website filesystem.

Configure the Symfony Application

PHP has been configured and your code has been pushed with Git. The last step is to configure the application and install the third party dependencies it requires that aren't tracked by Git. Switch back to the online **Console** of the Kudu application and execute the following commands in it:

Listing 39-7

```
1 $ cd site\wwwroot
2 $ curl -sS https://getcomposer.org/installer | php
3 $ php -d extension=php_intl.dll composer.phar install
```

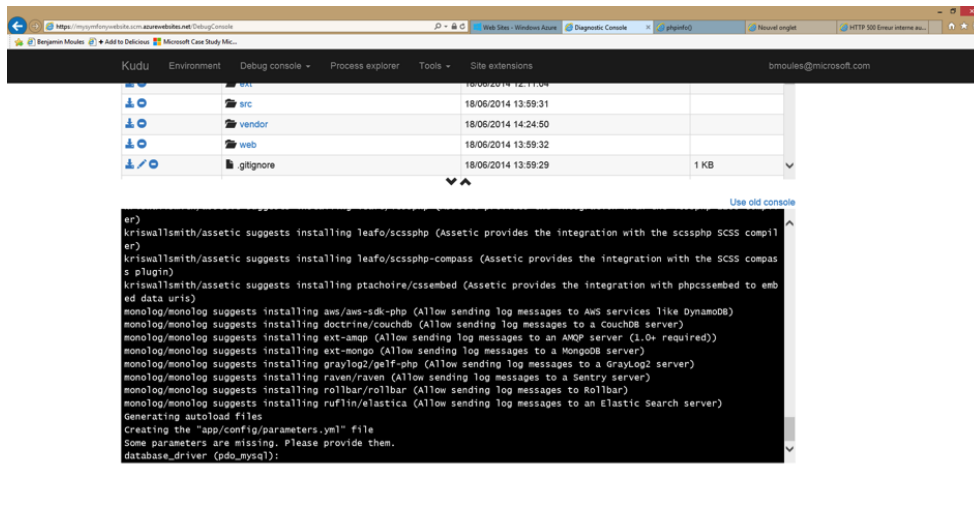
The **curl** command retrieves and downloads the Composer command line tool and installs it at the root of the **site/wwwroot** directory. Then, running the Composer **install** command downloads and installs all necessary third-party libraries.

This may take a while depending on the number of third-party dependencies you've configured in your **composer.json** file.

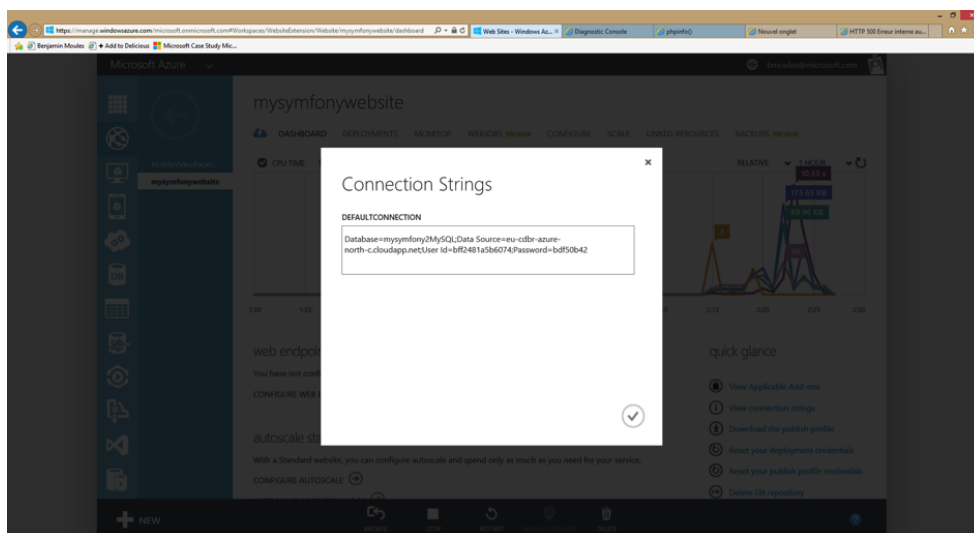


The **-d** switch allows you to quickly override/add any **php.ini** settings. In this command, we are forcing PHP to use the **intl** extension, because it is not enabled by default in Azure Website at the moment. Soon, this **-d** option will no longer be needed since Microsoft will enable the **intl** extension by default.

At the end of the **composer install** command, you will be prompted to fill in the values of some Symfony settings like database credentials, locale, mailer credentials, CSRF token protection, etc. These parameters come from the **app/config/parameters.yml.dist** file.



The most important thing in this cookbook is to correctly set up your database settings. You can get your MySQL database settings on the right sidebar of the **Azure Website Dashboard** panel. Simply click on the **View Connection Strings** link to make them appear in a pop-in.



The displayed MySQL database settings should be something similar to the code below. Of course, each value depends on what you've already configured.

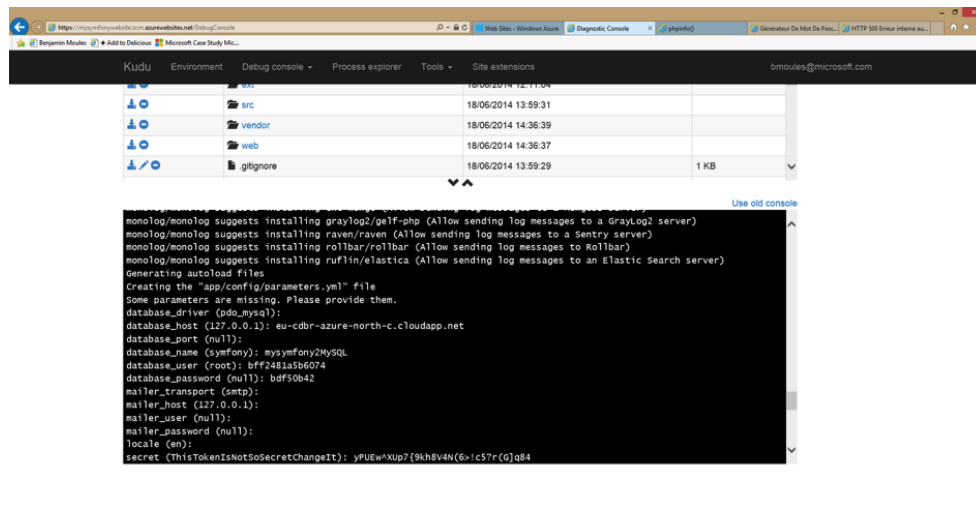
Listing 39-8 1 Database=mysymfonyMySQL;Data Source=eu-cdbr-azure-north-c.cloudapp.net;User Id=bff2481a5b6074;Password=bd50b42

Switch back to the console and answer the prompted questions and provide the following answers. Don't forget to adapt the values below with your real values from the MySQL connection string.

Listing 39-9 1 database_driver: pdo_mysql
2 database_host: u-cdbr-azure-north-c.cloudapp.net
3 database_port: null
4 database_name: mysymfonyMySQL
5 database_user: bff2481a5b6074
6 database_password: bd50b42
7 // ...

Don't forget to answer all the questions. It's important to set a unique random string for the **secret** variable. For the mailer configuration, Azure Website doesn't provide a built-in mailer service. You

should consider configuring the host-name and credentials of some other third-party mailing service if your application needs to send emails.



Your Symfony application is now configured and should be almost operational. The final step is to build the database schema. This can easily be done with the command line interface if you're using Doctrine. In the online **Console** tool of the Kudu application, run the following command to mount the tables into your MySQL database.

Listing 39-10 1 `$ php bin/console doctrine:schema:update --force`

This command builds the tables and indexes for your MySQL database. If your Symfony application is more complex than a basic Symfony Standard Edition, you may have additional commands to execute for setup (see *How to Deploy a Symfony Application*).

Make sure that your application is running by browsing the **app.php** front controller with your web browser and the following URL:

Listing 39-11 1 `http://<your-website-name>.azurewebsites.net/web/app.php`

If Symfony is correctly installed, you should see the front page of your Symfony application showing.

Configure the Web Server

At this point, the Symfony application has been deployed and works perfectly on the Azure Website. However, the **web** folder is still part of the URL, which you definitely don't want. But don't worry! You can easily configure the web server to point to the **web** folder and remove the **web** in the URL (and guarantee that nobody can access files outside of the **web** directory.)

To do this, create and deploy (see previous section about Git) the following **web.config** file. This file must be located at the root of your project next to the **composer.json** file. This file is the Microsoft IIS Server equivalent to the well-known **.htaccess** file from Apache. For a Symfony application, configure it with the following content:

Listing 39-12 1 `<!-- web.config -->`
2 `<?xml version="1.0" encoding="UTF-8"?>`
3 `<configuration>`
4 `<system.webServer>`
5 `<rewrite>`

```

6     <rules>
7         <clear />
8         <rule name="BlockAccessToPublic" patternSyntax="Wildcard" stopProcessing="true">
9             <match url="*" />
10            <conditions logicalGrouping="MatchAll" trackAllCaptures="false">
11                <add input="{URL}" pattern="/web/*" />
12            </conditions>
13            <action type="CustomResponse" statusCode="403" statusReason="Forbidden: Access is denied."
14 statusDescription="You do not have permission to view this directory or page using the credentials that you
15 supplied." />
16        </rule>
17        <rule name="RewriteAssetsToPublic" stopProcessing="true">
18            <match url="^(.*)"(.css|.js|.jpg|.png|.gif)$" />
19            <conditions logicalGrouping="MatchAll" trackAllCaptures="false">
20            </conditions>
21            <action type="Rewrite" url="web/{R:0}" />
22        </rule>
23        <rule name="RewriteRequestsToPublic" stopProcessing="true">
24            <match url="^(.*)" />
25            <conditions logicalGrouping="MatchAll" trackAllCaptures="false">
26            </conditions>
27            <action type="Rewrite" url="web/app.php/{R:0}" />
28        </rule>
29    </rules>
30 </rewrite>
    </system.webServer>
</configuration>

```

As you can see, the latest rule **RewriteRequestsToPublic** is responsible for rewriting any URLs to the **web/app.php** front controller which allows you to skip the **web/** folder in the URL. The first rule called **BlockAccessToPublic** matches all URL patterns that contain the **web/** folder and serves a **403 Forbidden** HTTP response instead. This example is based on Benjamin Eberlei's sample you can find on GitHub in the *SymfonyAzureEdition*⁷ bundle.

Deploy this file under the **site/wwwroot** directory of the Azure Website and browse to your application without the **web/app.php** segment in the URL.

Conclusion

Nice work! You've now deployed your Symfony application to the Microsoft Azure Website Cloud platform. You also saw that Symfony can be easily configured and executed on a Microsoft IIS web server. The process is simple and easy to implement. And as a bonus, Microsoft is continuing to reduce the number of steps needed so that deployment becomes even easier.

7. <https://github.com/beberlei/symfony-azure-edition/>



Chapter 40

Deploying to Heroku Cloud

This step by step cookbook describes how to deploy a Symfony web application to the Heroku cloud platform. Its contents are based on *the original article*¹ published by Heroku.

Setting up

To set up a new Heroku website, first *sign up with Heroku*² or sign in with your credentials. Then download and install the *Heroku Toolbelt*³ on your local computer.

You can also check out the *getting Started with PHP on Heroku*⁴ guide to gain more familiarity with the specifics of working with PHP applications on Heroku.

Preparing your Application

Deploying a Symfony application to Heroku doesn't require any change in its code, but it requires some minor tweaks to its configuration.

By default, the Symfony app will log into your application's `app/log/` directory. This is not ideal as Heroku uses an *ephemeral file system*⁵. On Heroku, the best way to handle logging is using *Logplex*⁶. And the best way to send log data to Logplex is by writing to `STDERR` or `STDOUT`. Luckily, Symfony uses the excellent Monolog library for logging. So, a new log destination is just a change to a config file away.

Open the `app/config/config_prod.yml` file, locate the `monolog/handlers/nested` section (or create it if it doesn't exist yet) and change the value of `path` from `"%kernel.logs_dir%/%kernel.environment%.log"` to `"php://stderr"`:

Listing 40-1

```
1 # app/config/config_prod.yml
2 monolog:
```

-
1. <https://devcenter.heroku.com/articles/getting-started-with-symfony2>
 2. <https://signup.heroku.com/signup/dc>
 3. <https://devcenter.heroku.com/articles/getting-started-with-php#set-up>
 4. <https://devcenter.heroku.com/articles/getting-started-with-php>
 5. <https://devcenter.heroku.com/articles/dynos#ephemeral-filesystem>
 6. <https://devcenter.heroku.com/articles/logplex>

```

3     # ...
4     handlers:
5         # ...
6         nested:
7             # ...
8             path: 'php://stderr'

```

Once the application is deployed, run `heroku logs --tail` to keep the stream of logs from Heroku open in your terminal.

Creating a new Application on Heroku

To create a new Heroku application that you can push to, use the CLI `create` command:

Listing 40-2

```

1 $ heroku create
2
3 Creating mighty-hamlet-1981 in organization heroku... done, stack is cedar
4 http://mighty-hamlet-1981.herokuapp.com/ | git@heroku.com:mighty-hamlet-1981.git
5 Git remote heroku added

```

You are now ready to deploy the application as explained in the next section.

Deploying your Application on Heroku

Before your first deploy, you need to do just three more things, which are explained below:

1. Create a Procfile
2. Set the Environment to prod
3. Push your Code to Heroku

1) Create a Procfile

By default, Heroku will launch an Apache web server together with PHP to serve applications. However, two special circumstances apply to Symfony applications:

1. The document root is in the `web/` directory and not in the root directory of the application;
2. The Composer `bin-dir`, where vendor binaries (and thus Heroku's own boot scripts) are placed, is `bin/`, and not the default `vendor/bin`.



Vendor binaries are usually installed to `vendor/bin` by Composer, but sometimes (e.g. when running a Symfony Standard Edition project!), the location will be different. If in doubt, you can always run `composer config bin-dir` to figure out the right location.

Create a new file called **Procfile** (without any extension) at the root directory of the application and add just the following content:

Listing 40-3

```

1 web: bin/heroku-php-apache2 web/

```



If you prefer to use Nginx, which is also available on Heroku, you can create a configuration file for it and point to it from your Procfile as described in the *Heroku documentation*⁷:

Listing 40-4

```

1 web: bin/heroku-php-nginx -C nginx_app.conf web/

```

7. <https://devcenter.heroku.com/articles/custom-php-settings#nginx>

If you prefer working on the command console, execute the following commands to create the **Procfile** file and to add it to the repository:

```
Listing 40-5 1 $ echo "web: bin/heroku-php-apache2 web/" > Procfile
2 $ git add .
3 $ git commit -m "Procfile for Apache and PHP"
4 [master 35075db] Procfile for Apache and PHP
5 1 file changed, 1 insertion(+)
```

2) Set the Environment to prod

During a deployment, Heroku runs **composer install --no-dev** to install all the dependencies your application requires. However, typical *post-install-commands*⁸ in **composer.json**, e.g. to install assets or clear (or pre-warm) caches, run using Symfony's **dev** environment by default.

This is clearly not what you want - the app runs in "production" (even if you use it just for an experiment, or as a staging environment), and so any build steps should use the same **prod** environment as well.

Thankfully, the solution to this problem is very simple: Symfony will pick up an environment variable named **SYMFONY_ENV** and use that environment if nothing else is explicitly set. As Heroku exposes all *config vars*⁹ as environment variables, you can issue a single command to prepare your app for a deployment:

```
Listing 40-6 1 $ heroku config:set SYMFONY_ENV=prod
```



Be aware that dependencies from **composer.json** listed in the **require-dev** section are never installed during a deploy on Heroku. This may cause problems if your Symfony environment relies on such packages. The solution is to move these packages from **require-dev** to the **require** section.

3) Push your Code to Heroku

Next up, it's finally time to deploy your application to Heroku. If you are doing this for the very first time, you may see a message such as the following:

```
Listing 40-7 1 The authenticity of host 'heroku.com (50.19.85.132)' can't be established.
2 RSA key fingerprint is 8b:48:5e:67:0e:c9:16:47:32:f2:87:0c:1f:c8:60:ad.
3 Are you sure you want to continue connecting (yes/no)?
```

In this case, you need to confirm by typing **yes** and hitting **<Enter>** key - ideally after you've *verified that the RSA key fingerprint is correct*¹⁰.

Then, deploy your application executing this command:

```
Listing 40-8 1 $ git push heroku master
2
3 Initializing repository, done.
4 Counting objects: 130, done.
5 Delta compression using up to 4 threads.
6 Compressing objects: 100% (107/107), done.
7 Writing objects: 100% (130/130), 70.88 KiB | 0 bytes/s, done.
8 Total 130 (delta 17), reused 0 (delta 0)
9
10 -----> PHP app detected
```

8. <https://getcomposer.org/doc/articles/scripts.md>

9. <https://devcenter.heroku.com/articles/config-vars>

10. <https://devcenter.heroku.com/articles/git-repository-ssh-fingerprints>


```

11
12 -----> Setting up runtime environment...
13     - PHP 5.5.12
14     - Apache 2.4.9
15     - Nginx 1.4.6
16
17 -----> Installing PHP extensions:
18     - opcache (automatic; bundled, using 'ext-opcache.ini')
19
20 -----> Installing dependencies...
21     Composer version 64ac32fca9e64eb38e50abfadc6eb6f2d0470039 2014-05-24 20:57:50
22     Loading composer repositories with package information
23     Installing dependencies from lock file
24     - ...
25
26     Generating optimized autoload files
27     Creating the "app/config/parameters.yml" file
28     Clearing the cache for the dev environment with debug true
29     Installing assets using the hard copy option
30     Installing assets for Symfony\Bundle\FrameworkBundle into web/bundles/framework
31     Installing assets for Acme\DemoBundle into web/bundles/acmedemo
32     Installing assets for Sensio\Bundle\DistributionBundle into web/bundles/sensiodistribution
33
34 -----> Building runtime environment...
35
36 -----> Discovering process types
37     Procfile declares types -> web
38
39 -----> Compressing... done, 61.5MB
40
41 -----> Launching... done, v3
42     http://mighty-hamlet-1981.herokuapp.com/ deployed to Heroku
43
44 To git@heroku.com:mighty-hamlet-1981.git
45 * [new branch]      master -> master

```

And that's it! If you now open your browser, either by manually pointing it to the URL **heroku create** gave you, or by using the Heroku Toolbelt, the application will respond:

Listing 40-9

```

1 $ heroku open
2 Opening mighty-hamlet-1981... done

```

You should be seeing your Symfony application in your browser.



If you take your first steps on Heroku using a fresh installation of the Symfony Standard Edition, you may run into a 404 page not found error. This is because the route for `/` is defined by the `AcmeDemoBundle`, but the `AcmeDemoBundle` is only loaded in the dev environment (check out your `AppKernel` class). Try opening `/app/example` from the `AppBundle`.

Custom Compile Steps

If you wish to execute additional custom commands during a build, you can leverage Heroku's *custom compile steps*¹¹. Imagine you want to remove the `dev` front controller from your production environment on Heroku in order to avoid a potential vulnerability. Adding a command to remove `web/app_dev.php` to Composer's *post-install-commands*¹² would work, but it also removes the controller in your local development environment on each `composer install` or `composer update` respectively. Instead, you can add a *custom Composer command*¹³ named `compile` (this key name is a Heroku convention)

11. <https://devcenter.heroku.com/articles/php-support#custom-compile-step>

12. <https://getcomposer.org/doc/articles/scripts.md>

13. <https://getcomposer.org/doc/articles/scripts.md#writing-custom-commands>

to the **scripts** section of your **composer.json**. The listed commands hook into Heroku's deploy process:

```
Listing 40-10 1 {
2     "scripts": {
3         "compile": [
4             "rm web/app_dev.php"
5         ]
6     }
7 }
```

This is also very useful to build assets on the production system, e.g. with Assetic:

```
Listing 40-11 1 {
2     "scripts": {
3         "compile": [
4             "bin/console assetic:dump"
5         ]
6     }
7 }
```



Node.js Dependencies

Building assets may depend on node packages, e.g. **uglifyjs** or **uglifycss** for asset minification. Installing node packages during the deploy requires a node installation. But currently, Heroku compiles your app using the PHP buildpack, which is auto-detected by the presence of a **composer.json** file, and does not include a node installation. Because the Node.js buildpack has a higher precedence than the PHP buildpack (see *Heroku buildpacks*¹⁴), adding a **package.json** listing your node dependencies makes Heroku opt for the Node.js buildpack instead:

```
Listing 40-12 1 {
2     "name": "myApp",
3     "engines": {
4         "node": "0.12.x"
5     },
6     "dependencies": {
7         "uglifycss": "*",
8         "uglify-js": "*"
9     }
10 }
```

With the next deploy, Heroku compiles your app using the Node.js buildpack and your npm packages become installed. On the other hand, your **composer.json** is now ignored. To compile your app with both buildpacks, Node.js *and* PHP, you need to use both buildpacks. To override buildpack auto-detection, you need to explicitly set the buildpack:

```
Listing 40-13 1 $ heroku buildpacks:set heroku/nodejs
2 Buildpack set. Next release on your-application will use heroku/nodejs.
3 Run git push heroku master to create a new release using this buildpack.
4 $ heroku buildpacks:set heroku/php --index 2
5 Buildpack set. Next release on your-application will use:
6   1. heroku/nodejs
7   2. heroku/php
8 Run git push heroku master to create a new release using these buildpacks.
```

With the next deploy, you can benefit from both buildpacks. This setup also enables your Heroku environment to make use of node based automatic build tools like *Grunt*¹⁵ or *gulp*¹⁶.

14. <https://devcenter.heroku.com/articles/buildpacks>

15. <http://gruntjs.com>

16. <http://gulpjs.com>



Chapter 41

Deploying to Platform.sh

This step-by-step cookbook describes how to deploy a Symfony web application to *Platform.sh*¹. You can read more about using Symfony with Platform.sh on the official *Platform.sh documentation*².

Deploy an Existing Site

In this guide, it is assumed your codebase is already versioned with Git.

Get a Project on Platform.sh

You need to subscribe to a *Platform.sh project*³. Choose the development plan and go through the checkout process. Once your project is ready, give it a name and choose: **Import an existing site**.

Prepare Your Application

To deploy your Symfony application on Platform.sh, you simply need to add a `.platform.app.yaml` at the root of your Git repository which will tell Platform.sh how to deploy your application (read more about *Platform.sh configuration files*⁴).

Listing 41-1

```
1  # .platform.app.yaml
2
3  # This file describes an application. You can have multiple applications
4  # in the same project.
5
6  # The name of this app. Must be unique within a project.
7  name: myphpproject
8
9  # The type of the application to build.
10 type: php:5.6
11 build:
12     flavor: symfony
```

-
1. <https://platform.sh>
 2. <https://docs.platform.sh/toolstacks/symfony/symfony-getting-started>
 3. <https://marketplace.commerceguys.com/platform/buy-now>
 4. <https://docs.platform.sh/reference/configuration-files>

```

13
14 # The relationships of the application with services or other applications.
15 # The left-hand side is the name of the relationship as it will be exposed
16 # to the application in the PLATFORM_RELATIONSHIPS variable. The right-hand
17 # side is in the form `<service name>:<endpoint name>`.
18 relationships:
19     database: 'mysql:mysql'
20
21 # The configuration of app when it is exposed to the web.
22 web:
23     # The public directory of the app, relative to its root.
24     document_root: '/web'
25     # The front-controller script to send non-static requests to.
26     passthru: '/app.php'
27
28 # The size of the persistent disk of the application (in MB).
29 disk: 2048
30
31 # The mounts that will be performed when the package is deployed.
32 mounts:
33     '/var/cache': 'shared:files/cache'
34     '/var/logs': 'shared:files/logs'
35
36 # The hooks that will be performed when the package is deployed.
37 hooks:
38     build: |
39         rm web/app_dev.php
40         bin/console --env=prod assetic:dump --no-debug
41     deploy: |
42         bin/console --env=prod cache:clear

```

For best practices, you should also add a `.platform` folder at the root of your Git repository which contains the following files:

Listing 41-2

```

1 # .platform/routes.yaml
2 "http://{default}":
3     type: upstream
4     # the first part should be your project name
5     upstream: 'myphpproject:php'

```

Listing 41-3

```

1 # .platform/services.yaml
2 mysql:
3     type: mysql
4     disk: 2048

```

An example of these configurations can be found on *GitHub*⁵. The list of *available services*⁶ can be found on the Platform.sh documentation.

Configure Database Access

Platform.sh overrides your database specific configuration via importing the following file (it's your role to add this file to your code base):

Listing 41-4

```

1 // app/config/parameters_platform.php
2 <?php
3 $relationships = getenv("PLATFORM_RELATIONSHIPS");
4 if (!$relationships) {
5     return;
6 }
7
8 $relationships = json_decode(base64_decode($relationships), true);
9

```

5. <https://github.com/platformsh/platformsh-examples>

6. <https://docs.platform.sh/reference/configuration-files/#configure-services>

```

10 foreach ($relationships['database'] as $endpoint) {
11     if (empty($endpoint['query']['is_master'])) {
12         continue;
13     }
14
15     $container->setParameter('database_driver', 'pdo_' . $endpoint['scheme']);
16     $container->setParameter('database_host', $endpoint['host']);
17     $container->setParameter('database_port', $endpoint['port']);
18     $container->setParameter('database_name', $endpoint['path']);
19     $container->setParameter('database_user', $endpoint['username']);
20     $container->setParameter('database_password', $endpoint['password']);
21     $container->setParameter('database_path', '');
22 }
23
24 # Store session into /tmp.
25 ini_set('session.save_path', '/tmp/sessions');

```

Make sure this file is listed in your *imports*:

Listing 41-5

```

1 # app/config/config.yml
2 imports:
3     - { resource: parameters_platform.php }

```

Deploy your Application

Now you need to add a remote to Platform.sh in your Git repository (copy the command that you see on the Platform.sh web UI):

Listing 41-6

```

1 $ git remote add platform [PROJECT-ID]@git.[CLUSTER].platform.sh:[PROJECT-ID].git

```

PROJECT-ID

Unique identifier of your project. Something like kjh43kbobssae

CLUSTER

Server location where your project is deployed. It can be eu or us

Commit the Platform.sh specific files created in the previous section:

Listing 41-7

```

1 $ git add .platform.app.yml .platform/*
2 $ git add app/config/config.yml app/config/parameters_platform.php
3 $ git commit -m "Adding Platform.sh configuration files."

```

Push your code base to the newly added remote:

Listing 41-8

```

1 $ git push platform master

```

That's it! Your application is being deployed on Platform.sh and you'll soon be able to access it in your browser.

Every code change that you do from now on will be pushed to Git in order to redeploy your environment on Platform.sh.

More information about *migrating your database and files*⁷ can be found on the Platform.sh documentation.

7. <https://docs.platform.sh/toolstacks/php/symfony/migrate-existing-site/>

Deploy a new Site

You can start a new *Platform.sh project*⁸. Choose the development plan and go through the checkout process.

Once your project is ready, give it a name and choose: **Create a new site**. Choose the *Symfony* stack and a starting point such as *Standard*.

That's it! Your Symfony application will be bootstrapped and deployed. You'll soon be able to see it in your browser.

8. <https://marketplace.commerceguys.com/platform/buy-now>



Chapter 42

Deploying to fortrabbit

This step-by-step cookbook describes how to deploy a Symfony web application to *fortrabbit*¹. You can read more about using Symfony with fortrabbit on the official fortrabbit *Symfony install guide*².

Setting up fortrabbit

Before getting started, you should have done a few things on the fortrabbit side:

- *Sign up*³;
- Add an SSH key to your Account (to deploy via Git);
- Create an App.

Preparing your Application

You don't need to change any code to deploy a Symfony application to fortrabbit. But it requires some minor tweaks to its configuration.

Configure Logging

Per default Symfony logs to a file. Modify the `app/config/config_prod.yml` file to redirect it to *error_log*⁴:

Listing 42-1

```
1 # app/config/config_prod.yml
2 monolog:
3     # ...
4     handlers:
5         nested:
6             type: error_log
```

1. <https://www.fortrabbit.com>

2. <https://help.fortrabbit.com/install-symfony>

3. <https://dashboard.fortrabbit.com>

4. <http://php.net/manual/en/function.error-log.php>

Configuring Database Access & Session Handler

You can use the fortrabbit App Secrets to attain your database credentials. Create the file `app/config/config_prod_secrets.php` with the following contents:

```
Listing 42-2 1 // get the path to the secrets.json file
2 $secrets = getenv("APP_SECRETS")
3 if (!$secrets) {
4     return;
5 }
6
7 // read the file and decode json to an array
8 $secrets = json_decode(file_get_contents($secrets), true);
9
10 // set database parameters to the container
11 if (isset($secrets['MYSQL'])) {
12     $container->setParameter('database_driver', 'pdo_mysql');
13     $container->setParameter('database_host', $secrets['MYSQL']['HOST']);
14     $container->setParameter('database_name', $secrets['MYSQL']['DATABASE']);
15     $container->setParameter('database_user', $secrets['MYSQL']['USER']);
16     $container->setParameter('database_password', $secrets['MYSQL']['PASSWORD']);
17 }
18
19 // check if the Memcache component is present
20 if (isset($secrets['MEMCACHE'])) {
21     $memcache = $secrets['MEMCACHE'];
22     $handlers = array();
23
24     foreach (range(1, $memcache['COUNT']) as $num) {
25         $handlers[] = $memcache['HOST'].$num.':'.$memcache['PORT'].$num;
26     }
27
28     // apply ini settings
29     ini_set('session.save_handler', 'memcached');
30     ini_set('session.save_path', implode(',', $handlers));
31
32     if ("2" === $memcache['COUNT']) {
33         ini_set('memcached.sess_number_of_replicas', 1);
34         ini_set('memcached.sess_consistent_hash', 1);
35         ini_set('memcached.sess_binary', 1);
36     }
37 }
```

Make sure this file is imported into the main config file:

```
Listing 42-3 1 # app/config/config_prod.yml
2     imports:
3         - { resource: config.yml }
4         - { resource: config_prod_secrets.php }
5
6     # ..
7     framework:
8         session:
9             # set handler_id to null to use default session handler from php.ini (memcached)
10             handler_id: ~
11     # ..
```

Configuring the Environment in the Dashboard

PHP Settings

The PHP version and enabled extensions are configurable under the PHP settings of your App within the fortrabbit Dashboard.

Environment Variables

Set the `SYMFONY_ENV` environment variable to **prod** to make sure the right config files get loaded. ENV vars are configurable in fortrabbit Dashboard as well.

Document Root

The document root is configurable for every custom domain you setup for your App. The default is `/htdocs`, but for Symfony you probably want to change it to `/htdocs/web`. You also do so in the fortrabbit Dashboard under **Domain** settings.

Deploying to fortrabbit

It is assumed that your codebase is under version-control with Git and dependencies are managed with Composer (locally).

Every time you push to fortrabbit composer install runs before your code gets deployed. To finetune the deployment behavior put a *fortrabbit.yml*⁵ deployment file (optional) in the project root.

Add fortrabbit as a (additional) Git remote and add your configuration changes:

Listing 42-4

```
1 $ git remote add fortrabbit git@deploy.eu2.frbt.com:<your-app>.git
2 $ git add composer.json composer.lock
3 $ git add app/config/config_prod_secrets.php
```

Commit and push

Listing 42-5

```
1 $ git commit -m 'fortrabbit config'
2 $ git push fortrabbit master -u
```



Replace `<your-app>` with the name of your fortrabbit App.

Listing 42-6

```
1 Commit received, starting build of branch master
2
3 ----- f -----
4
5 B U I L D
6
7 Checksum:
8   def1bb29911a62de26b1ddac6ef97fc76a5c647b
9
10 Deployment file:
11   fortrabbit.yml
12
13 Pre-script:
14   not found
15   Oms
16
17 Composer:
18   - - -
19 Loading composer repositories with package information
20 Installing dependencies (including require-dev) from lock file
21 Nothing to install or update
22 Generating autoload files
23
```

5. <https://help.fortrabbit.com/deployment-file-v2>

```
24 - - -
25 172ms
26
27 Post-script:
28   not found
29   0ms
30
31 R E L E A S E
32
33 Packaging:
34   930ms
35
36 Revision:
37   1455788127289043421.def1bb29911a62de26b1ddac6ef97fc76a5c647b
38
39 Size:
40   9.7MB
41
42 Uploading:
43   500ms
44
45 Build & release done in 1625ms, now queued for final distribution.
```



The first **git push** takes much longer as all composer dependencies get downloaded. All subsequent deploys are done within seconds.

That's it! Your application is being deployed on fortrabbit. More information about *database migrations and tunneling*⁶ can be found in the fortrabbit documentation.

6. <https://help.fortrabbit.com/install-symfony-2#toc-migrate-amp-other-database-commands>



Chapter 43

How to use Doctrine Extensions: Timestampable, Sluggable, Translatable, etc.

Doctrine2 is very flexible, and the community has already created a series of useful Doctrine extensions to help you with common entity-related tasks.

One library in particular - the *DoctrineExtensions*¹ library - provides integration functionality for *Sluggable*², *Translatable*³, *Timestampable*⁴, *Loggable*⁵, *Tree*⁶ and *Sortable*⁷ behaviors.

The usage for each of these extensions is explained in that repository.

However, to install/activate each extension you must register and activate an *Event Listener*. To do this, you have two options:

1. Use the *StofDoctrineExtensionsBundle*⁸, which integrates the above library.
2. Implement this services directly by following the documentation for integration with Symfony:
*Install Gedmo Doctrine2 extensions in Symfony2*⁹

1. <https://github.com/Atlantic18/DoctrineExtensions>

2. <https://github.com/Atlantic18/DoctrineExtensions/blob/master/doc/sluggable.md>

3. <https://github.com/Atlantic18/DoctrineExtensions/blob/master/doc/translatable.md>

4. <https://github.com/Atlantic18/DoctrineExtensions/blob/master/doc/timestampable.md>

5. <https://github.com/Atlantic18/DoctrineExtensions/blob/master/doc/loggable.md>

6. <https://github.com/Atlantic18/DoctrineExtensions/blob/master/doc/tree.md>

7. <https://github.com/Atlantic18/DoctrineExtensions/blob/master/doc/sortable.md>

8. <https://github.com/stof/StofDoctrineExtensionsBundle>

9. <https://github.com/Atlantic18/DoctrineExtensions/blob/master/doc/symfony2.md>



Chapter 44

How to Register Event Listeners and Subscribers

Doctrine packages a rich event system that fires events when almost anything happens inside the system. For you, this means that you can create arbitrary *services* and tell Doctrine to notify those objects whenever a certain action (e.g. **prePersist**) happens within Doctrine. This could be useful, for example, to create an independent search index whenever an object in your database is saved.

Doctrine defines two types of objects that can listen to Doctrine events: listeners and subscribers. Both are very similar, but listeners are a bit more straightforward. For more, see *The Event System*¹ on Doctrine's website.

The Doctrine website also explains all existing events that can be listened to.

Configuring the Listener/Subscriber

To register a service to act as an event listener or subscriber you just have to tag it with the appropriate name. Depending on your use-case, you can hook a listener into every DBAL connection and ORM entity manager or just into one specific DBAL connection and all the entity managers that use this connection.

Listing 44-1

```
1 doctrine:
2   dbal:
3     default_connection: default
4     connections:
5       default:
6         driver: pdo_sqlite
7         memory: true
8
9   services:
10    my.listener:
11      class: AppBundle\EventListener\SearchIndexer
12      tags:
13        - { name: doctrine.event_listener, event: postPersist }
14    my.listener2:
15      class: AppBundle\EventListener\SearchIndexer2
```

1. <http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/events.html>

```

16     tags:
17         - { name: doctrine.event_listener, event: postPersist, connection: default }
18     my.subscriber:
19         class: AppBundle\EventListener\SearchIndexerSubscriber
20         tags:
21             - { name: doctrine.event_subscriber, connection: default }

```

Creating the Listener Class

In the previous example, a service `my.listener` was configured as a Doctrine listener on the event `postPersist`. The class behind that service must have a `postPersist` method, which will be called when the event is dispatched:

Listing 44-2

```

1  // src/AppBundle/EventListener/SearchIndexer.php
2  namespace AppBundle\EventListener;
3
4  use Doctrine\ORM\Event\LifecycleEventArgs;
5  use AppBundle\Entity\Product;
6
7  class SearchIndexer
8  {
9      public function postPersist(LifecycleEventArgs $args)
10     {
11         $entity = $args->getEntity();
12
13         // only act on some "Product" entity
14         if (!$entity instanceof Product) {
15             return;
16         }
17
18         $entityManager = $args->getEntityManager();
19         // ... do something with the Product
20     }
21 }

```

In each event, you have access to a `LifecycleEventArgs` object, which gives you access to both the entity object of the event and the entity manager itself.

One important thing to notice is that a listener will be listening for *all* entities in your application. So, if you're interested in only handling a specific type of entity (e.g. a **Product** entity but not a **BlogPost** entity), you should check for the entity's class type in your method (as shown above).



In Doctrine 2.4, a feature called Entity Listeners was introduced. It is a lifecycle listener class used for an entity. You can read about it in *the Doctrine Documentation*².

Creating the Subscriber Class

A Doctrine event subscriber must implement the `Doctrine\Common\EventSubscriber` interface and have an event method for each event it subscribes to:

Listing 44-3

```

1  // src/AppBundle/EventListener/SearchIndexerSubscriber.php
2  namespace AppBundle\EventListener;
3
4  use Doctrine\Common\EventSubscriber;
5  use Doctrine\ORM\Event\LifecycleEventArgs;

```

2. <http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/events.html#entity-listeners>

```

6 // for Doctrine 2.4: Doctrine\Common\Persistence\Event\LifecycleEventArgs;
7 use AppBundle\Entity\Product;
8
9 class SearchIndexerSubscriber implements EventSubscriber
10 {
11     public function getSubscribedEvents()
12     {
13         return array(
14             'postPersist',
15             'postUpdate',
16         );
17     }
18
19     public function postUpdate(LifecycleEventArgs $args)
20     {
21         $this->index($args);
22     }
23
24     public function postPersist(LifecycleEventArgs $args)
25     {
26         $this->index($args);
27     }
28
29     public function index(LifecycleEventArgs $args)
30     {
31         $entity = $args->getEntity();
32
33         // perhaps you only want to act on some "Product" entity
34         if ($entity instanceof Product) {
35             $entityManager = $args->getEntityManager();
36             // ... do something with the Product
37         }
38     }
39 }

```



Doctrine event subscribers can not return a flexible array of methods to call for the events like the Symfony event subscriber can. Doctrine event subscribers must return a simple array of the event names they subscribe to. Doctrine will then expect methods on the subscriber with the same name as each subscribed event, just as when using an event listener.

For a full reference, see chapter *The Event System*³ in the Doctrine documentation.

3. <http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/events.html>



Chapter 45

How to Use Doctrine DBAL



This article is about the Doctrine DBAL. Typically, you'll work with the higher level Doctrine ORM layer, which simply uses the DBAL behind the scenes to actually communicate with the database. To read more about the Doctrine ORM, see "*Databases and Doctrine*".

The *Doctrine*¹ Database Abstraction Layer (DBAL) is an abstraction layer that sits on top of *PDO*² and offers an intuitive and flexible API for communicating with the most popular relational databases. In other words, the DBAL library makes it easy to execute queries and perform other database actions.



Read the official Doctrine *DBAL Documentation*³ to learn all the details and capabilities of Doctrine's DBAL library.

To get started, configure the database connection parameters:

Listing 45-1

```
1 # app/config/config.yml
2 doctrine:
3   dbal:
4     driver:   pdo_mysql
5     dbname:  Symfony
6     user:    root
7     password: null
8     charset: UTF8
9     server_version: 5.6
```

For full DBAL configuration options, or to learn how to configure multiple connections, see Doctrine DBAL Configuration.

You can then access the Doctrine DBAL connection by accessing the `database_connection` service:

Listing 45-2

```
1 class UserController extends Controller
2 {
3     public function indexAction()
```

1. <http://www.doctrine-project.org>

2. <http://www.php.net/pdo>

3. <http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/index.html>

```

4     {
5         $conn = $this->get('database_connection');
6         $users = $conn->fetchAll('SELECT * FROM users');
7
8         // ...
9     }
10 }

```

Registering custom Mapping Types

You can register custom mapping types through Symfony's configuration. They will be added to all configured connections. For more information on custom mapping types, read Doctrine's *Custom Mapping Types*⁴ section of their documentation.

Listing 45-3

```

1 # app/config/config.yml
2 doctrine:
3     dbal:
4         types:
5             custom_first: AppBundle\Type\CustomFirst
6             custom_second: AppBundle\Type\CustomSecond

```

Registering custom Mapping Types in the SchemaTool

The SchemaTool is used to inspect the database to compare the schema. To achieve this task, it needs to know which mapping type needs to be used for each database types. Registering new ones can be done through the configuration.

Now, map the ENUM type (not supported by DBAL by default) to the **string** mapping type:

Listing 45-4

```

1 # app/config/config.yml
2 doctrine:
3     dbal:
4         mapping_types:
5             enum: string

```

4. <http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/types.html#custom-mapping-types>



Chapter 46

How to Generate Entities from an Existing Database

When starting work on a brand new project that uses a database, two different situations comes naturally. In most cases, the database model is designed and built from scratch. Sometimes, however, you'll start with an existing and probably unchangeable database model. Fortunately, Doctrine comes with a bunch of tools to help generate model classes from your existing database.



As the *Doctrine tools documentation*¹ says, reverse engineering is a one-time process to get started on a project. Doctrine is able to convert approximately 70-80% of the necessary mapping information based on fields, indexes and foreign key constraints. Doctrine can't discover inverse associations, inheritance types, entities with foreign keys as primary keys or semantical operations on associations such as cascade or lifecycle events. Some additional work on the generated entities will be necessary afterwards to design each to fit your domain model specificities.

This tutorial assumes you're using a simple blog application with the following two tables: **blog_post** and **blog_comment**. A comment record is linked to a post record thanks to a foreign key constraint.

Listing 46-1

```
1 CREATE TABLE `blog_post` (  
2   `id` bigint(20) NOT NULL AUTO_INCREMENT,  
3   `title` varchar(100) COLLATE utf8_unicode_ci NOT NULL,  
4   `content` longtext COLLATE utf8_unicode_ci NOT NULL,  
5   `created_at` datetime NOT NULL,  
6   PRIMARY KEY (`id`)  
7 ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;  
8  
9 CREATE TABLE `blog_comment` (  
10  `id` bigint(20) NOT NULL AUTO_INCREMENT,  
11  `post_id` bigint(20) NOT NULL,  
12  `author` varchar(20) COLLATE utf8_unicode_ci NOT NULL,  
13  `content` longtext COLLATE utf8_unicode_ci NOT NULL,  
14  `created_at` datetime NOT NULL,  
15  PRIMARY KEY (`id`),  
16  KEY `blog_comment_post_id_idx` (`post_id`),
```

1. <http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/tools.html#reverse-engineering>

```

17     CONSTRAINT `blog_post_id` FOREIGN KEY (`post_id`) REFERENCES `blog_post` (`id`) ON DELETE CASCADE
18 ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;

```

Before diving into the recipe, be sure your database connection parameters are correctly setup in the `app/config/parameters.yml` file (or wherever your database configuration is kept) and that you have initialized a bundle that will host your future entity class. In this tutorial it's assumed that an `AcmeBlogBundle` exists and is located under the `src/Acme/BlogBundle` folder.

The first step towards building entity classes from an existing database is to ask Doctrine to introspect the database and generate the corresponding metadata files. Metadata files describe the entity class to generate based on table fields.

Listing 46-2 1 \$ php bin/console doctrine:mapping:import --force AcmeBlogBundle xml

This command line tool asks Doctrine to introspect the database and generate the XML metadata files under the `src/Acme/BlogBundle/Resources/config/doctrine` folder of your bundle. This generates two files: `BlogPost.orm.xml` and `BlogComment.orm.xml`.



It's also possible to generate the metadata files in YAML format by changing the last argument to `yml`.

The generated `BlogPost.orm.xml` metadata file looks as follows:

Listing 46-3

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://doctrine-project.org/schemas/
4 orm/doctrine-mapping http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd">
5   <entity name="Acme\BlogBundle\Entity\BlogPost" table="blog_post">
6     <id name="id" type="bigint" column="id">
7       <generator strategy="IDENTITY"/>
8     </id>
9     <field name="title" type="string" column="title" length="100" nullable="false"/>
10    <field name="content" type="text" column="content" nullable="false"/>
11    <field name="createdAt" type="datetime" column="created_at" nullable="false"/>
12  </entity>
13 </doctrine-mapping>

```

Once the metadata files are generated, you can ask Doctrine to build related entity classes by executing the following two commands.

Listing 46-4

```

1 $ php bin/console doctrine:mapping:convert annotation ./src
2 $ php bin/console doctrine:generate:entities AcmeBlogBundle

```

The first command generates entity classes with annotation mappings. But if you want to use YAML or XML mapping instead of annotations, you should execute the second command only.



If you want to use annotations, you must remove the XML (or YAML) files after running these two commands. This is necessary as it is not possible to mix mapping configuration formats

For example, the newly created `BlogComment` entity class looks as follow:

Listing 46-5

```

1 // src/Acme/BlogBundle/Entity/BlogComment.php
2 namespace Acme\BlogBundle\Entity;
3
4 use Doctrine\ORM\Mapping as ORM;
5
6 /**

```

```

7  * Acme\BlogBundle\Entity\BlogComment
8  *
9  * @ORM\Table(name="blog_comment")
10 * @ORM\Entity
11 */
12 class BlogComment
13 {
14     /**
15      * @var integer $id
16      *
17      * @ORM\Column(name="id", type="bigint")
18      * @ORM\Id
19      * @ORM\GeneratedValue(strategy="IDENTITY")
20      */
21     private $id;
22
23     /**
24      * @var string $author
25      *
26      * @ORM\Column(name="author", type="string", length=100, nullable=false)
27      */
28     private $author;
29
30     /**
31      * @var text $content
32      *
33      * @ORM\Column(name="content", type="text", nullable=false)
34      */
35     private $content;
36
37     /**
38      * @var datetime $createdAt
39      *
40      * @ORM\Column(name="created_at", type="datetime", nullable=false)
41      */
42     private $createdAt;
43
44     /**
45      * @var BlogPost
46      *
47      * @ORM\ManyToOne(targetEntity="BlogPost")
48      * @ORM\JoinColumn(name="post_id", referencedColumnName="id")
49      */
50     private $post;
51 }

```

As you can see, Doctrine converts all table fields to pure private and annotated class properties. The most impressive thing is that it also discovered the relationship with the **BlogPost** entity class based on the foreign key constraint. Consequently, you can find a private **\$post** property mapped with a **BlogPost** entity in the **BlogComment** entity class.



If you want to have a one-to-many relationship, you will need to add it manually into the entity or to the generated XML or YAML files. Add a section on the specific entities for one-to-many defining the **inversedBy** and the **mappedBy** pieces.

The generated entities are now ready to be used. Have fun!



Chapter 47

How to Work with multiple Entity Managers and Connections

You can use multiple Doctrine entity managers or connections in a Symfony application. This is necessary if you are using different databases or even vendors with entirely different sets of entities. In other words, one entity manager that connects to one database will handle some entities while another entity manager that connects to another database might handle the rest.



Using multiple entity managers is pretty easy, but more advanced and not usually required. Be sure you actually need multiple entity managers before adding in this layer of complexity.

The following configuration code shows how you can configure two entity managers:

Listing 47-1

```
1 doctrine:
2   dbal:
3     default_connection: default
4     connections:
5       default:
6         driver: pdo_mysql
7         host: '%database_host%'
8         port: '%database_port%'
9         dbname: '%database_name%'
10        user: '%database_user%'
11        password: '%database_password%'
12        charset: UTF8
13      customer:
14        driver: pdo_mysql
15        host: '%database_host2%'
16        port: '%database_port2%'
17        dbname: '%database_name2%'
18        user: '%database_user2%'
19        password: '%database_password2%'
20        charset: UTF8
21
22  orm:
23    default_entity_manager: default
24    entity_managers:
25      default:
```

```

26         connection: default
27         mappings:
28             AppBundle: ~
29             AcmeStoreBundle: ~
30     customer:
31         connection: customer
32         mappings:
33             AcmeCustomerBundle: ~

```

In this case, you've defined two entity managers and called them **default** and **customer**. The **default** entity manager manages entities in the AppBundle and AcmeStoreBundle, while the **customer** entity manager manages entities in the AcmeCustomerBundle. You've also defined two connections, one for each entity manager.



When working with multiple connections and entity managers, you should be explicit about which configuration you want. If you *do* omit the name of the connection or entity manager, the default (i.e. **default**) is used.

When working with multiple connections to create your databases:

Listing 47-2

```

1  # Play only with "default" connection
2  $ php bin/console doctrine:database:create
3
4  # Play only with "customer" connection
5  $ php bin/console doctrine:database:create --connection=customer

```

When working with multiple entity managers to update your schema:

Listing 47-3

```

1  # Play only with "default" mappings
2  $ php bin/console doctrine:schema:update --force
3
4  # Play only with "customer" mappings
5  $ php bin/console doctrine:schema:update --force --em=customer

```

If you *do* omit the entity manager's name when asking for it, the default entity manager (i.e. **default**) is returned:

Listing 47-4

```

1  class UserController extends Controller
2  {
3      public function indexAction()
4      {
5          // All three return the "default" entity manager
6          $em = $this->get('doctrine')->getManager();
7          $em = $this->get('doctrine')->getManager('default');
8          $em = $this->get('doctrine.orm.default_entity_manager');
9
10         // Both of these return the "customer" entity manager
11         $customerEm = $this->get('doctrine')->getManager('customer');
12         $customerEm = $this->get('doctrine.orm.customer_entity_manager');
13     }
14 }

```

You can now use Doctrine just as you did before - using the **default** entity manager to persist and fetch entities that it manages and the **customer** entity manager to persist and fetch its entities.

The same applies to repository calls:

Listing 47-5

```

1  class UserController extends Controller
2  {
3      public function indexAction()
4      {
5          // Retrieves a repository managed by the "default" em

```

```

6     $products = $this->get('doctrine')
7         ->getRepository('AcmeStoreBundle:Product')
8         ->findAll()
9     ;
10
11     // Explicit way to deal with the "default" em
12     $products = $this->get('doctrine')
13         ->getRepository('AcmeStoreBundle:Product', 'default')
14         ->findAll()
15     ;
16
17     // Retrieves a repository managed by the "customer" em
18     $customers = $this->get('doctrine')
19         ->getRepository('AcmeCustomerBundle:Customer', 'customer')
20         ->findAll()
21     ;
22 }
23 }

```



Chapter 48

How to Register custom DQL Functions

Doctrine allows you to specify custom DQL functions. For more information on this topic, read Doctrine's cookbook article "[DQL User Defined Functions](http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/cookbook/dql-user-defined-functions.html)¹".

In Symfony, you can register your custom DQL functions as follows:

Listing 48-1

```
1  # app/config/config.yml
2  doctrine:
3      orm:
4          # ...
5          dql:
6              string_functions:
7                  test_string: AppBundle\DQL\StringFunction
8                  second_string: AppBundle\DQL\SecondStringFunction
9              numeric_functions:
10                 test_numeric: AppBundle\DQL\NumericFunction
11             datetime_functions:
12                 test_datetime: AppBundle\DQL\DatetimeFunction
```

1. <http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/cookbook/dql-user-defined-functions.html>



Chapter 49

How to Define Relationships with Abstract Classes and Interfaces

One of the goals of bundles is to create discreet bundles of functionality that do not have many (if any) dependencies, allowing you to use that functionality in other applications without including unnecessary items.

Doctrine 2.2 includes a new utility called the **ResolveTargetEntityListener**, that functions by intercepting certain calls inside Doctrine and rewriting **targetEntity** parameters in your metadata mapping at runtime. It means that in your bundle you are able to use an interface or abstract class in your mappings and expect correct mapping to a concrete entity at runtime.

This functionality allows you to define relationships between different entities without making them hard dependencies.

Background

Suppose you have an *InvoiceBundle* which provides invoicing functionality and a *CustomerBundle* that contains customer management tools. You want to keep these separated, because they can be used in other systems without each other, but for your application you want to use them together.

In this case, you have an **Invoice** entity with a relationship to a non-existent object, an **InvoiceSubjectInterface**. The goal is to get the **ResolveTargetEntityListener** to replace any mention of the interface with a real object that implements that interface.

Set up

This article uses the following two basic entities (which are incomplete for brevity) to explain how to set up and use the **ResolveTargetEntityListener**.

A Customer entity:

Listing 49-1


```

1 // src/Acme/AppBundle/Entity/Customer.php
2
3 namespace Acme\AppBundle\Entity;
4
5 use Doctrine\ORM\Mapping as ORM;
6 use Acme\CustomerBundle\Entity\Customer as BaseCustomer;
7 use Acme\InvoiceBundle\Model\InvoiceSubjectInterface;
8
9 /**
10  * @ORM\Entity
11  * @ORM\Table(name="customer")
12  */
13 class Customer extends BaseCustomer implements InvoiceSubjectInterface
14 {
15     // In this example, any methods defined in the InvoiceSubjectInterface
16     // are already implemented in the BaseCustomer
17 }

```

An Invoice entity:

Listing 49-2

```

1 // src/Acme/InvoiceBundle/Entity/Invoice.php
2
3 namespace Acme\InvoiceBundle\Entity;
4
5 use Doctrine\ORM\Mapping as ORM;
6 use Acme\InvoiceBundle\Model\InvoiceSubjectInterface;
7
8 /**
9  * Represents an Invoice.
10  *
11  * @ORM\Entity
12  * @ORM\Table(name="invoice")
13  */
14 class Invoice
15 {
16     /**
17      * @ORM\ManyToOne(targetEntity="Acme\InvoiceBundle\Model\InvoiceSubjectInterface")
18      * @var InvoiceSubjectInterface
19      */
20     protected $subject;
21 }

```

An InvoiceSubjectInterface:

Listing 49-3

```

1 // src/Acme/InvoiceBundle/Model/InvoiceSubjectInterface.php
2
3 namespace Acme\InvoiceBundle\Model;
4
5 /**
6  * An interface that the invoice Subject object should implement.
7  * In most circumstances, only a single object should implement
8  * this interface as the ResolveTargetEntityListener can only
9  * change the target to a single object.
10  */
11 interface InvoiceSubjectInterface
12 {
13     // List any additional methods that your InvoiceBundle
14     // will need to access on the subject so that you can
15     // be sure that you have access to those methods.
16
17     /**
18      * @return string
19      */
20     public function getName();
21 }

```

Next, you need to configure the listener, which tells the DoctrineBundle about the replacement:

Listing 49-4

```
1 # app/config/config.yml
2 doctrine:
3     # ...
4     orm:
5         # ...
6         resolve_target_entities:
7             Acme\InvoiceBundle\Model\InvoiceSubjectInterface: Acme\AppBundle\Entity\Customer
```

Final Thoughts

With the **ResolveTargetEntityListener**, you are able to decouple your bundles, keeping them usable by themselves, but still being able to define relationships between different objects. By using this method, your bundles will end up being easier to maintain independently.



Chapter 50

How to Provide Model Classes for several Doctrine Implementations

When building a bundle that could be used not only with Doctrine ORM but also the CouchDB ODM, MongoDB ODM or PHPCR ODM, you should still only write one model class. The Doctrine bundles provide a compiler pass to register the mappings for your model classes.



For non-reusable bundles, the easiest option is to put your model classes in the default locations: **Entity** for the Doctrine ORM or **Document** for one of the ODMs. For reusable bundles, rather than duplicate model classes just to get the auto-mapping, use the compiler pass.

In your bundle class, write the following code to register the compiler pass. This one is written for the `CmfRoutingBundle`, so parts of it will need to be adapted for your case:

```
Listing 50-1 1 use Doctrine\Bundle\DoctrineBundle\DependencyInjection\Compiler\DoctrineOrmMappingsPass;
2 use Doctrine\Bundle\MongoDBBundle\DependencyInjection\Compiler\DoctrineMongoDBMappingsPass;
3 use Doctrine\Bundle\CouchDBBundle\DependencyInjection\Compiler\DoctrineCouchDBMappingsPass;
4 use Doctrine\Bundle\PHPCRBundle\DependencyInjection\Compiler\DoctrinePhpcrMappingsPass;
5
6 class CmfRoutingBundle extends Bundle
7 {
8     public function build(ContainerBuilder $container)
9     {
10         parent::build($container);
11         // ...
12
13         $modelDir = realpath(__DIR__.'/Resources/config/doctrine/model');
14         $mappings = array(
15             $modelDir => 'Symfony\Cmf\RoutingBundle\Model',
16         );
17
18         $ormCompilerClass =
19 'Doctrine\Bundle\DoctrineBundle\DependencyInjection\Compiler\DoctrineOrmMappingsPass';
20         if (class_exists($ormCompilerClass)) {
21             $container->addCompilerPass(
22                 DoctrineOrmMappingsPass::createXmlMappingDriver(
23                     $mappings,
24                     array('cmf_routing.model_manager_name'),
25                     'cmf_routing.backend_type_orm',
```

```

26         array('CmfRoutingBundle' => 'Symfony\Cmf\RoutingBundle\Model')
27     ));
28 }
29
30 $mongoCompilerClass =
31 'Doctrine\Bundle\MongoDBBundle\DependencyInjection\Compiler\DoctrineMongoDBMappingsPass';
32 if (class_exists($mongoCompilerClass)) {
33     $container->addCompilerPass(
34         DoctrineMongoDBMappingsPass::createXmlMappingDriver(
35             $mappings,
36             array('cmf_routing.model_manager_name'),
37             'cmf_routing.backend_type_mongodb',
38             array('CmfRoutingBundle' => 'Symfony\Cmf\RoutingBundle\Model')
39         ));
40 }
41
42 $couchCompilerClass =
43 'Doctrine\Bundle\CouchDBBundle\DependencyInjection\Compiler\DoctrineCouchDBMappingsPass';
44 if (class_exists($couchCompilerClass)) {
45     $container->addCompilerPass(
46         DoctrineCouchDBMappingsPass::createXmlMappingDriver(
47             $mappings,
48             array('cmf_routing.model_manager_name'),
49             'cmf_routing.backend_type_couchdb',
50             array('CmfRoutingBundle' => 'Symfony\Cmf\RoutingBundle\Model')
51         ));
52 }
53
54 $phpcrCompilerClass =
55 'Doctrine\Bundle\PHPCRBundle\DependencyInjection\Compiler\DoctrinePhpcrMappingsPass';
56 if (class_exists($phpcrCompilerClass)) {
57     $container->addCompilerPass(
58         DoctrinePhpcrMappingsPass::createXmlMappingDriver(
59             $mappings,
60             array('cmf_routing.model_manager_name'),
61             'cmf_routing.backend_type_phpcr',
62             array('CmfRoutingBundle' => 'Symfony\Cmf\RoutingBundle\Model')
63         ));
64 }
65 }
66 }

```

Note the *class_exists*¹ check. This is crucial, as you do not want your bundle to have a hard dependency on all Doctrine bundles but let the user decide which to use.

The compiler pass provides factory methods for all drivers provided by Doctrine: Annotations, XML, Yaml, PHP and StaticPHP. The arguments are:

- A map/hash of absolute directory path to namespace;
- An array of container parameters that your bundle uses to specify the name of the Doctrine manager that it is using. In the example above, the CmfRoutingBundle stores the manager name that's being used under the `cmf_routing.model_manager_name` parameter. The compiler pass will append the parameter Doctrine is using to specify the name of the default manager. The first parameter found is used and the mappings are registered with that manager;
- An optional container parameter name that will be used by the compiler pass to determine if this Doctrine type is used at all. This is relevant if your user has more than one type of Doctrine bundle installed, but your bundle is only used with one type of Doctrine;
- A map/hash of aliases to namespace. This should be the same convention used by Doctrine auto-mapping. In the example above, this allows the user to call `$om->getRepository('CmfRoutingBundle:Route')`.

1. <http://php.net/manual/en/function.class-exists.php>



The factory method is using the **SymfonyFileLocator** of Doctrine, meaning it will only see XML and YML mapping files if they do not contain the full namespace as the filename. This is by design: the **SymfonyFileLocator** simplifies things by assuming the files are just the "short" version of the class as their filename (e.g. **BlogPost.orm.xml**)

If you also need to map a base class, you can register a compiler pass with the **DefaultFileLocator** like this. This code is taken from the **DoctrineOrmMappingsPass** and adapted to use the **DefaultFileLocator** instead of the **SymfonyFileLocator**:

Listing 50-2

```
1 private function buildMappingCompilerPass()
2 {
3     $arguments = array(array(realpath(__DIR__ . '/Resources/config/doctrine-base'), '.orm.xml'));
4     $locator = new Definition('Doctrine\Common\Persistence\Mapping\Driver\DefaultFileLocator',
5     $arguments);
6     $driver = new Definition('Doctrine\ORM\Mapping\Driver\XmlDriver', array($locator));
7
8     return new DoctrineOrmMappingsPass(
9         $driver,
10        array('Full\Namespace'),
11        array('your_bundle.manager_name'),
12        'your_bundle.orm_enabled'
13    );
14 }
```

Note that you do not need to provide a namespace alias unless your users are expected to ask Doctrine for the base classes.

Now place your mapping file into **/Resources/config/doctrine-base** with the fully qualified class name, separated by **.** instead of ****, for example **Other.Namespace.Model.Name.orm.xml**. You may not mix the two as otherwise the **SymfonyFileLocator** will get confused.

Adjust accordingly for the other Doctrine implementations.



Chapter 51

How to Implement a Simple Registration Form

Creating a registration form is pretty easy - it *really* means just creating a form that will update some **User** model object (a Doctrine entity in this example) and then save it.



The popular *FOSUserBundle*¹ provides a registration form, reset password form and other user management functionality.

If you don't already have a **User** entity and a working login system, first start with *How to Load Security Users from the Database (the Entity Provider)*.

Your **User** entity will probably at least have the following fields:

username

This will be used for logging in, unless you instead want your user to login via email (in that case, this field is unnecessary).

email

A nice piece of information to collect. You can also allow users to login via email.

password

The encoded password.

plainPassword

This field is *not* persisted: (notice no `@ORM\Column` above it). It temporarily stores the plain password from the registration form. This field can be validated and is then used to populate the `password` field.

With some validation added, your class may look something like this:

Listing 51-1

```
1 // src/AppBundle/Entity/User.php
2 namespace AppBundle\Entity;
3
4 use Doctrine\ORM\Mapping as ORM;
5 use Symfony\Component\Validator\Constraints as Assert;
6 use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
7 use Symfony\Component\Security\Core\User\UserInterface;
```

1. <https://github.com/FriendsOfSymfony/FOSUserBundle>

```

8
9 /**
10  * @ORM\Entity
11  * @UniqueEntity(fields="email", message="Email already taken")
12  * @UniqueEntity(fields="username", message="Username already taken")
13  */
14 class User implements UserInterface
15 {
16     /**
17      * @ORM\Id
18      * @ORM\Column(type="integer")
19      * @ORM\GeneratedValue(strategy="AUTO")
20      */
21     private $id;
22
23     /**
24      * @ORM\Column(type="string", length=255, unique=true)
25      * @Assert\NotBlank()
26      * @Assert\Email()
27      */
28     private $email;
29
30     /**
31      * @ORM\Column(type="string", length=255, unique=true)
32      * @Assert\NotBlank()
33      */
34     private $username;
35
36     /**
37      * @Assert\NotBlank()
38      * @Assert\Length(max=4096)
39      */
40     private $plainPassword;
41
42     /**
43      * The below length depends on the "algorithm" you use for encoding
44      * the password, but this works well with bcrypt.
45      *
46      * @ORM\Column(type="string", length=64)
47      */
48     private $password;
49
50     // other properties and methods
51
52     public function getEmail()
53     {
54         return $this->email;
55     }
56
57     public function setEmail($email)
58     {
59         $this->email = $email;
60     }
61
62     public function getUsername()
63     {
64         return $this->username;
65     }
66
67     public function setUsername($username)
68     {
69         $this->username = $username;
70     }
71
72     public function getPlainPassword()
73     {
74         return $this->plainPassword;
75     }
76
77     public function setPlainPassword($password)
78     {

```

```

79     $this->plainPassword = $password;
80 }
81
82 public function setPassword($password)
83 {
84     $this->password = $password;
85 }
86
87 public function getSalt()
88 {
89     // The bcrypt algorithm doesn't require a separate salt.
90     // You *may* need a real salt if you choose a different encoder.
91     return null;
92 }
93
94 // other methods, including security methods like getRoles()
95 }

```

The *UserInterface*² requires a few other methods and your `security.yml` file needs to be configured properly to work with the `User` entity. For a more complete example, see the Entity Provider article.



Why the 4096 Password Limit?

Notice that the `plainPassword` field has a max length of 4096 characters. For security purposes (CVE-2013-5750³), Symfony limits the plain password length to 4096 characters when encoding it. Adding this constraint makes sure that your form will give a validation error if anyone tries a super-long password.

You'll need to add this constraint anywhere in your application where your user submits a plaintext password (e.g. change password form). The only place where you don't need to worry about this is your login form, since Symfony's Security component handles this for you.

Create a Form for the Entity

Next, create the form for the `User` entity:

Listing 51-2

```

1 // src/AppBundle/Form/ UserType.php
2 namespace AppBundle\Form;
3
4 use Symfony\Component\Form\AbstractType;
5 use Symfony\Component\Form\FormBuilderInterface;
6 use Symfony\Component\OptionsResolver\OptionsResolver;
7 use Symfony\Component\Form\Extension\Core\Type\EmailType;
8 use Symfony\Component\Form\Extension\Core\Type\TextType;
9 use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
10 use Symfony\Component\Form\Extension\Core\Type\PasswordType;
11
12 class UserType extends AbstractType
13 {
14     public function buildForm(FormBuilderInterface $builder, array $options)
15     {
16         $builder
17             ->add('email', EmailType::class)
18             ->add('username', TextType::class)
19             ->add('plainPassword', RepeatedType::class, array(
20                 'type' => PasswordType::class,
21                 'first_options' => array('label' => 'Password'),
22                 'second_options' => array('label' => 'Repeat Password'),

```

2. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html>

3. <https://symfony.com/blog/cve-2013-5750-security-issue-in-fosuserbundle-login-form>


```

23     )
24     );
25 }
26
27 public function configureOptions(OptionsResolver $resolver)
28 {
29     $resolver->setDefaults(array(
30         'data_class' => 'AppBundle\Entity\User',
31     ));
32 }
33 }

```

There are just three fields: **email**, **username** and **plainPassword** (repeated to confirm the entered password).



To explore more things about the Form component, read the *chapter about forms* in the book.

Handling the Form Submission

Next, you need a controller to handle the form rendering and submission. If the form is submitted, the controller performs the validation and saves the data into the database:

Listing 51-3

```

1  // src/AppBundle/Controller/RegistrationController.php
2  namespace AppBundle\Controller;
3
4  use AppBundle\Form\UserType;
5  use AppBundle\Entity\User;
6  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
7  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8  use Symfony\Component\HttpFoundation\Request;
9
10 class RegistrationController extends Controller
11 {
12     /**
13      * @Route("/register", name="user_registration")
14      */
15     public function registerAction(Request $request)
16     {
17         // 1) build the form
18         $user = new User();
19         $form = $this->createForm(UserType::class, $user);
20
21         // 2) handle the submit (will only happen on POST)
22         $form->handleRequest($request);
23         if ($form->isSubmitted() && $form->isValid()) {
24
25             // 3) Encode the password (you could also do this via Doctrine listener)
26             $password = $this->get('security.password_encoder')
27                 ->encodePassword($user, $user->getPlainPassword());
28             $user->setPassword($password);
29
30             // 4) save the User!
31             $em = $this->getDoctrine()->getManager();
32             $em->persist($user);
33             $em->flush();
34
35             // ... do any other work - like sending them an email, etc
36             // maybe set a "flash" success message for the user
37
38             return $this->redirectToRoute('replace_with_some_route');
39         }
40     }

```

```

41         return $this->render(
42             'registration/register.html.twig',
43             array('form' => $form->createView())
44         );
45     }
46 }

```

To define the algorithm used to encode the password in step 3 configure the encoder in the security configuration:

Listing 51-4

```

1  # app/config/security.yml
2  security:
3      encoders:
4          AppBundle\Entity\User: bcrypt

```

In this case the recommended **bcrypt** algorithm is used. To learn more about how to encode the users password have a look into the security chapter.



If you decide to NOT use annotation routing (shown above), then you'll need to create a route to this controller:

Listing 51-5

```

1  # app/config/routing.yml
2  user_registration:
3      path: /register
4      defaults: { _controller: AppBundle:Registration:register }

```

Next, create the template:

Listing 51-6

```

1  {# app/Resources/views/registration/register.html.twig #}
2
3  {{ form_start(form) }}
4      {{ form_row(form.username) }}
5      {{ form_row(form.email) }}
6      {{ form_row(form.plainPassword.first) }}
7      {{ form_row(form.plainPassword.second) }}
8
9      <button type="submit">Register!</button>
10 {{ form_end(form) }}

```

See *How to Customize Form Rendering* for more details.

Update your Database Schema

If you've updated the **User** entity during this tutorial, you have to update your database schema using this command:

Listing 51-7

```

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

```

That's it! Head to **/register** to try things out!

Having a Registration form with only Email (no Username)

If you want your users to login via email and you don't need a username, then you can remove it from your **User** entity entirely. Instead, make **getUsername()** return the **email** property:

Listing 51-8

```

1  // src/AppBundle/Entity/User.php
2  // ...
3
4  class User implements UserInterface
5  {
6      // ...
7
8      public function getUsername()
9      {
10         return $this->email;
11     }
12
13     // ...
14 }

```

Next, just update the **providers** section of your **security.yml** file so that Symfony knows how to load your users via the **email** property on login. See Using a Custom Query to Load the User.

Adding a "accept terms" Checkbox

Sometimes, you want a "Do you accept the terms and conditions" checkbox on your registration form. The only trick is that you want to add this field to your form without adding an unnecessary new **termsAccepted** property to your **User** entity that you'll never need.

To do this, add a **termsAccepted** field to your form, but set its mapped option to **false**:

Listing 51-9

```

1  // src/AppBundle/Form/UserIdType.php
2  // ...
3  use Symfony\Component\Validator\Constraints\IsTrue;
4  use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
5  use Symfony\Component\Form\Extension\Core\Type\EmailType;
6
7  class UserIdType extends AbstractType
8  {
9      public function buildForm(FormBuilderInterface $builder, array $options)
10     {
11         $builder
12             ->add('email', EmailType::class);
13         // ...
14         ->add('termsAccepted', CheckboxType::class, array(
15             'mapped' => false,
16             'constraints' => new IsTrue(),
17         ));
18     };
19 }
20 }

```

The constraints option is also used, which allows us to add validation, even though there is no **termsAccepted** property on **User**.



Chapter 52

How to Use PdoSessionHandler to Store Sessions in the Database

The default Symfony session storage writes the session information to files. Most medium to large websites use a database to store the session values instead of files, because databases are easier to use and scale in a multiple web server environment.

Symfony has a built-in solution for database session storage called *PdoSessionHandler*¹. To use it, you just need to change some parameters in the main configuration file:

Listing 52-1

```
1 # app/config/config.yml
2 framework:
3     session:
4         # ...
5         handler_id: session.handler.pdo
6
7 services:
8     session.handler.pdo:
9         class:      Symfony\Component\HttpFoundation\Session\Storage\Handler\PdoSessionHandler
10        public:     false
11        arguments:
12            - 'mysql:dbname=mydatabase'
13            - { db_username: myuser, db_password: mypassword }
```

Configuring the Table and Column Names

This will expect a **sessions** table with a number of different columns. The table name, and all of the column names, can be configured by passing a second array argument to **PdoSessionHandler**:

Listing 52-2

```
1 # app/config/config.yml
2 services:
3     # ...
4     session.handler.pdo:
5         class:      Symfony\Component\HttpFoundation\Session\Storage\Handler\PdoSessionHandler
```

1. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Handler/PdoSessionHandler.html>

```

6         public:    false
7         arguments:
8             - 'mysql:dbname=mydatabase'
9             - { db_table: sessions, db_username: myuser, db_password: mypassword }

```

These are parameters that you must configure:

db_table (default sessions):

The name of the session table in your database;

db_id_col (default sess_id):

The name of the id column in your session table (VARCHAR(128));

db_data_col (default sess_data):

The name of the value column in your session table (BLOB);

db_time_col (default sess_time):

The name of the time column in your session table (INTEGER);

db_lifetime_col (default sess_lifetime):

The name of the lifetime column in your session table (INTEGER).

Sharing your Database Connection Information

With the given configuration, the database connection settings are defined for the session storage connection only. This is OK when you use a separate database for the session data.

But if you'd like to store the session data in the same database as the rest of your project's data, you can use the connection settings from the `parameters.yml` file by referencing the database-related parameters defined there:

Listing 52-3

```

1  services:
2      session.handler.pdo:
3          class:    Symfony\Component\HttpFoundation\Session\Storage\Handler\PdoSessionHandler
4          public:   false
5          arguments:
6              - 'mysql:host=%database_host%;port=%database_port%;dbname=%database_name%'
7              - { db_username: '%database_user%', db_password: '%database_password%' }

```

Preparing the Database to Store Sessions

Before storing sessions in the database, you must create the table that stores the information. The following sections contain some examples of the SQL statements you may use for your specific database engine.

MySQL

Listing 52-4

```

1  CREATE TABLE `sessions` (
2      `sess_id` VARBINARY(128) NOT NULL PRIMARY KEY,
3      `sess_data` BLOB NOT NULL,
4      `sess_time` INTEGER UNSIGNED NOT NULL,
5      `sess_lifetime` MEDIUMINT NOT NULL
6  ) COLLATE utf8_bin, ENGINE = InnoDB;

```



A **BLOB** column type can only store up to 64 kb. If the data stored in a user's session exceeds this, an exception may be thrown or their session will be silently reset. Consider using a **MEDIUMBLOB** if you need more space.

PostgreSQL

Listing 52-5

```
1 CREATE TABLE sessions (
2     sess_id VARCHAR(128) NOT NULL PRIMARY KEY,
3     sess_data BYTEA NOT NULL,
4     sess_time INTEGER NOT NULL,
5     sess_lifetime INTEGER NOT NULL
6 );
```

Microsoft SQL Server

Listing 52-6

```
1 CREATE TABLE [dbo].[sessions](
2     [sess_id] [nvarchar](255) NOT NULL,
3     [sess_data] [ntext] NOT NULL,
4     [sess_time] [int] NOT NULL,
5     [sess_lifetime] [int] NOT NULL,
6     PRIMARY KEY CLUSTERED(
7         [sess_id] ASC
8     ) WITH (
9         PAD_INDEX = OFF,
10        STATISTICS_NORECOMPUTE = OFF,
11        IGNORE_DUP_KEY = OFF,
12        ALLOW_ROW_LOCKS = ON,
13        ALLOW_PAGE_LOCKS = ON
14    ) ON [PRIMARY]
15 ) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
```



If the session data doesn't fit in the data column, it might get truncated by the database engine. To make matters worse, when the session data gets corrupted, PHP ignores the data without giving a warning.

If the application stores large amounts of session data, this problem can be solved by increasing the column size (use **BLOB** or even **MEDIUMBLOB**). When using MySQL as the database engine, you can also enable the *strict SQL mode*² to get noticed when such an error happens.

2. <https://dev.mysql.com/doc/refman/5.7/en/sql-mode.html>



Chapter 53

How to Use MongoDBSessionHandler to Store Sessions in a MongoDB Database

The default Symfony session storage writes the session information to files. Some medium to large websites use a NoSQL database called MongoDB to store the session values instead of files, because databases are easier to use and scale in a multi-webserver environment.

Symfony has a built-in solution for NoSQL database session storage called *MongoDbSessionHandler*¹. MongoDB is an open-source document database that provides high performance, high availability and automatic scaling. This article assumes that you have already *installed and configured a MongoDB server*². To use it, you just need to change/add some parameters in the main configuration file:

Listing 53-1

```
1  # app/config/config.yml
2  framework:
3      session:
4          # ...
5          handler_id: session.handler.mongo
6          cookie_lifetime: 2592000 # optional, it is set to 30 days here
7          gc_maxlifetime: 2592000 # optional, it is set to 30 days here
8
9  services:
10     # ...
11     mongo_client:
12         class: MongoClient
13         # if using a username and password
14         arguments: ['mongodb://%mongodb_username%:%mongodb_password%@%mongodb_host%:27017']
15         # if not using a username and password
16         arguments: ['mongodb://%mongodb_host%:27017']
17     session.handler.mongo:
18         class: Symfony\Component\HttpFoundation\Session\Storage\Handler\MongoDbSessionHandler
19         arguments: ['@mongo_client', '%mongo.session.options%']
```

The parameters used above should be defined somewhere in your application, often in your main parameters configuration:

Listing 53-2

-
1. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Handler/MongoDbSessionHandler.html>
 2. <http://docs.mongodb.org/manual/installation/>

```

1 # app/config/parameters.yml
2 parameters:
3   # ...
4   mongo.session.options:
5     database: session_db # your MongoDB database name
6     collection: session # your MongoDB collection name
7     mongodb_host: 1.2.3.4 # your MongoDB server's IP
8     mongodb_username: my_username
9     mongodb_password: my_password

```

Setting Up the MongoDB Collection

Because MongoDB uses dynamic collection schemas, you do not need to do anything to initialize your session collection. However, you may want to add an index to improve garbage collection performance. From the *MongoDB shell*³:

Listing 53-3

```

1 use session_db
2 db.session.ensureIndex( { "expires_at": 1 }, { expireAfterSeconds: 0 } )

```

3. <http://docs.mongodb.org/v2.2/tutorial/getting-started-with-the-mongo-shell/>



Chapter 54

Console Commands

The Doctrine2 ORM integration offers several console commands under the **doctrine** namespace. To view the command list you can use the **list** command:

Listing 54-1 1 \$ php bin/console list doctrine

A list of available commands will print out. You can find out more information about any of these commands (or any Symfony command) by running the **help** command. For example, to get details about the **doctrine:database:create** task, run:

Listing 54-2 1 \$ php bin/console help doctrine:database:create

Some notable or interesting tasks include:

- **doctrine:ensure-production-settings** - checks to see if the current environment is configured efficiently for production. This should always be run in the **prod** environment:

Listing 54-3 1 \$ php bin/console doctrine:ensure-production-settings --env=prod

- **doctrine:mapping:import** - allows Doctrine to introspect an existing database and create mapping information. For more information, see *How to Generate Entities from an Existing Database*.
- **doctrine:mapping:info** - tells you all of the entities that Doctrine is aware of and whether or not there are any basic errors with the mapping.
- **doctrine:query:dql** and **doctrine:query:sql** - allow you to execute DQL or SQL queries directly from the command line.



Chapter 55

How to Send an Email

Sending emails is a classic task for any web application and one that has special complications and potential pitfalls. Instead of recreating the wheel, one solution to send emails is to use the `SwiftMailerBundle`, which leverages the power of the *Swift Mailer*¹ library. This bundle comes with the Symfony Standard Edition.

Configuration

To use Swift Mailer, you'll need to configure it for your mail server.



Instead of setting up/using your own mail server, you may want to use a hosted mail provider such as *Mandrill*², *SendGrid*³, *Amazon SES*⁴ or others. These give you an SMTP server, username and password (sometimes called keys) that can be used with the Swift Mailer configuration.

In a standard Symfony installation, some `swiftmailer` configuration is already included:

Listing 55-1

```
1 # app/config/config.yml
2 swiftmailer:
3     transport: '%mailer_transport%'
4     host:      '%mailer_host%'
5     username:  '%mailer_user%'
6     password:  '%mailer_password%'
```

These values (e.g. `%mailer_transport%`), are reading from the parameters that are set in the `parameters.yml` file. You can modify the values in that file, or set the values directly here.

The following configuration attributes are available:

- `transport` (smtp, mail, sendmail, or gmail)
- `username`
- `password`

1. <http://swiftmailer.org/>
2. <https://mandrill.com/>
3. <https://sendgrid.com/>
4. <http://aws.amazon.com/ses/>

- host
- port
- encryption (tls, or ssl)
- auth_mode (plain, login, or cram-md5)
- spool
 - type (how to queue the messages, file or memory is supported, see *How to Spool Emails*)
 - path (where to store the messages)
- delivery_address (an email address where to send ALL emails)
- disable_delivery (set to true to disable delivery completely)

Sending Emails

The Swift Mailer library works by creating, configuring and then sending **Swift_Message** objects. The "mailer" is responsible for the actual delivery of the message and is accessible via the **mailer** service. Overall, sending an email is pretty straightforward:

Listing 55-2

```

1  public function indexAction($name)
2  {
3      $message = \Swift_Message::newInstance()
4      ->setSubject('Hello Email')
5      ->setFrom('send@example.com')
6      ->setTo('recipient@example.com')
7      ->setBody(
8          $this->renderView(
9              // app/Resources/views/Emails/registration.html.twig
10             'Emails/registration.html.twig',
11             array('name' => $name)
12         ),
13         'text/html'
14     )
15     /*
16     * If you also want to include a plaintext version of the message
17     ->addPart(
18         $this->renderView(
19             'Emails/registration.txt.twig',
20             array('name' => $name)
21         ),
22         'text/plain'
23     )
24     */
25 ;
26 $this->get('mailer')->send($message);
27
28 return $this->render(...);
29 }
```

To keep things decoupled, the email body has been stored in a template and rendered with the **renderView()** method. The **registration.html.twig** template might look something like this:

Listing 55-3

```

1  {% app/Resources/views/Emails/registration.html.twig %}
2  <h3>You did it! You registered!</h3>
3
4  Hi {{ name }}! You're successfully registered.
5
6  {% example, assuming you have a route named "login" %}
7  To login, go to: <a href="{{ url('login') }}">...</a>.
8
9  Thanks!
10
11 {% Makes an absolute URL to the /images/logo.png file %}
12 
```

The `$message` object supports many more options, such as including attachments, adding HTML content, and much more. Fortunately, Swift Mailer covers the topic of *Creating Messages*⁵ in great detail in its documentation.



Several other cookbook articles are available related to sending emails in Symfony:

- *How to Use Gmail to Send Emails*
- *How to Work with Emails during Development*
- *How to Spool Emails*

5. <http://swiftmailer.org/docs/messages.html>



Chapter 56

How to Use Gmail to Send Emails

During development, instead of using a regular SMTP server to send emails, you might find using Gmail easier and more practical. The SwiftmailerBundle makes it really easy.

In the development configuration file, change the **transport** setting to **gmail** and set the **username** and **password** to the Google credentials:

Listing 56-1

```
1 # app/config/config_dev.yml
2 swiftmailer:
3     transport: gmail
4     username:  your_gmail_username
5     password:  your_gmail_password
```



Listing 56-2

It's more convenient to configure these options in the **parameters.yml** file:

```
1 # app/config/parameters.yml
2 parameters:
3     # ...
4     mailer_user:    your_gmail_username
5     mailer_password: your_gmail_password
```

Listing 56-3

```
1 # app/config/config_dev.yml
2 swiftmailer:
3     transport: gmail
4     username:  '%mailer_user%'
5     password:  '%mailer_password%'
```

Redefining the Default Configuration Parameters

The **gmail** transport is simply a shortcut that uses the **smtp** transport and sets these options:

Option	Value
encryption	ssl

Option	Value
auth_mode	login
host	smtp.gmail.com

If your application uses **tls** encryption or **oauth** authentication, you must override the default options by defining the **encryption** and **auth_mode** parameters.

If your Gmail account uses 2-Step-Verification, you must *generate an App password*¹ and use it as the value of the **mailer_password** parameter. You must also ensure that you *allow less secure apps to access your Gmail account*².

See the Swiftmailer configuration reference for more details.

1. <https://support.google.com/accounts/answer/185833>

2. <https://support.google.com/accounts/answer/6010255>



Chapter 57

How to Use the Cloud to Send Emails

Requirements for sending emails from a production system differ from your development setup as you don't want to be limited in the number of emails, the sending rate or the sender address. Thus, *using Gmail* or similar services is not an option. If setting up and maintaining your own reliable mail server causes you a headache there's a simple solution: Leverage the cloud to send your emails.

This cookbook shows how easy it is to integrate *Amazon's Simple Email Service (SES)*¹ into Symfony.



You can use the same technique for other mail services, as most of the time there is nothing more to it than configuring an SMTP endpoint for Swift Mailer.

In the Symfony configuration, change the Swift Mailer settings **transport**, **host**, **port** and **encryption** according to the information provided in the *SES console*². Create your individual SMTP credentials in the SES console and complete the configuration with the provided **username** and **password**:

Listing 57-1

```
1 # app/config/config.yml
2 swiftmailer:
3     transport: smtp
4     host:      email-smtp.us-east-1.amazonaws.com
5     port:      587 # different ports are available, see SES console
6     encryption: tls # TLS encryption is required
7     username:  AWS_SES_SMTP_USERNAME # to be created in the SES console
8     password:  AWS_SES_SMTP_PASSWORD # to be created in the SES console
```

The **port** and **encryption** keys are not present in the Symfony Standard Edition configuration by default, but you can simply add them as needed.

And that's it, you're ready to start sending emails through the cloud!

1. <http://aws.amazon.com/ses>

2. <https://console.aws.amazon.com/ses>



If you are using the Symfony Standard Edition, configure the parameters in `parameters.yml` and use them in your configuration files. This allows for different Swift Mailer configurations for each installation of your application. For instance, use Gmail during development and the cloud in production.

Listing 57-2

```
1 # app/config/parameters.yml
2 parameters:
3     # ...
4     mailer_transport: smtp
5     mailer_host:      email-smtp.us-east-1.amazonaws.com
6     mailer_port:      587 # different ports are available, see SES console
7     mailer_encryption: tls # TLS encryption is required
8     mailer_user:      AWS_SES_SMTP_USERNAME # to be created in the SES console
9     mailer_password:  AWS_SES_SMTP_PASSWORD # to be created in the SES console
```



If you intend to use Amazon SES, please note the following:

- You have to sign up to *Amazon Web Services (AWS)*³;
- Every sender address used in the From or Return-Path (bounce address) header needs to be confirmed by the owner. You can also confirm an entire domain;
- Initially you are in a restricted sandbox mode. You need to request production access before being allowed to send to arbitrary recipients;
- SES may be subject to a charge.

3. <http://aws.amazon.com>



Chapter 58

How to Work with Emails during Development

When developing an application which sends email, you will often not want to actually send the email to the specified recipient during development. If you are using the SwiftmailerBundle with Symfony, you can easily achieve this through configuration settings without having to make any changes to your application's code at all. There are two main choices when it comes to handling email during development: (a) disabling the sending of email altogether or (b) sending all email to a specific address (with optional exceptions).

Disabling Sending

You can disable sending email by setting the `disable_delivery` option to `true`. This is the default in the `test` environment in the Standard distribution. If you do this in the `test` specific config then email will not be sent when you run tests, but will continue to be sent in the `prod` and `dev` environments:

Listing 58-1

```
1 # app/config/config_test.yml
2 swiftmailer:
3     disable_delivery: true
```

If you'd also like to disable deliver in the `dev` environment, simply add this same configuration to the `config_dev.yml` file.

Sending to a Specified Address

You can also choose to have all email sent to a specific address, instead of the address actually specified when sending the message. This can be done via the `delivery_address` option:

Listing 58-2

```
1 # app/config/config_dev.yml
2 swiftmailer:
3     delivery_address: 'dev@example.com'
```

Now, suppose you're sending an email to `recipient@example.com`.

Listing 58-3

```

1 public function indexAction($name)
2 {
3     $message = \Swift_Message::newInstance()
4         ->setSubject('Hello Email')
5         ->setFrom('send@example.com')
6         ->setTo('recipient@example.com')
7         ->setBody(
8             $this->renderView(
9                 'HelloBundle:Hello:email.txt.twig',
10                array('name' => $name)
11            )
12        )
13    ;
14    $this->get('mailer')->send($message);
15
16    return $this->render(..);
17 }

```

In the **dev** environment, the email will instead be sent to **dev@example.com**. Swift Mailer will add an extra header to the email, **X-Swift-To**, containing the replaced address, so you can still see who it would have been sent to.



In addition to the **to** addresses, this will also stop the email being sent to any **CC** and **BCC** addresses set for it. Swift Mailer will add additional headers to the email with the overridden addresses in them. These are **X-Swift-Cc** and **X-Swift-Bcc** for the **CC** and **BCC** addresses respectively.

Sending to a Specified Address but with Exceptions

Suppose you want to have all email redirected to a specific address, (like in the above scenario to **dev@example.com**). But then you may want email sent to some specific email addresses to go through after all, and not be redirected (even if it is in the dev environment). This can be done by adding the **delivery_whitelist** option:

Listing 58-4

```

1 # app/config/config_dev.yml
2 swiftmailer:
3     delivery_address: dev@example.com
4     delivery_whitelist:
5         # all email addresses matching these regexes will be delivered
6         # like normal, as well as being sent to dev@example.com
7         - '/@specialdomain\.com$/'
8         - '/^admin@mydomain\.com$/'

```

In the above example all email messages will be redirected to **dev@example.com** and messages sent to the **admin@mydomain.com** address or to any email address belonging to the domain **specialdomain.com** will also be delivered as normal.

Viewing from the Web Debug Toolbar

You can view any email sent during a single response when you are in the **dev** environment using the web debug toolbar. The email icon in the toolbar will show how many emails were sent. If you click it, a report will open showing the details of the sent emails.

If you're sending an email and then immediately redirecting to another page, the web debug toolbar will not display an email icon or a report on the next page.

Instead, you can set the **intercept_redirects** option to **true** in the **config_dev.yml** file, which will cause the redirect to stop and allow you to open the report with details of the sent emails.

Listing 58-5

```
1 # app/config/config_dev.yml
2 web_profiler:
3     intercept_redirects: true
```



Alternatively, you can open the profiler after the redirect and search by the submit URL used on the previous request (e.g. **/contact/handle**). The profiler's search feature allows you to load the profiler information for any past requests.



Chapter 59

How to Spool Emails

When you are using the `SwiftmailerBundle` to send an email from a Symfony application, it will default to sending the email immediately. You may, however, want to avoid the performance hit of the communication between Swift Mailer and the email transport, which could cause the user to wait for the next page to load while the email is sending. This can be avoided by choosing to "spool" the emails instead of sending them directly. This means that Swift Mailer does not attempt to send the email but instead saves the message to somewhere such as a file. Another process can then read from the spool and take care of sending the emails in the spool. Currently only spooling to file or memory is supported by Swift Mailer.

Spool Using Memory

When you use spooling to store the emails to memory, they will get sent right before the kernel terminates. This means the email only gets sent if the whole request got executed without any unhandled exception or any errors. To configure swiftmailer with the memory option, use the following configuration:

Listing 59-1

```
1 # app/config/config.yml
2 swiftmailer:
3     # ...
4     spool: { type: memory }
```

Spool Using Files

When you use the filesystem for spooling, Symfony creates a folder in the given path for each mail service (e.g. "default" for the default service). This folder will contain files for each email in the spool. So make sure this directory is writable by Symfony (or your webserver/php)!

In order to use the spool with files, use the following configuration:

Listing 59-2

```
1 # app/config/config.yml
2 swiftmailer:
3     # ...
```

```
4     spool:
5         type: file
6         path: /path/to/spooldir
```



If you want to store the spool somewhere with your project directory, remember that you can use the `%kernel.root_dir%` parameter to reference the project's root:

Listing 59-3 1 `path: '%kernel.root_dir%/spool'`

Now, when your app sends an email, it will not actually be sent but instead added to the spool. Sending the messages from the spool is done separately. There is a console command to send the messages in the spool:

Listing 59-4 1 `$ php bin/console swiftmailer:spool:send --env=prod`

It has an option to limit the number of messages to be sent:

Listing 59-5 1 `$ php bin/console swiftmailer:spool:send --message-limit=10 --env=prod`

You can also set the time limit in seconds:

Listing 59-6 1 `$ php bin/console swiftmailer:spool:send --time-limit=10 --env=prod`

Of course you will not want to run this manually in reality. Instead, the console command should be triggered by a cron job or scheduled task and run at a regular interval.



Chapter 60

How to Test that an Email is Sent in a Functional Test

Sending emails with Symfony is pretty straightforward thanks to the `SwiftmailerBundle`, which leverages the power of the *Swift Mailer*¹ library.

To functionally test that an email was sent, and even assert the email subject, content or any other headers, you can use *the Symfony Profiler*.

Start with an easy controller action that sends an email:

Listing 60-1

```
1 public function sendEmailAction($name)
2 {
3     $message = \Swift_Message::newInstance()
4         ->setSubject('Hello Email')
5         ->setFrom('send@example.com')
6         ->setTo('recipient@example.com')
7         ->setBody('You should see me from the profiler!')
8     ;
9
10    $this->get('mailer')->send($message);
11
12    return $this->render(...);
13 }
```



Don't forget to enable the profiler as explained in *How to Use the Profiler in a Functional Test*.

In your functional test, use the `swiftmailer` collector on the profiler to get information about the messages sent on the previous request:

Listing 60-2

```
1 // tests/AppBundle/Controller/MailControllerTest.php
2 namespace Tests\AppBundle\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;
```

1. <http://swiftmailer.org/>

```

5
6 class MailControllerTest extends WebTestCase
7 {
8     public function testMailIsSentAndContentIsOk()
9     {
10         $client = static::createClient();
11
12         // Enable the profiler for the next request (it does nothing if the profiler is not available)
13         $client->enableProfiler();
14
15         $crawler = $client->request('POST', '/path/to/above/action');
16
17         $mailCollector = $client->getProfile()->getCollector('swiftmailer');
18
19         // Check that an email was sent
20         $this->assertEquals(1, $mailCollector->getMessageCount());
21
22         $collectedMessages = $mailCollector->getMessages();
23         $message = $collectedMessages[0];
24
25         // Asserting email data
26         $this->assertInstanceOf('Swift_Message', $message);
27         $this->assertEquals('Hello Email', $message->getSubject());
28         $this->assertEquals('send@example.com', key($message->getFrom()));
29         $this->assertEquals('recipient@example.com', key($message->getTo()));
30         $this->assertEquals(
31             'You should see me from the profiler!',
32             $message->getBody()
33         );
34     }
35 }

```



Chapter 61

How to Create Event Listeners and Subscribers

During the execution of a Symfony application, lots of event notifications are triggered. Your application can listen to these notifications and respond to them by executing any piece of code.

Internal events provided by Symfony itself are defined in the *KernelEvents*¹ class. Third-party bundles and libraries also trigger lots of events and your own application can trigger *custom events*.

All the examples shown in this article use the same `KernelEvents::EXCEPTION` event for consistency purposes. In your own application, you can use any event and even mix several of them in the same subscriber.

Creating an Event Listener

The most common way to listen to an event is to register an **event listener**:

```
Listing 61-1 1 // src/AppBundle/EventListener/ExceptionListener.php
2 namespace AppBundle\EventListener;
3
4 use Symfony\Component\HttpFoundation\Response;
5 use Symfony\Component\HttpFoundation\Response;
6 use Symfony\Component\HttpFoundation\Response;
7
8 class ExceptionListener
9 {
10     public function onKernelException(GetResponseForExceptionEvent $event)
11     {
12         // You get the exception object from the received event
13         $exception = $event->getException();
14         $message = sprintf(
15             'My Error says: %s with code: %s',
16             $exception->getMessage(),
17             $exception->getCode()
18         );
19
20         // Customize your response object to display the exception details
21         $response = new Response();
22         $response->setContent($message);
23     }
24 }
```

1. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/KernelEvents.html>


```

24     // HttpExceptionInterface is a special type of exception that
25     // holds status code and header details
26     if ($exception instanceof HttpExceptionInterface) {
27         $response->setStatusCode($exception->getStatusCode());
28         $response->headers->replace($exception->getHeaders());
29     } else {
30         $response->setStatusCode(Response::HTTP_INTERNAL_SERVER_ERROR);
31     }
32
33     // Send the modified response object to the event
34     $event->setResponse($response);
35 }
36 }

```



Each event receives a slightly different type of `$event` object. For the `kernel.exception` event, it is *GetResponseForExceptionEvent*². To see what type of object each event listener receives, see *KernelEvents*³ or the documentation about the specific event you're listening to.

Now that the class is created, you just need to register it as a service and notify Symfony that it is a "listener" on the `kernel.exception` event by using a special "tag":

Listing 61-2

```

1 # app/config/services.yml
2 services:
3     app.exception_listener:
4         class: AppBundle\EventListener\ExceptionListener
5         tags:
6             - { name: kernel.event_listener, event: kernel.exception }

```



There is an optional tag attribute called `method` which defines which method to execute when the event is triggered. By default the name of the method is `on` + "camel-cased event name". If the event is `kernel.exception` the method executed by default is `onKernelException()`.

The other optional tag attribute is called `priority`, which defaults to `0` and it controls the order in which listeners are executed (the highest the priority, the earlier a listener is executed). This is useful when you need to guarantee that one listener is executed before another. The priorities of the internal Symfony listeners usually range from `-255` to `255` but your own listeners can use any positive or negative integer.

Creating an Event Subscriber

Another way to listen to events is via an **event subscriber**, which is a class that defines one or more methods that listen to one or various events. The main difference with the event listeners is that subscribers always know which events they are listening to.

In a given subscriber, different methods can listen to the same event. The order in which methods are executed is defined by the `priority` parameter of each method (the higher the priority the earlier the method is called). To learn more about event subscribers, read *The EventDispatcher Component*.

The following example shows an event subscriber that defines several methods which listen to the same `kernel.exception` event:

Listing 61-3

```

1 // src/AppBundle/EventSubscriber/ExceptionSubscriber.php
2 namespace AppBundle\EventSubscriber;

```

2. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html>
3. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/KernelEvents.html>

```

3
4 use Symfony\Component\EventDispatcher\EventSubscriberInterface;
5 use Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent;
6 use Symfony\Component\HttpKernel\KernelEvents;
7
8 class ExceptionSubscriber implements EventSubscriberInterface
9 {
10     public static function getSubscribedEvents()
11     {
12         // return the subscribed events, their methods and priorities
13         return array(
14             KernelEvents::EXCEPTION => array(
15                 array('processException', 10),
16                 array('logException', 0),
17                 array('notifyException', -10),
18             )
19         );
20     }
21
22     public function processException(GetResponseForExceptionEvent $event)
23     {
24         // ...
25     }
26
27     public function logException(GetResponseForExceptionEvent $event)
28     {
29         // ...
30     }
31
32     public function notifyException(GetResponseForExceptionEvent $event)
33     {
34         // ...
35     }
36 }

```

Now, you just need to register the class as a service and add the `kernel.event_subscriber` tag to tell Symfony that this is an event subscriber:

Listing 61-4

```

1 # app/config/services.yml
2 services:
3     app.exception_subscriber:
4         class: AppBundle\EventSubscriber\ExceptionSubscriber
5         tags:
6             - { name: kernel.event_subscriber }

```

Request Events, Checking Types

A single page can make several requests (one master request, and then multiple sub-requests - typically by Embedding Controllers). For the core Symfony events, you might need to check to see if the event is for a "master" request or a "sub request":

Listing 61-5

```

1 // src/AppBundle/EventListener/RequestListener.php
2 namespace AppBundle\EventListener;
3
4 use Symfony\Component\HttpKernel\Event\GetResponseEvent;
5 use Symfony\Component\HttpKernel\HttpKernel;
6 use Symfony\Component\HttpKernel\HttpKernelInterface;
7
8 class RequestListener
9 {
10     public function onKernelRequest(GetResponseEvent $event)
11     {
12         if (!$event->isMasterRequest()) {
13             // don't do anything if it's not the master request
14             return;
15         }
16     }
17 }

```

```
15         }  
16  
17         // ...  
18     }  
19 }
```

Certain things, like checking information on the *real* request, may not need to be done on the sub-request listeners.

Listeners or Subscribers

Listeners and subscribers can be used in the same application indistinctly. The decision to use either of them is usually a matter of personal taste. However, there are some minor advantages for each of them:

- **Subscribers are easier to reuse** because the knowledge of the events is kept in the class rather than in the service definition. This is the reason why Symfony uses subscribers internally;
- **Listeners are more flexible** because bundles can enable or disable each of them conditionally depending on some configuration value.

Debugging Event Listeners

You can find out what listeners are registered in the event dispatcher using the console. To show all events and their listeners, run:

```
Listing 61-6 1 $ php bin/console debug:event-dispatcher
```

You can get registered listeners for a particular event by specifying its name:

```
Listing 61-7 1 $ php bin/console debug:event-dispatcher kernel.exception
```



Chapter 62

How to Set Up Before and After Filters

It is quite common in web application development to need some logic to be executed just before or just after your controller actions acting as filters or hooks.

Some web frameworks define methods like `preExecute()` and `postExecute()`, but there is no such thing in Symfony. The good news is that there is a much better way to interfere with the Request -> Response process using the *EventDispatcher* component.

Token Validation Example

Imagine that you need to develop an API where some controllers are public but some others are restricted to one or some clients. For these private features, you might provide a token to your clients to identify themselves.

So, before executing your controller action, you need to check if the action is restricted or not. If it is restricted, you need to validate the provided token.



Please note that for simplicity in this recipe, tokens will be defined in config and neither database setup nor authentication via the Security component will be used.

Before Filters with the `kernel.controller` Event

First, store some basic token configuration using `config.yml` and the `parameters` key:

Listing 62-1

```
1 # app/config/config.yml
2 parameters:
3     tokens:
4         client1: pass1
5         client2: pass2
```

Tag Controllers to Be Checked

A `kernel.controller` listener gets notified on *every* request, right before the controller is executed. So, first, you need some way to identify if the controller that matches the request needs token validation.

A clean and easy way is to create an empty interface and make the controllers implement it:

```
Listing 62-2 1 namespace AppBundle\Controller;
2
3 interface TokenAuthenticatedController
4 {
5     // ...
6 }
```

A controller that implements this interface simply looks like this:

```
Listing 62-3 1 namespace AppBundle\Controller;
2
3 use AppBundle\Controller\TokenAuthenticatedController;
4 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5
6 class FooController extends Controller implements TokenAuthenticatedController
7 {
8     // An action that needs authentication
9     public function barAction()
10    {
11        // ...
12    }
13 }
```

Creating an Event Listener

Next, you'll need to create an event listener, which will hold the logic that you want executed before your controllers. If you're not familiar with event listeners, you can learn more about them at *How to Create Event Listeners and Subscribers*:

```
Listing 62-4 1 // src/AppBundle/EventListener/TokenListener.php
2 namespace AppBundle\EventListener;
3
4 use AppBundle\Controller\TokenAuthenticatedController;
5 use Symfony\Component\HttpKernel\Exception\AccessDeniedHttpException;
6 use Symfony\Component\HttpKernel\Event\FilterControllerEvent;
7
8 class TokenListener
9 {
10     private $tokens;
11
12     public function __construct($tokens)
13     {
14         $this->tokens = $tokens;
15     }
16
17     public function onKernelController(FilterControllerEvent $event)
18     {
19         $controller = $event->getController();
20
21         /*
22          * $controller passed can be either a class or a Closure.
23          * This is not usual in Symfony but it may happen.
24          * If it is a class, it comes in array format
25          */
26         if (!is_array($controller)) {
27             return;
28         }
29
30         if ($controller[0] instanceof TokenAuthenticatedController) {
31             $token = $event->getRequest()->query->get('token');
```

```

32         if (!in_array($token, $this->tokens)) {
33             throw new AccessDeniedHttpException('This action needs a valid token!');
34         }
35     }
36 }
37 }

```

Registering the Listener

Finally, register your listener as a service and tag it as an event listener. By listening on **kernel.controller**, you're telling Symfony that you want your listener to be called just before any controller is executed.

Listing 62-5

```

1 # app/config/services.yml
2 services:
3     app.tokens.action_listener:
4         class: AppBundle\EventListener\TokenListener
5         arguments: ['%tokens%']
6         tags:
7             - { name: kernel.event_listener, event: kernel.controller, method: onKernelController }

```

With this configuration, your **TokenListener onKernelController** method will be executed on each request. If the controller that is about to be executed implements **TokenAuthenticatedController**, token authentication is applied. This lets you have a "before" filter on any controller that you want.

After Filters with the **kernel.response** Event

In addition to having a "hook" that's executed *before* your controller, you can also add a hook that's executed *after* your controller. For this example, imagine that you want to add a sha1 hash (with a salt using that token) to all responses that have passed this token authentication.

Another core Symfony event - called **kernel.response** - is notified on every request, but after the controller returns a Response object. Creating an "after" listener is as easy as creating a listener class and registering it as a service on this event.

For example, take the **TokenListener** from the previous example and first record the authentication token inside the request attributes. This will serve as a basic flag that this request underwent token authentication:

Listing 62-6

```

1 public function onKernelController(FilterControllerEvent $event)
2 {
3     // ...
4
5     if ($controller[0] instanceof TokenAuthenticatedController) {
6         $token = $event->getRequest()->query->get('token');
7         if (!in_array($token, $this->tokens)) {
8             throw new AccessDeniedHttpException('This action needs a valid token!');
9         }
10
11         // mark the request as having passed token authentication
12         $event->getRequest()->attributes->set('auth_token', $token);
13     }
14 }

```

Now, add another method to this class - **onKernelResponse** - that looks for this flag on the request object and sets a custom header on the response if it's found:

Listing 62-7

```

1  // add the new use statement at the top of your file
2  use Symfony\Component\HttpKernel\Event\FilterResponseEvent;
3
4  public function onKernelResponse(FilterResponseEvent $event)
5  {
6      // check to see if onKernelController marked this as a token "auth'ed" request
7      if (!$token = $event->getRequest()->attributes->get('auth_token')) {
8          return;
9      }
10
11     $response = $event->getResponse();
12
13     // create a hash and set it as a response header
14     $hash = sha1($response->getContent().$token);
15     $response->headers->set('X-CONTENT-HASH', $hash);
16 }

```

Finally, a second "tag" is needed in the service definition to notify Symfony that the `onKernelResponse` event should be notified for the `kernel.response` event:

Listing 62-8

```

1  # app/config/services.yml
2  services:
3      app.tokens.action_listener:
4          class: AppBundle\EventListener\TokenListener
5          arguments: ['%tokens%']
6          tags:
7              - { name: kernel.event_listener, event: kernel.controller, method: onKernelController }
8              - { name: kernel.event_listener, event: kernel.response, method: onKernelResponse }

```

That's it! The `TokenListener` is now notified before every controller is executed (`onKernelController`) and after every controller returns a response (`onKernelResponse`). By making specific controllers implement the `TokenAuthenticatedController` interface, your listener knows which controllers it should take action on. And by storing a value in the request's "attributes" bag, the `onKernelResponse` method knows to add the extra header. Have fun!



Chapter 63

How to Extend a Class without Using Inheritance

To allow multiple classes to add methods to another one, you can define the magic `__call()` method in the class you want to be extended like this:

Listing 63-1

```
1 class Foo
2 {
3     // ...
4
5     public function __call($method, $arguments)
6     {
7         // create an event named 'foo.method_is_not_found'
8         $event = new HandleUndefinedMethodEvent($this, $method, $arguments);
9         $this->dispatcher->dispatch('foo.method_is_not_found', $event);
10
11         // no listener was able to process the event? The method does not exist
12         if (!$event->isProcessed()) {
13             throw new \Exception(sprintf('Call to undefined method %s::%s.', get_class($this), $method));
14         }
15
16         // return the listener returned value
17         return $event->getReturnValue();
18     }
19 }
```

This uses a special `HandleUndefinedMethodEvent` that should also be created. This is a generic class that could be reused each time you need to use this pattern of class extension:

Listing 63-2

```
1 use Symfony\Component\EventDispatcher\Event;
2
3 class HandleUndefinedMethodEvent extends Event
4 {
5     protected $subject;
6     protected $method;
7     protected $arguments;
8     protected $returnValue;
9     protected $isProcessed = false;
10
11     public function __construct($subject, $method, $arguments)
```



```

12     {
13         $this->subject = $subject;
14         $this->method = $method;
15         $this->arguments = $arguments;
16     }
17
18     public function getSubject()
19     {
20         return $this->subject;
21     }
22
23     public function getMethod()
24     {
25         return $this->method;
26     }
27
28     public function getArguments()
29     {
30         return $this->arguments;
31     }
32
33     /**
34      * Sets the value to return and stops other listeners from being notified
35      */
36     public function setReturnValue($val)
37     {
38         $this->returnValue = $val;
39         $this->isProcessed = true;
40         $this->stopPropagation();
41     }
42
43     public function getReturnValue()
44     {
45         return $this->returnValue;
46     }
47
48     public function isProcessed()
49     {
50         return $this->isProcessed;
51     }
52 }

```

Next, create a class that will listen to the `foo.method_is_not_found` event and *add* the method `bar()`:

Listing 63-3

```

1  class Bar
2  {
3      public function onFooMethodIsNotFound(HandleUndefinedMethodEvent $event)
4      {
5          // only respond to the calls to the 'bar' method
6          if ('bar' != $event->getMethod()) {
7              // allow another listener to take care of this unknown method
8              return;
9          }
10
11          // the subject object (the foo instance)
12          $foo = $event->getSubject();
13
14          // the bar method arguments
15          $arguments = $event->getArguments();
16
17          // ... do something
18
19          // set the return value
20          $event->setReturnValue($someValue);
21      }
22  }

```

Finally, add the new `bar` method to the `Foo` class by registering an instance of `Bar` with the `foo.method_is_not_found` event:

```
Listing 63-4 1 $bar = new Bar();  
2 $dispatcher->addListener('foo.method_is_not_found', array($bar, 'onFooMethodIsNotFound'));
```



Chapter 64

How to Customize a Method Behavior without Using Inheritance

Doing something before or after a Method Call

If you want to do something just before, or just after a method is called, you can dispatch an event respectively at the beginning or at the end of the method:

Listing 64-1

```
1  class Foo
2  {
3      // ...
4
5      public function send($foo, $bar)
6      {
7          // do something before the method
8          $event = new FilterBeforeSendEvent($foo, $bar);
9          $this->dispatcher->dispatch('foo.pre_send', $event);
10
11         // get $foo and $bar from the event, they may have been modified
12         $foo = $event->getFoo();
13         $bar = $event->getBar();
14
15         // the real method implementation is here
16         $ret = ...;
17
18         // do something after the method
19         $event = new FilterSendReturnValue($ret);
20         $this->dispatcher->dispatch('foo.post_send', $event);
21
22         return $event->getReturnValue();
23     }
24 }
```

In this example, two events are thrown: **foo.pre_send**, before the method is executed, and **foo.post_send** after the method is executed. Each uses a custom Event class to communicate information to the listeners of the two events. These event classes would need to be created by you and should allow, in this example, the variables **\$foo**, **\$bar** and **\$ret** to be retrieved and set by the listeners.

For example, assuming the `FilterSendReturnValue` has a `setReturnValue` method, one listener might look like this:

Listing 64-2

```
1 public function onFooPostSend(FilterSendReturnValue $event)
2 {
3     $ret = $event->getReturnValue();
4     // modify the original ``$ret`` value
5
6     $event->setReturnValue($ret);
7 }
```



Chapter 65

How to use Expressions in Security, Routing, Services, and Validation

Symfony comes with a powerful *ExpressionLanguage* component. It allows you to add highly customized logic inside configuration.

The Symfony Framework leverages expressions out of the box in the following ways:

- Configuring services;
- Route matching conditions;
- Checking security (explained below) and access controls with `allow_if`;
- *Validation*.

For more information about how to create and work with expressions, see *The Expression Syntax*.

Security: Complex Access Controls with Expressions

In addition to a role like `ROLE_ADMIN`, the `isGranted` method also accepts an *Expression*¹ object:

Listing 65-1

```
1 use Symfony\Component\ExpressionLanguage\Expression;
2 // ...
3
4 public function indexAction()
5 {
6     $this->denyAccessUnlessGranted(new Expression(
7         '"ROLE_ADMIN" in roles or (user and user.isSuperAdmin())'
8     ));
9
10    // ...
11 }
```

In this example, if the current user has `ROLE_ADMIN` or if the current user object's `isSuperAdmin()` method returns `true`, then access will be granted (note: your User object may not have an `isSuperAdmin` method, that method is invented for this example).

1. <http://api.symfony.com/3.0/Symfony/Component/ExpressionLanguage/Expression.html>

This uses an expression and you can learn more about the expression language syntax, see *The Expression Syntax*.

Inside the expression, you have access to a number of variables:

user

The user object (or the string `anon` if you're not authenticated).

roles

The array of roles the user has, including from the role hierarchy but not including the `IS_AUTHENTICATED_*` attributes (see the functions below).

object

The object (if any) that's passed as the second argument to `isGranted`.

token

The token object.

trust_resolver

The *AuthenticationTrustResolverInterface*², object: you'll probably use the `is_*` functions below instead.

Additionally, you have access to a number of functions inside the expression:

is_authenticated

Returns true if the user is authenticated via "remember-me" or authenticated "fully" - i.e. returns true if the user is "logged in".

is_anonymous

Equal to using `IS_AUTHENTICATED_ANONYMOUSLY` with the `isGranted` function.

is_remember_me

Similar, but not equal to `IS_AUTHENTICATED_REMEMBERED`, see below.

is_fully_authenticated

Similar, but not equal to `IS_AUTHENTICATED_FULLY`, see below.

has_role

Checks to see if the user has the given role - equivalent to an expression like `'ROLE_ADMIN'` in `roles`.

2. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/AuthenticationTrustResolverInterface.html>



`is_remember_me` is different than checking `IS_AUTHENTICATED_REMEMBERED`

The `is_remember_me` and `is_authenticated_fully` functions are *similar* to using `IS_AUTHENTICATED_REMEMBERED` and `IS_AUTHENTICATED_FULLY` with the `isGranted` function - but they are **not** the same. The following shows the difference:

Listing 65-2

```
1 use Symfony\Component\ExpressionLanguage\Expression;
2 // ...
3
4 $ac = $this->get('security.authorization_checker');
5 $access1 = $ac->isGranted('IS_AUTHENTICATED_REMEMBERED');
6
7 $access2 = $ac->isGranted(new Expression(
8     'is_remember_me() or is_fully_authenticated()'
9 ));
```

Here, `$access1` and `$access2` will be the same value. Unlike the behavior of `IS_AUTHENTICATED_REMEMBERED` and `IS_AUTHENTICATED_FULLY`, the `is_remember_me` function *only* returns true if the user is authenticated via a remember-me cookie and `is_fully_authenticated` *only* returns true if the user has actually logged in during this session (i.e. is full-fledged).



Chapter 66

How to Customize Form Rendering

Symfony gives you a wide variety of ways to customize how a form is rendered. In this guide, you'll learn how to customize every possible part of your form with as little effort as possible whether you use Twig or PHP as your templating engine.

Form Rendering Basics

Recall that the label, error and HTML widget of a form field can easily be rendered by using the `form_row` Twig function or the `row` PHP helper method:

Listing 66-1

```
1 {{ form_row(form.age) }}
```

You can also render each of the three parts of the field individually:

Listing 66-2

```
1 <div>
2     {{ form_label(form.age) }}
3     {{ form_errors(form.age) }}
4     {{ form_widget(form.age) }}
5 </div>
```

In both cases, the form label, errors and HTML widget are rendered by using a set of markup that ships standard with Symfony. For example, both of the above templates would render:

Listing 66-3

```
1 <div>
2     <label for="form_age">Age</label>
3     <ul>
4         <li>This field is required</li>
5     </ul>
6     <input type="number" id="form_age" name="form[age]" />
7 </div>
```

To quickly prototype and test a form, you can render the entire form with just one line:

Listing 66-4

```
1 {# renders all fields #}
2 {{ form_widget(form) }}
3
```



```

4  {# renders all fields *and* the form start and end tags #}
5  {{ form(form) }}

```

The remainder of this recipe will explain how every part of the form's markup can be modified at several different levels. For more information about form rendering in general, see [Rendering a Form in a Template](#).

What are Form Themes?

Symfony uses form fragments - a small piece of a template that renders just one part of a form - to render each part of a form - field labels, errors, **input** text fields, **select** tags, etc.

The fragments are defined as blocks in Twig and as template files in PHP.

A *theme* is nothing more than a set of fragments that you want to use when rendering a form. In other words, if you want to customize one portion of how a form is rendered, you'll import a *theme* which contains a customization of the appropriate form fragments.

Symfony comes with some **built-in form themes** that define each and every fragment needed to render every part of a form:

- *form_div_layout.html.twig*¹, wraps each form field inside a `<div>` element.
- *form_table_layout.html.twig*², wraps the entire form inside a `<table>` element and each form field inside a `<tr>` element.
- *bootstrap_3_layout.html.twig*³, wraps each form field inside a `<div>` element with the appropriate CSS classes to apply the default *Bootstrap 3 CSS framework*⁴ styles.
- *bootstrap_3_horizontal_layout.html.twig*⁵, it's similar to the previous theme, but the CSS classes applied are the ones used to display the forms horizontally (i.e. the label and the widget in the same row).
- *foundation_5_layout.html.twig*⁶, wraps each form field inside a `<div>` element with the appropriate CSS classes to apply the default *Foundation CSS framework*⁷ styles.



When you use the Bootstrap form themes and render the fields manually, calling `form_label()` for a checkbox/radio field doesn't show anything. Due to Bootstrap internals, the label is already shown by `form_widget()`.

In the next section you will learn how to customize a theme by overriding some or all of its fragments.

For example, when the widget of an **integer** type field is rendered, an **input number** field is generated

Listing 66-5 1 {{ form_widget(form.age) }}

renders:

Listing 66-6 1 <input type="number" id="form_age" name="form[age]" required="required" value="33" />

Internally, Symfony uses the **integer_widget** fragment to render the field. This is because the field type is **integer** and you're rendering its **widget** (as opposed to its **label** or **errors**).

1. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig
2. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_table_layout.html.twig
3. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/bootstrap_3_layout.html.twig
4. <http://getbootstrap.com/>
5. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/bootstrap_3_horizontal_layout.html.twig
6. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/foundation_5_layout.html.twig
7. <http://foundation.zurb.com/>

In Twig that would default to the block `integer_widget` from the `form_div_layout.html.twig`⁸ template.

In PHP it would rather be the `integer_widget.html.php` file located in the `FrameworkBundle/Resources/views/Form` folder.

The default implementation of the `integer_widget` fragment looks like this:

Listing 66-7

```
1  {% form_div_layout.html.twig %}
2  {% block integer_widget %}
3      {% set type = type|default('number') %}
4      {{ block('form_widget_simple') }}
5  {% endblock integer_widget %}
```

As you can see, this fragment itself renders another fragment - `form_widget_simple`:

Listing 66-8

```
1  {% form_div_layout.html.twig %}
2  {% block form_widget_simple %}
3      {% set type = type|default('text') %}
4      <input type="{{ type }}" {{ block('widget_attributes') }} {% if value is not empty %}value="{{ value }}"
5  {% endif %}/>
6  {% endblock form_widget_simple %}
```

The point is, the fragments dictate the HTML output of each part of a form. To customize the form output, you just need to identify and override the correct fragment. A set of these form fragment customizations is known as a form "theme". When rendering a form, you can choose which form theme(s) you want to apply.

In Twig a theme is a single template file and the fragments are the blocks defined in this file.

In PHP a theme is a folder and the fragments are individual template files in this folder.



Knowing which Block to Customize

In this example, the customized fragment name is `integer_widget` because you want to override the HTML `widget` for all `integer` field types. If you need to customize `textarea` fields, you would customize `textarea_widget`.

The `integer` part comes from the class name: `IntegerType` becomes `integer`, based on a standard.

As you can see, the fragment name is a combination of the field type and which part of the field is being rendered (e.g. `widget`, `label`, `errors`, `row`). As such, to customize how errors are rendered for just input `text` fields, you should customize the `text_errors` fragment.

More commonly, however, you'll want to customize how errors are displayed across *all* fields. You can do this by customizing the `form_errors` fragment. This takes advantage of field type inheritance. Specifically, since the `text` type extends from the `form` type, the Form component will first look for the type-specific fragment (e.g. `text_errors`) before falling back to its parent fragment name if it doesn't exist (e.g. `form_errors`).

For more information on this topic, see Form Fragment Naming.

Form Theming

To see the power of form theming, suppose you want to wrap every input `number` field with a `div` tag. The key to doing this is to customize the `integer_widget` fragment.

8. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

Form Theming in Twig

When customizing the form field block in Twig, you have two options on *where* the customized form block can live:

Method	Pros	Cons
Inside the same template as the form	Quick and easy	Can't be reused in other templates
Inside a separate template	Can be reused by many templates	Requires an extra template to be created

Both methods have the same effect but are better in different situations.

Method 1: Inside the same Template as the Form

The easiest way to customize the `integer_widget` block is to customize it directly in the template that's actually rendering the form.

Listing 66-9

```
1 {% extends 'base.html.twig' %}
2
3 {% form_theme form_self %}
4
5 {% block integer_widget %}
6     <div class="integer_widget">
7         {% set type = type|default('number') %}
8         {{ block('form_widget_simple') }}
9     </div>
10 {% endblock %}
11
12 {% block content %}
13     {# ... render the form #}
14
15     {{ form_row(form.age) }}
16 {% endblock %}
```

By using the special `{% form_theme form_self %}` tag, Twig looks inside the same template for any overridden form blocks. Assuming the `form.age` field is an `integer` type field, when its widget is rendered, the customized `integer_widget` block will be used.

The disadvantage of this method is that the customized form block can't be reused when rendering other forms in other templates. In other words, this method is most useful when making form customizations that are specific to a single form in your application. If you want to reuse a form customization across several (or all) forms in your application, read on to the next section.

Method 2: Inside a separate Template

You can also choose to put the customized `integer_widget` form block in a separate template entirely. The code and end-result are the same, but you can now re-use the form customization across many templates:

Listing 66-10

```
1 {# app/Resources/views/form/fields.html.twig #}
2 {% block integer_widget %}
3     <div class="integer_widget">
4         {% set type = type|default('number') %}
5         {{ block('form_widget_simple') }}
6     </div>
7 {% endblock %}
```

Now that you've created the customized form block, you need to tell Symfony to use it. Inside the template where you're actually rendering your form, tell Symfony to use the template via the `form_theme` tag:

```
Listing 66-11 1 {% form_theme form 'form/fields.html.twig' %}
                2
                3 {{ form_widget(form.age) }}
```

When the `form.age` widget is rendered, Symfony will use the `integer_widget` block from the new template and the `input` tag will be wrapped in the `div` element specified in the customized block.

Multiple Templates

A form can also be customized by applying several templates. To do this, pass the name of all the templates as an array using the `with` keyword:

```
Listing 66-12 1 {% form_theme form with ['common.html.twig', 'form/fields.html.twig'] %}
                2
                3 {# ... #}
```

The templates can also be located in different bundles, use the functional name to reference these templates, e.g. `AcmeFormExtraBundle:form:fields.html.twig`.

Child Forms

You can also apply a form theme to a specific child of your form:

```
Listing 66-13 1 {% form_theme form.child 'form/fields.html.twig' %}
```

This is useful when you want to have a custom theme for a nested form that's different than the one of your main form. Just specify both your themes:

```
Listing 66-14 1 {% form_theme form 'form/fields.html.twig' %}
                2
                3 {% form_theme form.child 'form/fields_child.html.twig' %}
```

Form Theming in PHP

When using PHP as a templating engine, the only method to customize a fragment is to create a new template file - this is similar to the second method used by Twig.

The template file must be named after the fragment. You must create a `integer_widget.html.php` file in order to customize the `integer_widget` fragment.

```
Listing 66-15 1 <!-- app/Resources/views/form/integer_widget.html.php -->
                2 <div class="integer_widget">
                3     <?php echo $view['form']->block(
                4         $form,
                5         'form_widget_simple',
                6         array('type' => isset($type) ? $type : "number")
                7     ) ?>
                8 </div>
```

Now that you've created the customized form template, you need to tell Symfony to use it. Inside the template where you're actually rendering your form, tell Symfony to use the theme via the `setTheme` helper method:

```
Listing 66-16 1 <?php $view['form']->setTheme($form, array(':form')); ?>
2
3 <?php $view['form']->widget($form['age']) ?>
```

When the `form.age` widget is rendered, Symfony will use the customized `integer_widget.html.php` template and the `input` tag will be wrapped in the `div` element.

If you want to apply a theme to a specific child form, pass it to the `setTheme` method:

```
Listing 66-17 1 <?php $view['form']->setTheme($form['child'], ':form'); ?>
```



The `:form` syntax is based on the functional names for templates: `Bundle:Directory`. As the form directory lives in the `app/Resources/views` directory, the `Bundle` part is empty, resulting in `:form`.

Referencing base Form Blocks (Twig specific)

So far, to override a particular form block, the best method is to copy the default block from `form_div_layout.html.twig`⁹, paste it into a different template, and then customize it. In many cases, you can avoid doing this by referencing the base block when customizing it.

This is easy to do, but varies slightly depending on if your form block customizations are in the same template as the form or a separate template.

Referencing Blocks from inside the same Template as the Form

Import the blocks by adding a `use` tag in the template where you're rendering the form:

```
Listing 66-18 1 {% use 'form_div_layout.html.twig' with integer_widget as base_integer_widget %}
```

Now, when the blocks from `form_div_layout.html.twig`¹⁰ are imported, the `integer_widget` block is called `base_integer_widget`. This means that when you redefine the `integer_widget` block, you can reference the default markup via `base_integer_widget`:

```
Listing 66-19 1 {% block integer_widget %}
2     <div class="integer_widget">
3         {{ block('base_integer_widget') }}
4     </div>
5 {% endblock %}
```

Referencing base Blocks from an external Template

If your form customizations live inside an external template, you can reference the base block by using the `parent()` Twig function:

```
Listing 66-20 1 {% app/Resources/views/Form/fields.html.twig #}
2 {% extends 'form_div_layout.html.twig' %}
3
4 {% block integer_widget %}
5     <div class="integer_widget">
6         {{ parent() }}
```

9. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

10. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

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



It is not possible to reference the base block when using PHP as the templating engine. You have to manually copy the content from the base block to your new template file.

Making Application-wide Customizations

If you'd like a certain form customization to be global to your application, you can accomplish this by making the form customizations in an external template and then importing it inside your application configuration.

Twig

By using the following configuration, any customized form blocks inside the `form/fields.html.twig` template will be used globally when a form is rendered.

```
Listing 66-21 1 # app/config/config.yml
2 twig:
3     form_themes:
4         - 'form/fields.html.twig'
5     # ...
```

By default, Twig uses a *div* layout when rendering forms. Some people, however, may prefer to render forms in a *table* layout. Use the `form_table_layout.html.twig` resource to use such a layout:

```
Listing 66-22 1 # app/config/config.yml
2 twig:
3     form_themes:
4         - 'form_table_layout.html.twig'
5     # ...
```

If you only want to make the change in one template, add the following line to your template file rather than adding the template as a resource:

```
Listing 66-23 1 {% form_theme form 'form_table_layout.html.twig' %}
```

Note that the `form` variable in the above code is the form view variable that you passed to your template.

PHP

By using the following configuration, any customized form fragments inside the `app/Resources/views/Form` folder will be used globally when a form is rendered.

```
Listing 66-24 1 # app/config/config.yml
2 framework:
3     templating:
4         form:
5             resources:
6                 - 'AppBundle:Form'
7     # ...
```

By default, the PHP engine uses a *div* layout when rendering forms. Some people, however, may prefer to render forms in a *table* layout. Use the `FrameworkBundle:FormTable` resource to use such a layout:

Listing 66-25

```

1 # app/config/config.yml
2 framework:
3     templating:
4         form:
5             resources:
6                 - 'FrameworkBundle:FormTable'

```

If you only want to make the change in one template, add the following line to your template file rather than adding the template as a resource:

Listing 66-26

```

1 <?php $view['form']->setTheme($form, array('FrameworkBundle:FormTable')); ?>

```

Note that the `$form` variable in the above code is the form view variable that you passed to your template.

How to Customize an individual Field

So far, you've seen the different ways you can customize the widget output of all text field types. You can also customize individual fields. For example, suppose you have two **text** fields in a **product** form - **name** and **description** - but you only want to customize one of the fields. This can be accomplished by customizing a fragment whose name is a combination of the field's **id** attribute and which part of the field is being customized. For example, to customize the **name** field only:

Listing 66-27

```

1 {% form_theme form _self %}
2
3 {% block _product_name_widget %}
4     <div class="text_widget">
5         {{ block('form_widget_simple') }}
6     </div>
7 {% endblock %}
8
9 {{ form_widget(form.name) }}

```

Here, the `_product_name_widget` fragment defines the template to use for the field whose *id* is `product_name` (and name is `product[name]`).



The **product** portion of the field is the form name, which may be set manually or generated automatically based on your form type name (e.g. **ProductType** equates to **product**). If you're not sure what your form name is, just view the source of your generated form.

If you want to change the **product** or **name** portion of the block name `_product_name_widget` you can set the **block_name** option in your form type:

Listing 66-28

```

1 use Symfony\Component\Form\FormBuilderInterface;
2 use Symfony\Component\Form\Extension\Core\Type\TextType;
3
4 public function buildForm(FormBuilderInterface $builder, array $options)
5 {
6     // ...
7
8     $builder->add('name', TextType::class, array(
9         'block_name' => 'custom_name',
10     ));
11 }

```

Then the block name will be `_product_custom_name_widget`.

You can also override the markup for an entire field row using the same method:

Listing 66-29

```

1  {% form_theme form _self %}
2
3  {% block _product_name_row %}
4      <div class="name_row">
5          {{ form_label(form) }}
6          {{ form_errors(form) }}
7          {{ form_widget(form) }}
8      </div>
9  {% endblock %}
10
11 {{ form_row(form.name) }}
```

How to Customize a Collection Prototype

When using a *collection of forms*, the prototype can be overridden with a completely custom prototype by overriding a block. For example, if your form field is named **tasks**, you will be able to change the widget for each task as follows:

Listing 66-30

```

1  {% form_theme form _self %}
2
3  {% block _tasks_entry_widget %}
4      <tr>
5          <td>{{ form_widget(form.task) }}</td>
6          <td>{{ form_widget(form.dueDate) }}</td>
7      </tr>
8  {% endblock %}
```

Not only can you override the rendered widget, but you can also change the complete form row or the label as well. For the **tasks** field given above, the block names would be the following:

Part of the Form	Block Name
label	<code>_tasks_entry_label</code>
widget	<code>_tasks_entry_widget</code>
row	<code>_tasks_entry_row</code>

Other common Customizations

So far, this recipe has shown you several different ways to customize a single piece of how a form is rendered. The key is to customize a specific fragment that corresponds to the portion of the form you want to control (see naming form blocks).

In the next sections, you'll see how you can make several common form customizations. To apply these customizations, use one of the methods described in the Form Theming section.

Customizing Error Output



The Form component only handles *how* the validation errors are rendered, and not the actual validation error messages. The error messages themselves are determined by the validation constraints you apply to your objects. For more information, see the chapter on *validation*.

There are many different ways to customize how errors are rendered when a form is submitted with errors. The error messages for a field are rendered when you use the **form_errors** helper:

Listing 66-31 1 `{{ form_errors(form.age) }}`

By default, the errors are rendered inside an unordered list:

Listing 66-32 1 ``
2 `This field is required`
3 ``

To override how errors are rendered for *all* fields, simply copy, paste and customize the `form_errors` fragment.

Listing 66-33 1 `{# form_errors.html.twig #}`
2 `{% block form_errors %}`
3 `{% spaceless %}`
4 `{% if errors|length > 0 %}`
5 ``
6 `{% for error in errors %}`
7 `{{ error.message }}`
8 `{% endfor %}`
9 ``
10 `{% endif %}`
11 `{% endspaceless %}`
12 `{% endblock form_errors %}`



See Form Theming for how to apply this customization.

You can also customize the error output for just one specific field type. To customize *only* the markup used for these errors, follow the same directions as above but put the contents in a relative `_errors` block (or file in case of PHP templates). For example: `text_errors` (or `text_errors.html.php`).



See Form Fragment Naming to find out which specific block or file you have to customize.

Certain errors that are more global to your form (i.e. not specific to just one field) are rendered separately, usually at the top of your form:

Listing 66-34 1 `{{ form_errors(form) }}`

To customize *only* the markup used for these errors, follow the same directions as above, but now check if the `compound` variable is set to `true`. If it is `true`, it means that what's being currently rendered is a collection of fields (e.g. a whole form), and not just an individual field.

Listing 66-35 1 `{# form_errors.html.twig #}`
2 `{% block form_errors %}`
3 `{% spaceless %}`
4 `{% if errors|length > 0 %}`
5 `{% if compound %}`
6 ``
7 `{% for error in errors %}`
8 `{{ error.message }}`
9 `{% endfor %}`
10 ``
11 `{% else %}`
12 `{# ... display the errors for a single field #}`
13 `{% endif %}`
14 `{% endif %}`

```

15     {% endspaceless %}
16 {% endblock form_errors %}

```

Customizing the "Form Row"

When you can manage it, the easiest way to render a form field is via the `form_row` function, which renders the label, errors and HTML widget of a field. To customize the markup used for rendering *all* form field rows, override the `form_row` fragment. For example, suppose you want to add a class to the `div` element around each row:

Listing 66-36

```

1  {% form_row.html.twig %}
2  {% block form_row %}
3      <div class="form_row">
4          {{ form_label(form) }}
5          {{ form_errors(form) }}
6          {{ form_widget(form) }}
7      </div>
8  {% endblock form_row %}

```



See Form Theming for how to apply this customization.

Adding a "Required" Asterisk to Field Labels

If you want to denote all of your required fields with a required asterisk (*), you can do this by customizing the `form_label` fragment.

In Twig, if you're making the form customization inside the same template as your form, modify the `use` tag and add the following:

Listing 66-37

```

1  {% use 'form_div_layout.html.twig' with form_label as base_form_label %}
2
3  {% block form_label %}
4      {{ block('base_form_label') }}
5
6      {% if required %}
7          <span class="required" title="This field is required">*</span>
8      {% endif %}
9  {% endblock %}

```

In Twig, if you're making the form customization inside a separate template, use the following:

Listing 66-38

```

1  {% extends 'form_div_layout.html.twig' %}
2
3  {% block form_label %}
4      {{ parent() }}
5
6      {% if required %}
7          <span class="required" title="This field is required">*</span>
8      {% endif %}
9  {% endblock %}

```

When using PHP as a templating engine you have to copy the content from the original template:

Listing 66-39

```

1  <!-- form_label.html.php -->
2
3  <!-- original content -->
4  <?php if ($required) { $label_attr['class'] = trim((isset($label_attr['class']) ? $label_attr['class'] :

```

```

5  '').' required'); } ??>
6  <?php if (!$compound) { $label_attr['for'] = $id; } ??>
7  <?php if (!$label) { $label = $view['form']->humanize($name); } ??>
8  <label <?php foreach ($label_attr as $k => $v) { printf('%s="%s" ', $view->escape($k), $view->escape($v)); }
9  ?><?php echo $view->escape($view['translator']->trans($label, array(), $translation_domain)) ?></label>
10
11 <!-- customization -->
12 <?php if ($required) : ??>
    <span class="required" title="This field is required">*</span>
    <?php endif ??>

```



See Form Theming for how to apply this customization.



Using CSS only

By default, `label` tags of required fields are rendered with a `required` CSS class. Thus, you can also add an asterisk using CSS only:

Listing 66-40

```

1  label.required:before {
2      content: "* ";
3  }

```

Adding "help" Messages

You can also customize your form widgets to have an optional "help" message.

In Twig, if you're making the form customization inside the same template as your form, modify the `use` tag and add the following:

Listing 66-41

```

1  {% use 'form_div_layout.html.twig' with form_widget_simple as base_form_widget_simple %}
2
3  {% block form_widget_simple %}
4      {{ block('base_form_widget_simple') }}
5
6      {% if help is defined %}
7          <span class="help-block">{{ help }}</span>
8      {% endif %}
9  {% endblock %}

```

In Twig, if you're making the form customization inside a separate template, use the following:

Listing 66-42

```

1  {% extends 'form_div_layout.html.twig' %}
2
3  {% block form_widget_simple %}
4      {{ parent() }}
5
6      {% if help is defined %}
7          <span class="help-block">{{ help }}</span>
8      {% endif %}
9  {% endblock %}

```

When using PHP as a templating engine you have to copy the content from the original template:

Listing 66-43

```

1  <!-- form_widget_simple.html.php -->
2
3  <!-- Original content -->
4  <input
5      type="<?php echo isset($type) ? $view->escape($type) : 'text' ?>"

```

```

6     <?php if (!empty($value)): ?>value="<?php echo $view->escape($value) ?>"<?php endif ?>
7     <?php echo $view['form']->block($form, 'widget_attributes') ?>
8 />
9
10 <!-- Customization -->
11 <?php if (isset($help)) : ?>
12     <span class="help"><?php echo $view->escape($help) ?></span>
13 <?php endif ?>

```

To render a help message below a field, pass in a **help** variable:

Listing 66-44 1 `{{ form_widget(form.title, {'help': 'foobar'}) }}`



See Form Theming for how to apply this customization.

Using Form Variables

Most of the functions available for rendering different parts of a form (e.g. the form widget, form label, form errors, etc.) also allow you to make certain customizations directly. Look at the following example:

Listing 66-45 1 `{# render a widget, but add a "foo" class to it #}`
 2 `{{ form_widget(form.name, {'attr': {'class': 'foo'}}) }}`

The array passed as the second argument contains form "variables". For more details about this concept in Twig, see [More about Form Variables](#).



Chapter 67

How to Use Data Transformers

Data transformers are used to translate the data for a field into a format that can be displayed in a form (and back on submit). They're already used internally for many field types. For example, the *DateTime* field can be rendered as a **yyyy-MM-dd**-formatted input textbox. Internally, a data transformer converts the starting **DateTime** value of the field into the **yyyy-MM-dd** string to render the form, and then back into a **DateTime** object on submit.



When a form field has the `inherit_data` option set, Data Transformers won't be applied to that field.

Simple Example: Transforming String Tags from User Input to an Array

Suppose you have a Task form with a tags `text` type:

```
Listing 67-1 1 // src/AppBundle/Form/TaskType.php
2 namespace AppBundle\Form\Type;
3
4 use Symfony\Component\Form\FormBuilderInterface;
5 use Symfony\Component\OptionsResolver\OptionsResolver;
6 use Symfony\Component\Form\Extension\Core\Type\TextType;
7
8 // ...
9 class TaskType extends AbstractType
10 {
11     public function buildForm(FormBuilderInterface $builder, array $options)
12     {
13         $builder->add('tags', TextType::class)
14     }
15
16     public function configureOptions(OptionsResolver $resolver)
17     {
18         $resolver->setDefaults(array(
19             'data_class' => 'AppBundle\Entity\Task',
20         ));
21     }
22 }
```

```

23     // ...
24 }

```

Internally the **tags** are stored as an array, but displayed to the user as a simple comma separated string to make them easier to edit.

This is a *perfect* time to attach a custom data transformer to the **tags** field. The easiest way to do this is with the *CallbackTransformer*¹ class:

Listing 67-2

```

1  // src/AppBundle/Form/TaskType.php
2  namespace AppBundle\Form\Type;
3
4  use Symfony\Component\Form\CallbackTransformer;
5  use Symfony\Component\Form\FormBuilderInterface;
6  use Symfony\Component\Form\Extension\Core\Type\TextType;
7  // ...
8
9  class TaskType extends AbstractType
10 {
11     public function buildForm(FormBuilderInterface $builder, array $options)
12     {
13         $builder->add('tags', TextType::class);
14
15         $builder->get('tags')
16             ->addModelTransformer(new CallbackTransformer(
17                 function ($tagsAsArray) {
18                     // transform the array to a string
19                     return implode(', ', $tagsAsArray);
20                 },
21                 function ($tagsAsString) {
22                     // transform the string back to an array
23                     return explode(', ', $tagsAsString);
24                 }
25             ));
26     }
27 }
28
29 // ...
30 }

```

The **CallbackTransformer** takes two callback functions as arguments. The first transforms the original value into a format that'll be used to render the field. The second does the reverse: it transforms the submitted value back into the format you'll use in your code.



The **addModelTransformer()** method accepts *any* object that implements *DataTransformerInterface*² - so you can create your own classes, instead of putting all the logic in the form (see the next section).

You can also add the transformer, right when adding the field by changing the format slightly:

Listing 67-3

```

1  use Symfony\Component\Form\Extension\Core\Type\TextType;
2
3  $builder->add(
4      $builder
5          ->create('tags', TextType::class)
6          ->addModelTransformer(...)
7  );

```

1. <http://api.symfony.com/3.0/Symfony/Component/Form/CallbackTransformer.html>

2. <http://api.symfony.com/3.0/Symfony/Component/Form/DataTransformerInterface.html>

Harder Example: Transforming an Issue Number into an Issue Entity

Say you have a many-to-one relation from the Task entity to an Issue entity (i.e. each Task has an optional foreign key to its related Issue). Adding a listbox with all possible issues could eventually get *really* long and take a long time to load. Instead, you decide you want to add a textbox, where the user can simply enter the issue number.

Start by setting up the text field like normal:

```
Listing 67-4 1 // src/AppBundle/Form/TaskType.php
2 namespace AppBundle\Form\Type;
3
4 use Symfony\Component\Form\Extension\Core\Type\TextareaType;
5 use Symfony\Component\Form\Extension\Core\Type\TextType;
6
7 // ...
8 class TaskType extends AbstractType
9 {
10     public function buildForm(FormBuilderInterface $builder, array $options)
11     {
12         $builder
13             ->add('description', TextareaType::class)
14             ->add('issue', TextType::class)
15         ;
16     }
17
18     public function configureOptions(OptionsResolver $resolver)
19     {
20         $resolver->setDefaults(array(
21             'data_class' => 'AppBundle\Entity\Task'
22         ));
23     }
24
25     // ...
26 }
```

Good start! But if you stopped here and submitted the form, the Task's **issue** property would be a string (e.g. "55"). How can you transform this into an **Issue** entity on submit?

Creating the Transformer

You could use the **CallbackTransformer** like earlier. But since this is a bit more complex, creating a new transformer class will keep the **TaskType** form class simpler.

Create an **IssueToNumberTransformer** class: it will be responsible for converting to and from the issue number and the **Issue** object:

```
Listing 67-5 1 // src/AppBundle/Form/DataTransformer/IssueToNumberTransformer.php
2 namespace AppBundle\Form\DataTransformer;
3
4 use AppBundle\Entity\Issue;
5 use Doctrine\Common\Persistence\ObjectManager;
6 use Symfony\Component\Form\DataTransformerInterface;
7 use Symfony\Component\Form\Exception\TransformationFailedException;
8
9 class IssueToNumberTransformer implements DataTransformerInterface
10 {
11     private $manager;
12
13     public function __construct(ObjectManager $manager)
14     {
15         $this->manager = $manager;
16     }
17
18     /**
```

```

19      * Transforms an object (issue) to a string (number).
20      *
21      * @param Issue|null $issue
22      * @return string
23      */
24      public function transform($issue)
25      {
26          if (null === $issue) {
27              return '';
28          }
29
30          return $issue->getId();
31      }
32
33      /**
34       * Transforms a string (number) to an object (issue).
35       *
36       * @param string $issueNumber
37       * @return Issue|null
38       * @throws TransformationFailedException if object (issue) is not found.
39       */
40      public function reverseTransform($issueNumber)
41      {
42          // no issue number? It's optional, so that's ok
43          if (!$issueNumber) {
44              return;
45          }
46
47          $issue = $this->manager
48              ->getRepository('AppBundle:Issue')
49              // query for the issue with this id
50              ->find($issueNumber)
51          ;
52
53          if (null === $issue) {
54              // causes a validation error
55              // this message is not shown to the user
56              // see the invalid_message option
57              throw new TransformationFailedException(sprintf(
58                  'An issue with number "%s" does not exist!',
59                  $issueNumber
60              ));
61          }
62
63          return $issue;
64      }
65  }

```

Just like in the first example, a transformer has two directions. The **transform()** method is responsible for converting the data used in your code to a format that can be rendered in your form (e.g. an **Issue** object to its **id**, a string). The **reverseTransform()** method does the reverse: it converts the submitted value back into the format you want (e.g. convert the **id** back to the **Issue** object).

To cause a validation error, throw a *TransformationFailedException*³. But the message you pass to this exception won't be shown to the user. You'll set that message with the **invalid_message** option (see below).



When **null** is passed to the **transform()** method, your transformer should return an equivalent value of the type it is transforming to (e.g. an empty string, 0 for integers or 0.0 for floats).

3. <http://api.symfony.com/3.0/Symfony/Component/Form/Exception/TransformationFailedException.html>

Using the Transformer

Next, you need to instantiate the `IssueToNumberTransformer` class from inside `TaskType` and add it to the `issue` field. But to do that, you'll need an instance of the entity manager (because `IssueToNumberTransformer` needs this).

No problem! Just add a `__construct()` function to `TaskType` and force this to be passed in by registering `TaskType` as a service:

```
Listing 67-6 1 // src/AppBundle/Form/TaskType.php
2 namespace AppBundle\Form\Type;
3
4 use AppBundle\Form\DataTransformer\IssueToNumberTransformer;
5 use Doctrine\Common\Persistence\ObjectManager;
6 use Symfony\Component\Form\Extension\Core\Type\TextareaType;
7 use Symfony\Component\Form\Extension\Core\Type\TextType;
8
9 // ...
10 class TaskType extends AbstractType
11 {
12     private $manager;
13
14     public function __construct(ObjectManager $manager)
15     {
16         $this->manager = $manager;
17     }
18
19     public function buildForm(FormBuilderInterface $builder, array $options)
20     {
21         $builder
22             ->add('description', TextareaType::class)
23             ->add('issue', TextType::class, array(
24                 // validation message if the data transformer fails
25                 'invalid_message' => 'That is not a valid issue number',
26             ));
27
28         // ...
29
30         $builder->get('issue')
31             ->addModelTransformer(new IssueToNumberTransformer($this->manager));
32     }
33
34     // ...
35 }
```

Define the form type as a service in your configuration files.

```
Listing 67-7 1 # src/AppBundle/Resources/config/services.yml
2 services:
3     app.form.type.task:
4         class: AppBundle\Form\Type\TaskType
5         arguments: ["@doctrine.orm.entity_manager"]
6         tags:
7             - { name: form.type }
```



For more information about defining form types as services, read [register your form type as a service](#).

Now, you can easily use your `TaskType`:

```
Listing 67-8 // e.g. in a controller somewhere
$form = $this->createForm(TaskType::class, $task);

// ...
```

Cool, you're done! Your user will be able to enter an issue number into the text field and it will be transformed back into an Issue object. This means that, after a successful submission, the Form component will pass a real **Issue** object to **Task::setIssue()** instead of the issue number.

If the issue isn't found, a form error will be created for that field and its error message can be controlled with the **invalid_message** field option.



Be careful when adding your transformers. For example, the following is **wrong**, as the transformer would be applied to the entire form, instead of just this field:

Listing 67-9 *// THIS IS WRONG - TRANSFORMER WILL BE APPLIED TO THE ENTIRE FORM*
// see above example for correct code

```
$builder->add('issue', TextType::class)
->addModelTransformer($transformer);
```

Creating a Reusable issue_selector Field

In the above example, you applied the transformer to a normal **text** field. But if you do this transformation a lot, it might be better to *create a custom field type*. that does this automatically.

First, create the custom field type class:

Listing 67-10

```
1 // src/AppBundle/Form/IssueSelectorType.php
2 namespace AppBundle\Form;
3
4 use AppBundle\Form\DataTransformer\IssueToNumberTransformer;
5 use Doctrine\Common\Persistence\ObjectManager;
6 use Symfony\Component\Form\AbstractType;
7 use Symfony\Component\Form\FormBuilderInterface;
8 use Symfony\Component\OptionsResolver\OptionsResolver;
9
10 class IssueSelectorType extends AbstractType
11 {
12     private $manager;
13
14     public function __construct(ObjectManager $manager)
15     {
16         $this->manager = $manager;
17     }
18
19     public function buildForm(FormBuilderInterface $builder, array $options)
20     {
21         $transformer = new IssueToNumberTransformer($this->manager);
22         $builder->addModelTransformer($transformer);
23     }
24
25     public function configureOptions(OptionsResolver $resolver)
26     {
27         $resolver->setDefaults(array(
28             'invalid_message' => 'The selected issue does not exist',
29         ));
30     }
31
32     public function getParent()
33     {
34         return TextType::class;
35     }
36 }
```

Great! This will act and render like a text field (**getParent()**), but will automatically have the data transformer *and* a nice default value for the **invalid_message** option.

Next, register your type as a service and tag it with **form.type** so that it's recognized as a custom field type:

Listing 67-11

```

1  # app/config/services.yml
2  services:
3      app.type.issue_selector:
4          class: AppBundle\Form\IssueSelectorType
5          arguments: ['@doctrine.orm.entity_manager']
6          tags:
7              - { name: form.type }

```

Now, whenever you need to use your special `issue_selector` field type, it's quite easy:

Listing 67-12

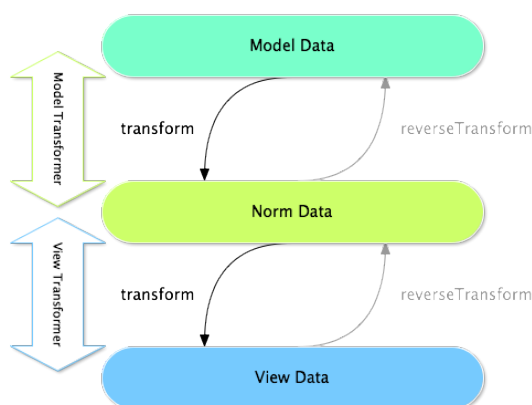
```

1  // src/AppBundle/Form/TaskType.php
2  namespace AppBundle\Form\Type;
3
4  use AppBundle\Form\DataTransformer\IssueToNumberTransformer;
5  use Symfony\Component\Form\Extension\Core\Type\TextareaType;
6  // ...
7
8  class TaskType extends AbstractType
9  {
10     public function buildForm(FormBuilderInterface $builder, array $options)
11     {
12         $builder
13             ->add('description', TextareaType::class)
14             ->add('issue', IssueSelectorType::class)
15         ;
16     }
17
18     // ...
19 }

```

About Model and View Transformers

In the above example, the transformer was used as a "model" transformer. In fact, there are two different types of transformers and three different types of underlying data.



In any form, the three different types of data are:

1. **Model data** - This is the data in the format used in your application (e.g. an `Issue` object). If you call `Form::getData()` or `Form::setData()`, you're dealing with the "model" data.
2. **Norm Data** - This is a normalized version of your data and is commonly the same as your "model" data (though not in our example). It's not commonly used directly.

3. **View Data** - This is the format that's used to fill in the form fields themselves. It's also the format in which the user will submit the data. When you call `Form::submit($data)`, the `$data` is in the "view" data format.

The two different types of transformers help convert to and from each of these types of data:

Model transformers:

- `transform: "model data" => "norm data"`
- `reverseTransform: "norm data" => "model data"`

View transformers:

- `transform: "norm data" => "view data"`
- `reverseTransform: "view data" => "norm data"`

Which transformer you need depends on your situation.

To use the view transformer, call `addViewTransformer`.

So why Use the Model Transformer?

In this example, the field is a **text** field, and a text field is always expected to be a simple, scalar format in the "norm" and "view" formats. For this reason, the most appropriate transformer was the "model" transformer (which converts to/from the *norm* format - string issue number - to the *model* format - Issue object).

The difference between the transformers is subtle and you should always think about what the "norm" data for a field should really be. For example, the "norm" data for a **text** field is a string, but is a **DateTime** object for a **date** field.



As a general rule, the normalized data should contain as much information as possible.



Chapter 68

How to Dynamically Modify Forms Using Form Events

Often times, a form can't be created statically. In this entry, you'll learn how to customize your form based on three common use-cases:

1. Customizing your Form Based on the Underlying Data

Example: you have a "Product" form and need to modify/add/remove a field

based on the data on the underlying Product being edited.

2. How to dynamically Generate Forms Based on user Data

Example: you create a "Friend Message" form and need to build a drop-down that contains only users that are friends with the *current* authenticated user.

3. Dynamic Generation for Submitted Forms

Example: on a registration form, you have a "country" field and a "state" field which should populate dynamically based on the value in the "country" field.

If you wish to learn more about the basics behind form events, you can take a look at the *Form Events* documentation.

Customizing your Form Based on the Underlying Data

Before jumping right into dynamic form generation, hold on and recall what a bare form class looks like:

```
Listing 68-1 1 // src/AppBundle/Form/Type/ProductType.php
2 namespace AppBundle\Form\Type;
3
4 use Symfony\Component\Form\AbstractType;
5 use Symfony\Component\Form\FormBuilderInterface;
6 use Symfony\Component\OptionsResolver\OptionsResolver;
7
8 class ProductType extends AbstractType
9 {
10     public function buildForm(FormBuilderInterface $builder, array $options)
11     {
```

```

12     $builder->add('name');
13     $builder->add('price');
14 }
15
16 public function configureOptions(OptionsResolver $resolver)
17 {
18     $resolver->setDefaults(array(
19         'data_class' => 'AppBundle\Entity\Product'
20     ));
21 }
22 }

```



If this particular section of code isn't already familiar to you, you probably need to take a step back and first review the *Forms chapter* before proceeding.

Assume for a moment that this form utilizes an imaginary "Product" class that has only two properties ("name" and "price"). The form generated from this class will look the exact same regardless if a new Product is being created or if an existing product is being edited (e.g. a product fetched from the database).

Suppose now, that you don't want the user to be able to change the **name** value once the object has been created. To do this, you can rely on Symfony's *EventDispatcher component* system to analyze the data on the object and modify the form based on the Product object's data. In this entry, you'll learn how to add this level of flexibility to your forms.

Adding an Event Listener to a Form Class

So, instead of directly adding that **name** widget, the responsibility of creating that particular field is delegated to an event listener:

Listing 68-2

```

1 // src/AppBundle/Form/Type/ProductType.php
2 namespace AppBundle\Form\Type;
3
4 // ...
5 use Symfony\Component\Form\FormEvent;
6 use Symfony\Component\Form\FormEvents;
7
8 class ProductType extends AbstractType
9 {
10     public function buildForm(FormBuilderInterface $builder, array $options)
11     {
12         $builder->add('price');
13
14         $builder->addEventListener(FormEvents::PRE_SET_DATA, function (FormEvent $event) {
15             // ... adding the name field if needed
16         });
17     }
18
19     // ...
20 }

```

The goal is to create a **name** field *only* if the underlying **Product** object is new (e.g. hasn't been persisted to the database). Based on that, the event listener might look like the following:

Listing 68-3

```

1 // ...
2 public function buildForm(FormBuilderInterface $builder, array $options)
3 {
4     // ...
5     $builder->addEventListener(FormEvents::PRE_SET_DATA, function (FormEvent $event) {
6         $product = $event->getData();
7         $form = $event->getForm();

```

```

8
9      // check if the Product object is "new"
10     // If no data is passed to the form, the data is "null".
11     // This should be considered a new "Product"
12     if (!$product || null === $product->getId()) {
13         $form->add('name', TextType::class);
14     }
15     });
16 }

```



The `FormEvents::PRE_SET_DATA` line actually resolves to the string `form.pre_set_data`. *FormEvents*¹ serves an organizational purpose. It is a centralized location in which you can find all of the various form events available. You can view the full list of form events via the *FormEvents*² class.

Adding an Event Subscriber to a Form Class

For better reusability or if there is some heavy logic in your event listener, you can also move the logic for creating the `name` field to an event subscriber:

Listing 68-4

```

1  // src/AppBundle/Form/Type/ProductType.php
2  namespace AppBundle\Form\Type;
3
4  // ...
5  use AppBundle\Form\EventListener\AddNameFieldSubscriber;
6
7  class ProductType extends AbstractType
8  {
9      public function buildForm(FormBuilderInterface $builder, array $options)
10     {
11         $builder->add('price');
12
13         $builder->addEventSubscriber(new AddNameFieldSubscriber());
14     }
15
16     // ...
17 }

```

Now the logic for creating the `name` field resides in its own subscriber class:

Listing 68-5

```

1  // src/AppBundle/Form/EventListener/AddNameFieldSubscriber.php
2  namespace AppBundle\Form\EventListener;
3
4  use Symfony\Component\Form\FormEvent;
5  use Symfony\Component\Form\FormEvents;
6  use Symfony\Component\EventDispatcher\EventSubscriberInterface;
7  use Symfony\Component\Form\Extension\Core\Type\TextType;
8
9  class AddNameFieldSubscriber implements EventSubscriberInterface
10 {
11     public static function getSubscribedEvents()
12     {
13         // Tells the dispatcher that you want to listen on the form.pre_set_data
14         // event and that the preSetData method should be called.
15         return array(FormEvents::PRE_SET_DATA => 'preSetData');
16     }
17
18     public function preSetData(FormEvent $event)
19     {
20         $product = $event->getData();

```

1. <http://api.symfony.com/3.0/Symfony/Component/Form/FormEvents.html>
2. <http://api.symfony.com/3.0/Symfony/Component/Form/FormEvents.html>

```

21     $form = $event->getForm();
22
23     if (!$product || null === $product->getId()) {
24         $form->add('name', TextType::class);
25     }
26 }
27 }

```

How to dynamically Generate Forms Based on user Data

Sometimes you want a form to be generated dynamically based not only on data from the form but also on something else - like some data from the current user. Suppose you have a social website where a user can only message people marked as friends on the website. In this case, a "choice list" of whom to message should only contain users that are the current user's friends.

Creating the Form Type

Using an event listener, your form might look like this:

Listing 68-6

```

1  // src/AppBundle/Form/Type/FriendMessageFormType.php
2  namespace AppBundle\Form\Type;
3
4  use Symfony\Component\Form\AbstractType;
5  use Symfony\Component\Form\FormBuilderInterface;
6  use Symfony\Component\Form\FormEvents;
7  use Symfony\Component\Form\FormEvent;
8  use Symfony\Component\Security\Core\Authentication\Token\Storage\TokenStorageInterface;
9  use Symfony\Component\Form\Extension\Core\Type\TextType;
10 use Symfony\Component\Form\Extension\Core\Type\TextareaType;
11
12 class FriendMessageFormType extends AbstractType
13 {
14     public function buildForm(FormBuilderInterface $builder, array $options)
15     {
16         $builder
17             ->add('subject', TextType::class)
18             ->add('body', TextareaType::class)
19         ;
20         $builder->addEventListener(FormEvents::PRE_SET_DATA, function (FormEvent $event) {
21             // ... add a choice list of friends of the current application user
22         });
23     }
24 }

```

The problem is now to get the current user and create a choice field that contains only this user's friends. Luckily it is pretty easy to inject a service inside of the form. This can be done in the constructor:

Listing 68-7

```

1  private $tokenStorage;
2
3  public function __construct(TokenStorageInterface $tokenStorage)
4  {
5      $this->tokenStorage = $tokenStorage;
6  }

```



You might wonder, now that you have access to the User (through the token storage), why not just use it directly in **buildForm** and omit the event listener? This is because doing so in the **buildForm** method would result in the whole form type being modified and not just this one form instance. This may not usually be a problem, but technically a single form type could be used on a single request to create many forms or fields.

Customizing the Form Type

Now that you have all the basics in place you can take advantage of the `TokenStorageInterface` and fill in the listener logic:

```
Listing 68-8 1 // src/AppBundle/FormType/FriendMessageFormType.php
2
3 use Symfony\Component\Security\Core\Authentication\Token\Storage\TokenStorageInterface;
4 use Doctrine\ORM\EntityRepository;
5 use Symfony\Component\Form\Extension\Core\Type\TextType;
6 use Symfony\Component\Form\Extension\Core\Type\TextareaType;
7 use Symfony\Bridge\Doctrine\Form\Type\EntityType;
8
9 // ...
10
11 class FriendMessageFormType extends AbstractType
12 {
13     private $tokenStorage;
14
15     public function __construct(TokenStorageInterface $tokenStorage)
16     {
17         $this->tokenStorage = $tokenStorage;
18     }
19
20     public function buildForm(FormBuilderInterface $builder, array $options)
21     {
22         $builder
23             ->add('subject', TextType::class)
24             ->add('body', TextareaType::class)
25         ;
26
27         // grab the user, do a quick sanity check that one exists
28         $user = $this->tokenStorage->getToken()->getUser();
29         if (!$user) {
30             throw new \LogicException(
31                 'The FriendMessageFormType cannot be used without an authenticated user!'
32             );
33         }
34
35         $builder->addEventListener(
36             FormEvents::PRE_SET_DATA,
37             function (FormEvent $event) use ($user) {
38                 $form = $event->getForm();
39
40                 $formOptions = array(
41                     'class' => 'AppBundle\Entity\User',
42                     'property' => 'fullName',
43                     'query_builder' => function (EntityRepository $er) use ($user) {
44                         // build a custom query
45                         // return $er->createQueryBuilder('u')->addOrderBy('fullName', 'DESC');
46
47                         // or call a method on your repository that returns the query builder
48                         // the $er is an instance of your UserRepository
49                         // return $er->createOrderByFullNameQueryBuilder();
50                     },
51                 );
52
53                 // create the field, this is similar the $builder->add()
54                 // field name, field type, data, options
55                 $form->add('friend', EntityType::class, $formOptions);
56             }
57         );
58     }
59
60     // ...
61 }
```



The **multiple** and **expanded** form options will default to false because the type of the friend field is `EntityType::class`.

Using the Form

Our form is now ready to use. But first, because it has a `__construct()` method, you need to register it as a service and tag it with `form.type`:

Listing 68-9

```
1 # app/config/config.yml
2 services:
3     app.form.friend_message:
4         class: AppBundle\Form\Type\FriendMessageFormType
5         arguments: ['@security.token_storage']
6         tags:
7             - { name: form.type }
```

In a controller that extends the *Controller*³ class, you can simply call:

Listing 68-10

```
1 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
2
3 class FriendMessageController extends Controller
4 {
5     public function newAction(Request $request)
6     {
7         $form = $this->createForm(FriendMessageFormType::class);
8
9         // ...
10    }
11 }
```

You can also easily embed the form type into another form:

Listing 68-11

```
1 // inside some other "form type" class
2 public function buildForm(FormBuilderInterface $builder, array $options)
3 {
4     $builder->add('message', FriendMessageFormType::class);
5 }
```

Dynamic Generation for Submitted Forms

Another case that can appear is that you want to customize the form specific to the data that was submitted by the user. For example, imagine you have a registration form for sports gatherings. Some events will allow you to specify your preferred position on the field. This would be a **choice** field for example. However the possible choices will depend on each sport. Football will have attack, defense, goalkeeper etc... Baseball will have a pitcher but will not have a goalkeeper. You will need the correct options in order for validation to pass.

The meetup is passed as an entity field to the form. So we can access each sport like this:

Listing 68-12

```
1 // src/AppBundle/Form/Type/SportMeetupType.php
2 namespace AppBundle\Form\Type;
3
4 use Symfony\Component\Form\AbstractType;
5 use Symfony\Component\Form\FormBuilderInterface;
6 use Symfony\Component\Form\FormEvent;
7 use Symfony\Component\Form\FormEvents;
```

3. <http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html>

```

8 use Symfony\Bridge\Doctrine\Form\Type\EntityType;
9 // ...
10
11 class SportMeetupType extends AbstractType
12 {
13     public function buildForm(FormBuilderInterface $builder, array $options)
14     {
15         $builder
16             ->add('sport', EntityType::class, array(
17                 'class' => 'AppBundle:Sport',
18                 'placeholder' => '',
19             ))
20         ;
21
22         $builder->addEventListener(
23             FormEvents::PRE_SET_DATA,
24             function (FormEvent $event) {
25                 $form = $event->getForm();
26
27                 // this would be your entity, i.e. SportMeetup
28                 $data = $event->getData();
29
30                 $sport = $data->getSport();
31                 $positions = null === $sport ? array() : $sport->getAvailablePositions();
32
33                 $form->add('position', EntityType::class, array(
34                     'class' => 'AppBundle:Position',
35                     'placeholder' => '',
36                     'choices' => $positions,
37                 ));
38             }
39         );
40     }
41
42     // ...
43 }

```

When you're building this form to display to the user for the first time, then this example works perfectly. However, things get more difficult when you handle the form submission. This is because the `PRE_SET_DATA` event tells us the data that you're starting with (e.g. an empty `SportMeetup` object), *not* the submitted data.

On a form, we can usually listen to the following events:

- `PRE_SET_DATA`
- `POST_SET_DATA`
- `PRE_SUBMIT`
- `SUBMIT`
- `POST_SUBMIT`

The key is to add a `POST_SUBMIT` listener to the field that your new field depends on. If you add a `POST_SUBMIT` listener to a form child (e.g. `sport`), and add new children to the parent form, the Form component will detect the new field automatically and map it to the submitted client data.

The type would now look like:

Listing 68-13

```

1 // src/AppBundle/Form/Type/SportMeetupType.php
2 namespace AppBundle\Form\Type;
3
4 // ...
5 use Symfony\Component\Form\FormInterface;
6 use Symfony\Bridge\Doctrine\Form\Type\EntityType;
7 use AppBundle\Entity\Sport;
8
9 class SportMeetupType extends AbstractType
10 {

```

```

11     public function buildForm(FormBuilderInterface $builder, array $options)
12     {
13         $builder
14             ->add('sport', EntityType::class, array(
15                 'class' => 'AppBundle:Sport',
16                 'placeholder' => '',
17             ));
18     };
19
20     $formModifier = function (FormInterface $form, Sport $sport = null) {
21         $positions = null === $sport ? array() : $sport->getAvailablePositions();
22
23         $form->add('position', EntityType::class, array(
24             'class' => 'AppBundle:Position',
25             'placeholder' => '',
26             'choices' => $positions,
27         ));
28     };
29
30     $builder->addEventListener(
31         FormEvents::PRE_SET_DATA,
32         function (FormEvent $event) use ($formModifier) {
33             // this would be your entity, i.e. SportMeetup
34             $data = $event->getData();
35
36             $formModifier($event->getForm(), $data->getSport());
37         }
38     );
39
40     $builder->get('sport')->addEventListener(
41         FormEvents::POST_SUBMIT,
42         function (FormEvent $event) use ($formModifier) {
43             // It's important here to fetch $event->getForm()->getData(), as
44             // $event->getData() will get you the client data (that is, the ID)
45             $sport = $event->getForm()->getData();
46
47             // since we've added the listener to the child, we'll have to pass on
48             // the parent to the callback functions!
49             $formModifier($event->getForm()->getParent(), $sport);
50         }
51     );
52 }
53
54 // ...
55 }

```

You can see that you need to listen on these two events and have different callbacks only because in two different scenarios, the data that you can use is available in different events. Other than that, the listeners always perform exactly the same things on a given form.

One piece that is still missing is the client-side updating of your form after the sport is selected. This should be handled by making an AJAX call back to your application. Assume that you have a sport meetup creation controller:

Listing 68-14

```

1  // src/AppBundle/Controller/MeetupController.php
2  namespace AppBundle\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5  use Symfony\Component\HttpFoundation\Request;
6  use AppBundle\Entity\SportMeetup;
7  use AppBundle\Form\Type\SportMeetupType;
8  // ...
9
10 class MeetupController extends Controller
11 {
12     public function createAction(Request $request)
13     {
14         $meetup = new SportMeetup();
15         $form = $this->createForm(SportMeetupType::class, $meetup);

```

```

16     $form->handleRequest($request);
17     if ($form->isValid()) {
18         // ... save the meetup, redirect etc.
19     }
20
21     return $this->render(
22         'AppBundle:Meetup:create.html.twig',
23         array('form' => $form->createView())
24     );
25 }
26
27 // ...
28 }

```

The associated template uses some JavaScript to update the **position** form field according to the current selection in the **sport** field:

Listing 68-15

```

1  {# app/Resources/views/Meetup/create.html.twig #}
2  {{ form_start(form) }}
3      {{ form_row(form.sport) }} {# <select id="meetup_sport" ... #}
4      {{ form_row(form.position) }} {# <select id="meetup_position" ... #}
5      {# ... #}
6  {{ form_end(form) }}
7
8  <script>
9      var $sport = $('#meetup_sport');
10     // When sport gets selected ...
11     $sport.change(function() {
12         // ... retrieve the corresponding form.
13         var $form = $(this).closest('form');
14         // Simulate form data, but only include the selected sport value.
15         var data = {};
16         data[$sport.attr('name')] = $sport.val();
17         // Submit data via AJAX to the form's action path.
18         $.ajax({
19             url : $form.attr('action'),
20             type: $form.attr('method'),
21             data : data,
22             success: function(html) {
23                 // Replace current position field ...
24                 $('#meetup_position').replaceWith(
25                     // ... with the returned one from the AJAX response.
26                     $(html).find('#meetup_position')
27                 );
28                 // Position field now displays the appropriate positions.
29             }
30         });
31     });
32 </script>

```

The major benefit of submitting the whole form to just extract the updated **position** field is that no additional server-side code is needed; all the code from above to generate the submitted form can be reused.

Suppressing Form Validation

To suppress form validation you can use the `POST_SUBMIT` event and prevent the *ValidationListener*⁴ from being called.

The reason for needing to do this is that even if you set `validation_groups` to `false` there are still some integrity checks executed. For example an uploaded file will still be checked to see if it is too large

4. <http://api.symfony.com/3.0/Symfony/Component/Form/Extension/Validator/EventListener/ValidationListener.html>

and the form will still check to see if non-existing fields were submitted. To disable all of this, use a listener:

Listing 68-16

```
1 use Symfony\Component\Form\FormBuilderInterface;
2 use Symfony\Component\Form\FormEvents;
3 use Symfony\Component\Form\FormEvent;
4
5 public function buildForm(FormBuilderInterface $builder, array $options)
6 {
7     $builder->addEventListener(FormEvents::POST_SUBMIT, function (FormEvent $event) {
8         $event->stopPropagation();
9     }, 900); // Always set a higher priority than ValidationListener
10
11     // ...
12 }
```



By doing this, you may accidentally disable something more than just form validation, since the `POST_SUBMIT` event may have other listeners.



Chapter 69

How to Embed a Collection of Forms

In this entry, you'll learn how to create a form that embeds a collection of many other forms. This could be useful, for example, if you had a **Task** class and you wanted to edit/create/remove many **Tag** objects related to that Task, right inside the same form.



In this entry, it's loosely assumed that you're using Doctrine as your database store. But if you're not using Doctrine (e.g. Propel or just a database connection), it's all very similar. There are only a few parts of this tutorial that really care about "persistence".

If you *are* using Doctrine, you'll need to add the Doctrine metadata, including the **ManyToOne** association mapping definition on the Task's **tags** property.

First, suppose that each **Task** belongs to multiple **Tag** objects. Start by creating a simple **Task** class:

```
Listing 69-1 1 // src/AppBundle/Entity/Task.php
2 namespace AppBundle\Entity;
3
4 use Doctrine\Common\Collections\ArrayCollection;
5
6 class Task
7 {
8     protected $description;
9
10    protected $tags;
11
12    public function __construct()
13    {
14        $this->tags = new ArrayCollection();
15    }
16
17    public function getDescription()
18    {
19        return $this->description;
20    }
21
22    public function setDescription($description)
23    {
24        $this->description = $description;
25    }
26
27    public function getTags()
```

```

28     {
29         return $this->tags;
30     }
31 }

```



The **ArrayCollection** is specific to Doctrine and is basically the same as using an **array** (but it must be an **ArrayCollection** if you're using Doctrine).

Now, create a **Tag** class. As you saw above, a **Task** can have many **Tag** objects:

Listing 69-2

```

1  // src/AppBundle/Entity/Tag.php
2  namespace AppBundle\Entity;
3
4  class Tag
5  {
6      private $name;
7
8      public function getName()
9      {
10         return $this->name;
11     }
12
13     public function setName($name)
14     {
15         $this->name = $name;
16     }
17 }

```

Then, create a form class so that a **Tag** object can be modified by the user:

Listing 69-3

```

1  // src/AppBundle/Form/Type/TagType.php
2  namespace AppBundle\Form\Type;
3
4  use Symfony\Component\Form\AbstractType;
5  use Symfony\Component\Form\FormBuilderInterface;
6  use Symfony\Component\OptionsResolver\OptionsResolver;
7
8  class TagType extends AbstractType
9  {
10     public function buildForm(FormBuilderInterface $builder, array $options)
11     {
12         $builder->add('name');
13     }
14
15     public function configureOptions(OptionsResolver $resolver)
16     {
17         $resolver->setDefaults(array(
18             'data_class' => 'AppBundle\Entity\Tag',
19         ));
20     }
21 }

```

With this, you have enough to render a tag form by itself. But since the end goal is to allow the tags of a **Task** to be modified right inside the task form itself, create a form for the **Task** class.

Notice that you embed a collection of **TagType** forms using the *CollectionType* field:

Listing 69-4

```

1  // src/AppBundle/Form/Type/TaskType.php
2  namespace AppBundle\Form\Type;
3
4  use Symfony\Component\Form\AbstractType;
5  use Symfony\Component\Form\FormBuilderInterface;
6  use Symfony\Component\OptionsResolver\OptionsResolver;
7  use Symfony\Component\Form\Extension\Core\Type\CollectionType;

```



```

8
9 class TaskType extends AbstractType
10 {
11     public function buildForm(FormBuilderInterface $builder, array $options)
12     {
13         $builder->add('description');
14
15         $builder->add('tags', CollectionType::class, array(
16             'entry_type' => TagType::class
17         ));
18     }
19
20     public function configureOptions(OptionsResolver $resolver)
21     {
22         $resolver->setDefaults(array(
23             'data_class' => 'AppBundle\Entity\Task',
24         ));
25     }
26 }

```

In your controller, you'll create a new form from the **TaskType**:

Listing 69-5

```

1 // src/AppBundle/Controller/TaskController.php
2 namespace AppBundle\Controller;
3
4 use AppBundle\Entity\Task;
5 use AppBundle\Entity\Tag;
6 use AppBundle\Form\Type\TaskType;
7 use Symfony\Component\HttpFoundation\Request;
8 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
9
10 class TaskController extends Controller
11 {
12     public function newAction(Request $request)
13     {
14         $task = new Task();
15
16         // dummy code - this is here just so that the Task has some tags
17         // otherwise, this isn't an interesting example
18         $tag1 = new Tag();
19         $tag1->setName('tag1');
20         $task->getTags()->add($tag1);
21         $tag2 = new Tag();
22         $tag2->setName('tag2');
23         $task->getTags()->add($tag2);
24         // end dummy code
25
26         $form = $this->createForm(TaskType::class, $task);
27
28         $form->handleRequest($request);
29
30         if ($form->isValid()) {
31             // ... maybe do some form processing, like saving the Task and Tag objects
32         }
33
34         return $this->render('AppBundle:Task:new.html.twig', array(
35             'form' => $form->createView(),
36         ));
37     }
38 }

```

The corresponding template is now able to render both the **description** field for the task form as well as all the **TagType** forms for any tags that are already related to this **Task**. In the above controller, I added some dummy code so that you can see this in action (since a **Task** has zero tags when first created).

Listing 69-6

```

1  {# src/AppBundle/Resources/views/Task/new.html.twig #}
2
3  {# ... #}
4
5  {{ form_start(form) }}
6      {# render the task's only field: description #}
7      {{ form_row(form.description) }}
8
9      <h3>Tags</h3>
10     <ul class="tags">
11         {# iterate over each existing tag and render its only field: name #}
12         {% for tag in form.tags %}
13             <li>{{ form_row(tag.name) }}</li>
14         {% endfor %}
15     </ul>
16 {{ form_end(form) }}
17
18 {# ... #}

```

When the user submits the form, the submitted data for the **tags** field are used to construct an **ArrayCollection** of **Tag** objects, which is then set on the **tag** field of the **Task** instance.

The **tags** collection is accessible naturally via `$task->getTags()` and can be persisted to the database or used however you need.

So far, this works great, but this doesn't allow you to dynamically add new tags or delete existing tags. So, while editing existing tags will work great, your user can't actually add any new tags yet.



In this entry, you embed only one collection, but you are not limited to this. You can also embed nested collection as many levels down as you like. But if you use Xdebug in your development setup, you may receive a **Maximum function nesting level of '100' reached, aborting!** error. This is due to the `xdebug.max_nesting_level` PHP setting, which defaults to **100**.

This directive limits recursion to 100 calls which may not be enough for rendering the form in the template if you render the whole form at once (e.g `form_widget(form)`). To fix this you can set this directive to a higher value (either via a `php.ini` file or via `ini_set`¹, for example in `app/autoload.php`) or render each form field by hand using `form_row`.

Allowing "new" Tags with the "Prototype"

Allowing the user to dynamically add new tags means that you'll need to use some JavaScript. Previously you added two tags to your form in the controller. Now let the user add as many tag forms as they need directly in the browser. This will be done through a bit of JavaScript.

The first thing you need to do is to let the form collection know that it will receive an unknown number of tags. So far you've added two tags and the form type expects to receive exactly two, otherwise an error will be thrown: **This form should not contain extra fields**. To make this flexible, add the `allow_add` option to your collection field:

Listing 69-7

```

1  // src/AppBundle/Form/Type/TaskType.php
2
3  // ...
4  use Symfony\Component\Form\FormBuilderInterface;
5
6  public function buildForm(FormBuilderInterface $builder, array $options)
7  {
8      $builder->add('description');
9

```

1. <http://php.net/manual/en/function.ini-set.php>

```

10     $builder->add('tags', CollectionType::class, array(
11         'entry_type' => TagType::class,
12         'allow_add'   => true,
13     ));
14 }

```

In addition to telling the field to accept any number of submitted objects, the **allow_add** also makes a "prototype" variable available to you. This "prototype" is a little "template" that contains all the HTML to be able to render any new "tag" forms. To render it, make the following change to your template:

Listing 69-8

```

1 <ul class="tags" data-prototype="{{ form_widget(form.tags.vars.prototype)|e('html_attr') }}">
2     ...
3 </ul>

```



If you render your whole "tags" sub-form at once (e.g. `form_row(form.tags)`), then the prototype is automatically available on the outer **div** as the **data-prototype** attribute, similar to what you see above.



The `form.tags.vars.prototype` is a form element that looks and feels just like the individual `form_widget(tag)` elements inside your **for** loop. This means that you can call `form_widget`, `form_row` or `form_label` on it. You could even choose to render only one of its fields (e.g. the `name` field):

Listing 69-9

```

1 {{ form_widget(form.tags.vars.prototype.name)|e }}

```

On the rendered page, the result will look something like this:

Listing 69-10

```

1 <ul class="tags" data-prototype="&lt;div&gt;&lt;label class=&quot; required&quot;&gt;__name__&lt;/label&gt;&lt;div id=&quot;task_tags__name__&quot;&gt;&lt;div&gt;&lt;label
for=&quot;task_tags__name__name&quot; class=&quot; required&quot;&gt;Name&lt;/label&gt;&lt;input
type=&quot;text&quot; id=&quot;task_tags__name__name&quot; name=&quot;task[tags][__name__][name]&quot;
required=&quot;required&quot; maxlength=&quot;255&quot; /&gt;&lt;/div&gt;&lt;/div&gt;&lt;/div&gt;">

```

The goal of this section will be to use JavaScript to read this attribute and dynamically add new tag forms when the user clicks a "Add a tag" link. To make things simple, this example uses jQuery and assumes you have it included somewhere on your page.

Add a **script** tag somewhere on your page so you can start writing some JavaScript.

First, add a link to the bottom of the "tags" list via JavaScript. Second, bind to the "click" event of that link so you can add a new tag form (`addTagForm` will be shown next):

Listing 69-11

```

1 var $collectionHolder;
2
3 // setup an "add a tag" link
4 var $addTagLink = $('<a href="#" class="add_tag_link">Add a tag</a>');
5 var $newLinkLi = $('<li></li>').append($addTagLink);
6
7 jQuery(document).ready(function() {
8     // Get the ul that holds the collection of tags
9     $collectionHolder = $('ul.tags');
10
11     // add the "add a tag" anchor and li to the tags ul
12     $collectionHolder.append($newLinkLi);
13
14     // count the current form inputs we have (e.g. 2), use that as the new
15     // index when inserting a new item (e.g. 2)
16     $collectionHolder.data('index', $collectionHolder.find('input').length);
17

```

```

18     $addTagLink.on('click', function(e) {
19         // prevent the link from creating a "#" on the URL
20         e.preventDefault();
21
22         // add a new tag form (see next code block)
23         addTagForm($collectionHolder, $newLinkLi);
24     });
25 });

```

The `addTagForm` function's job will be to use the `data-prototype` attribute to dynamically add a new form when this link is clicked. The `data-prototype` HTML contains the tag `text` input element with a name of `task[tags][__name__][name]` and id of `task_tags__name__name`. The `__name__` is a little "placeholder", which you'll replace with a unique, incrementing number (e.g. `task[tags][3][name]`).

The actual code needed to make this all work can vary quite a bit, but here's one example:

Listing 69-12

```

1 function addTagForm($collectionHolder, $newLinkLi) {
2     // Get the data-prototype explained earlier
3     var prototype = $collectionHolder.data('prototype');
4
5     // get the new index
6     var index = $collectionHolder.data('index');
7
8     // Replace '__name__' in the prototype's HTML to
9     // instead be a number based on how many items we have
10    var newForm = prototype.replace(/__name__/g, index);
11
12    // increase the index with one for the next item
13    $collectionHolder.data('index', index + 1);
14
15    // Display the form in the page in an li, before the "Add a tag" link li
16    var $newFormLi = $('<li></li>').append(newForm);
17    $newLinkLi.before($newFormLi);
18 }

```



It is better to separate your JavaScript in real JavaScript files than to write it inside the HTML as is done here.

Now, each time a user clicks the **Add a tag** link, a new sub form will appear on the page. When the form is submitted, any new tag forms will be converted into new **Tag** objects and added to the `tags` property of the **Task** object.

You can find a working example in this [JSFiddle](http://jsfiddle.net/847Kf/4/)².

If you want to customize the HTML code in the prototype, read *How to Customize a Collection Prototype*.

To make handling these new tags easier, add an "adder" and a "remover" method for the tags in the **Task** class:

Listing 69-13

```

1 // src/AppBundle/Entity/Task.php
2 namespace AppBundle\Entity;
3
4 // ...
5 class Task
6 {
7     // ...
8 }

```

2. <http://jsfiddle.net/847Kf/4/>

```

9     public function addTag(Tag $tag)
10    {
11        $this->tags->add($tag);
12    }
13
14    public function removeTag(Tag $tag)
15    {
16        // ...
17    }
18 }

```

Next, add a **by_reference** option to the **tags** field and set it to **false**:

Listing 69-14

```

1  // src/AppBundle/Form/Type/TaskType.php
2
3  // ...
4  public function buildForm(FormBuilderInterface $builder, array $options)
5  {
6      // ...
7
8      $builder->add('tags', CollectionType::class, array(
9          // ...
10         'by_reference' => false,
11     ));
12 }

```

With these two changes, when the form is submitted, each new **Tag** object is added to the **Task** class by calling the **addTag** method. Before this change, they were added internally by the form by calling **\$task->getTags()->add(\$tag)**. That was just fine, but forcing the use of the "adder" method makes handling these new **Tag** objects easier (especially if you're using Doctrine, which you will learn about next!).



You have to create **both** **addTag** and **removeTag** methods, otherwise the form will still use **setTag** even if **by_reference** is **false**. You'll learn more about the **removeTag** method later in this article.



Doctrine: Cascading Relations and saving the "Inverse" side

To save the new tags with Doctrine, you need to consider a couple more things. First, unless you iterate over all of the new **Tag** objects and call `$em->persist($tag)` on each, you'll receive an error from Doctrine:

A new entity was found through the relationship AppBundle\Entity\Task#tags that was not configured to cascade persist operations for entity...

To fix this, you may choose to "cascade" the persist operation automatically from the **Task** object to any related tags. To do this, add the **cascade** option to your **ManyToOne** metadata:

Listing 69-15

```
1 // src/AppBundle/Entity/Task.php
2
3 // ...
4
5 /**
6  * @ORM\ManyToOne(targetEntity="Tag", cascade={"persist"})
7  */
8 protected $tags;
```

A second potential issue deals with the *Owning Side and Inverse Side*³ of Doctrine relationships. In this example, if the "owning" side of the relationship is "Task", then persistence will work fine as the tags are properly added to the Task. However, if the owning side is on "Tag", then you'll need to do a little bit more work to ensure that the correct side of the relationship is modified.

The trick is to make sure that the single "Task" is set on each "Tag". One easy way to do this is to add some extra logic to `addTag()`, which is called by the form type since `by_reference` is set to `false`:

Listing 69-16

```
1 // src/AppBundle/Entity/Task.php
2
3 // ...
4 public function addTag(Tag $tag)
5 {
6     $tag->addTask($this);
7
8     $this->tags->add($tag);
9 }
```

Inside **Tag**, just make sure you have an `addTask` method:

Listing 69-17

```
1 // src/AppBundle/Entity/Tag.php
2
3 // ...
4 public function addTask(Task $task)
5 {
6     if (!$this->tasks->contains($task)) {
7         $this->tasks->add($task);
8     }
9 }
```

If you have a one-to-many relationship, then the workaround is similar, except that you can simply call `setTask` from inside `addTag`.

3. <http://docs.doctrine-project.org/en/latest/reference/unitofwork-associations.html>

Allowing Tags to be Removed

The next step is to allow the deletion of a particular item in the collection. The solution is similar to allowing tags to be added.

Start by adding the `allow_delete` option in the form Type:

```
Listing 69-18 1 // src/AppBundle/Form/Type/TaskType.php
2
3 // ...
4 public function buildForm(FormBuilderInterface $builder, array $options)
5 {
6     // ...
7
8     $builder->add('tags', CollectionType::class, array(
9         // ...
10        'allow_delete' => true,
11    ));
12 }
```

Now, you need to put some code into the `removeTag` method of `Task`:

```
Listing 69-19 1 // src/AppBundle/Entity/Task.php
2
3 // ...
4 class Task
5 {
6     // ...
7
8     public function removeTag(Tag $tag)
9     {
10        $this->tags->removeElement($tag);
11    }
12 }
```

Template Modifications

The `allow_delete` option has one consequence: if an item of a collection isn't sent on submission, the related data is removed from the collection on the server. The solution is thus to remove the form element from the DOM.

First, add a "delete this tag" link to each tag form:

```
Listing 69-20 1 jQuery(document).ready(function() {
2     // Get the ul that holds the collection of tags
3     $collectionHolder = $('ul.tags');
4
5     // add a delete link to all of the existing tag form li elements
6     $collectionHolder.find('li').each(function() {
7         addTagFormDeleteLink($(this));
8     });
9
10    // ... the rest of the block from above
11 });
12
13 function addTagForm() {
14     // ...
15
16     // add a delete link to the new form
17     addTagFormDeleteLink($newFormLi);
18 }
```

The `addTagFormDeleteLink` function will look something like this:

Listing 69-21

```

1 function addTagFormDeleteLink($tagFormLi) {
2     var $removeFormA = $('<a href="#">delete this tag</a>');
3     $tagFormLi.append($removeFormA);
4
5     $removeFormA.on('click', function(e) {
6         // prevent the link from creating a "#" on the URL
7         e.preventDefault();
8
9         // remove the li for the tag form
10        $tagFormLi.remove();
11    });
12 }

```

When a tag form is removed from the DOM and submitted, the removed **Tag** object will not be included in the collection passed to **setTags**. Depending on your persistence layer, this may or may not be enough to actually remove the relationship between the removed **Tag** and **Task** object.



Doctrine: Ensuring the database persistence

When removing objects in this way, you may need to do a little bit more work to ensure that the relationship between the **Task** and the removed **Tag** is properly removed.

In Doctrine, you have two sides of the relationship: the owning side and the inverse side. Normally in this case you'll have a many-to-many relationship and the deleted tags will disappear and persist correctly (adding new tags also works effortlessly).

But if you have a one-to-many relationship or a many-to-many relationship with a **mappedBy** on the Task entity (meaning Task is the "inverse" side), you'll need to do more work for the removed tags to persist correctly.

In this case, you can modify the controller to remove the relationship on the removed tag. This assumes that you have some **editAction** which is handling the "update" of your Task:

```
Listing 69-22 1 // src/AppBundle/Controller/TaskController.php
2
3 use Doctrine\Common\Collections\ArrayCollection;
4
5 // ...
6 public function editAction($id, Request $request)
7 {
8     $em = $this->getDoctrine()->getManager();
9     $task = $em->getRepository('AppBundle:Task')->find($id);
10
11     if (!$task) {
12         throw $this->createNotFoundException('No task found for id '.$id);
13     }
14
15     $originalTags = new ArrayCollection();
16
17     // Create an ArrayCollection of the current Tag objects in the database
18     foreach ($task->getTags() as $tag) {
19         $originalTags->add($tag);
20     }
21
22     $editForm = $this->createForm(TaskType::class, $task);
23
24     $editForm->handleRequest($request);
25
26     if ($editForm->isValid()) {
27
28         // remove the relationship between the tag and the Task
29         foreach ($originalTags as $tag) {
30             if (false === $task->getTags()->contains($tag)) {
31                 // remove the Task from the Tag
32                 $tag->getTasks()->removeElement($task);
33
34                 // if it was a many-to-one relationship, remove the relationship like this
35                 // $tag->setTask(null);
36
37                 $em->persist($tag);
38
39                 // if you wanted to delete the Tag entirely, you can also do that
40                 // $em->remove($tag);
41             }
42         }
43
44         $em->persist($task);
45         $em->flush();
46
47         // redirect back to some edit page
48         return $this->redirectToRoute('task_edit', array('id' => $id));
49     }
50
51     // render some form template
52 }
```

As you can see, adding and removing the elements correctly can be tricky. Unless you have a many-to-many relationship where Task is the "owning" side, you'll need to do extra work to make sure that

the relationship is properly updated (whether you're adding new tags or removing existing tags) on each Tag object itself.



Chapter 70

How to Create a Custom Form Field Type

Symfony comes with a bunch of core field types available for building forms. However there are situations where you may want to create a custom form field type for a specific purpose. This recipe assumes you need a field definition that holds a person's gender, based on the existing choice field. This section explains how the field is defined, how you can customize its layout and finally, how you can register it for use in your application.

Defining the Field Type

In order to create the custom field type, first you have to create the class representing the field. In this situation the class holding the field type will be called **GenderType** and the file will be stored in the default location for form fields, which is `<BundleName>\Form\Type`. Make sure the field extends *AbstractType*¹:

Listing 70-1

```
1  // src/AppBundle/Form/Type/GenderType.php
2  namespace AppBundle\Form\Type;
3
4  use Symfony\Component\Form\AbstractType;
5  use Symfony\Component\OptionsResolver\OptionsResolver;
6  use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
7
8  class GenderType extends AbstractType
9  {
10     public function configureOptions(OptionsResolver $resolver)
11     {
12         $resolver->setDefaults(array(
13             'choices' => array(
14                 'm' => 'Male',
15                 'f' => 'Female',
16             )
17         ));
18     }
19
20     public function getParent()
21     {
22         return ChoiceType::class;
```

1. <http://api.symfony.com/3.0/Symfony/Component/Form/AbstractType.html>

```
23     }
24 }
```



The location of this file is not important - the `Form\Type` directory is just a convention.

Here, the return value of the `getParent` function indicates that you're extending the `ChoiceType` field. This means that, by default, you inherit all of the logic and rendering of that field type. To see some of the logic, check out the `ChoiceType`² class. There are three methods that are particularly important:

`buildForm()`

Each field type has a `buildForm` method, which is where you configure and build any field(s). Notice that this is the same method you use to setup *your* forms, and it works the same here.

`buildView()`

This method is used to set any extra variables you'll need when rendering your field in a template. For example, in `ChoiceType`³, a `multiple` variable is set and used in the template to set (or not set) the `multiple` attribute on the `select` field. See [Creating a Template for the Field](#) for more details.

`configureOptions()`

This defines options for your form type that can be used in `buildForm()` and `buildView()`. There are a lot of options common to all fields (see *FormType Field*), but you can create any others that you need here.



If you're creating a field that consists of many fields, then be sure to set your "parent" type as `form` or something that extends `form`. Also, if you need to modify the "view" of any of your child types from your parent type, use the `finishView()` method.

The goal of this field was to extend the choice type to enable selection of a gender. This is achieved by fixing the `choices` to a list of possible genders.

Creating a Template for the Field

Each field type is rendered by a template fragment, which is determined in part by the class name of your type. For more information, see [What are Form Themes?](#)



The first part of the prefix (e.g. `gender`) comes from the class name (`GenderType` -> `gender`). This can be controlled by overriding `getBlockPrefix()` in `GenderType`.



When the name of your form class matches any of the built-in field types, your form might not be rendered correctly. A form type named `AppBundle\Form>PasswordType` will have the same block name as the built-in `PasswordType` and won't be rendered correctly. Override the `getBlockPrefix()` method to return a unique block prefix (e.g. `app_password`) to avoid collisions.

In this case, since the parent field is `ChoiceType`, you don't *need* to do any work as the custom field type will automatically be rendered like a `ChoiceType`. But for the sake of this example, suppose that

2. <https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Form/Extension/Core/Type/ChoiceType.php>

3. <https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Form/Extension/Core/Type/ChoiceType.php>

when your field is "expanded" (i.e. radio buttons or checkboxes, instead of a select field), you want to always render it in a `ul` element. In your form theme template (see above link for details), create a `gender_widget` block to handle this:

Listing 70-2

```

1  {# app/Resources/views/form/fields.html.twig #}
2  {% block gender_widget %}
3      {% spaceless %}
4          {% if expanded %}
5              <ul {{ block('widget_container_attributes') }}>
6                  {% for child in form %}
7                      <li>
8                          {{ form_widget(child) }}
9                          {{ form_label(child) }}
10                     </li>
11                 {% endfor %}
12             </ul>
13         {% else %}
14             {# just let the choice widget render the select tag #}
15             {{ block('choice_widget') }}
16         {% endif %}
17     {% endspaceless %}
18 {% endblock %}

```



Make sure the correct widget prefix is used. In this example the name should be `gender_widget` (see What are Form Themes?). Further, the main config file should point to the custom form template so that it's used when rendering all forms.

When using Twig this is:

Listing 70-3

```

1  # app/config/config.yml
2  twig:
3      form_themes:
4          - 'form/fields.html.twig'

```

For the PHP templating engine, your configuration should look like this:

Listing 70-4

```

1  # app/config/config.yml
2  framework:
3      templating:
4          form:
5              resources:
6                  - ':form:fields.html.php'

```

Using the Field Type

You can now use your custom field type immediately, simply by creating a new instance of the type in one of your forms:

Listing 70-5

```

1  // src/AppBundle/Form/Type/AuthorType.php
2  namespace AppBundle\Form\Type;
3
4  use Symfony\Component\Form\AbstractType;
5  use Symfony\Component\Form\FormBuilderInterface;
6  use AppBundle\Form\Type\GenderType;
7
8  class AuthorType extends AbstractType
9  {
10     public function buildForm(FormBuilderInterface $builder, array $options)
11     {
12         $builder->add('gender_code', GenderType::class, array(
13             'placeholder' => 'Choose a gender',
14         ));
15     }
16 }

```

```

15     }
16 }

```

But this only works because the **GenderType** is very simple. What if the gender codes were stored in configuration or in a database? The next section explains how more complex field types solve this problem.

Creating your Field Type as a Service

So far, this entry has assumed that you have a very simple custom field type. But if you need access to configuration, a database connection, or some other service, then you'll want to register your custom type as a service. For example, suppose that you're storing the gender parameters in configuration:

Listing 70-6

```

1 # app/config/config.yml
2 parameters:
3     genders:
4         m: Male
5         f: Female

```

To use the parameter, define your custom field type as a service, injecting the **genders** parameter value as the first argument to its to-be-created **__construct** function:

Listing 70-7

```

1 # src/AppBundle/Resources/config/services.yml
2 services:
3     app.form.type.gender:
4         class: AppBundle\Form\Type\GenderType
5         arguments:
6             - '%genders%'
7         tags:
8             - { name: form.type }

```



Make sure the services file is being imported. See Importing Configuration with imports for details.

First, add a **__construct** method to **GenderType**, which receives the gender configuration:

Listing 70-8

```

1 // src/AppBundle/Form/Type/GenderType.php
2 namespace AppBundle\Form\Type;
3
4 use Symfony\Component\OptionsResolver\OptionsResolver;
5
6 // ...
7
8 // ...
9 class GenderType extends AbstractType
10 {
11     private $genderChoices;
12
13     public function __construct(array $genderChoices)
14     {
15         $this->genderChoices = $genderChoices;
16     }
17
18     public function configureOptions(OptionsResolver $resolver)
19     {
20         $resolver->setDefaults(array(
21             'choices' => $this->genderChoices,
22         ));
23     }
24 }

```

```

24
25     // ...
26 }

```

Great! The **GenderType** is now fueled by the configuration parameters and registered as a service. Because you used the **form.type** alias in its configuration, your service will be used instead of creating a new **GenderType**. In other words, your controller *does not need to change*, it still looks like this:

Listing 70-9

```

1  // src/AppBundle/Form/Type/AuthorType.php
2  namespace AppBundle\Form\Type;
3
4  use Symfony\Component\Form\AbstractType;
5  use Symfony\Component\Form\FormBuilderInterface;
6  use AppBundle\Form\Type\GenderType;
7
8  class AuthorType extends AbstractType
9  {
10     public function buildForm(FormBuilderInterface $builder, array $options)
11     {
12         $builder->add('gender_code', GenderType::class, array(
13             'placeholder' => 'Choose a gender',
14         ));
15     }
16 }

```

Have fun!



Chapter 71

How to Create a Form Type Extension

Custom form field types are great when you need field types with a specific purpose, such as a gender selector, or a VAT number input.

But sometimes, you don't really need to add new field types - you want to add features on top of existing types. This is where form type extensions come in.

Form type extensions have 2 main use-cases:

1. You want to add a **specific feature to a single type** (such as adding a "download" feature to the `FileType` field type);
2. You want to add a **generic feature to several types** (such as adding a "help" text to every "input text"-like type).

It might be possible to achieve your goal with custom form rendering or custom form field types. But using form type extensions can be cleaner (by limiting the amount of business logic in templates) and more flexible (you can add several type extensions to a single form type).

Form type extensions can achieve most of what custom field types can do, but instead of being field types of their own, **they plug into existing types**.

Imagine that you manage a **Media** entity, and that each media is associated to a file. Your **Media** form uses a file type, but when editing the entity, you would like to see its image automatically rendered next to the file input.

You could of course do this by customizing how this field is rendered in a template. But field type extensions allow you to do this in a nice DRY fashion.

Defining the Form Type Extension

Your first task will be to create the form type extension class (called **ImageTypeExtension** in this article). By standard, form extensions usually live in the **Form\Extension** directory of one of your bundles.

When creating a form type extension, you can either implement the *FormTypeExtensionInterface*¹ interface or extend the *AbstractTypeExtension*² class. In most cases, it's easier to extend the abstract class:

Listing 71-1

```
1 // src/AppBundle/Form/Extension/ImageTypeExtension.php
2 namespace AppBundle\Form\Extension;
3
4 use Symfony\Component\Form\AbstractTypeExtension;
5 use Symfony\Component\Form\Extension\Core\Type\FileType;
6
7 class ImageTypeExtension extends AbstractTypeExtension
8 {
9     /**
10      * Returns the name of the type being extended.
11      *
12      * @return string The name of the type being extended
13      */
14     public function getExtendedType()
15     {
16         return FileType::class;
17     }
18 }
```

The only method you **must** implement is the `getExtendedType` function. It is used to indicate the name of the form type that will be extended by your extension.



The value you return in the `getExtendedType` method corresponds to the fully qualified class name of the form type class you wish to extend.

In addition to the `getExtendedType` function, you will probably want to override one of the following methods:

- `buildForm()`
- `buildView()`
- `configureOptions()`
- `finishView()`

For more information on what those methods do, you can refer to the *Creating Custom Field Types* cookbook article.

Registering your Form Type Extension as a Service

The next step is to make Symfony aware of your extension. All you need to do is to declare it as a service by using the `form.type_extension` tag:

Listing 71-2

```
1 services:
2     app.image_type_extension:
3         class: AppBundle\Form\Extension\ImageTypeExtension
4         tags:
5             - { name: form.type_extension, extended_type: Symfony\Component\Form\Extension\Core\Type\FileType }
```

The `extended_type` key of the tag is the type of field that this extension should be applied to. In your case, as you want to extend the `Symfony\Component\Form\Extension\Core\Type\FileType` field type, you will use that as the `extended_type`.

1. <http://api.symfony.com/3.0/Symfony/Component/Form/FormTypeExtensionInterface.html>
2. <http://api.symfony.com/3.0/Symfony/Component/Form/AbstractTypeExtension.html>

Adding the extension Business Logic

The goal of your extension is to display nice images next to file inputs (when the underlying model contains images). For that purpose, suppose that you use an approach similar to the one described in *How to handle File Uploads with Doctrine*: you have a Media model with a path property, corresponding to the image path in the database:

```
Listing 71-3 1 // src/AppBundle/Entity/Media.php
2 namespace AppBundle\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class Media
7 {
8     // ...
9
10    /**
11     * @var string The path - typically stored in the database
12     */
13    private $path;
14
15    // ...
16
17    /**
18     * Get the image URL
19     *
20     * @return null|string
21     */
22    public function getWebPath()
23    {
24        // ... $webPath being the full image URL, to be used in templates
25
26        return $webPath;
27    }
28 }
```

Your form type extension class will need to do two things in order to extend the `FileType::class` form type:

1. Override the `configureOptions` method in order to add an `image_path` option;
2. Override the `buildView` methods in order to pass the image URL to the view.

The logic is the following: when adding a form field of type `FileType::class`, you will be able to specify a new option: `image_path`. This option will tell the file field how to get the actual image URL in order to display it in the view:

```
Listing 71-4 1 // src/AppBundle/Form/Extension/ImageTypeExtension.php
2 namespace AppBundle\Form\Extension;
3
4 use Symfony\Component\Form\AbstractTypeExtension;
5 use Symfony\Component\Form\FormView;
6 use Symfony\Component\Form\FormInterface;
7 use Symfony\Component\PropertyAccess\PropertyAccess;
8 use Symfony\Component\OptionsResolver\OptionsResolver;
9 use Symfony\Component\Form\Extension\Core\Type\FileType;
10
11 class ImageTypeExtension extends AbstractTypeExtension
12 {
13     /**
14      * Returns the name of the type being extended.
15      *
16      * @return string The name of the type being extended
17      */
18     public function getExtendedType()
19     {
20         return FileType::class;
21     }
22 }
```

```

22
23  /**
24   * Add the image_path option
25   *
26   * @param OptionsResolver $resolver
27   */
28  public function configureOptions(OptionsResolver $resolver)
29  {
30      $resolver->setDefined(array('image_path'));
31  }
32
33  /**
34   * Pass the image URL to the view
35   *
36   * @param FormView $view
37   * @param FormInterface $form
38   * @param array $options
39   */
40  public function buildView(FormView $view, FormInterface $form, array $options)
41  {
42      if (isset($options['image_path'])) {
43          $parentData = $form->getParent()->getData();
44
45          $imageUrl = null;
46          if (null !== $parentData) {
47              $accessor = PropertyAccess::createPropertyAccessor();
48              $imageUrl = $accessor->getValue($parentData, $options['image_path']);
49          }
50
51          // set an "image_url" variable that will be available when rendering this field
52          $view->vars['image_url'] = $imageUrl;
53      }
54  }
55
56  }

```

Override the File Widget Template Fragment

Each field type is rendered by a template fragment. Those template fragments can be overridden in order to customize form rendering. For more information, you can refer to the [What are Form Themes?](#) article.

In your extension class, you have added a new variable (`image_url`), but you still need to take advantage of this new variable in your templates. Specifically, you need to override the `file_widget` block:

Listing 71-5

```

1  {# src/AppBundle/Resources/views/Form/fields.html.twig #}
2  {% extends 'form_div_layout.html.twig' %}
3
4  {% block file_widget %}
5      {% spaceless %}
6
7          {{ block('form_widget') }}
8          {% if image_url is not null %}
9              
10             {% endif %}
11
12         {% endspaceless %}
13     {% endblock %}

```



You will need to change your config file or explicitly specify how you want your form to be themed in order for Symfony to use your overridden block. See [What are Form Themes?](#) for more information.

Using the Form Type Extension

From now on, when adding a field of type `FileType::class` in your form, you can specify an `image_path` option that will be used to display an image next to the file field. For example:

Listing 71-6

```
1  // src/AppBundle/Form/Type/MediaType.php
2  namespace AppBundle\Form\Type;
3
4  use Symfony\Component\Form\AbstractType;
5  use Symfony\Component\Form\FormBuilderInterface;
6  use Symfony\Component\Form\Extension\Core\Type\TextType;
7  use Symfony\Component\Form\Extension\Core\Type\FileType;
8
9  class MediaType extends AbstractType
10 {
11     public function buildForm(FormBuilderInterface $builder, array $options)
12     {
13         $builder
14             ->add('name', TextType::class)
15             ->add('file', FileType::class, array('image_path' => 'webPath'));
16     }
17 }
```

When displaying the form, if the underlying model has already been associated with an image, you will see it displayed next to the file input.

Generic Form Type Extensions

You can modify several form types at once by specifying their common parent (*Form Types Reference*). For example, several form types natively available in Symfony inherit from the `TextType` form type (such as `EmailType`, `SearchType`, `UrlType`, etc.). A form type extension applying to `TextType` (i.e. whose `getExtendedType` method returns `TextType::class`) would apply to all of these form types.

In the same way, since **most** form types natively available in Symfony inherit from the `FormType` form type, a form type extension applying to `FormType` would apply to all of these. A notable exception are the `ButtonType` form types. Also keep in mind that a custom form type which extends neither the `FormType` nor the `ButtonType` type could always be created.



Chapter 72

How to Reduce Code Duplication with "inherit_data"

The `inherit_data` form field option can be very useful when you have some duplicated fields in different entities. For example, imagine you have two entities, a **Company** and a **Customer**:

Listing 72-1

```
1 // src/AppBundle/Entity/Company.php
2 namespace AppBundle\Entity;
3
4 class Company
5 {
6     private $name;
7     private $website;
8
9     private $address;
10    private $zipcode;
11    private $city;
12    private $country;
13 }
```

Listing 72-2

```
1 // src/AppBundle/Entity/Customer.php
2 namespace AppBundle\Entity;
3
4 class Customer
5 {
6     private $firstName;
7     private $lastName;
8
9     private $address;
10    private $zipcode;
11    private $city;
12    private $country;
13 }
```

As you can see, each entity shares a few of the same fields: **address**, **zipcode**, **city**, **country**.

Start with building two forms for these entities, **CompanyType** and **CustomerType**:

Listing 72-3

```

1  // src/AppBundle/Form/Type/CompanyType.php
2  namespace AppBundle\Form\Type;
3
4  use Symfony\Component\Form\AbstractType;
5  use Symfony\Component\Form\FormBuilderInterface;
6  use Symfony\Component\Form\Extension\Core\Type\TextType;
7
8  class CompanyType extends AbstractType
9  {
10     public function buildForm(FormBuilderInterface $builder, array $options)
11     {
12         $builder
13             ->add('name', TextType::class)
14             ->add('website', TextType::class);
15     }
16 }

```

Listing 72-4

```

1  // src/AppBundle/Form/Type/CustomerType.php
2  namespace AppBundle\Form\Type;
3
4  use Symfony\Component\Form\FormBuilderInterface;
5  use Symfony\Component\Form\AbstractType;
6  use Symfony\Component\Form\Extension\Core\Type\TextType;
7
8  class CustomerType extends AbstractType
9  {
10     public function buildForm(FormBuilderInterface $builder, array $options)
11     {
12         $builder
13             ->add('firstName', TextType::class)
14             ->add('lastName', TextType::class);
15     }
16 }

```

Instead of including the duplicated fields **address**, **zipcode**, **city** and **country** in both of these forms, create a third form called **LocationType** for that:

Listing 72-5

```

1  // src/AppBundle/Form/Type/LocationType.php
2  namespace AppBundle\Form\Type;
3
4  use Symfony\Component\Form\AbstractType;
5  use Symfony\Component\Form\FormBuilderInterface;
6  use Symfony\Component\OptionsResolver\OptionsResolver;
7  use Symfony\Component\Form\Extension\Core\Type\TextareaType;
8  use Symfony\Component\Form\Extension\Core\Type\TextType;
9
10 class LocationType extends AbstractType
11 {
12     public function buildForm(FormBuilderInterface $builder, array $options)
13     {
14         $builder
15             ->add('address', TextareaType::class)
16             ->add('zipcode', TextType::class)
17             ->add('city', TextType::class)
18             ->add('country', TextType::class);
19     }
20
21     public function configureOptions(OptionsResolver $resolver)
22     {
23         $resolver->setDefaults(array(
24             'inherit_data' => true
25         ));
26     }
27 }

```

The location form has an interesting option set, namely **inherit_data**. This option lets the form inherit its data from its parent form. If embedded in the company form, the fields of the location form

will access the properties of the **Company** instance. If embedded in the customer form, the fields will access the properties of the **Customer** instance instead. Easy, eh?



Instead of setting the `inherit_data` option inside `LocationType`, you can also (just like with any option) pass it in the third argument of `$builder->add()`.

Finally, make this work by adding the location form to your two original forms:

Listing 72-6

```
1 // src/AppBundle/Form/Type/CompanyType.php
2 public function buildForm(FormBuilderInterface $builder, array $options)
3 {
4     // ...
5
6     $builder->add('foo', LocationType::class, array(
7         'data_class' => 'AppBundle\Entity\Company'
8     ));
9 }
```

Listing 72-7

```
1 // src/AppBundle/Form/Type/CustomerType.php
2 public function buildForm(FormBuilderInterface $builder, array $options)
3 {
4     // ...
5
6     $builder->add('bar', LocationType::class, array(
7         'data_class' => 'AppBundle\Entity\Customer'
8     ));
9 }
```

That's it! You have extracted duplicated field definitions to a separate location form that you can reuse wherever you need it.



Forms with the `inherit_data` option set cannot have `*_SET_DATA` event listeners.



Chapter 73

How to Unit Test your Forms

The Form component consists of 3 core objects: a form type (implementing *FormTypeInterface*¹), the *Form*² and the *FormView*³.

The only class that is usually manipulated by programmers is the form type class which serves as a form blueprint. It is used to generate the **Form** and the **FormView**. You could test it directly by mocking its interactions with the factory but it would be complex. It is better to pass it to **FormFactory** like it is done in a real application. It is simple to bootstrap and you can trust the Symfony components enough to use them as a testing base.

There is already a class that you can benefit from for simple FormTypes testing: *TypeTestCase*⁴. It is used to test the core types and you can use it to test your types too.



Depending on the way you installed your Symfony or Symfony Form component the tests may not be downloaded. Use the **--prefer-source** option with Composer if this is the case.

The Basics

The simplest **TypeTestCase** implementation looks like the following:

Listing 73-1

```
1 // tests/AppBundle/Form/Type/TestedTypeTest.php
2 namespace Tests\AppBundle\Form\Type;
3
4 use AppBundle\Form\Type\TestedType;
5 use AppBundle\Model\TestObject;
6 use Symfony\Component\Form\Test\TypeTestCase;
7
8 class TestedTypeTest extends TypeTestCase
9 {
10     public function testSubmitValidData()
```

-
1. <http://api.symfony.com/3.0/Symfony/Component/Form/FormTypeInterface.html>
 2. <http://api.symfony.com/3.0/Symfony/Component/Form/Form.html>
 3. <http://api.symfony.com/3.0/Symfony/Component/Form/FormView.html>
 4. <http://api.symfony.com/3.0/Symfony/Component/Form/Test/TypeTestCase.html>


```

11     {
12         $formData = array(
13             'test' => 'test',
14             'test2' => 'test2',
15         );
16
17         $form = $this->factory->create(TestedType::class);
18
19         $object = TestObject::fromArray($formData);
20
21         // submit the data to the form directly
22         $form->submit($formData);
23
24         $this->assertTrue($form->isSynchronized());
25         $this->assertEquals($object, $form->getData());
26
27         $view = $form->createView();
28         $children = $view->children;
29
30         foreach (array_keys($formData) as $key) {
31             $this->assertArrayHasKey($key, $children);
32         }
33     }
34 }

```

So, what does it test? Here comes a detailed explanation.

First you verify if the **FormType** compiles. This includes basic class inheritance, the **buildForm** function and options resolution. This should be the first test you write:

Listing 73-2 `$form = $this->factory->create(TestedType::class);`

This test checks that none of your data transformers used by the form failed. The *isSynchronized()*⁵ method is only set to **false** if a data transformer throws an exception:

Listing 73-3 `$form->submit($formData);`
`$this->assertTrue($form->isSynchronized());`



Don't test the validation: it is applied by a listener that is not active in the test case and it relies on validation configuration. Instead, unit test your custom constraints directly.

Next, verify the submission and mapping of the form. The test below checks if all the fields are correctly specified:

Listing 73-4 `$this->assertEquals($object, $form->getData());`

Finally, check the creation of the **FormView**. You should check if all widgets you want to display are available in the children property:

Listing 73-5

```

1 $view = $form->createView();
2 $children = $view->children;
3
4 foreach (array_keys($formData) as $key) {
5     $this->assertArrayHasKey($key, $children);
6 }

```

5. http://api.symfony.com/3.0/Symfony/Component/Form/FormInterface.html#method_isSynchronized

Testings Types from the Service Container

Your form may be used as a service, as it depends on other services (e.g. the Doctrine entity manager). In these cases, using the above code won't work, as the Form component just instantiates the form type without passing any arguments to the constructor.

To solve this, you have to mock the injected dependencies, instantiate your own form type and use the *PreloadedExtension*⁶ to make sure the **FormRegistry** uses the created instance:

```
Listing 73-6 1 // tests/AppBundle/Form/Type/TestedTypeTests.php
2 namespace Tests\AppBundle\Form\Type;
3
4 use Symfony\Component\Form\PreloadedExtension;
5 // ...
6
7 class TestedTypeTest extends TypeTestCase
8 {
9     private $entityManager;
10
11     protected function setUp()
12     {
13         // mock any dependencies
14         $this->entityManager = $this->getMock('Doctrine\Common\Persistence\ObjectManager');
15
16         parent::setUp();
17     }
18
19     protected function getExtensions()
20     {
21         // create a type instance with the mocked dependencies
22         $type = new TestedType($this->entityManager);
23
24         return array(
25             // register the type instances with the PreloadedExtension
26             new PreloadedExtension(array($type), array()),
27         );
28     }
29
30     public function testSubmitValidData()
31     {
32         // Instead of creating a new instance, the one created in
33         // getExtensions() will be used.
34         $form = $this->factory->create(TestedType::class);
35
36         // ... your test
37     }
38 }
```

Adding Custom Extensions

It often happens that you use some options that are added by *form extensions*. One of the cases may be the **ValidatorExtension** with its `invalid_message` option. The **TypeTestCase** only loads the core form extension, which means an *InvalidOptionsException*⁷ will be raised if you try to test a class that depends on other extensions. The *getExtensions()*⁸ method allows you to return a list of extensions to register:

```
Listing 73-7 1 // tests/AppBundle/Form/Type/TestedTypeTests.php
2 namespace Tests\AppBundle\Form\Type;
```

6. <http://api.symfony.com/3.0/Symfony/Component/Form/PreloadedExtension.html>

7. <http://api.symfony.com/3.0/Symfony/Component/OptionsResolver/Exception/InvalidOptionsException.html>

8. http://api.symfony.com/3.0/Symfony/Component/Form/Test/TypeTestCase.html#method_getExtensions

```

3
4 // ...
5 use Symfony\Component\Form\Extension\Validator\ValidatorExtension;
6 use Symfony\Component\Validator\ConstraintViolationList;
7
8 class TestedTypeTest extends TypeTestCase
9 {
10     private $validator;
11
12     protected function getExtensions()
13     {
14         $this->validator = $this->getMock(
15             'Symfony\Component\Validator\Validator\ValidatorInterface'
16         );
17         $this->validator
18             ->method('validate')
19             ->will($this->returnValue(new ConstraintViolationList()));
20
21         return array(
22             new ValidatorExtension($this->validator),
23         );
24     }
25
26     // ... your tests
27 }

```

Testing against Different Sets of Data

If you are not familiar yet with PHPUnit's *data providers*⁹, this might be a good opportunity to use them:

Listing 73-8

```

1 // tests/AppBundle/Form/Type/TestedTypeTests.php
2 namespace Tests\AppBundle\Form\Type;
3
4 use AppBundle\Form\Type\TestedType;
5 use AppBundle\Model\TestObject;
6 use Symfony\Component\Form\Test\TypeTestCase;
7
8 class TestedTypeTest extends TypeTestCase
9 {
10     /**
11      * @dataProvider getValidTestData
12      */
13     public function testForm($data)
14     {
15         // ... your test
16     }
17
18     public function getValidTestData()
19     {
20         return array(
21             array(
22                 'data' => array(
23                     'test' => 'test',
24                     'test2' => 'test2',
25                 ),
26             ),
27             array(
28                 'data' => array(),
29             ),
30             array(
31                 'data' => array(
32                     'test' => null,
33                     'test2' => null,
34                 ),
35             ),
36         );
37     }
38 }

```

9. <https://phpunit.de/manual/current/en/writing-tests-for-phpunit.html#writing-tests-for-phpunit.data-providers>

```
35         ),
36     );
37 }
38 }
```

The code above will run your test three times with 3 different sets of data. This allows for decoupling the test fixtures from the tests and easily testing against multiple sets of data.

You can also pass another argument, such as a boolean if the form has to be synchronized with the given set of data or not etc.



Chapter 74

How to Configure empty Data for a Form Class

The `empty_data` option allows you to specify an empty data set for your form class. This empty data set would be used if you submit your form, but haven't called `setData()` on your form or passed in data when you created your form. For example:

Listing 74-1

```
1 public function indexAction()
2 {
3     $blog = ...;
4
5     // $blog is passed in as the data, so the empty_data
6     // option is not needed
7     $form = $this->createForm(BlogType::class, $blog);
8
9     // no data is passed in, so empty_data is
10    // used to get the "starting data"
11    $form = $this->createForm(BlogType::class);
12 }
```

By default, `empty_data` is set to `null`. Or, if you have specified a `data_class` option for your form class, it will default to a new instance of that class. That instance will be created by calling the constructor with no arguments.

If you want to override this default behavior, there are two ways to do this.

Option 1: Instantiate a new Class

One reason you might use this option is if you want to use a constructor that takes arguments. Remember, the default `data_class` option calls that constructor with no arguments:

Listing 74-2

```
1 // src/AppBundle/Form/Type/BlogType.php
2
3 // ...
4 use Symfony\Component\Form\AbstractType;
5 use AppBundle\Entity\Blog;
6 use Symfony\Component\OptionsResolver\OptionsResolver;
7
8 class BlogType extends AbstractType
9 {
```

```

10     private $someDependency;
11
12     public function __construct($someDependency)
13     {
14         $this->someDependency = $someDependency;
15     }
16     // ...
17
18     public function configureOptions(OptionsResolver $resolver)
19     {
20         $resolver->setDefaults(array(
21             'empty_data' => new Blog($this->someDependency),
22         ));
23     }
24 }

```

You can instantiate your class however you want. In this example, you pass some dependency into the **BlogType** then use that to instantiate the **Blog** class. The point is, you can set **empty_data** to the exact "new" object that you want to use.



In order to pass arguments to the **BlogType** constructor, you'll need to register it as a service and tag with **form.type**.

Option 2: Provide a Closure

Using a closure is the preferred method, since it will only create the object if it is needed.

The closure must accept a **FormInterface** instance as the first argument:

Listing 74-3

```

1  use Symfony\Component\OptionsResolver\OptionsResolver;
2  use Symfony\Component\Form\FormInterface;
3  // ...
4
5  public function configureOptions(OptionsResolver $resolver)
6  {
7      $resolver->setDefaults(array(
8          'empty_data' => function (FormInterface $form) {
9              return new Blog($form->get('title')->getData());
10          },
11      ));
12  }

```



Chapter 75

How to Use the submit() Function to Handle Form Submissions

With the `handleRequest()` method, it is really easy to handle form submissions:

Listing 75-1

```
1 use Symfony\Component\HttpFoundation\Request;
2 // ...
3
4 public function newAction(Request $request)
5 {
6     $form = $this->createFormBuilder()
7         // ...
8         ->getForm();
9
10    $form->handleRequest($request);
11
12    if ($form->isValid()) {
13        // perform some action...
14
15        return $this->redirectToRoute('task_success');
16    }
17
18    return $this->render('AppBundle:Default:new.html.twig', array(
19        'form' => $form->createView(),
20    ));
21 }
```



To see more about this method, read [Handling Form Submissions](#).

Calling Form::submit() manually

In some cases, you want better control over when exactly your form is submitted and what data is passed to it. Instead of using the `handleRequest()`¹ method, pass the submitted data directly to `submit()`²:

Listing 75-2

```
1 use Symfony\Component\HttpFoundation\Request;
2 // ...
3
4 public function newAction(Request $request)
5 {
6     $form = $this->createFormBuilder()
7         // ...
8         ->getForm();
9
10    if ($request->isMethod('POST')) {
11        $form->submit($request->request->get($form->getName()));
12
13        if ($form->isValid()) {
14            // perform some action...
15
16            return $this->redirectToRoute('task_success');
17        }
18    }
19
20    return $this->render('AppBundle:Default:new.html.twig', array(
21        'form' => $form->createView(),
22    ));
23 }
```



Forms consisting of nested fields expect an array in `submit()`³. You can also submit individual fields by calling `submit()`⁴ directly on the field:

Listing 75-3

```
$form->get('firstName')->submit('Fabien');
```



When submitting a form via a "PATCH" request, you may want to update only a few submitted fields. To achieve this, you may pass an optional second boolean parameter to `submit()`. Passing `false` will remove any missing fields within the form object. Otherwise, the missing fields will be set to `null`.

-
1. http://api.symfony.com/3.0/Symfony/Component/Form/FormInterface.html#method_handleRequest
 2. http://api.symfony.com/3.0/Symfony/Component/Form/FormInterface.html#method_submit
 3. http://api.symfony.com/3.0/Symfony/Component/Form/FormInterface.html#method_submit
 4. http://api.symfony.com/3.0/Symfony/Component/Form/FormInterface.html#method_submit



Chapter 76

How to Use the virtual Form Field Option



As of Symfony 2.3, the **virtual** option is renamed to **inherit_data**. You can read everything about the new option in "*How to Reduce Code Duplication with 'inherit_data'*".



Chapter 77

Using Bower with Symfony

Symfony and all its packages are perfectly managed by Composer. Bower is a dependency management tool for front-end dependencies, like Bootstrap or jQuery. As Symfony is purely a back-end framework, it can't help you much with Bower. Fortunately, it is very easy to use!

Installing Bower

*Bower*¹ is built on top of *Node.js*². Make sure you have that installed and then run:

Listing 77-1 1 \$ npm install -g bower

After this command has finished, run **bower** in your terminal to find out if it's installed correctly.



If you don't want to have NodeJS on your computer, you can also use *BowerPHP*³ (an unofficial PHP port of Bower). Beware that this is currently in beta status. If you're using BowerPHP, use **bowerphp** instead of **bower** in the examples.

Configuring Bower in your Project

Normally, Bower downloads everything into a **bower_components/** directory. In Symfony, only files in the **web/** directory are publicly accessible, so you need to configure Bower to download things there instead. To do that, just create a **.bowerrc** file with a new destination (like **web/assets/vendor**):

Listing 77-2

```
1 {  
2   "directory": "web/assets/vendor/"  
3 }
```

1. <http://bower.io>
2. <https://nodejs.org>
3. <http://bowerphp.org/>



If you're using a front-end build system like *Gulp*⁴ or *Grunt*⁵, then you can set the directory to whatever you want. Typically, you'll use these tools to ultimately move all assets into the **web/** directory.

An Example: Installing Bootstrap

Believe it or not, but you're now ready to use Bower in your Symfony application. As an example, you'll now install Bootstrap in your project and include it in your layout.

Installing the Dependency

To create a **bower.json** file, just run **bower init**. Now you're ready to start adding things to your project. For example, to add *Bootstrap*⁶ to your **bower.json** and download it, just run:

Listing 77-3 1 `$ bower install --save bootstrap`

This will install Bootstrap and its dependencies in **web/assets/vendor/** (or whatever directory you configured in **.bowerrc**).

For more details on how to use Bower, check out Bower documentation⁷.

Including the Dependency in your Template

Now that the dependencies are installed, you can include bootstrap in your template like normal CSS/JS:

Listing 77-4

```

1  {# app/Resources/views/layout.html.twig #}
2  <!doctype html>
3  <html>
4      <head>
5          {# ... #}
6
7          <link rel="stylesheet"
8              href="{{ asset('assets/vendor/bootstrap/dist/css/bootstrap.min.css') }}">
9      </head>
10
11     {# ... #}
12 </html>
```

Great job! Your site is now using Bootstrap. You can now easily upgrade bootstrap to the latest version and manage other front-end dependencies too.

Should I Git Ignore or Commit Bower Assets?

Currently, you should probably *commit* the assets downloaded by Bower instead of adding the directory (e.g. **web/assets/vendor**) to your **.gitignore** file:

Listing 77-5 1 `$ git add web/assets/vendor`

4. <http://gulpjs.com/>
5. <http://gruntjs.com/>
6. <http://getbootstrap.com/>
7. <http://bower.io/>

Why? Unlike Composer, Bower currently does not have a "lock" feature, which means that there's no guarantee that running **bower install** on a different server will give you the *exact* assets that you have on other machines. For more details, read the article *Checking in front-end dependencies*⁸.

But, it's very possible that Bower will add a lock feature in the future (e.g. *bower/bower#1748*⁹).

If you don't care too much about having *exact* the same versions, you can only commit the **bower.json** file. Running **bower install** will give you the latest versions within the specified version range of each package in **bower.json**. Using strict version constraints (e.g. **1.10.***) is often enough to ensure only bringing in compatible versions.

8. <http://addyosmani.com/blog/checking-in-front-end-dependencies/>

9. <https://github.com/bower/bower/pull/1748>



Chapter 78

How to Install or Upgrade to the Latest, Unreleased Symfony Version

In this article, you'll learn how to install and use new Symfony versions before they are released as stable versions.

Creating a New Project Based on an Unstable Symfony Version

Suppose that Symfony 2.7 version hasn't been released yet and you want to create a new project to test its features. First, *install the Composer* package manager. Then, open a command console, enter your project's directory and execute the following command:

Listing 78-1 1 `$ composer create-project symfony/framework-standard-edition my_project "2.7.*" --stability=dev`

Once the command finishes its execution, you'll have a new Symfony project created in the `my_project/` directory and based on the most recent code found in the 2.7 branch.

If you want to test a beta version, use **beta** as the value of the **stability** option:

Listing 78-2 1 `$ composer create-project symfony/framework-standard-edition my_project "2.7.*" --stability=beta`

Upgrading your Project to an Unstable Symfony Version

Suppose again that Symfony 2.7 hasn't been released yet and you want to upgrade an existing application to test that your project works with it.

First, open the `composer.json` file located in the root directory of your project. Then, edit the value of the version defined for the `symfony/symfony` dependency as follows:

Listing 78-3 1 `{`
2 `"require": {`

```
3     "symfony/symfony" : "2.7.*@dev"
4   }
5 }
```

Finally, open a command console, enter your project directory and execute the following command to update your project dependencies:

Listing 78-4 1 `$ composer update symfony/symfony`

If you prefer to test a Symfony beta version, replace the `"2.7.*@dev"` constraint by `"2.7.0-beta1"` to install a specific beta number or `2.7.*@beta` to get the most recent beta version.

After upgrading the Symfony version, read the *Symfony Upgrading Guide* to learn how you should proceed to update your application's code in case the new Symfony version has deprecated some of its features.



If you use Git to manage the project's code, it's a good practice to create a new branch to test the new Symfony version. This solution avoids introducing any issue in your application and allows you to test the new version with total confidence:

Listing 78-5

```
1 $ cd projects/my_project/
2 $ git checkout -b testing_new_symfony
3 # ... update composer.json configuration
4 $ composer update symfony/symfony
5
6 # ... after testing the new Symfony version
7 $ git checkout master
8 $ git branch -D testing_new_symfony
```



Chapter 79

How to Use Monolog to Write Logs

*Monolog*¹ is a logging library for PHP used by Symfony. It is inspired by the Python LogBook library.

Usage

To log a message simply get the **logger** service from the container in your controller:

Listing 79-1

```
1 public function indexAction()
2 {
3     $logger = $this->get('logger');
4     $logger->info('I just got the logger');
5     $logger->error('An error occurred');
6
7     // ...
8 }
```

The **logger** service has different methods for different logging levels. See *LoggerInterface*² for details on which methods are available.

Handlers and Channels: Writing Logs to different Locations

In Monolog each logger defines a logging channel, which organizes your log messages into different "categories". Then, each channel has a stack of handlers to write the logs (the handlers can be shared).



When injecting the logger in a service you can use a custom channel control which "channel" the logger will log to.

The basic handler is the **StreamHandler** which writes logs in a stream (by default in the **var/logs/prod.log** in the prod environment and **var/logs/dev.log** in the dev environment).

1. <https://github.com/Seldaek/monolog>

2. <https://github.com/php-fig/log/blob/master/Psr/Log/LoggerInterface.php>

Monolog comes also with a powerful built-in handler for the logging in prod environment: **FingersCrossedHandler**. It allows you to store the messages in a buffer and to log them only if a message reaches the action level (**error** in the configuration provided in the Symfony Standard Edition) by forwarding the messages to another handler.

Using several Handlers

The logger uses a stack of handlers which are called successively. This allows you to log the messages in several ways easily.

Listing 79-2

```
1  # app/config/config.yml
2  monolog:
3      handlers:
4          applog:
5              type: stream
6              path: /var/log/symfony.log
7              level: error
8          main:
9              type: fingers_crossed
10             action_level: warning
11             handler: file
12         file:
13             type: stream
14             level: debug
15     syslog:
16         type: syslog
17         level: error
```

The above configuration defines a stack of handlers which will be called in the order they are defined.



The handler named "file" will not be included in the stack itself as it is used as a nested handler of the **fingers_crossed** handler.



If you want to change the config of MonologBundle in another config file you need to redefine the whole stack. It cannot be merged because the order matters and a merge does not allow to control the order.

Changing the Formatter

The handler uses a **Formatter** to format the record before logging it. All Monolog handlers use an instance of **Monolog\Formatter\LineFormatter** by default but you can replace it easily. Your formatter must implement **Monolog\Formatter\FormatterInterface**.

Listing 79-3

```
1  # app/config/config.yml
2  services:
3      my_formatter:
4          class: Monolog\Formatter\JsonFormatter
5  monolog:
6      handlers:
7          file:
8              type: stream
9              level: debug
10             formatter: my_formatter
```


How to Rotate your Log Files

Over time, log files can grow to be *huge*, both while developing and on production. One best-practice solution is to use a tool like the *logrotate*³ Linux command to rotate log files before they become too large.

Another option is to have Monolog rotate the files for you by using the **rotating_file** handler. This handler creates a new log file every day and can also remove old files automatically. To use it, just set the **type** option of your handler to **rotating_file**:

Listing 79-4

```
1 # app/config/config_dev.yml
2 monolog:
3     handlers:
4         main:
5             type: rotating_file
6             path: '%kernel.logs_dir%/%kernel.environment%.log'
7             level: debug
8             # max number of log files to keep
9             # defaults to zero, which means infinite files
10            max_files: 10
```

How to Disable Microseconds Precision

New in version 2.11: The **use_microseconds** option was introduced in MonologBundle 2.11.

Setting the parameter **use_microseconds** to **false** forces the logger to reduce the precision in the **datetime** field of the log messages from microsecond to second, avoiding a call to the **microtime(true)** function and the subsequent parsing. Disabling the use of microseconds can provide a small performance gain speeding up the log generation. This is recommended for systems that generate a large number of log events.

Listing 79-5

```
1 # app/config/config.yml
2 monolog:
3     use_microseconds: false
4     handlers:
5         applog:
6             type: stream
7             path: /var/log/symfony.log
8             level: error
```

Adding some extra Data in the Log Messages

Monolog allows you to process the record before logging it to add some extra data. A processor can be applied for the whole handler stack or only for a specific handler.

A processor is simply a callable receiving the record as its first argument. Processors are configured using the **monolog.processor** DIC tag. See the reference about it.

Adding a Session/Request Token

Sometimes it is hard to tell which entries in the log belong to which session and/or request. The following example will add a unique token for each request using a processor.

Listing 79-6

3. <https://fedorahosted.org/logrotate/>

```

1 namespace AppBundle;
2
3 use Symfony\Component\HttpFoundation\Session\Session;
4
5 class SessionRequestProcessor
6 {
7     private $session;
8     private $token;
9
10    public function __construct(Session $session)
11    {
12        $this->session = $session;
13    }
14
15    public function processRecord(array $record)
16    {
17        if (null === $this->token) {
18            try {
19                $this->token = substr($this->session->getId(), 0, 8);
20            } catch (\RuntimeException $e) {
21                $this->token = '???????';
22            }
23            $this->token .= '-' . substr(uniqid(), -8);
24        }
25        $record['extra']['token'] = $this->token;
26
27        return $record;
28    }
29 }

```

Listing 79-7

```

1 # app/config/config.yml
2 services:
3     monolog.formatter.session_request:
4         class: Monolog\Formatter\LineFormatter
5         arguments:
6             - "[%datetime%] [%extra.token%] %%channel%%.%%level_name%%: %%message%% %%context%%
7             %%extra%%\n"
8
9     monolog.processor.session_request:
10        class: AppBundle\SessionRequestProcessor
11        arguments: ['@session']
12        tags:
13            - { name: monolog.processor, method: processRecord }
14
15 monolog:
16     handlers:
17         main:
18             type: stream
19             path: '%kernel.logs_dir%/%kernel.environment%.log'
20             level: debug
21             formatter: monolog.formatter.session_request

```



If you use several handlers, you can also register a processor at the handler level or at the channel level instead of registering it globally (see the following sections).

Registering Processors per Handler

You can register a processor per handler using the **handler** option of the **monolog.processor** tag:

Listing 79-8

```

1 # app/config/config.yml
2 services:
3     monolog.processor.session_request:

```

```

4      class: AppBundle\SessionRequestProcessor
5      arguments: [ '@session' ]
6      tags:
7          - { name: monolog.processor, method: processRecord, handler: main }

```

Registering Processors per Channel

You can register a processor per channel using the `channel` option of the `monolog.processor` tag:

Listing 79-9

```

1  # app/config/config.yml
2  services:
3      monolog.processor.session_request:
4          class: AppBundle\SessionRequestProcessor
5          arguments: [ '@session' ]
6          tags:
7              - { name: monolog.processor, method: processRecord, channel: main }

```



Chapter 80

How to Configure Monolog to Email Errors

*Monolog*¹ can be configured to send an email when an error occurs with an application. The configuration for this requires a few nested handlers in order to avoid receiving too many emails. This configuration looks complicated at first but each handler is fairly straightforward when it is broken down.

Listing 80-1

```
1  # app/config/config_prod.yml
2  monolog:
3      handlers:
4          mail:
5              type:          fingers_crossed
6              # 500 errors are logged at the critical level
7              action_level: critical
8              # to also log 400 level errors (but not 404's):
9              # action_level: error
10             # excluded_404s:
11             #     - ^/
12             handler:        deduplicated
13         deduplicated:
14             type:            deduplication
15             handler:         swift
16         swift:
17             type:            swift_mailer
18             from_email:      'error@example.com'
19             to_email:        'error@example.com'
20             # or list of recipients
21             # to_email:      ['dev1@example.com', 'dev2@example.com', ...]
22             subject:         'An Error Occurred! %%message%%'
23             level:           debug
24             formatter:        monolog.formatter.html
25             content_type:    text/html
```

The **mail** handler is a **fingers_crossed** handler which means that it is only triggered when the action level, in this case **critical** is reached. The **critical** level is only triggered for 5xx HTTP code errors. If this level is reached once, the **fingers_crossed** handler will log all messages regardless of their level. The **handler** setting means that the output is then passed onto the **deduplicated** handler.

1. <https://github.com/Seldaek/monolog>



If you want both 400 level and 500 level errors to trigger an email, set the `action_level` to `error` instead of `critical`. See the code above for an example.

The `deduplicated` handler simply keeps all the messages for a request and then passes them onto the nested handler in one go, but only if the records are unique over a given period of time (60 seconds by default). If the records are duplicates they are simply discarded. Adding this handler reduces the amount of notifications to a manageable level, specially in critical failure scenarios.

The messages are then passed to the `swift` handler. This is the handler that actually deals with emailing you the error. The settings for this are straightforward, the to and from addresses, the formatter, the content type and the subject.

You can combine these handlers with other handlers so that the errors still get logged on the server as well as the emails being sent:

Listing 80-2

```
1  # app/config/config_prod.yml
2  monolog:
3      handlers:
4          main:
5              type:          fingers_crossed
6              action_level: critical
7              handler:       grouped
8          grouped:
9              type:          group
10             members: [streamed, deduplicated]
11          streamed:
12              type:          stream
13              path:          '%kernel.logs_dir%/%kernel.environment%.log'
14              level:         debug
15          deduplicated:
16              type:          deduplication
17              handler:       swift
18          swift:
19              type:          swift_mailer
20              from_email:    'error@example.com'
21              to_email:      'error@example.com'
22              subject:       'An Error Occurred! %%message%%'
23              level:         debug
24              formatter:     monolog.formatter.html
25              content_type:  text/html
```

This uses the `group` handler to send the messages to the two group members, the `deduplicated` and the `stream` handlers. The messages will now be both written to the log file and emailed.



Chapter 81

How to Configure Monolog to Display Console Messages

It is possible to use the console to print messages for certain verbosity levels using the *OutputInterface*¹ instance that is passed when a command gets executed.

Alternatively, you can use the standalone PSR-3 logger provided with the console component.

When a lot of logging has to happen, it's cumbersome to print information depending on the verbosity settings (**-v**, **-vv**, **-vvv**) because the calls need to be wrapped in conditions. The code quickly gets verbose or dirty. For example:

Listing 81-1

```
1 use Symfony\Component\Console\Input\InputInterface;
2 use Symfony\Component\Console\Output\OutputInterface;
3
4 protected function execute(InputInterface $input, OutputInterface $output)
5 {
6     if ($output->getVerbosity() >= OutputInterface::VERBOSITY_DEBUG) {
7         $output->writeln('Some info');
8     }
9
10    if ($output->getVerbosity() >= OutputInterface::VERBOSITY_VERBOSE) {
11        $output->writeln('Some more info');
12    }
13 }
```

Instead of using these semantic methods to test for each of the verbosity levels, the *MonologBridge*² provides a *ConsoleHandler*³ that listens to console events and writes log messages to the console output depending on the current log level and the console verbosity.

The example above could then be rewritten as:

Listing 81-2

```
1 use Symfony\Component\Console\Input\InputInterface;
2 use Symfony\Component\Console\Output\OutputInterface;
```

-
1. <http://api.symfony.com/3.0/Symfony/Component/Console/Output/OutputInterface.html>
 2. <https://github.com/symfony/MonologBridge>
 3. <https://github.com/symfony/MonologBridge/blob/master/Handler/ConsoleHandler.php>

```

3
4 protected function execute(InputInterface $input, OutputInterface $output)
5 {
6     // assuming the Command extends ContainerAwareCommand...
7     $logger = $this->getContainer()->get('logger');
8     $logger->debug('Some info');
9
10    $logger->notice('Some more info');
11 }

```

Depending on the verbosity level that the command is run in and the user's configuration (see below), these messages may or may not be displayed to the console. If they are displayed, they are timestamped and colored appropriately. Additionally, error logs are written to the error output (php://stderr). There is no need to conditionally handle the verbosity settings anymore.

The Monolog console handler is enabled in the Monolog configuration.

Listing 81-3

```

1 # app/config/config.yml
2 monolog:
3     handlers:
4         console:
5             type: console

```

With the **verbosity_levels** option you can adapt the mapping between verbosity and log level. In the given example it will also show notices in normal verbosity mode (instead of warnings only). Additionally, it will only use messages logged with the custom **my_channel** channel and it changes the display style via a custom formatter (see the *MonologBundle* reference for more information):

Listing 81-4

```

1 # app/config/config.yml
2 monolog:
3     handlers:
4         console:
5             type: console
6             verbosity_levels:
7                 VERBOSITY_NORMAL: NOTICE
8             channels: my_channel
9             formatter: my_formatter

```

Listing 81-5

```

1 # app/config/services.yml
2 services:
3     my_formatter:
4         class: Symfony\Bridge\Monolog\Formatter\ConsoleFormatter
5         arguments:
6             - "[%datetime%] %start_tag%%message%%end_tag% (%level_name%) %context% %extra%\n"

```



Chapter 82

How to Configure Monolog to Exclude 404 Errors from the Log

Sometimes your logs become flooded with unwanted 404 HTTP errors, for example, when an attacker scans your app for some well-known application paths (e.g. */phpmyadmin*). When using a **fingers_crossed** handler, you can exclude logging these 404 errors based on a regular expression in the MonologBundle configuration:

Listing 82-1

```
1 # app/config/config.yml
2 monolog:
3   handlers:
4     main:
5       # ...
6       type: fingers_crossed
7       handler: ...
8       excluded_404s:
9         - ^/phpmyadmin
```




Chapter 83

How to Log Messages to different Files

The Symfony Framework organizes log messages into channels. By default, there are several channels, including **doctrine**, **event**, **security**, **request** and more. The channel is printed in the log message and can also be used to direct different channels to different places/files.

By default, Symfony logs every message into a single file (regardless of the channel).



Each channel corresponds to a logger service (**monolog.logger.XXX**) in the container (use the **debug:container** command to see a full list) and those are injected into different services.

Switching a Channel to a different Handler

Now, suppose you want to log the **security** channel to a different file. To do this, just create a new handler and configure it to log only messages from the **security** channel. You might add this in **config.yml** to log in all environments, or just **config_prod.yml** to happen only in **prod**:

Listing 83-1

```
1  # app/config/config.yml
2  monolog:
3      handlers:
4          security:
5              # log all messages (since debug is the lowest level)
6              level:    debug
7              type:     stream
8              path:     '%kernel.logs_dir%/security.log'
9              channels: [security]
10
11      # an example of *not* logging security channel messages for this handler
12      main:
13          # ...
14          # channels: ['!security']
```



The **channels** configuration only works for top level handlers. Handlers that are nested inside a group, buffer, filter, fingers crossed or other such handler will ignore this configuration and will process every message passed to them.

YAML Specification

You can specify the configuration by many forms:

Listing 83-2

```
1 channels: ~      # Include all the channels
2
3 channels: foo    # Include only channel 'foo'
4 channels: '!foo' # Include all channels, except 'foo'
5
6 channels: [foo, bar] # Include only channels 'foo' and 'bar'
7 channels: ['!foo', '!bar'] # Include all channels, except 'foo' and 'bar'
```

Creating your own Channel

You can change the channel monolog logs to one service at a time. This is done either via the configuration below or by tagging your service with `monolog.logger` and specifying which channel the service should log to. With the tag, the logger that is injected into that service is preconfigured to use the channel you've specified.

Configure Additional Channels without Tagged Services

You can also configure additional channels without the need to tag your services:

Listing 83-3

```
1 # app/config/config.yml
2 monolog:
3     channels: ['foo', 'bar']
```

With this, you can now send log messages to the `foo` channel by using the automatically registered logger service `monolog.logger.foo`.

Learn more from the Cookbook

- *How to Use Monolog to Write Logs*



Chapter 84

How to Create a custom Data Collector

The *Symfony Profiler* delegates data collection to some special classes called data collectors. Symfony comes bundled with a few of them, but you can easily create your own.

Creating a custom Data Collector

Creating a custom data collector is as simple as implementing the *DataCollectorInterface*¹:

Listing 84-1

```
1 interface DataCollectorInterface
2 {
3     function collect(Request $request, Response $response, \Exception $exception = null);
4     function getName();
5 }
```

The *getName()*² method returns the name of the data collector and must be unique in the application. This value is also used to access the information later on (see *How to Use the Profiler in a Functional Test* for instance).

The *collect()*³ method is responsible for storing the collected data in local properties.

Most of the time, it is convenient to extend *DataCollector*⁴ and populate the *\$this->data* property (it takes care of serializing the *\$this->data* property). Imagine you create a new data collector that collects the method and accepted content types from the request:

Listing 84-2

```
1 // src/AppBundle/DataCollector/RequestCollector.php
2 namespace AppBundle\DataCollector;
3
4 use Symfony\Component\HttpKernel\DataCollector\DataCollector;
5
6 class RequestCollector extends DataCollector
7 {
8     public function collect(Request $request, Response $response, \Exception $exception = null)
```

-
1. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/DataCollector/DataCollectorInterface.html>
 2. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/DataCollector/DataCollectorInterface.html#method_getName
 3. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/DataCollector/DataCollectorInterface.html#method_collect
 4. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/DataCollector/DataCollector.html>

```

9      {
10         $this->data = array(
11             'method' => $request->getMethod(),
12             'acceptable_content_types' => $request->getAcceptableContentTypes(),
13         );
14     }
15
16     public function getMethod()
17     {
18         return $this->data['method'];
19     }
20
21     public function getAcceptableContentTypes()
22     {
23         return $this->data['acceptable_content_types'];
24     }
25
26     public function getName()
27     {
28         return 'app.request_collector';
29     }
30 }

```

The getters are added to give the template access to the collected information.



As the profiler serializes data collector instances, you should not store objects that cannot be serialized (like PDO objects) or you need to provide your own **serialize()** method.

Enabling Custom Data Collectors

To enable a data collector, define it as a regular service and tag it as **data_collector**:

Listing 84-3

```

1  # app/config/services.yml
2  services:
3      app.request_collector:
4          class: AppBundle\DataCollector\RequestCollector
5          public: false
6          tags:
7              - { name: data_collector }

```

Adding Web Profiler Templates

The information collected by your data collector can be displayed both in the web debug toolbar and in the web profiler. To do so, you need to create a Twig template that includes some specific blocks.

In the simplest case, you just want to display the information in the toolbar without providing a profiler panel. This requires to define the **toolbar** block and set the value of two variables called **icon** and **text**:

Listing 84-4

```

1  {% extends 'WebProfilerBundle:Profiler:layout.html.twig' %}
2
3  {% block toolbar %}
4      {% set icon %}
5          {# this is the content displayed as a panel in the toolbar #}
6          <span class="icon"></span>
7          <span class="sf-toolbar-status">Request</span>
8      {% endset %}
9

```

```

10     {% set text %}
11         {# this is the content displayed when hovering the mouse over
12         the toolbar panel #}
13         <div class="sf-toolbar-info-piece">
14             <b>Method</b>
15             <span>{{ collector.method }}</span>
16         </div>
17
18         <div class="sf-toolbar-info-piece">
19             <b>Accepted content type</b>
20             <span>{{ collector.acceptableContentTypes|join(', ') }}</span>
21         </div>
22     {% endset %}
23
24     {# the 'link' value set to 'false' means that this panel doesn't
25     show a section in the web profiler #}
26     {{ include('@WebProfiler/Profiler/toolbar_item.html.twig', { link: false }) }}
27 {% endblock %}

```



Built-in collector templates define all their images as embedded base64-encoded images. This makes them work everywhere without having to mess with web assets links:

Listing 84-5 1

Another solution is to define the images as SVG files. In addition to being resolution-independent, these images can be easily embedded in the Twig template or included from an external file to reuse them in several templates:

Listing 84-6 1 {{ include('@App/data_collector/icon.svg') }}

You are encouraged to use the latter technique for your own toolbar panels.

If the toolbar panel includes extended web profiler information, the Twig template must also define additional blocks:

Listing 84-7

```

1  {% extends '@WebProfiler/Profiler/layout.html.twig' %}
2
3  {% block toolbar %}
4      {% set icon %}
5          <span class="icon"></span>
6          <span class="sf-toolbar-status">Request</span>
7      {% endset %}
8
9      {% set text %}
10         <div class="sf-toolbar-info-piece">
11             {# ... #}
12         </div>
13     {% endset %}
14
15     {{ include('@WebProfiler/Profiler/toolbar_item.html.twig', { 'link': true }) }}
16 {% endblock %}
17
18 {% block head %}
19     {# Optional. Here you can link to or define your own CSS and JS contents. #}
20     {# Use {{ parent() }} to extend the default styles instead of overriding them. #}
21 {% endblock %}
22
23 {% block menu %}
24     {# This left-hand menu appears when using the full-screen profiler. #}
25     <span class="label">
26         <span class="icon"></span>
27         <strong>Request</strong>
28     </span>
29 {% endblock %}
30

```

```

31 {% block panel %}
32     {# Optional, for showing the most details. #}
33     <h2>Acceptable Content Types</h2>
34     <table>
35         <tr>
36             <th>Content Type</th>
37         </tr>
38
39         {% for type in collector.acceptableContentTypes %}
40         <tr>
41             <td>{{ type }}</td>
42         </tr>
43         {% endfor %}
44     </table>
45 {% endblock %}

```

The **menu** and **panel** blocks are the only required blocks to define the contents displayed in the web profiler panel associated with this data collector. All blocks have access to the **collector** object.

Finally, to enable the data collector template, add a **template** attribute to the **data_collector** tag in your service configuration:

Listing 84-8

```

1  # app/config/services.yml
2  services:
3      app.request_collector:
4          class: AppBundle\DataCollector\RequestCollector
5          tags:
6              -
7              name:      data_collector
8              template: 'data_collector/template.html.twig'
9              id:        'app.request_collector'
10         public: false

```



The **id** attribute must match the value returned by the **getName()** method.

The position of each panel in the toolbar is determined by the priority defined by each collector. Most built-in collectors use **255** as their priority. If you want your collector to be displayed before them, use a higher value:

Listing 84-9

```

1  # app/config/services.yml
2  services:
3      app.request_collector:
4          class: AppBundle\DataCollector\RequestCollector
5          tags:
6              - { name: data_collector, template: '...', id: '...', priority: 300 }

```



Chapter 85

How to Use Matchers to Enable the Profiler Conditionally

The Symfony profiler is only activated in the development environment to not hurt your application performance. However, sometimes it may be useful to conditionally enable the profiler in the production environment to assist you in debugging issues. This behavior is implemented with the **Request Matchers**.

Using the built-in Matcher

A request matcher is a class that checks whether a given **Request** instance matches a set of conditions. Symfony provides a *built-in matcher*¹ which matches paths and IPs. For example, if you want to only show the profiler when accessing the page with the **168.0.0.1** IP, then you can use this configuration:

Listing 85-1

```
1 # app/config/config.yml
2 framework:
3     # ...
4     profiler:
5         matcher:
6             ip: 168.0.0.1
```

You can also set a **path** option to define the path on which the profiler should be enabled. For instance, setting it to **^/admin/** will enable the profiler only for the URLs which start with **/admin/**.

Creating a Custom Matcher

Leveraging the concept of Request Matchers you can define a custom matcher to enable the profiler conditionally in your application. To do so, create a class which implements *RequestMatcherInterface*². This interface requires one method: *matches()*³. This method

1. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/RequestMatcher.html>

returns **false** when the request doesn't match the conditions and **true** otherwise. Therefore, the custom matcher must return **false** to disable the profiler and **true** to enable it.

Suppose that the profiler must be enabled whenever a user with a **ROLE_SUPER_ADMIN** is logged in. This is the only code needed for that custom matcher:

```
Listing 85-2 1 // src/AppBundle/Profiler/SuperAdminMatcher.php
2 namespace AppBundle\Profiler;
3
4 use Symfony\Component\Security\Core\Authorization\AuthorizationCheckerInterface;
5 use Symfony\Component\HttpFoundation\Request;
6 use Symfony\Component\HttpFoundation\RequestMatcherInterface;
7
8 class SuperAdminMatcher implements RequestMatcherInterface
9 {
10     protected $authorizationChecker;
11
12     public function __construct(AuthorizationCheckerInterface $authorizationChecker)
13     {
14         $this->authorizationChecker = $authorizationChecker;
15     }
16
17     public function matches(Request $request)
18     {
19         return $this->authorizationChecker->isGranted('ROLE_SUPER_ADMIN');
20     }
21 }
```

Then, configure a new service and set it as **private** because the application won't use it directly:

```
Listing 85-3 1 # app/config/services.yml
2 services:
3     app.super_admin_matcher:
4         class: AppBundle\Profiler\SuperAdminMatcher
5         arguments: ['@security.authorization_checker']
6         public: false
```

Once the service is registered, the only thing left to do is configure the profiler to use this service as the matcher:

```
Listing 85-4 1 # app/config/config.yml
2 framework:
3     # ...
4     profiler:
5         matcher:
6             service: app.super_admin_matcher
```

2. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/RequestMatcherInterface.html>

3. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/RequestMatcherInterface.html#method_matches



Chapter 86

Switching the Profiler Storage

In Symfony versions prior to 3.0, profiles could be stored in files, databases, services like Redis and Memcache, etc. Starting from Symfony 3.0, the only storage mechanism with built-in support is the filesystem.

By default the profiler stores the collected data in the `%kernel.cache_dir%/profiler/` directory. If you want to use another location to store the profiles, define the `dsn` option of the `framework.profiler`:

Listing 86-1

```
1 # app/config/config.yml
2 framework:
3     profiler:
4         dsn: 'file:/tmp/symfony/profiler'
```

You can also create your own profile storage service implementing the `class:Symfony\Component\HttpKernel\Profiler\ProfilerStorageInterface` and overriding the `profiler.storage` service.



Chapter 87

How to Access Profiling Data Programmatically

Most of the times, the profiler information is accessed and analyzed using its web-based visualizer. However, you can also retrieve profiling information programmatically thanks to the methods provided by the **profiler** service.

When the response object is available, use the *loadProfileFromResponse()*¹ method to access to its associated profile:

Listing 87-1

```
// ... $profiler is the 'profiler' service
$profile = $profiler->loadProfileFromResponse($response);
```

When the profiler stores data about a request, it also associates a token with it; this token is available in the **X-Debug-Token** HTTP header of the response. Using this token, you can access the profile of any past response thanks to the *loadProfile()*² method:

Listing 87-2

```
$token = $response->headers->get('X-Debug-Token');
$profile = $container->get('profiler')->loadProfile($token);
```



When the profiler is enabled but not the web debug toolbar, inspect the page with your browser's developer tools to get the value of the **X-Debug-Token** HTTP header.

The **profiler** service also provides the *find()*³ method to look for tokens based on some criteria:

Listing 87-3

```
1 // get the latest 10 tokens
2 $tokens = $container->get('profiler')->find('', '', 10, '', '', '');
3
4 // get the latest 10 tokens for all URL containing /admin/
5 $tokens = $container->get('profiler')->find('', '/admin/', 10, '', '', '');
6
7 // get the latest 10 tokens for local POST requests
8 $tokens = $container->get('profiler')->find('127.0.0.1', '', 10, 'POST', '', '');
9
10 // get the latest 10 tokens for requests that happened between 2 and 4 days ago
```

1. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Profiler/Profiler.html#method_loadProfileFromResponse
2. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Profiler/Profiler.html#method_loadProfile
3. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Profiler/Profiler.html#method_find

```
11 $tokens = $container->get('profiler')
12     ->find('', '', 10, '', '4 days ago', '2 days ago');
```



Chapter 88

The PSR-7 Bridge

The PSR-7 bridge converts *HttpFoundation* objects from and to objects implementing HTTP message interfaces defined by the *PSR-7*¹.

Installation

You can install the component in 2 different ways:

- Install it via Composer (*symfony/psr-http-message-bridge* on Packagist²);
- Use the official Git repository (<https://github.com/symfony/psr-http-message-bridge>).

The bridge also needs a PSR-7 implementation to allow converting *HttpFoundation* objects to PSR-7 objects. It provides native support for *Zend Diactoros*³. Use Composer (*zendframework/zend-diactoros* on Packagist⁴) or refer to the project documentation to install it.

Usage

Converting from *HttpFoundation* Objects to PSR-7

The bridge provides an interface of a factory called *HttpMessageFactoryInterface*⁵ that builds objects implementing PSR-7 interfaces from *HttpFoundation* objects. It also provide a default implementation using *Zend Diactoros* internally.

1. <http://www.php-fig.org/psr/psr-7/>

2. <https://packagist.org/packages/symfony/psr-http-message-bridge>

3. <https://github.com/zendframework/zend-diactoros>

4. <https://packagist.org/packages/zendframework/zend-diactoros>

5. <http://api.symfony.com/3.0/Symfony/Bridge/ PsrHttpMessage/HttpMessageFactoryInterface.html>

The following code snippet explain how to convert a *Request*⁶ to a Zend Diactoros *ServerRequest*⁷ implementing the *ServerRequestInterface*⁸ interface:

Listing 88-1

```
1 use Symfony\Bridge\PsrHttpMessage\Factory\DiactorosFactory;
2 use Symfony\Component\HttpFoundation\Request;
3
4 $symfonyRequest = new Request(array(), array(), array(), array(), array(), array('HTTP_HOST' => 'dunglas.fr'),
5 'Content');
6 // The HTTP_HOST server key must be set to avoid an unexpected error
7
8 $psr7Factory = new DiactorosFactory();
9 $psrRequest = $psr7Factory->createRequest($symfonyRequest);
```

And now from a *Response*⁹ to a Zend Diactoros *Response*¹⁰ implementing the *ResponseInterface*¹¹ interface:

Listing 88-2

```
1 use Symfony\Bridge\PsrHttpMessage\Factory\DiactorosFactory;
2 use Symfony\Component\HttpFoundation\Response;
3
4 $symfonyResponse = new Response('Content');
5
6 $psr7Factory = new DiactorosFactory();
7 $psrResponse = $psr7Factory->createResponse($symfonyResponse);
```

Converting Objects implementing PSR-7 Interfaces to HttpFoundation

On the other hand, the bridge provide a factory interface called *HttpFoundationFactoryInterface*¹² that builds HttpFoundation objects from objects implementing PSR-7 interfaces.

The next snippet explain how to convert an object implementing the *ServerRequestInterface*¹³ interface to a *Request*¹⁴ instance:

Listing 88-3

```
1 use Symfony\Bridge\PsrHttpMessage\Factory\HttpFoundationFactory;
2
3 // $psrRequest is an instance of Psr\Http\Message\ServerRequestInterface
4
5 $httpFoundationFactory = new HttpFoundationFactory();
6 $symfonyRequest = $httpFoundationFactory->createRequest($psrRequest);
```

From an object implementing the *ResponseInterface*¹⁵ to a *Response*¹⁶ instance:

Listing 88-4

```
1 use Symfony\Bridge\PsrHttpMessage\Factory\HttpFoundationFactory;
2
3 // $psrResponse is an instance of Psr\Http\Message\ResponseInterface
4
5 $httpFoundationFactory = new HttpFoundationFactory();
6 $symfonyResponse = $httpFoundationFactory->createResponse($psrResponse);
```

-
- 6. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html>
 - 7. <http://api.symfony.com/3.0/Zend/Diactoros/ServerRequest.html>
 - 8. <http://api.symfony.com/3.0/Psr/Http/Message/ServerRequestInterface.html>
 - 9. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html>
 - 10. <http://api.symfony.com/3.0/Zend/Diactoros/Response.html>
 - 11. <http://api.symfony.com/3.0/Psr/Http/Message/ResponseInterface.html>
 - 12. <http://api.symfony.com/3.0/Symfony/Bridge/PsrHttpMessage/HttpFoundationFactoryInterface.html>
 - 13. <http://api.symfony.com/3.0/Psr/Http/Message/ServerRequestInterface.html>
 - 14. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html>
 - 15. <http://api.symfony.com/3.0/Psr/Http/Message/ResponseInterface.html>
 - 16. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html>



Chapter 89

How to Configure Symfony to Work behind a Load Balancer or a Reverse Proxy

When you deploy your application, you may be behind a load balancer (e.g. an AWS Elastic Load Balancer) or a reverse proxy (e.g. Varnish for *caching*).

For the most part, this doesn't cause any problems with Symfony. But, when a request passes through a proxy, certain request information is sent using either the standard **Forwarded** header or non-standard special **X-Forwarded-*** headers. For example, instead of reading the **REMOTE_ADDR** header (which will now be the IP address of your reverse proxy), the user's true IP will be stored in a standard **Forwarded: for="..."** header or a non standard **X-Forwarded-For** header.

If you don't configure Symfony to look for these headers, you'll get incorrect information about the client's IP address, whether or not the client is connecting via HTTPS, the client's port and the hostname being requested.

Solution: `trusted_proxies`

This is no problem, but you *do* need to tell Symfony what is happening and which reverse proxy IP addresses will be doing this type of thing:

Listing 89-1

```
1 # app/config/config.yml
2 # ...
3 framework:
4     trusted_proxies: [192.0.0.1, 10.0.0.0/8]
```

In this example, you're saying that your reverse proxy (or proxies) has the IP address **192.0.0.1** or matches the range of IP addresses that use the CIDR notation **10.0.0.0/8**. For more details, see the `framework.trusted_proxies` option.

You are also saying that you trust that the proxy does not send conflicting headers, e.g. sending both **X-Forwarded-For** and **Forwarded** in the same request.

That's it! Symfony will now look for the correct headers to get information like the client's IP address, host, port and whether the request is using HTTPS.

But what if the IP of my Reverse Proxy Changes Constantly!

Some reverse proxies (like Amazon's Elastic Load Balancers) don't have a static IP address or even a range that you can target with the CIDR notation. In this case, you'll need to - *very carefully* - trust *all* proxies.

1. Configure your web server(s) to *not* respond to traffic from *any* clients other than your load balancers. For AWS, this can be done with *security groups*¹.
2. Once you've guaranteed that traffic will only come from your trusted reverse proxies, configure Symfony to *always* trust incoming request. This is done inside of your front controller:

Listing 89-2

```
1 // web/app.php
2
3 // ...
4 Request::setTrustedProxies(array('127.0.0.1', $request->server->get('REMOTE_ADDR')));
5
6 $response = $kernel->handle($request);
7 // ...
```

3. Ensure that the `trusted_proxies` setting in your `app/config/config.yml` is not set or it will overwrite the `setTrustedProxies` call above.

That's it! It's critical that you prevent traffic from all non-trusted sources. If you allow outside traffic, they could "spoof" their true IP address and other information.

My Reverse Proxy Sends X-Forwarded-For but Does not Filter the Forwarded Header

Many popular proxy implementations do not yet support the **Forwarded** header and do not filter it by default. Ideally, you would configure this in your proxy. If this is not possible, you can tell Symfony to distrust the **Forwarded** header, while still trusting your proxy's **X-Forwarded-For** header.

This is done inside of your front controller:

Listing 89-3

```
1 // web/app.php
2
3 // ...
4 Request::setTrustedHeaderName(Request::HEADER_FORWARDED, null);
5
6 $response = $kernel->handle($request);
7 // ...
```

Configuring the proxy server trust is very important, as not doing so will allow malicious users to "spoof" their IP address.

My Reverse Proxy Uses Non-Standard (not X-Forwarded) Headers

Although *RFC 7239*² recently defined a standard **Forwarded** header to disclose all proxy information, most reverse proxies store information in non-standard **X-Forwarded-*** headers.

But if your reverse proxy uses other non-standard header names, you can configure these (see "*Trusting Proxies*").

The code for doing this will need to live in your front controller (e.g. `web/app.php`).

1. <http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/using-elb-security-groups.html>

2. <http://tools.ietf.org/html/rfc7239>



Chapter 90

How to Register a new Request Format and Mime Type

Every **Request** has a "format" (e.g. **html**, **json**), which is used to determine what type of content to return in the **Response**. In fact, the request format, accessible via `getRequestFormat()`¹, is used to set the MIME type of the **Content-Type** header on the **Response** object. Internally, Symfony contains a map of the most common formats (e.g. **html**, **json**) and their associated MIME types (e.g. **text/html**, **application/json**). Of course, additional format-MIME type entries can easily be added. This document will show how you can add the **jsonp** format and corresponding MIME type.

Configure your New Format

The FrameworkBundle registers a subscriber that will add formats to incoming requests.

All you have to do is to configure the **jsonp** format:

Listing 90-1

```
1 # app/config/config.yml
2 framework:
3     request:
4         formats:
5             jsonp: 'application/javascript'
```



You can also associate multiple mime types to a format, but please note that the preferred one must be the first as it will be used as the content type:

Listing 90-2

```
1 # app/config/config.yml
2 framework:
3     request:
4         formats:
5             csv: ['text/csv', 'text/plain']
```

1. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getRequestFormat



Chapter 91

How to Force Routes to always Use HTTPS or HTTP

Sometimes, you want to secure some routes and be sure that they are always accessed via the HTTPS protocol. The Routing component allows you to enforce the URI scheme via schemes:

Listing 91-1

```
1 secure:
2     path:    /secure
3     defaults: { _controller: AppBundle:Main:secure }
4     schemes: [https]
```

The above configuration forces the **secure** route to always use HTTPS.

When generating the **secure** URL, and if the current scheme is HTTP, Symfony will automatically generate an absolute URL with HTTPS as the scheme:

Listing 91-2

```
1  {# If the current scheme is HTTPS #}
2  {{ path('secure') }}
3  {# generates /secure #}
4
5  {# If the current scheme is HTTP #}
6  {{ path('secure') }}
7  {# generates https://example.com/secure #}
```

The requirement is also enforced for incoming requests. If you try to access the **/secure** path with HTTP, you will automatically be redirected to the same URL, but with the HTTPS scheme.

The above example uses **https** for the scheme, but you can also force a URL to always use **http**.



The Security component provides another way to enforce HTTP or HTTPS via the **requires_channel** setting. This alternative method is better suited to secure an "area" of your website (all URLs under **/admin**) or when you want to secure URLs defined in a third party bundle (see *How to Force HTTPS or HTTP for different URLs* for more details).



Chapter 92

How to Allow a "/" Character in a Route Parameter

Sometimes, you need to compose URLs with parameters that can contain a slash /. For example, take the classic `/hello/{username}` route. By default, `/hello/Fabien` will match this route but not `/hello/Fabien/Kris`. This is because Symfony uses this character as separator between route parts.

This guide covers how you can modify a route so that `/hello/Fabien/Kris` matches the `/hello/{username}` route, where `{username}` equals `Fabien/Kris`.

Configure the Route

By default, the Symfony Routing component requires that the parameters match the following regex path: `[^/]+`. This means that all characters are allowed except /.

You must explicitly allow / to be part of your parameter by specifying a more permissive regex path.

Listing 92-1

```
1 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
2
3 class DemoController
4 {
5     /**
6      * @Route("/hello/{username}", name="_hello", requirements={"username"=".*"})
7      */
8     public function helloAction($username)
9     {
10         // ...
11     }
12 }
```

That's it! Now, the `{username}` parameter can contain the / character.



Chapter 93

How to Configure a Redirect without a custom Controller

Sometimes, a URL needs to redirect to another URL. You can do that by creating a new controller action whose only task is to redirect, but using the *RedirectController*¹ of the FrameworkBundle is even easier.

You can redirect to a specific path (e.g. */about*) or to a specific route using its name (e.g. *homepage*).

Redirecting Using a Path

Assume there is no default controller for the */* path of your application and you want to redirect these requests to */app*. You will need to use the *urlRedirectAction()*² action to redirect to this new url:

Listing 93-1

```
1  # app/config/routing.yml
2
3  # load some routes - one should ultimately have the path "/app"
4  AppBundle:
5      resource: '@AppBundle/Controller/'
6      type:     annotation
7      prefix:   /app
8
9  # redirecting the root
10 root:
11     path: /
12     defaults:
13         _controller: FrameworkBundle\Redirect:urlRedirect
14         path: /app
15         permanent: true
```

In this example, you configured a route for the */* path and let the *RedirectController* redirect it to */app*. The *permanent* switch tells the action to issue a **301** HTTP status code instead of the default **302** HTTP status code.

1. <http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/RedirectController.html>

2. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/RedirectController.html#method_urlRedirectAction

Redirecting Using a Route

Assume you are migrating your website from WordPress to Symfony, you want to redirect `/wp-admin` to the route `sonata_admin_dashboard`. You don't know the path, only the route name. This can be achieved using the `redirectAction()`³ action:

Listing 93-2

```
1  # app/config/routing.yml
2
3  # ...
4
5  # redirecting the admin home
6  root:
7      path: /wp-admin
8      defaults:
9          _controller: FrameworkBundle\Redirect:redirect
10         route: sonata_admin_dashboard
11         permanent: true
```



Because you are redirecting to a route instead of a path, the required option is called **route** in the **redirect** action, instead of **path** in the **urlRedirect** action.

3. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/RedirectController.html#method_redirectAction



Chapter 94

How to Use HTTP Methods beyond GET and POST in Routes

The HTTP method of a request is one of the requirements that can be checked when seeing if it matches a route. This is introduced in the routing chapter of the book "*Routing*" with examples using GET and POST. You can also use other HTTP verbs in this way. For example, if you have a blog post entry then you could use the same URL path to show it, make changes to it and delete it by matching on GET, PUT and DELETE.

Listing 94-1

```
1  blog_show:
2      path:      /blog/{slug}
3      defaults: { _controller: AppBundle:Blog:show }
4      methods:   [GET]
5
6  blog_update:
7      path:      /blog/{slug}
8      defaults: { _controller: AppBundle:Blog:update }
9      methods:   [PUT]
10
11 blog_delete:
12     path:      /blog/{slug}
13     defaults: { _controller: AppBundle:Blog:delete }
14     methods:   [DELETE]
```

Faking the Method with `_method`

Unfortunately, life isn't quite this simple, since most browsers do not support sending PUT and DELETE requests via the `method` attribute in an HTML form. Fortunately, Symfony provides you with a simple way of working around this limitation. By including a `_method` parameter in the query string or parameters of an HTTP request, Symfony will use this as the method when matching routes. Forms automatically include a hidden field for this parameter if their submission method is not GET or POST. See the related chapter in the forms documentation for more information.



This feature can be disabled using the `http_method_override` option.



Chapter 95

How to Use Service Container Parameters in your Routes

Sometimes you may find it useful to make some parts of your routes globally configurable. For instance, if you build an internationalized site, you'll probably start with one or two locales. Surely you'll add a requirement to your routes to prevent a user from matching a locale other than the locales you support.

You *could* hardcode your `_locale` requirement in all your routes, but a better solution is to use a configurable service container parameter right inside your routing configuration:

Listing 95-1

```
1 # app/config/routing.yml
2 contact:
3     path:      /{_locale}/contact
4     defaults: { _controller: AppBundle:Main:contact }
5     requirements:
6         _locale: '%app.locales%'
```

You can now control and set the `app.locales` parameter somewhere in your container:

Listing 95-2

```
1 # app/config/config.yml
2 parameters:
3     app.locales: en|es
```

You can also use a parameter to define your route path (or part of your path):

Listing 95-3

```
1 # app/config/routing.yml
2 some_route:
3     path:      /%app.route_prefix%/contact
4     defaults: { _controller: AppBundle:Main:contact }
```



Just like in normal service container configuration files, if you actually need a `%` in your route, you can escape the percent sign by doubling it, e.g. `/score-50%%`, which would resolve to `/score-50%`.

However, as the `%` characters included in any URL are automatically encoded, the resulting URL of this example would be `/score-50%25` (`%25` is the result of encoding the `%` character).

For parameter handling within a Dependency Injection Class see Using Parameters within a Dependency Injection Class.



Chapter 96

How to Create a custom Route Loader

What is a Custom Route Loader

A custom route loader enables you to generate routes based on some conventions or patterns. A great example for this use-case is the *FOSRestBundle*¹ where routes are generated based on the names of the action methods in a controller.

You still need to modify your routing configuration (e.g. `app/config/routing.yml`) manually, even when using a custom route loader.



There are many bundles out there that use their own route loaders to accomplish cases like those described above, for instance *FOSRestBundle*², *JMSI18nRoutingBundle*³, *KnpradBundle*⁴ and *SonataAdminBundle*⁵.

Loading Routes

The routes in a Symfony application are loaded by the *DelegatingLoader*⁶. This loader uses several other loaders (delegates) to load resources of different types, for instance YAML files or `@Route` and `@Method` annotations in controller files. The specialized loaders implement *LoaderInterface*⁷ and therefore have two important methods: *supports()*⁸ and *load()*⁹.

Take these lines from the `routing.yml` in the Symfony Standard Edition:

-
1. <https://github.com/FriendsOfSymfony/FOSRestBundle>
 2. <https://github.com/FriendsOfSymfony/FOSRestBundle>
 3. <https://github.com/schmittjoh/JMSI18nRoutingBundle>
 4. <https://github.com/Knplabs/KnpRadBundle>
 5. <https://github.com/sonata-project/SonataAdminBundle>
 6. <http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Routing/DelegatingLoader.html>
 7. <http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderInterface.html>
 8. http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderInterface.html#method_supports
 9. http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderInterface.html#method_load

Listing 96-1

```
1 # app/config/routing.yml
2 app:
3     resource: '@AppBundle/Controller/'
4     type:     annotation
```

When the main loader parses this, it tries all registered delegate loaders and calls their *supports()*¹⁰ method with the given resource (`@AppBundle/Controller/`) and type (`annotation`) as arguments. When one of the loader returns `true`, its *load()*¹¹ method will be called, which should return a *RouteCollection*¹² containing *Route*¹³ objects.



Routes loaded this way will be cached by the Router the same way as when they are defined in one of the default formats (e.g. XML, YML, PHP file).

Creating a custom Loader

To load routes from some custom source (i.e. from something other than annotations, YAML or XML files), you need to create a custom route loader. This loader has to implement *LoaderInterface*¹⁴.

In most cases it is easier to extend from *Loader*¹⁵ instead of implementing *LoaderInterface*¹⁶ yourself.

The sample loader below supports loading routing resources with a type of `extra`. The type name should not clash with other loaders that might support the same type of resource. Just make up a name specific to what you do. The resource name itself is not actually used in the example:

Listing 96-2

```
1 // src/AppBundle/Routing/ExtraLoader.php
2 namespace AppBundle\Routing;
3
4 use Symfony\Component\Config\Loader\Loader;
5 use Symfony\Component\Routing\Route;
6 use Symfony\Component\Routing\RouteCollection;
7
8 class ExtraLoader extends Loader
9 {
10     private $loaded = false;
11
12     public function load($resource, $type = null)
13     {
14         if (true === $this->loaded) {
15             throw new \RuntimeException('Do not add the "extra" loader twice');
16         }
17
18         $routes = new RouteCollection();
19
20         // prepare a new route
21         $path = '/extra/{parameter}';
22         $defaults = array(
23             '_controller' => 'AppBundle:Extra:extra',
24         );
25         $requirements = array(
26             'parameter' => '\d+',
```

10. http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderInterface.html#method_supports

11. http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderInterface.html#method_load

12. <http://api.symfony.com/3.0/Symfony/Component/Routing/RouteCollection.html>

13. <http://api.symfony.com/3.0/Symfony/Component/Routing/Route.html>

14. <http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderInterface.html>

15. <http://api.symfony.com/3.0/Symfony/Component/Config/Loader/Loader.html>

16. <http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderInterface.html>

```

27     );
28     $route = new Route($path, $defaults, $requirements);
29
30     // add the new route to the route collection
31     $routeName = 'extraRoute';
32     $routes->add($routeName, $route);
33
34     $this->loaded = true;
35
36     return $routes;
37 }
38
39 public function supports($resource, $type = null)
40 {
41     return 'extra' === $type;
42 }
43 }

```

Make sure the controller you specify really exists. In this case you have to create an **extraAction** method in the **ExtraController** of the **AppBundle**:

Listing 96-3

```

1 // src/AppBundle/Controller/ExtraController.php
2 namespace AppBundle\Controller;
3
4 use Symfony\Component\HttpFoundation\Response;
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6
7 class ExtraController extends Controller
8 {
9     public function extraAction($parameter)
10    {
11        return new Response($parameter);
12    }
13 }

```

Now define a service for the **ExtraLoader**:

Listing 96-4

```

1 # app/config/services.yml
2 services:
3     app.routing_loader:
4         class: AppBundle\Routing\ExtraLoader
5         tags:
6             - { name: routing.loader }

```

Notice the tag **routing.loader**. All services with this *tag* will be marked as potential route loaders and added as specialized route loaders to the **routing.loader** service, which is an instance of *DelegatingLoader*¹⁷.

Using the custom Loader

If you did nothing else, your custom routing loader would *not* be called. What remains to do is adding a few lines to the routing configuration:

Listing 96-5

```

1 # app/config/routing.yml
2 app_extra:
3     resource: .
4     type: extra

```

17. <http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Routing/DelegatingLoader.html>

The important part here is the **type** key. Its value should be "extra" as this is the type which the **ExtraLoader** supports and this will make sure its **load()** method gets called. The **resource** key is insignificant for the **ExtraLoader**, so it is set to ".".



The routes defined using custom route loaders will be automatically cached by the framework. So whenever you change something in the loader class itself, don't forget to clear the cache.

More advanced Loaders

If your custom route loader extends from *Loader*¹⁸ as shown above, you can also make use of the provided resolver, an instance of *LoaderResolver*¹⁹, to load secondary routing resources.

Of course you still need to implement *supports()*²⁰ and *load()*²¹. Whenever you want to load another resource - for instance a YAML routing configuration file - you can call the *import()*²² method:

Listing 96-6

```
1  // src/AppBundle/Routing/AdvancedLoader.php
2  namespace AppBundle\Routing;
3
4  use Symfony\Component\Config\Loader\Loader;
5  use Symfony\Component\Routing\RouteCollection;
6
7  class AdvancedLoader extends Loader
8  {
9      public function load($resource, $type = null)
10     {
11         $collection = new RouteCollection();
12
13         $resource = '@AppBundle/Resources/config/import_routing.yml';
14         $type = 'yaml';
15
16         $importedRoutes = $this->import($resource, $type);
17
18         $collection->addCollection($importedRoutes);
19
20         return $collection;
21     }
22
23     public function supports($resource, $type = null)
24     {
25         return 'advanced_extra' === $type;
26     }
27 }
```



The resource name and type of the imported routing configuration can be anything that would normally be supported by the routing configuration loader (YAML, XML, PHP, annotation, etc.).

18. <http://api.symfony.com/3.0/Symfony/Component/Config/Loader/Loader.html>

19. <http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderResolver.html>

20. http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderInterface.html#method_supports

21. http://api.symfony.com/3.0/Symfony/Component/Config/Loader/LoaderInterface.html#method_load

22. http://api.symfony.com/3.0/Symfony/Component/Config/Loader/Loader.html#method_import



Chapter 97

Redirect URLs with a Trailing Slash

The goal of this cookbook is to demonstrate how to redirect URLs with a trailing slash to the same URL without a trailing slash (for example `/en/blog/` to `/en/blog`).

Create a controller that will match any URL with a trailing slash, remove the trailing slash (keeping query parameters if any) and redirect to the new URL with a 301 response status code:

Listing 97-1

```
1 // src/AppBundle/Controller/RedirectingController.php
2 namespace AppBundle\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5 use Symfony\Component\HttpFoundation\Request;
6
7 class RedirectingController extends Controller
8 {
9     public function removeTrailingSlashAction(Request $request)
10     {
11         $pathInfo = $request->getPathInfo();
12         $requestUri = $request->getRequestUri();
13
14         $url = str_replace($pathInfo, rtrim($pathInfo, '/'), $requestUri);
15
16         return $this->redirect($url, 301);
17     }
18 }
```

After that, create a route to this controller that's matched whenever a URL with a trailing slash is requested. Be sure to put this route last in your system, as explained below:

Listing 97-2

```
1 // src/AppBundle/Controller/RedirectingController.php
2 namespace AppBundle\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5 use Symfony\Component\HttpFoundation\Request;
6
7 class RedirectingController extends Controller
8 {
9     /**
10      * @Route("/{url}", name="remove_trailing_slash",
11      *     requirements={"url" = ".*/$"}, methods={"GET"})
12      */
13     public function removeTrailingSlashAction(Request $request)
```

```
14     {  
15         // ...  
16     }  
17 }
```



Redirecting a POST request does not work well in old browsers. A 302 on a POST request would send a GET request after the redirection for legacy reasons. For that reason, the route here only matches GET requests.



Make sure to include this route in your routing configuration at the very end of your route listing. Otherwise, you risk redirecting real routes (including Symfony core routes) that actually *do* have a trailing slash in their path.



Chapter 98

How to Pass Extra Information from a Route to a Controller

Parameters inside the **defaults** collection don't necessarily have to match a placeholder in the route **path**. In fact, you can use the **defaults** array to specify extra parameters that will then be accessible as arguments to your controller, and as attributes of the **Request** object:

Listing 98-1

```
1 # app/config/routing.yml
2 blog:
3     path:      /blog/{page}
4     defaults:
5         _controller: AppBundle:Blog:index
6         page:      1
7         title:      "Hello world!"
```

Now, you can access this extra parameter in your controller, as an argument to the controller method:

Listing 98-2

```
public function indexAction($page, $title)
{
    // ...
}
```

Alternatively, the title could be accessed through the **Request** object:

Listing 98-3

```
1 use Symfony\Component\HttpFoundation\Request;
2
3 public function indexAction(Request $request, $page)
4 {
5     $title = $request->attributes->get('title');
6
7     // ...
8 }
```

As you can see, the **\$title** variable was never defined inside the route path, but you can still access its value from inside your controller, through the method's argument, or from the **Request** object's **attributes** bag.



Chapter 99

Looking up Routes from a Database: Symfony CMF DynamicRouter

The core Symfony Routing System is excellent at handling complex sets of routes. A highly optimized routing cache is dumped during deployments.

However, when working with large amounts of data that each need a nice readable URL (e.g. for search engine optimization purposes), the routing can get slowed down. Additionally, if routes need to be edited by users, the route cache would need to be rebuilt frequently.

For these cases, the **DynamicRouter** offers an alternative approach:

- Routes are stored in a database;
- There is a database index on the path field, the lookup scales to huge numbers of different routes;
- Writes only affect the index of the database, which is very efficient.

When all routes are known during deploy time and the number is not too high, using a *custom route loader* is the preferred way to add more routes. When working with just one type of objects, a slug parameter on the object and the **@ParamConverter** annotation work fine (see *FrameworkExtraBundle*¹).

The **DynamicRouter** is useful when you need **Route** objects with the full feature set of Symfony. Each route can define a specific controller so you can decouple the URL structure from your application logic.

The **DynamicRouter** comes with built-in support for Doctrine ORM and Doctrine PHPCR-ODM but offers the **ContentRepositoryInterface** to write a custom loader, e.g. for another database type or a REST API or anything else.

The **DynamicRouter** is explained in the *Symfony CMF documentation*².

1. <https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html>

2. <http://symfony.com/doc/master/cmf/book/routing.html>



Chapter 100

How to Build a Traditional Login Form



If you need a login form and are storing users in some sort of a database, then you should consider using *FOSUserBundle*¹, which helps you build your **User** object and gives you many routes and controllers for common tasks like login, registration and forgot password.

In this entry, you'll build a traditional login form. Of course, when the user logs in, you can load your users from anywhere - like the database. See B) Configuring how Users are Loaded for details.

First, enable form login under your firewall:

Listing 100-1

```
1  # app/config/security.yml
2  security:
3      # ...
4
5      firewalls:
6          main:
7              anonymous: ~
8              form_login:
9                  login_path: login
10                 check_path: login
```



The `login_path` and `check_path` can also be route names (but cannot have mandatory wildcards - e.g. `/login/{foo}` where `foo` has no default value).

Now, when the security system initiates the authentication process, it will redirect the user to the login form `/login`. Implementing this login form visually is your job. First, create a new **SecurityController** inside a bundle:

Listing 100-2

```
1  // src/AppBundle/Controller/SecurityController.php
2  namespace AppBundle\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5
6  class SecurityController extends Controller
```

1. <https://github.com/FriendsOfSymfony/FOSUserBundle>

```

7 {
8 }

```

Next, configure the route that you earlier used under your `form_login` configuration (`login`):

Listing 100-3

```

1 // src/AppBundle/Controller/SecurityController.php
2
3 // ...
4 use Symfony\Component\HttpFoundation\Request;
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6
7 class SecurityController extends Controller
8 {
9     /**
10      * @Route("/login", name="login")
11      */
12     public function loginAction(Request $request)
13     {
14     }
15 }

```

Great! Next, add the logic to `loginAction` that will display the login form:

Listing 100-4

```

1 // src/AppBundle/Controller/SecurityController.php
2
3 public function loginAction(Request $request)
4 {
5     $authenticationUtils = $this->get('security.authentication_utils');
6
7     // get the login error if there is one
8     $error = $authenticationUtils->getLastAuthenticationError();
9
10    // last username entered by the user
11    $lastUsername = $authenticationUtils->getLastUsername();
12
13    return $this->render(
14        'security/login.html.twig',
15        array(
16            // last username entered by the user
17            'last_username' => $lastUsername,
18            'error'         => $error,
19        )
20    );
21 }

```

Don't let this controller confuse you. As you'll see in a moment, when the user submits the form, the security system automatically handles the form submission for you. If the user had submitted an invalid username or password, this controller reads the form submission error from the security system so that it can be displayed back to the user.

In other words, your job is to *display* the login form and any login errors that may have occurred, but the security system itself takes care of checking the submitted username and password and authenticating the user.

Finally, create the template:

Listing 100-5

```

1 {# app/Resources/views/security/login.html.twig #}
2 {# ... you will probably extends your base template, like base.html.twig #}
3
4 {% if error %}
5     <div>{{ error.messageKey|trans(error.messageData, 'security') }}</div>
6 {% endif %}
7
8 <form action="{{ path('login') }}" method="post">
9     <label for="username">Username:</label>
10     <input type="text" id="username" name="_username" value="{{ last_username }}" />

```

```

11
12     <label for="password">Password:</label>
13     <input type="password" id="password" name="_password" />
14
15     {#
16         If you want to control the URL the user
17         is redirected to on success (more details below)
18         <input type="hidden" name="_target_path" value="/account" />
19     #}
20
21     <button type="submit">login</button>
22 </form>

```



The **error** variable passed into the template is an instance of *AuthenticationException*². It may contain more information - or even sensitive information - about the authentication failure, so use it wisely!

The form can look like anything, but has a few requirements:

- The form must POST to the `login` route, since that's what you configured under the `form_login` key in `security.yml`.
- The username must have the name `_username` and the password must have the name `_password`.



Actually, all of this can be configured under the `form_login` key. See Form Login Configuration for more details.



This login form is currently not protected against CSRF attacks. Read *Using CSRF Protection in the Login Form* on how to protect your login form.

And that's it! When you submit the form, the security system will automatically check the user's credentials and either authenticate the user or send the user back to the login form where the error can be displayed.

To review the whole process:

1. The user tries to access a resource that is protected;
2. The firewall initiates the authentication process by redirecting the user to the login form (`/login`);
3. The `/login` page renders login form via the route and controller created in this example;
4. The user submits the login form to `/login`;
5. The security system intercepts the request, checks the user's submitted credentials, authenticates the user if they are correct, and sends the user back to the login form if they are not.

Redirecting after Success

If the submitted credentials are correct, the user will be redirected to the original page that was requested (e.g. `/admin/foo`). If the user originally went straight to the login page, they'll be redirected to the homepage. This can all be customized, allowing you to, for example, redirect the user to a specific URL.

For more details on this and how to customize the form login process in general, see *How to Customize your Form Login*.

2. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Exception/AuthenticationException.html>

Avoid Common Pitfalls

When setting up your login form, watch out for a few common pitfalls.

1. Create the Correct Routes

First, be sure that you've defined the `/login` route correctly and that it corresponds to the `login_path` and `check_path` config values. A misconfiguration here can mean that you're redirected to a 404 page instead of the login page, or that submitting the login form does nothing (you just see the login form over and over again).

2. Be Sure the Login Page Isn't Secure (Redirect Loop!)

Also, be sure that the login page is accessible by anonymous users. For example, the following configuration - which requires the `ROLE_ADMIN` role for all URLs (including the `/login` URL), will cause a redirect loop:

```
Listing 100-6 1 # app/config/security.yml
2
3 # ...
4 access_control:
5     - { path: ^/, roles: ROLE_ADMIN }
```

Adding an access control that matches `/login/*` and requires *no* authentication fixes the problem:

```
Listing 100-7 1 # app/config/security.yml
2
3 # ...
4 access_control:
5     - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
6     - { path: ^/, roles: ROLE_ADMIN }
```

Also, if your firewall does *not* allow for anonymous users (no `anonymous` key), you'll need to create a special firewall that allows anonymous users for the login page:

```
Listing 100-8 1 # app/config/security.yml
2
3 # ...
4 firewalls:
5     # order matters! This must be before the ^/ firewall
6     login_firewall:
7         pattern: ^/login$
8         anonymous: ~
9     secured_area:
10        pattern: ^/
11        form_login: ~
```

3. Be Sure check_path Is Behind a Firewall

Next, make sure that your `check_path` URL (e.g. `/login`) is behind the firewall you're using for your form login (in this example, the single firewall matches *all* URLs, including `/login`). If `/login` doesn't match any firewall, you'll receive a **Unable to find the controller for path "/login"** exception.

4. Multiple Firewalls Don't Share the Same Security Context

If you're using multiple firewalls and you authenticate against one firewall, you will *not* be authenticated against any other firewalls automatically. Different firewalls are like different security systems. To do this you have to explicitly specify the same Firewall Context for different firewalls. But usually for most applications, having one main firewall is enough.

5. Routing Error Pages Are not Covered by Firewalls

As routing is done *before* security, 404 error pages are not covered by any firewall. This means you can't check for security or even access the user object on these pages. See *How to Customize Error Pages* for more details.



Chapter 101

Authenticating against an LDAP server

Symfony provides different means to work with an LDAP server.

The Security component offers:

- The `ldap` user provider, using the `LdapUserProvider`¹ class. Like all other user providers, it can be used with any authentication provider.
- The `form_login_ldap` authentication provider, for authenticating against an LDAP server using a login form. Like all other authentication providers, it can be used with any user provider.
- The `http_basic_ldap` authentication provider, for authenticating against an LDAP server using HTTP Basic. Like all other authentication providers, it can be used with any user provider.

This means that the following scenarios will work:

- Checking a user's password and fetching user information against an LDAP server. This can be done using both the LDAP user provider and either the LDAP form login or LDAP HTTP Basic authentication providers.
- Checking a user's password against an LDAP server while fetching user information from another source (database using `FOSUserBundle`, for example).
- Loading user information from an LDAP server, while using another authentication strategy (token-based pre-authentication, for example).

Ldap Configuration Reference

See *SecurityBundle Configuration ("security")* for the full LDAP configuration reference (`form_login_ldap`, `http_basic_ldap`, `ldap`). Some of the more interesting options are explained below.

1. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/LdapUserProvider.html>

Configuring the LDAP client

All mechanisms actually need an LDAP client previously configured. The providers are configured to use a default service named `ldap`, but you can override this setting in the security component's configuration.

An LDAP client can be simply configured, using the following service definition:

```
Listing 101-1 1 # app/config/services.yml
2 services:
3     ldap:
4         class: 'Symfony\Component\Ldap\LdapClient'
5         arguments:
6             - my-server # host
7             - 389 # port
8             - 3 # version
9             - false # SSL
10            - true # TLS
```

Fetching Users Using the LDAP User Provider

If you want to fetch user information from an LDAP server, you may want to use the `ldap` user provider.

```
Listing 101-2 1 # app/config/security.yml
2 security:
3     # ...
4
5     providers:
6         my_ldap:
7             ldap:
8                 service: ldap
9                 base_dn: dc=example,dc=com
10                search_dn: "cn=read-only-admin,dc=example,dc=com"
11                search_password: password
12                default_roles: ROLE_USER
13                uid_key: uid
```

The `ldap` user provider supports many different configuration options:

service

type: string **default:** `ldap`

This is the name of your configured LDAP client. You can freely chose the name, but it must be unique in your application and it cannot start with a number or contain white spaces.

base_dn

type: string **default:** `null`

This is the base DN for the directory

search_dn

type: string **default:** `null`

This is your read-only user's DN, which will be used to authenticate against the LDAP server in order to fetch the user's information.

search_password

type: string **default:** null

This is your read-only user's password, which will be used to authenticate against the LDAP server in order to fetch the user's information.

default_roles

type: array **default:** []

This is the default role you wish to give to a user fetched from the LDAP server. If you do not configure this key, your users won't have any roles, and will not be considered as authenticated fully.

uid_key

type: string **default:** sAMAccountName

This is the entry's key to use as its UID. Depends on your LDAP server implementation. Commonly used values are:

- sAMAccountName
- userPrincipalName
- uid

filter

type: string **default:** ({uid_key}={username})

This key lets you configure which LDAP query will be used. The `{uid_key}` string will be replaced by the value of the `uid_key` configuration value (by default, `sAMAccountName`), and the `{username}` string will be replaced by the username you are trying to load.

For example, with a `uid_key` of `uid`, and if you are trying to load the user `fabpot`, the final string will be: `(uid=fabpot)`.

Of course, the username will be escaped, in order to prevent *LDAP injection*².

The syntax for the `filter` key is defined by *RFC4515*³.

Authenticating against an LDAP server

Authenticating against an LDAP server can be done using either the form login or the HTTP Basic authentication providers.

They are configured exactly as their non-LDAP counterparts, with the addition of two configuration keys:

service

type: string **default:** ldap

This is the name of your configured LDAP client. You can freely choose the name, but it must be unique in your application and it cannot start with a number or contain white spaces.

2. <http://projects.webappsec.org/w/page/13246947/LDAP%20Injection>

3. <http://www.faqs.org/rfcs/rfc4515.html>

dn_string

type: string **default:** {username}

This key defines the form of the string used in order to compose the DN of the user, from the username. The `{username}` string is replaced by the actual username of the person trying to authenticate.

For example, if your users have DN strings in the form `uid=einstein,dc=example,dc=com`, then the `dn_string` will be `uid={username},dc=example,dc=com`.

Examples are provided below, for both `form_login_ldap` and `http_basic_ldap`.

Configuration example for form login

Listing 101-3

```
1 # app/config/security.yml
2 security:
3     # ...
4
5     firewalls:
6         main:
7             # ...
8             form_login_ldap:
9                 login_path: login
10                check_path: login_check
11                # ...
12                service: ldap
13                dn_string: 'uid={username},dc=example,dc=com'
```

Configuration example for HTTP Basic

Listing 101-4

```
1 # app/config/security.yml
2 security:
3     # ...
4
5     firewalls:
6         main:
7             # ...
8             http_basic_ldap:
9                 # ...
10                service: ldap
11                dn_string: 'uid={username},dc=example,dc=com'
```



Chapter 102

How to Load Security Users from the Database (the Entity Provider)

Symfony's security system can load security users from anywhere - like a database, via Active Directory or an OAuth server. This article will show you how to load your users from the database via a Doctrine entity.

Introduction



Before you start, you should check out *FOSUserBundle*¹. This external bundle allows you to load users from the database (like you'll learn here) *and* gives you built-in routes & controllers for things like login, registration and forgot password. But, if you need to heavily customize your user system *or* if you want to learn how things work, this tutorial is even better.

Loading users via a Doctrine entity has 2 basic steps:

1. Create your User entity
2. Configure security.yml to load from your entity

Afterwards, you can learn more about forbidding inactive users, using a custom query and user serialization to the session

1) Create your User Entity

For this entry, suppose that you already have a **User** entity inside an **AppBundle** with the following fields: **id**, **username**, **password**, **email** and **isActive**:

Listing 102-1

```
1 // src/AppBundle/Entity/User.php
2 namespace AppBundle\Entity;
3
```

1. <https://github.com/FriendsOfSymfony/FOSUserBundle>

```

4 use Doctrine\ORM\Mapping as ORM;
5 use Symfony\Component\Security\Core\User\UserInterface;
6
7 /**
8  * @ORM\Table(name="app_users")
9  * @ORM\Entity(repositoryClass="AppBundle\Entity\UserRepository")
10 */
11 class User implements UserInterface, \Serializable
12 {
13     /**
14      * @ORM\Column(type="integer")
15      * @ORM\Id
16      * @ORM\GeneratedValue(strategy="AUTO")
17      */
18     private $id;
19
20     /**
21      * @ORM\Column(type="string", length=25, unique=true)
22      */
23     private $username;
24
25     /**
26      * @ORM\Column(type="string", length=64)
27      */
28     private $password;
29
30     /**
31      * @ORM\Column(type="string", length=60, unique=true)
32      */
33     private $email;
34
35     /**
36      * @ORM\Column(name="is_active", type="boolean")
37      */
38     private $isActive;
39
40     public function __construct()
41     {
42         $this->isActive = true;
43         // may not be needed, see section on salt below
44         // $this->salt = md5(uniqid(null, true));
45     }
46
47     public function getUsername()
48     {
49         return $this->username;
50     }
51
52     public function getSalt()
53     {
54         // you *may* need a real salt depending on your encoder
55         // see section on salt below
56         return null;
57     }
58
59     public function getPassword()
60     {
61         return $this->password;
62     }
63
64     public function getRoles()
65     {
66         return array('ROLE_USER');
67     }
68
69     public function eraseCredentials()
70     {
71     }
72
73     /** @see \Serializable::serialize() */
74     public function serialize()

```

```

75     {
76         return serialize(array(
77             $this->id,
78             $this->username,
79             $this->password,
80             // see section on salt below
81             // $this->salt,
82         ));
83     }
84
85     /** @see \Serializable::unserialize() */
86     public function unserialize($serialized)
87     {
88         list (
89             $this->id,
90             $this->username,
91             $this->password,
92             // see section on salt below
93             // $this->salt
94         ) = unserialize($serialized);
95     }
96 }

```

To make things shorter, some of the getter and setter methods aren't shown. But you can generate these by running:

Listing 102-2 `1 $ php bin/console doctrine:generate:entities AppBundle/Entity/User`

Next, make sure to create the database table:

Listing 102-3 `1 $ php bin/console doctrine:schema:update --force`

What's this `UserInterface`?

So far, this is just a normal entity. But to use this class in the security system, it must implement *`UserInterface`*². This forces the class to have the five following methods:

- `getRoles()`³
- `getPassword()`⁴
- `getSalt()`⁵
- `getUsername()`⁶
- `eraseCredentials()`⁷

To learn more about each of these, see *`UserInterface`*⁸.

What do the `serialize` and `unserialize` Methods do?

At the end of each request, the `User` object is serialized to the session. On the next request, it's unserialized. To help PHP do this correctly, you need to implement `Serializable`. But you don't need to serialize everything: you only need a few fields (the ones shown above plus a few extra if you decide to implement `AdvancedUserInterface`). On each request, the `id` is used to query for a fresh `User` object from the database.

Want to know more? See Understanding `serialize` and how a `User` is Saved in the Session.

2. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html>

3. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html#method_getRoles

4. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html#method_getPassword

5. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html#method_getSalt

6. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html#method_getUsername

7. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html#method_eraseCredentials

8. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html>

2) Configure Security to load from your Entity

Now that you have a **User** entity that implements **UserInterface**, you just need to tell Symfony's security system about it in **security.yml**.

In this example, the user will enter their username and password via HTTP basic authentication. Symfony will query for a **User** entity matching the username and then check the password (more on passwords in a moment):

```
Listing 102-4 1 # app/config/security.yml
2 security:
3     encoders:
4         AppBundle\Entity\User:
5             algorithm: bcrypt
6
7     # ...
8
9     providers:
10        our_db_provider:
11            entity:
12                class: AppBundle\User
13                property: username
14                # if you're using multiple entity managers
15                # manager_name: customer
16
17        firewalls:
18            main:
19                pattern: ^/
20                http_basic: ~
21                provider: our_db_provider
22
23        # ...
```

First, the **encoders** section tells Symfony to expect that the passwords in the database will be encoded using **bcrypt**. Second, the **providers** section creates a "user provider" called **our_db_provider** that knows to query from your **AppBundle\User** entity by the **username** property. The name **our_db_provider** isn't important: it just needs to match the value of the **provider** key under your firewall. Or, if you don't set the **provider** key under your firewall, the first "user provider" is automatically used.

Creating your First User

To add users, you can implement a *registration form* or add some *fixtures*⁹. This is just a normal entity, so there's nothing tricky, *except* that you need to encode each user's password. But don't worry, Symfony gives you a service that will do this for you. See Dynamically Encoding a Password for details.

Below is an export of the **app_users** table from MySQL with user **admin** and password **admin** (which has been encoded).

```
Listing 102-5 1 $ mysql> SELECT * FROM app_users;
2 +-----+-----+-----+-----+-----+
3 | id | username | password | email | is_active |
4 +-----+-----+-----+-----+-----+
5 | 1 | admin | $2a$08$jHZj/wJfcVKlIwr5AvR78euJxYK7Ku5kURNhNx.7.CSIJ3Pq6LEPC | admin@example.com | 1 |
6 +-----+-----+-----+-----+-----+
```

9. <https://symfony.com/doc/master/bundles/DoctrineFixturesBundle/index.html>



Do you need to use a Salt property?

If you use **bcrypt**, no. Otherwise, yes. All passwords must be hashed with a salt, but **bcrypt** does this internally. Since this tutorial *does* use **bcrypt**, the **getSalt()** method in **User** can just return **null** (it's not used). If you use a different algorithm, you'll need to uncomment the **salt** lines in the **User** entity and add a persisted **salt** property.

Forbid Inactive Users (AdvancedUserInterface)

If a User's **isActive** property is set to **false** (i.e. **is_active** is 0 in the database), the user will still be able to login to the site normally. This is easily fixable.

To exclude inactive users, change your **User** class to implement *AdvancedUserInterface*¹⁰. This extends *UserInterface*¹¹, so you only need the new interface:

Listing 102-6

```

1  // src/AppBundle/Entity/User.php
2
3  use Symfony\Component\Security\Core\User\AdvancedUserInterface;
4  // ...
5
6  class User implements AdvancedUserInterface, \Serializable
7  {
8      // ...
9
10     public function isAccountNonExpired()
11     {
12         return true;
13     }
14
15     public function isAccountNonLocked()
16     {
17         return true;
18     }
19
20     public function isCredentialsNonExpired()
21     {
22         return true;
23     }
24
25     public function isEnabled()
26     {
27         return $this->isActive;
28     }
29
30     // serialize and unserialize must be updated - see below
31     public function serialize()
32     {
33         return serialize(array(
34             // ...
35             $this->isActive
36         ));
37     }
38     public function unserialize($serialized)
39     {
40         list (
41             // ...
42             $this->isActive
43         ) = unserialize($serialized);
44     }
45 }
```

10. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/AdvancedUserInterface.html>

11. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html>

The *AdvancedUserInterface*¹² interface adds four extra methods to validate the account status:

- *isAccountNonExpired()*¹³ checks whether the user's account has expired;
- *isAccountNonLocked()*¹⁴ checks whether the user is locked;
- *isCredentialsNonExpired()*¹⁵ checks whether the user's credentials (password) has expired;
- *isEnabled()*¹⁶ checks whether the user is enabled.

If *any* of these return **false**, the user won't be allowed to login. You can choose to have persisted properties for all of these, or whatever you need (in this example, only **isActive** pulls from the database).

So what's the difference between the methods? Each returns a slightly different error message (and these can be translated when you render them in your login template to customize them further).



If you use **AdvancedUserInterface**, you also need to add any of the properties used by these methods (like **isActive**) to the **serialize()** and **unserialize()** methods. If you *don't* do this, your user may not be deserialized correctly from the session on each request.

Congrats! Your database-loading security system is all setup! Next, add a true *login form* instead of HTTP Basic or keep reading for other topics.

Using a Custom Query to Load the User

It would be great if a user could login with their username *or* email, as both are unique in the database. Unfortunately, the native entity provider is only able to handle querying via a single property on the user.

To do this, make your **UserRepository** implement a special *UserLoaderInterface*¹⁷. This interface only requires one method: **loadUserByUsername(\$username)**:

Listing 102-7

```
1 // src/AppBundle/Entity/UserRepository.php
2 namespace AppBundle\Entity;
3
4 use Symfony\Bridge\Doctrine\Security\User\UserLoaderInterface;
5 use Doctrine\ORM\EntityRepository;
6
7 class UserRepository extends EntityRepository implements UserLoaderInterface
8 {
9     public function loadUserByUsername($username)
10     {
11         return $this->createQueryBuilder('u')
12             ->where('u.username = :username OR u.email = :email')
13             ->setParameter('username', $username)
14             ->setParameter('email', $username)
15             ->getQuery()
16             ->getOneOrNullResult();
17     }
18 }
```



Don't forget to add the repository class to the mapping definition of your entity.

To finish this, just remove the **property** key from the user provider in **security.yml**:

-
- 12. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/AdvancedUserInterface.html>
 - 13. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/AdvancedUserInterface.html#method_isAccountNonExpired
 - 14. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/AdvancedUserInterface.html#method_isAccountNonLocked
 - 15. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/AdvancedUserInterface.html#method_isCredentialsNonExpired
 - 16. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/AdvancedUserInterface.html#method_isEnabled
 - 17. <http://api.symfony.com/3.0/Symfony/Bridge/Doctrine/Security/User/UserLoaderInterface.html>

Listing 102-8

```
1 # app/config/security.yml
2 security:
3     # ...
4
5     providers:
6         our_db_provider:
7             entity:
8                 class: AppBundle\User
```

This tells Symfony to *not* query automatically for the User. Instead, when someone logs in, the `loadUserByUsername()` method on `UserRepository` will be called.

Understanding serialize and how a User is Saved in the Session

If you're curious about the importance of the `serialize()` method inside the `User` class or how the User object is serialized or deserialized, then this section is for you. If not, feel free to skip this.

Once the user is logged in, the entire User object is serialized into the session. On the next request, the User object is deserialized. Then, the value of the `id` property is used to re-query for a fresh User object from the database. Finally, the fresh User object is compared to the deserialized User object to make sure that they represent the same user. For example, if the `username` on the 2 User objects doesn't match for some reason, then the user will be logged out for security reasons.

Even though this all happens automatically, there are a few important side-effects.

First, the *Serializable*¹⁸ interface and its `serialize` and `unserialize` methods have been added to allow the `User` class to be serialized to the session. This may or may not be needed depending on your setup, but it's probably a good idea. In theory, only the `id` needs to be serialized, because the *refreshUser()*¹⁹ method refreshes the user on each request by using the `id` (as explained above). This gives us a "fresh" User object.

But Symfony also uses the `username`, `salt`, and `password` to verify that the User has not changed between requests (it also calls your `AdvancedUserInterface` methods if you implement it). Failing to serialize these may cause you to be logged out on each request. If your User implements the *EquatableInterface*²⁰, then instead of these properties being checked, your `isEqualTo` method is simply called, and you can check whatever properties you want. Unless you understand this, you probably *won't* need to implement this interface or worry about it.

18. <http://php.net/manual/en/class.serializable.php>

19. http://api.symfony.com/3.0/Symfony/Bridge/Doctrine/Security/User/EntityUserProvider.html#method_refreshUser

20. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/EquatableInterface.html>



Chapter 103

How to Create a Custom Authentication System with Guard

Whether you need to build a traditional login form, an API token authentication system or you need to integrate with some proprietary single-sign-on system, the Guard component can make it easy... and fun! In this example, you'll build an API token authentication system and learn how to work with Guard.

Create a User and a User Provider

No matter how you authenticate, you need to create a User class that implements **UserInterface** and configure a *user provider*. In this example, users are stored in the database via Doctrine, and each user has an **apiKey** property they use to access their account via the API:

```
Listing 103-1 1 // src/AppBundle/Entity/User.php
2 namespace AppBundle\Entity;
3
4 use Symfony\Component\Security\Core\User\UserInterface;
5 use Doctrine\ORM\Mapping as ORM;
6
7 /**
8  * @ORM\Entity
9  * @ORM\Table(name="user")
10  */
11 class User implements UserInterface
12 {
13     /**
14      * @ORM\Id
15      * @ORM\GeneratedValue(strategy="AUTO")
16      * @ORM\Column(type="integer")
17      */
18     private $id;
19
20     /**
21      * @ORM\Column(type="string", unique=true)
22      */
23     private $username;
24 }
```

```

25  /**
26   * @ORM\Column(type="string", unique=true)
27   */
28   private $apiKey;
29
30   public function getUsername()
31   {
32       return $this->username;
33   }
34
35   public function getRoles()
36   {
37       return ['ROLE_USER'];
38   }
39
40   public function getPassword()
41   {
42   }
43   public function getSalt()
44   {
45   }
46   public function eraseCredentials()
47   {
48   }
49
50   // more getters/setters
51 }

```



This User doesn't have a password, but you can add a **password** property if you also want to allow this user to login with a password (e.g. via a login form).

Your **User** class doesn't need to be stored in Doctrine: do whatever you need. Next, make sure you've configured a "user provider" for the user:

Listing 103-2

```

1  # app/config/security.yml
2  security:
3      # ...
4
5      providers:
6          your_db_provider:
7              entity:
8                  class: AppBundle\User
9
10     # ...

```

That's it! Need more information about this step, see:

- *How to Load Security Users from the Database (the Entity Provider)*
- *How to Create a custom User Provider*

Step 1) Create the Authenticator Class

Suppose you have an API where your clients will send an **X-AUTH-TOKEN** header on each request with their API token. Your job is to read this and find the associated user (if any).

To create a custom authentication system, just create a class and make it implement *GuardAuthenticatorInterface*¹. Or, extend the simpler *AbstractGuardAuthenticator*². This requires you to implement six methods:

-
1. <http://api.symfony.com/3.0/Symfony/Component/Security/Guard/GuardAuthenticatorInterface.html>
 2. <http://api.symfony.com/3.0/Symfony/Component/Security/Guard/AbstractGuardAuthenticator.html>

```

1  // src/AppBundle/Security/TokenAuthenticator.php
2  namespace AppBundle\Security;
3
4  use Symfony\Component\HttpFoundation\Request;
5  use Symfony\Component\HttpFoundation\JsonResponse;
6  use Symfony\Component\Security\Core\User\UserInterface;
7  use Symfony\Component\Security\Guard\AbstractGuardAuthenticator;
8  use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
9  use Symfony\Component\Security\Core\Exception\AuthenticationException;
10 use Symfony\Component\Security\Core\User\UserProviderInterface;
11 use Doctrine\ORM\EntityManager;
12
13 class TokenAuthenticator extends AbstractGuardAuthenticator
14 {
15     private $em;
16
17     public function __construct(EntityManager $em)
18     {
19         $this->em = $em;
20     }
21
22     /**
23      * Called on every request. Return whatever credentials you want,
24      * or null to stop authentication.
25      */
26     public function getCredentials(Request $request)
27     {
28         if (!$token = $request->headers->get('X-AUTH-TOKEN')) {
29             // no token? Return null and no other methods will be called
30             return;
31         }
32
33         // What you return here will be passed to getUser() as $credentials
34         return array(
35             'token' => $token,
36         );
37     }
38
39     public function getUser($credentials, UserProviderInterface $userProvider)
40     {
41         $apiKey = $credentials['token'];
42
43         // if null, authentication will fail
44         // if a User object, checkCredentials() is called
45         return $this->em->getRepository('AppBundle:User')
46             ->findOneBy(array('apiKey' => $apiKey));
47     }
48
49     public function checkCredentials($credentials, UserInterface $user)
50     {
51         // check credentials - e.g. make sure the password is valid
52         // no credential check is needed in this case
53
54         // return true to cause authentication success
55         return true;
56     }
57
58     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
59     {
60         // on success, let the request continue
61         return null;
62     }
63
64     public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
65     {
66         $data = array(
67             'message' => substr($exception->getMessageKey(), $exception->getMessageData())
68
69             // or to translate this message
70             // $this->translator->trans($exception->getMessageKey(), $exception->getMessageData())
71         );

```

```

72         return new JsonResponse($data, 403);
73     }
74
75     /**
76      * Called when authentication is needed, but it's not sent
77      */
78     public function start(Request $request, AuthenticationException $authException = null)
79     {
80         $data = array(
81             // you might translate this message
82             'message' => 'Authentication Required'
83         );
84
85         return new JsonResponse($data, 401);
86     }
87
88     public function supportsRememberMe()
89     {
90         return false;
91     }
92 }
93

```

Nice work! Each method is explained below: The Guard Authenticator Methods.

Step 2) Configure the Authenticator

To finish this, register the class as a service:

Listing 103-4

```

1 # app/config/services.yml
2 services:
3     app.token_authenticator:
4         class: AppBundle\Security\TokenAuthenticator
5         arguments: ['@doctrine.orm.entity_manager']

```

Finally, configure your **firewalls** key in **security.yml** to use this authenticator:

Listing 103-5

```

1 # app/config/security.yml
2 security:
3     # ...
4
5     firewalls:
6         # ...
7
8         main:
9             anonymous: ~
10            logout: ~
11
12            guard:
13                authenticators:
14                    - app.token_authenticator
15
16            # if you want, disable storing the user in the session
17            # stateless: true
18
19            # maybe other things, like form_login, remember_me, etc
20            # ...

```

You did it! You now have a fully-working API token authentication system. If your homepage required **ROLE_USER**, then you could test it under different conditions:

Listing 103-6

```

1 # test with no token
2 curl http://localhost:8000/
3 # {"message": "Authentication Required"}

```

```

4
5 # test with a bad token
6 curl -H "X-AUTH-TOKEN: FAKE" http://localhost:8000/
7 # {"message":"Username could not be found."}
8
9 # test with a working token
10 curl -H "X-AUTH-TOKEN: REAL" http://localhost:8000/
11 # the homepage controller is executed: the page loads normally

```

Now, learn more about what each method does.

The Guard Authenticator Methods

Each authenticator needs the following methods:

getCredentials(Request \$request)

This will be called on *every* request and your job is to read the token (or whatever your "authentication" information is) from the request and return it. If you return `null`, the rest of the authentication process is skipped. Otherwise, `getUser()` will be called and the return value is passed as the first argument.

getUser(\$credentials, UserProviderInterface \$userProvider)

If `getCredentials()` returns a non-null value, then this method is called and its return value is passed here as the `$credentials` argument. Your job is to return an object that implements `UserInterface`. If you do, then `checkCredentials()` will be called. If you return `null` (or throw an `AuthenticationException`) authentication will fail.

checkCredentials(\$credentials, UserInterface \$user)

If `getUser()` returns a `User` object, this method is called. Your job is to verify if the credentials are correct. For a login form, this is where you would check that the password is correct for the user. To pass authentication, return `true`. If you return *anything* else (or throw an `AuthenticationException`), authentication will fail.

onAuthenticationSuccess(Request \$request, TokenInterface \$token, \$providerKey)

This is called after successful authentication and your job is to either return a `Response`³ object that will be sent to the client or `null` to continue the request (e.g. allow the route/controller to be called like normal). Since this is an API where each request authenticates itself, you want to return `null`.

onAuthenticationFailure(Request \$request, AuthenticationException \$exception)

This is called if authentication fails. Your job is to return the `Response`⁴ object that should be sent to the client. The `$exception` will tell you *what* went wrong during authentication.

start(Request \$request, AuthenticationException \$authException = null)

This is called if the client accesses a URI/resource that requires authentication, but no authentication details were sent (i.e. you returned `null` from `getCredentials()`). Your job is to return a `Response`⁵ object that helps the user authenticate (e.g. a 401 response that says "token is missing!").

supportsRememberMe

If you want to support "remember me" functionality, return `true` from this method. You will still need to active `remember_me` under your firewall for it to work. Since this is a stateless API, you do not want to support "remember me" functionality in this example.

3. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html>

4. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html>

5. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html>

Customizing Error Messages

When `onAuthenticationFailure()` is called, it is passed an `AuthenticationException` that describes *how* authentication failed via its `$e->getMessageKey()` (and `$e->getMessageData()`) method. The message will be different based on *where* authentication fails (i.e. `getUser()` versus `checkCredentials()`).

But, you can easily return a custom message by throwing a *`CustomUserMessageAuthenticationException`*⁶. You can throw this from `getCredentials()`, `getUser()` or `checkCredentials()` to cause a failure:

```
Listing 103-7 1 // src/AppBundle/Security/TokenAuthenticator.php
2 // ...
3
4 use Symfony\Component\Security\Core\Exception\CustomUserMessageAuthenticationException;
5
6 class TokenAuthenticator extends AbstractGuardAuthenticator
7 {
8     // ...
9
10    public function getCredentials(Request $request)
11    {
12        // ...
13
14        if ($token == 'ILuvAPIs') {
15            throw new CustomUserMessageAuthenticationException(
16                'ILuvAPIs is not a real API key: it\'s just a silly phrase'
17            );
18        }
19
20        // ...
21    }
22
23    // ...
24 }
```

In this case, since "ILuvAPIs" is a ridiculous API key, you could include an easter egg to return a custom message if someone tries this:

```
Listing 103-8 1 curl -H "X-AUTH-TOKEN: ILuvAPIs" http://localhost:8000/
2 # {"message": "ILuvAPIs is not a real API key: it's just a silly phrase"}
```

Frequently Asked Questions

Can I have Multiple Authenticators?

Yes! But when you do, you'll need choose just *one* authenticator to be your "entry_point". This means you'll need to choose *which* authenticator's `start()` method should be called when an anonymous user tries to access a protected resource. For example, suppose you have an `app.form_login_authenticator` that handles a traditional form login. When a user accesses a protected page anonymously, you want to use the `start()` method from the form authenticator and redirect them to the login page (instead of returning a JSON response):

```
Listing 103-9 1 # app/config/security.yml
2 security:
3     # ...
4
5     firewalls:
```

6. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Exception/CustomUserMessageAuthenticationException.html>

```

6      # ...
7
8      main:
9          anonymous: ~
10         logout: ~
11
12         guard:
13             authenticators:
14                 - app.token_authenticator
15
16             # if you want, disable storing the user in the session
17             # stateless: true
18
19             # maybe other things, like form_login, remember_me, etc
20             # ...

```

Can I use this with `form_login`?

Yes! `form_login` is *one* way to authenticate a user, so you could use it *and* then add one or more authenticators. Using a guard authenticator doesn't collide with other ways to authenticate.

Can I use this with FOSUserBundle?

Yes! Actually, FOSUserBundle doesn't handle security: it simply gives you a `User` object and some routes and controllers to help with login, registration, forgot password, etc. When you use FOSUserBundle, you typically use `form_login` to actually authenticate the user. You can continue doing that (see previous question) or use the `User` object from FOSUserBundle and create your own authenticator(s) (just like in this article).



Chapter 104

How to Add "Remember Me" Login Functionality

Once a user is authenticated, their credentials are typically stored in the session. This means that when the session ends they will be logged out and have to provide their login details again next time they wish to access the application. You can allow users to choose to stay logged in for longer than the session lasts using a cookie with the **remember_me** firewall option:

Listing 104-1

```
1  # app/config/security.yml
2  security:
3      # ...
4
5      firewalls:
6          main:
7              # ...
8              remember_me:
9                  secret: '%secret%'
10                 lifetime: 604800 # 1 week in seconds
11                 path: /
12                 # by default, the feature is enabled by checking a
13                 # checkbox in the login form (see below), uncomment the
14                 # following line to always enable it.
15                 #always_remember_me: true
```

The **remember_me** firewall defines the following configuration options:

secret (required)

The value used to encrypt the cookie's content. It's common to use the **secret** value defined in the `app/config/parameters.yml` file.

name (default value: REMEMBERME)

The name of the cookie used to keep the user logged in. If you enable the **remember_me** feature in several firewalls of the same application, make sure to choose a different name for the cookie of each firewall. Otherwise, you'll face lots of security related problems.

lifetime (default value: 31536000)

The number of seconds during which the user will remain logged in. By default users are logged in for one year.

path (default value: /)

The path where the cookie associated with this feature is used. By default the cookie will be applied to the entire website but you can restrict to a specific section (e.g. /forum, /admin).

domain (default value: null)

The domain where the cookie associated with this feature is used. By default cookies use the current domain obtained from `$_SERVER`.

secure (default value: false)

If true, the cookie associated with this feature is sent to the user through an HTTPS secure connection.

httponly (default value: true)

If true, the cookie associated with this feature is accessible only through the HTTP protocol. This means that the cookie won't be accessible by scripting languages, such as JavaScript.

remember_me_parameter (default value: _remember_me)

The name of the form field checked to decide if the "Remember Me" feature should be enabled or not. Keep reading this article to know how to enable this feature conditionally.

always_remember_me (default value: false)

If true, the value of the `remember_me_parameter` is ignored and the "Remember Me" feature is always enabled, regardless of the desire of the end user.

token_provider (default value: null)

Defines the service id of a token provider to use. By default, tokens are stored in a cookie. For example, you might want to store the token in a database, to not have a (hashed) version of the password in a cookie. The DoctrineBridge comes with a `Symfony\Bridge\Doctrine\Security\RememberMe\DoctrineTokenProvider` that you can use.

Forcing the User to Opt-Out of the Remember Me Feature

It's a good idea to provide the user with the option to use or not use the remember me functionality, as it will not always be appropriate. The usual way of doing this is to add a checkbox to the login form. By giving the checkbox the name `_remember_me` (or the name you configured using `remember_me_parameter`), the cookie will automatically be set when the checkbox is checked and the user successfully logs in. So, your specific login form might ultimately look like this:

```
Listing 104-2 1  {# app/Resources/views/security/login.html.twig #}
2  {% if error %}
3      <div>{{ error.message }}</div>
4  {% endif %}
5
6  <form action="{{ path('login') }}" method="post">
7      <label for="username">Username:</label>
8      <input type="text" id="username" name="_username" value="{{ last_username }}" />
9
10     <label for="password">Password:</label>
11     <input type="password" id="password" name="_password" />
12
13     <input type="checkbox" id="remember_me" name="_remember_me" checked />
14     <label for="remember_me">Keep me logged in</label>
15
16     <input type="submit" name="login" />
17 </form>
```

The user will then automatically be logged in on subsequent visits while the cookie remains valid.

Forcing the User to Re-Authenticate before Accessing certain Resources

When the user returns to your site, they are authenticated automatically based on the information stored in the remember me cookie. This allows the user to access protected resources as if the user had actually authenticated upon visiting the site.

In some cases, however, you may want to force the user to actually re-authenticate before accessing certain resources. For example, you might allow "remember me" users to see basic account information, but then require them to actually re-authenticate before modifying that information.

The Security component provides an easy way to do this. In addition to roles explicitly assigned to them, users are automatically given one of the following roles depending on how they are authenticated:

`IS_AUTHENTICATED_ANONYMOUSLY`

Automatically assigned to a user who is in a firewall protected part of the site but who has not actually logged in. This is only possible if anonymous access has been allowed.

`IS_AUTHENTICATED_REMEMBERED`

Automatically assigned to a user who was authenticated via a remember me cookie.

`IS_AUTHENTICATED_FULLY`

Automatically assigned to a user that has provided their login details during the current session.

You can use these to control access beyond the explicitly assigned roles.



If you have the `IS_AUTHENTICATED_REMEMBERED` role, then you also have the `IS_AUTHENTICATED_ANONYMOUSLY` role. If you have the `IS_AUTHENTICATED_FULLY` role, then you also have the other two roles. In other words, these roles represent three levels of increasing "strength" of authentication.

You can use these additional roles for finer grained control over access to parts of a site. For example, you may want your user to be able to view their account at `/account` when authenticated by cookie but to have to provide their login details to be able to edit the account details. You can do this by securing specific controller actions using these roles. The edit action in the controller could be secured using the service context.

In the following example, the action is only allowed if the user has the `IS_AUTHENTICATED_FULLY` role.

```
Listing 104-3 1 // ...
2 use Symfony\Component\Security\Core\Exception\AccessDeniedException
3
4 // ...
5 public function editAction()
6 {
7     $this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');
8
9     // ...
10 }
```

If your application is based on the Symfony Standard Edition, you can also secure your controller using annotations:

```
Listing 104-4 1 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
2
3 /**
4  * @Security("has_role('IS_AUTHENTICATED_FULLY')")
5  */
6 public function editAction($name)
7 {
```

```
8 // ...  
9 }
```



If you also had an access control in your security configuration that required the user to have a **ROLE_USER** role in order to access any of the account area, then you'd have the following situation:

- If a non-authenticated (or anonymously authenticated user) tries to access the account area, the user will be asked to authenticate.
- Once the user has entered their username and password, assuming the user receives the **ROLE_USER** role per your configuration, the user will have the **IS_AUTHENTICATED_FULLY** role and be able to access any page in the account section, including the `editAction` controller.
- If the user's session ends, when the user returns to the site, they will be able to access every account page - except for the edit page - without being forced to re-authenticate. However, when they try to access the `editAction` controller, they will be forced to re-authenticate, since they are not, yet, fully authenticated.

For more information on securing services or methods in this way, see *How to Secure any Service or Method in your Application*.



Chapter 105

How to Impersonate a User

Sometimes, it's useful to be able to switch from one user to another without having to log out and log in again (for instance when you are debugging or trying to understand a bug a user sees that you can't reproduce).



User impersonation is not compatible with *pre authenticated firewalls*. The reason is that impersonation requires the authentication state to be maintained server-side, but pre-authenticated information (SSL_CLIENT_S_DN_Email, REMOTE_USER or other) is sent in each request.

Impersonating the user can be easily done by activating the **switch_user** firewall listener:

Listing 105-1

```
1 # app/config/security.yml
2 security:
3     # ...
4
5     firewalls:
6         main:
7             # ...
8             switch_user: true
```

To switch to another user, just add a query string with the **_switch_user** parameter and the username as the value to the current URL:

Listing 105-2

```
1 http://example.com/somewhere?_switch_user=thomas
```

To switch back to the original user, use the special **_exit** username:

Listing 105-3

```
1 http://example.com/somewhere?_switch_user=_exit
```

During impersonation, the user is provided with a special role called **ROLE_PREVIOUS_ADMIN**. In a template, for instance, this role can be used to show a link to exit impersonation:

Listing 105-4

```
1 {% if is_granted('ROLE_PREVIOUS_ADMIN') %}
2     <a href="{{ path('homepage', {'_switch_user': '_exit'}) }}">Exit impersonation</a>
3 {% endif %}
```

In some cases you may need to get the object that represents the impersonating user rather than the impersonated user. Use the following snippet to iterate over the user's roles until you find one that a **SwitchUserRole** object:

```
Listing 105-5 1 use Symfony\Component\Security\Core\Role\SwitchUserRole;
2
3 $authChecker = $this->get('security.authorization_checker');
4 $tokenStorage = $this->get('security.token_storage');
5
6 if ($authChecker->isGranted('ROLE_PREVIOUS_ADMIN')) {
7     foreach ($tokenStorage->getToken()->getRoles() as $role) {
8         if ($role instanceof SwitchUserRole) {
9             $impersonatingUser = $role->getSource()->getUser();
10            break;
11        }
12    }
13 }
```

Of course, this feature needs to be made available to a small group of users. By default, access is restricted to users having the **ROLE_ALLOWED_TO_SWITCH** role. The name of this role can be modified via the **role** setting. For extra security, you can also change the query parameter name via the **parameter** setting:

```
Listing 105-6 1 # app/config/security.yml
2 security:
3     # ...
4
5     firewalls:
6         main:
7             # ...
8             switch_user: { role: ROLE_ADMIN, parameter: _want_to_be_this_user }
```

Events

The firewall dispatches the **security.switch_user** event right after the impersonation is completed. The *SwitchUserEvent*¹ is passed to the listener, and you can use this to get the user that you are now impersonating.

The cookbook article about *Making the Locale "Sticky" during a User's Session* does not update the locale when you impersonate a user. The following code sample will show how to change the sticky locale:

```
Listing 105-7 1 # app/config/services.yml
2 services:
3     app.switch_user_listener:
4         class: AppBundle\EventListener\SwitchUserListener
5         tags:
6             - { name: kernel.event_listener, event: security.switch_user, method: onSwitchUser }
```



The listener implementation assumes your **User** entity has a **getLocale()** method.

```
Listing 105-8 1 // src/AppBundle/EventListener/SwitchUserListener.php
2 namespace AppBundle\EventListener;
3
4 use Symfony\Component\Security\Http\Event\SwitchUserEvent;
```

1. <http://api.symfony.com/3.0/Symfony/Component/Security/Http/Event/SwitchUserEvent.html>

```
5
6 class SwitchUserListener
7 {
8     public function onSwitchUser(SwitchUserEvent $event)
9     {
10         $event->getRequest()->getSession()->set(
11             '_locale',
12             $event->getTargetUser()->getLocale()
13         );
14     }
15 }
```



Chapter 106

How to Customize your Form Login

Using a *form login* for authentication is a common, and flexible, method for handling authentication in Symfony. Pretty much every aspect of the form login can be customized. The full, default configuration is shown in the next section.

Form Login Configuration Reference

To see the full form login configuration reference, see *SecurityBundle Configuration ("security")*. Some of the more interesting options are explained below.

Redirecting after Success

You can change where the login form redirects after a successful login using the various config options. By default the form will redirect to the URL the user requested (i.e. the URL which triggered the login form being shown). For example, if the user requested `http://www.example.com/admin/post/18/edit`, then after they successfully log in, they will eventually be sent back to `http://www.example.com/admin/post/18/edit`. This is done by storing the requested URL in the session. If no URL is present in the session (perhaps the user went directly to the login page), then the user is redirected to the default page, which is `/` (i.e. the homepage) by default. You can change this behavior in several ways.



As mentioned, by default the user is redirected back to the page originally requested. Sometimes, this can cause problems, like if a background Ajax request "appears" to be the last visited URL, causing the user to be redirected there. For information on controlling this behavior, see *How to Change the default Target Path Behavior*.

Changing the default Page

First, the default page can be set (i.e. the page the user is redirected to if no previous page was stored in the session). To set it to the `default_security_target` route use the following config:

Listing 106-1

```

1 # app/config/security.yml
2 security:
3   # ...
4
5   firewalls:
6     main:
7       form_login:
8         # ...
9         default_target_path: default_security_target

```

Now, when no URL is set in the session, users will be sent to the `default_security_target` route.

Always Redirect to the default Page

You can make it so that users are always redirected to the default page regardless of what URL they had requested previously by setting the `always_use_default_target_path` option to true:

Listing 106-2

```

1 # app/config/security.yml
2 security:
3   # ...
4
5   firewalls:
6     main:
7       form_login:
8         # ...
9         always_use_default_target_path: true

```

Using the Referring URL

In case no previous URL was stored in the session, you may wish to try using the `HTTP_REFERER` instead, as this will often be the same. You can do this by setting `use_referer` to true (it defaults to false):

Listing 106-3

```

1 # app/config/security.yml
2 security:
3   # ...
4
5   firewalls:
6     main:
7       # ...
8       form_login:
9         # ...
10        use_referer: true

```

Control the Redirect URL from inside the Form

You can also override where the user is redirected to via the form itself by including a hidden field with the name `_target_path`. For example, to redirect to the URL defined by some `account` route, use the following:

Listing 106-4

```

1 {% src AppBundle/Resources/views/Security/login.html.twig %}
2 {% if error %}
3   <div>{{ error.message }}</div>
4 {% endif %}
5
6 <form action="{{ path('login') }}" method="post">
7   <label for="username">Username:</label>
8   <input type="text" id="username" name="_username" value="{{ last_username }}" />
9
10  <label for="password">Password:</label>

```



```

11     <input type="password" id="password" name="_password" />
12     <input type="hidden" name="_target_path" value="account" />
13     <input type="submit" name="login" />
14 </form>

```

Now, the user will be redirected to the value of the hidden form field. The value attribute can be a relative path, absolute URL, or a route name. You can even change the name of the hidden form field by changing the `target_path_parameter` option to another value.

Listing 106-5

```

1 # app/config/security.yml
2 security:
3     # ...
4
5     firewalls:
6         main:
7             # ...
8             form_login:
9                 target_path_parameter: redirect_url

```

Redirecting on Login Failure

In addition to redirecting the user after a successful login, you can also set the URL that the user should be redirected to after a failed login (e.g. an invalid username or password was submitted). By default, the user is redirected back to the login form itself. You can set this to a different route (e.g. `login_failure`) with the following config:

Listing 106-6

```

1 # app/config/security.yml
2 security:
3     # ...
4
5     firewalls:
6         main:
7             # ...
8             form_login:
9                 # ...
10                failure_path: login_failure

```



Chapter 107

How to Create a custom User Provider

Part of Symfony's standard authentication process depends on "user providers". When a user submits a username and password, the authentication layer asks the configured user provider to return a user object for a given username. Symfony then checks whether the password of this user is correct and generates a security token so the user stays authenticated during the current session. Out of the box, Symfony has four user providers: `in_memory`, `entity`, `ldap` and `chain`. In this entry you'll see how you can create your own user provider, which could be useful if your users are accessed via a custom database, a file, or - as shown in this example - a web service.

Create a User Class

First, regardless of *where* your user data is coming from, you'll need to create a `User` class that represents that data. The `User` can look however you want and contain any data. The only requirement is that the class implements `UserInterface`¹. The methods in this interface should therefore be defined in the custom user class: `getRoles()`², `getPassword()`³, `getSalt()`⁴, `getUsername()`⁵, `eraseCredentials()`⁶. It may also be useful to implement the `EquatableInterface`⁷ interface, which defines a method to check if the user is equal to the current user. This interface requires an `isEqualTo()`⁸ method.

This is how your `WebuserServiceUser` class looks in action:

Listing 107-1

```
1 // src/AppBundle/Security/User/WebuserServiceUser.php
2 namespace AppBundle\Security\User;
3
4 use Symfony\Component\Security\Core\User\UserInterface;
```

-
1. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html>
 2. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html#method_getRoles
 3. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html#method_getPassword
 4. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html#method_getSalt
 5. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html#method_getUsername
 6. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html#method_eraseCredentials
 7. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/EquatableInterface.html>
 8. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/EquatableInterface.html#method_isEqualTo

```

5 use Symfony\Component\Security\Core\User\EquatableInterface;
6
7 class WebserviceUser implements UserInterface, EquatableInterface
8 {
9     private $username;
10    private $password;
11    private $salt;
12    private $roles;
13
14    public function __construct($username, $password, $salt, array $roles)
15    {
16        $this->username = $username;
17        $this->password = $password;
18        $this->salt = $salt;
19        $this->roles = $roles;
20    }
21
22    public function getRoles()
23    {
24        return $this->roles;
25    }
26
27    public function getPassword()
28    {
29        return $this->password;
30    }
31
32    public function getSalt()
33    {
34        return $this->salt;
35    }
36
37    public function getUsername()
38    {
39        return $this->username;
40    }
41
42    public function eraseCredentials()
43    {
44    }
45
46    public function isEqualTo(UserInterface $user)
47    {
48        if (!$user instanceof WebserviceUser) {
49            return false;
50        }
51
52        if ($this->password !== $user->getPassword()) {
53            return false;
54        }
55
56        if ($this->salt !== $user->getSalt()) {
57            return false;
58        }
59
60        if ($this->username !== $user->getUsername()) {
61            return false;
62        }
63
64        return true;
65    }
66 }

```

If you have more information about your users - like a "first name" - then you can add a **firstName** field to hold that data.

Create a User Provider

Now that you have a **User** class, you'll create a user provider, which will grab user information from some web service, create a **WebuserService** object, and populate it with data.

The user provider is just a plain PHP class that has to implement the *UserProviderInterface*⁹, which requires three methods to be defined: `loadUserByUsername($username)`, `refreshUser(UserInterface $user)`, and `supportsClass($class)`. For more details, see *UserProviderInterface*¹⁰.

Here's an example of how this might look:

```
Listing 107-2 1 // src/AppBundle/Security/User/WebuserServiceProvider.php
2 namespace AppBundle\Security\User;
3
4 use Symfony\Component\Security\Core\User\UserProviderInterface;
5 use Symfony\Component\Security\Core\User\UserInterface;
6 use Symfony\Component\Security\Core\Exception\UsernameNotFoundException;
7 use Symfony\Component\Security\Core\Exception\UnsupportedUserException;
8
9 class WebuserServiceProvider implements UserProviderInterface
10 {
11     public function loadUserByUsername($username)
12     {
13         // make a call to your webservice here
14         $userData = ...
15         // pretend it returns an array on success, false if there is no user
16
17         if ($userData) {
18             $password = '...';
19
20             // ...
21
22             return new WebuserService($username, $password, $salt, $roles);
23         }
24
25         throw new UsernameNotFoundException(
26             sprintf('Username "%s" does not exist.', $username)
27         );
28     }
29
30     public function refreshUser(UserInterface $user)
31     {
32         if (!$user instanceof WebuserService) {
33             throw new UnsupportedUserException(
34                 sprintf('Instances of "%s" are not supported.', get_class($user))
35             );
36         }
37
38         return $this->loadUserByUsername($user->getUsername());
39     }
40
41     public function supportsClass($class)
42     {
43         return $class === 'AppBundle\Security\User\WebuserService';
44     }
45 }
```

Create a Service for the User Provider

Now you make the user provider available as a service:

9. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserProviderInterface.html>

10. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserProviderInterface.html>

Listing 107-3

```
1 # app/config/services.yml
2 services:
3     app.webservice_user_provider:
4         class: AppBundle\Security\User\WebserviceUserProvider
```



The real implementation of the user provider will probably have some dependencies or configuration options or other services. Add these as arguments in the service definition.



Make sure the services file is being imported. See Importing Configuration with imports for details.

Modify security.yml

Everything comes together in your security configuration. Add the user provider to the list of providers in the "security" section. Choose a name for the user provider (e.g. "webservice") and mention the **id** of the service you just defined.

Listing 107-4

```
1 # app/config/security.yml
2 security:
3     # ...
4
5     providers:
6         webservice:
7             id: app.webservice_user_provider
```

Symfony also needs to know how to encode passwords that are supplied by website users, e.g. by filling in a login form. You can do this by adding a line to the "encoders" section in your security configuration:

Listing 107-5

```
1 # app/config/security.yml
2 security:
3     # ...
4
5     encoders:
6         AppBundle\Security\User\WebserviceUser: bcrypt
```

The value here should correspond with however the passwords were originally encoded when creating your users (however those users were created). When a user submits their password, it's encoded using this algorithm and the result is compared to the hashed password returned by your **getPassword()** method.



Specifics on how Passwords are Encoded

Symfony uses a specific method to combine the salt and encode the password before comparing it to your encoded password. If `getSalt()` returns nothing, then the submitted password is simply encoded using the algorithm you specify in `security.yml`. If a salt is specified, then the following value is created and *then* hashed via the algorithm:

Listing 107-6 `$password.'{'.$salt.'}'`

If your external users have their passwords salted via a different method, then you'll need to do a bit more work so that Symfony properly encodes the password. That is beyond the scope of this entry, but would include sub-classing `MessageDigestPasswordEncoder` and overriding the `mergePasswordAndSalt` method.

Additionally, you can configure the details of the algorithm used to hash passwords. In this example, the application sets explicitly the cost of the bcrypt hashing:

Listing 107-7

```
1 # app/config/security.yml
2 security:
3     # ...
4
5     encoders:
6         AppBundle\Security\User\WebServiceUser:
7             algorithm: bcrypt
8             cost: 12
```



Chapter 108

How to Create a Custom Form Password Authenticator



Check out *How to Create a Custom Authentication System with Guard* for a simpler and more flexible way to accomplish custom authentication tasks like this.

Imagine you want to allow access to your website only between 2pm and 4pm UTC. In this entry, you'll learn how to do this for a login form (i.e. where your user submits their username and password).

The Password Authenticator

First, create a new class that implements *SimpleFormAuthenticatorInterface*¹. Eventually, this will allow you to create custom logic for authenticating the user:

Listing 108-1

```
1 // src/Acme/HelloBundle/Security/TimeAuthenticator.php
2 namespace Acme\HelloBundle\Security;
3
4 use Symfony\Component\HttpFoundation\Request;
5 use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
6 use Symfony\Component\Security\Core\Authentication\Token\UsernamePasswordToken;
7 use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
8 use Symfony\Component\Security\Core\Exception\CustomUserMessageAuthenticationException;
9 use Symfony\Component\Security\Core\Exception\UsernameNotFoundException;
10 use Symfony\Component\Security\Core\User\UserProviderInterface;
11 use Symfony\Component\Security\Http\Authentication\SimpleFormAuthenticatorInterface;
12
13 class TimeAuthenticator implements SimpleFormAuthenticatorInterface
14 {
15     private $encoder;
16
17     public function __construct(UserPasswordEncoderInterface $encoder)
18     {
19         $this->encoder = $encoder;
```

1. <http://api.symfony.com/3.0/Symfony/Component/Security/Http/Authentication/SimpleFormAuthenticatorInterface.html>

```

20     }
21
22     public function authenticateToken(TokenInterface $token, UserProviderInterface $userProvider,
23     $providerKey)
24     {
25         try {
26             $user = $userProvider->loadUserByUsername($token->getUsername());
27         } catch (UsernameNotFoundException $e) {
28             // CAUTION: this message will be returned to the client
29             // (so don't put any un-trusted messages / error strings here)
30             throw new CustomUserMessageAuthenticationException('Invalid username or password');
31         }
32
33         $passwordValid = $this->encoder->isPasswordValid($user, $token->getCredentials());
34
35         if ($passwordValid) {
36             $currentHour = date('G');
37             if ($currentHour < 14 || $currentHour > 16) {
38                 // CAUTION: this message will be returned to the client
39                 // (so don't put any un-trusted messages / error strings here)
40                 throw new CustomUserMessageAuthenticationException(
41                     'You can only log in between 2 and 4!',
42                     100
43                 );
44             }
45
46             return new UsernamePasswordToken(
47                 $user,
48                 $user->getPassword(),
49                 $providerKey,
50                 $user->getRoles()
51             );
52         }
53
54         // CAUTION: this message will be returned to the client
55         // (so don't put any un-trusted messages / error strings here)
56         throw new CustomUserMessageAuthenticationException('Invalid username or password');
57     }
58
59     public function supportsToken(TokenInterface $token, $providerKey)
60     {
61         return $token instanceof UsernamePasswordToken
62             && $token->getProviderKey() === $providerKey;
63     }
64
65     public function createToken(Request $request, $username, $password, $providerKey)
66     {
67         return new UsernamePasswordToken($username, $password, $providerKey);
68     }
69 }

```

How it Works

Great! Now you just need to setup some Configuration. But first, you can find out more about what each method in this class does.

1) createToken

When Symfony begins handling a request, `createToken()` is called, where you create a *TokenInterface*² object that contains whatever information you need in `authenticateToken()` to authenticate the user (e.g. the username and password).

Whatever token object you create here will be passed to you later in `authenticateToken()`.

2. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Token/TokenInterface.html>

2) supportsToken

After Symfony calls `createToken()`, it will then call `supportsToken()` on your class (and any other authentication listeners) to figure out who should handle the token. This is just a way to allow several authentication mechanisms to be used for the same firewall (that way, you can for instance first try to authenticate the user via a certificate or an API key and fall back to a form login).

Mostly, you just need to make sure that this method returns `true` for a token that has been created by `createToken()`. Your logic should probably look exactly like this example.

3) authenticateToken

If `supportsToken` returns `true`, Symfony will now call `authenticateToken()`. Your job here is to check that the token is allowed to log in by first getting the `User` object via the user provider and then, by checking the password and the current time.



The "flow" of how you get the `User` object and determine whether or not the token is valid (e.g. checking the password), may vary based on your requirements.

Ultimately, your job is to return a *new* token object that is "authenticated" (i.e. it has at least 1 role set on it) and which has the `User` object inside of it.

Inside this method, the password encoder is needed to check the password's validity:

Listing 108-2

```
$passwordValid = $this->encoder->isPasswordValid($user, $token->getCredentials());
```

This is a service that is already available in Symfony and it uses the password algorithm that is configured in the security configuration (e.g. `security.yml`) under the `encoders` key. Below, you'll see how to inject that into the `TimeAuthenticator`.

Configuration

Now, configure your `TimeAuthenticator` as a service:

Listing 108-3

```
1 # app/config/config.yml
2 services:
3     # ...
4
5     time_authenticator:
6         class:      Acme\HelloBundle\Security\TimeAuthenticator
7         arguments:  ["@security.password_encoder"]
```

Then, activate it in the `firewalls` section of the security configuration using the `simple_form` key:

Listing 108-4

```
1 # app/config/security.yml
2 security:
3     # ...
4
5     firewalls:
6         secured_area:
7             pattern: ^/admin
8             # ...
9             simple_form:
10                 authenticator: time_authenticator
11                 check_path:    login_check
12                 login_path:    login
```

The **simple_form** key has the same options as the normal **form_login** option, but with the additional **authenticator** key that points to the new service. For details, see Form Login Configuration.

If creating a login form in general is new to you or you don't understand the **check_path** or **login_path** options, see *How to Customize your Form Login*.



Chapter 109

How to Authenticate Users with API Keys



Check out *How to Create a Custom Authentication System with Guard* for a simpler and more flexible way to accomplish custom authentication tasks like this.

Nowadays, it's quite usual to authenticate the user via an API key (when developing a web service for instance). The API key is provided for every request and is passed as a query string parameter or via an HTTP header.

The API Key Authenticator

Authenticating a user based on the Request information should be done via a pre-authentication mechanism. The *SimplePreAuthenticatorInterface*¹ allows you to implement such a scheme really easily.

Your exact situation may differ, but in this example, a token is read from an **apikey** query parameter, the proper username is loaded from that value and then a User object is created:

```
Listing 109-1 1 // src/AppBundle/Security/ApiKeyAuthenticator.php
2 namespace AppBundle\Security;
3
4 use Symfony\Component\HttpFoundation\Request;
5 use Symfony\Component\Security\Core\Authentication\Token\PreAuthenticatedToken;
6 use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
7 use Symfony\Component\Security\Core\Exception\AuthenticationException;
8 use Symfony\Component\Security\Core\Exception\CustomUserMessageAuthenticationException;
9 use Symfony\Component\Security\Core\Exception\BadCredentialsException;
10 use Symfony\Component\Security\Core\User\UserProviderInterface;
11 use Symfony\Component\Security\Http\Authentication\SimplePreAuthenticatorInterface;
12
13 class ApiKeyAuthenticator implements SimplePreAuthenticatorInterface
14 {
15     public function createToken(Request $request, $providerKey)
16     {
17         // look for an apikey query parameter
```

1. <http://api.symfony.com/3.0/Symfony/Component/Security/Http/Authentication/SimplePreAuthenticatorInterface.html>

```

18     $apiKey = $request->query->get('apikey');
19
20     // or if you want to use an "apikey" header, then do something like this:
21     // $apiKey = $request->headers->get('apikey');
22
23     if (!$apiKey) {
24         throw new BadCredentialsException();
25
26         // or to just skip api key authentication
27         // return null;
28     }
29
30     return new PreAuthenticatedToken(
31         'anon.',
32         $apiKey,
33         $providerKey
34     );
35 }
36
37 public function supportsToken(TokenInterface $token, $providerKey)
38 {
39     return $token instanceof PreAuthenticatedToken && $token->getProviderKey() === $providerKey;
40 }
41
42 public function authenticateToken(TokenInterface $token, UserProviderInterface $userProvider,
43 $providerKey)
44 {
45     if (!$userProvider instanceof ApiKeyUserProvider) {
46         throw new \InvalidArgumentException(
47             sprintf(
48                 'The user provider must be an instance of ApiKeyUserProvider (%s was given).',
49                 get_class($userProvider)
50             )
51         );
52     }
53
54     $apiKey = $token->getCredentials();
55     $username = $userProvider->getUsernameForApiKey($apiKey);
56
57     if (!$username) {
58         // CAUTION: this message will be returned to the client
59         // (so don't put any un-trusted messages / error strings here)
60         throw new CustomUserMessageAuthenticationException(
61             sprintf('API Key "%s" does not exist.', $apiKey)
62         );
63     }
64
65     $user = $userProvider->loadUserByUsername($username);
66
67     return new PreAuthenticatedToken(
68         $user,
69         $apiKey,
70         $providerKey,
71         $user->getRoles()
72     );
73 }

```

Once you've configured everything, you'll be able to authenticate by adding an `apikey` parameter to the query string, like `http://example.com/api/foo?apikey=37b51d194a7513e45b56f6524f2d51f2`.

The authentication process has several steps, and your implementation will probably differ:

1. createToken

Early in the request cycle, Symfony calls `createToken()`. Your job here is to create a token object that contains all of the information from the request that you need to authenticate the user (e.g. the `apikey`

query parameter). If that information is missing, throwing a *BadCredentialsException*² will cause authentication to fail. You might want to return `null` instead to just skip the authentication, so Symfony can fallback to another authentication method, if any.



In case you return `null` from your `createToken()` method, be sure to enable `anonymous` in your firewall. This way you'll be able to get an `AnonymousToken`.

2. supportsToken

After Symfony calls `createToken()`, it will then call `supportsToken()` on your class (and any other authentication listeners) to figure out who should handle the token. This is just a way to allow several authentication mechanisms to be used for the same firewall (that way, you can for instance first try to authenticate the user via a certificate or an API key and fall back to a form login).

Mostly, you just need to make sure that this method returns `true` for a token that has been created by `createToken()`. Your logic should probably look exactly like this example.

3. authenticateToken

If `supportsToken()` returns `true`, Symfony will now call `authenticateToken()`. One key part is the `$userProvider`, which is an external class that helps you load information about the user. You'll learn more about this next.

In this specific example, the following things happen in `authenticateToken()`:

1. First, you use the `$userProvider` to somehow look up the `$username` that corresponds to the `$apiKey`;
2. Second, you use the `$userProvider` again to load or create a `User` object for the `$username`;
3. Finally, you create an *authenticated token* (i.e. a token with at least one role) that has the proper roles and the `User` object attached to it.

The goal is ultimately to use the `$apiKey` to find or create a `User` object. *How* you do this (e.g. query a database) and the exact class for your `User` object may vary. Those differences will be most obvious in your user provider.

The User Provider

The `$userProvider` can be any user provider (see *How to Create a custom User Provider*). In this example, the `$apiKey` is used to somehow find the username for the user. This work is done in a `getUsernameForApiKey()` method, which is created entirely custom for this use-case (i.e. this isn't a method that's used by Symfony's core user provider system).

The `$userProvider` might look something like this:

Listing 109-2

```
1 // src/AppBundle/Security/ApiKeyUserProvider.php
2 namespace AppBundle\Security;
3
4 use Symfony\Component\Security\Core\User\UserProviderInterface;
5 use Symfony\Component\Security\Core\User\User;
6 use Symfony\Component\Security\Core\User\UserInterface;
7 use Symfony\Component\Security\Core\Exception\UnsupportedUserException;
8
9 class ApiKeyUserProvider implements UserProviderInterface
10 {
11     public function getUsernameForApiKey($apiKey)
12     {
```

2. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Exception/BadCredentialsException.html>

```

13     // Look up the username based on the token in the database, via
14     // an API call, or do something entirely different
15     $username = ...;
16
17     return $username;
18 }
19
20 public function loadUserByUsername($username)
21 {
22     return new User(
23         $username,
24         null,
25         // the roles for the user - you may choose to determine
26         // these dynamically somehow based on the user
27         array('ROLE_API')
28     );
29 }
30
31 public function refreshUser(UserInterface $user)
32 {
33     // this is used for storing authentication in the session
34     // but in this example, the token is sent in each request,
35     // so authentication can be stateless. Throwing this exception
36     // is proper to make things stateless
37     throw new UnsupportedUserException();
38 }
39
40 public function supportsClass($class)
41 {
42     return 'Symfony\Component\Security\Core\User\User' === $class;
43 }
44 }

```

Now register your user provider as a service:

Listing 109-3

```

1 # app/config/services.yml
2 services:
3     api_key_user_provider:
4         class: AppBundle\Security\ApiKeyUserProvider

```



Read the dedicated article to learn *how to create a custom user provider*.

The logic inside `getUsernameForApiKey()` is up to you. You may somehow transform the API key (e.g. `37b51d`) into a username (e.g. `jondoe`) by looking up some information in a "token" database table.

The same is true for `loadUserByUsername()`. In this example, Symfony's core *User*³ class is simply created. This makes sense if you don't need to store any extra information on your User object (e.g. `firstName`). But if you do, you may instead have your *own* user class which you create and populate here by querying a database. This would allow you to have custom data on the *User* object.

Finally, just make sure that `supportsClass()` returns `true` for User objects with the same class as whatever user you return in `loadUserByUsername()`.

If your authentication is stateless like in this example (i.e. you expect the user to send the API key with every request and so you don't save the login to the session), then you can simply throw the `UnsupportedUserException` exception in `refreshUser()`.

3. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/User.html>



If you *do* want to store authentication data in the session so that the key doesn't need to be sent on every request, see [Storing Authentication in the Session](#).

Handling Authentication Failure

In order for your `ApiKeyAuthenticator` to correctly display a 401 http status when either bad credentials or authentication fails you will need to implement the *`AuthenticationFailureHandlerInterface`*⁴ on your Authenticator. This will provide a method `onAuthenticationFailure` which you can use to create an error `Response`.

Listing 109-4

```

1  // src/AppBundle/Security/ApiKeyAuthenticator.php
2  namespace AppBundle\Security;
3
4  use Symfony\Component\Security\Core\Exception\AuthenticationException;
5  use Symfony\Component\Security\Http\Authentication\AuthenticationFailureHandlerInterface;
6  use Symfony\Component\Security\Http\Authentication\SimplePreAuthenticatorInterface;
7  use Symfony\Component\HttpFoundation\Response;
8  use Symfony\Component\HttpFoundation\Request;
9
10 class ApiKeyAuthenticator implements SimplePreAuthenticatorInterface, AuthenticationFailureHandlerInterface
11 {
12     // ...
13
14     public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
15     {
16         return new Response(
17             // this contains information about *why* authentication failed
18             // use it, or return your own message
19             strstr($exception->getMessageKey(), $exception->getMessageData()),
20             401
21         );
22     }
23 }
```

Configuration

Once you have your `ApiKeyAuthenticator` all setup, you need to register it as a service and use it in your security configuration (e.g. `security.yml`). First, register it as a service.

Listing 109-5

```

1  # app/config/config.yml
2  services:
3      # ...
4
5      apikey_authenticator:
6          class: AppBundle\Security\ApiKeyAuthenticator
7          public: false
```

Now, activate it and your custom user provider (see *How to Create a custom User Provider*) in the **firewalls** section of your security configuration using the `simple_preauth` and `provider` keys respectively:

Listing 109-6

```

1  # app/config/security.yml
2  security:
3      # ...
4
```

4. <http://api.symfony.com/3.0/Symfony/Component/Security/Http/Authentication/AuthenticationFailureHandlerInterface.html>

```

5     firewalls:
6         secured_area:
7             pattern: ^/api
8             stateless: true
9             simple_preauth:
10                authenticator: apikey_authenticator
11                provider: api_key_user_provider
12
13     providers:
14         api_key_user_provider:
15             id: api_key_user_provider

```

If you have defined `access_control`, make sure to add a new entry:

Listing 109-7

```

1  # app/config/security.yml
2  security:
3      # ...
4
5      access_control:
6          - { path: ^/api, roles: ROLE_API }

```

That's it! Now, your **ApiKeyAuthenticator** should be called at the beginning of each request and your authentication process will take place.

The `stateless` configuration parameter prevents Symfony from trying to store the authentication information in the session, which isn't necessary since the client will send the **apikey** on each request. If you *do* need to store authentication in the session, keep reading!

Storing Authentication in the Session

So far, this entry has described a situation where some sort of authentication token is sent on every request. But in some situations (like an OAuth flow), the token may be sent on only *one* request. In this case, you will want to authenticate the user and store that authentication in the session so that the user is automatically logged in for every subsequent request.

To make this work, first remove the `stateless` key from your firewall configuration or set it to **false**:

Listing 109-8

```

1  # app/config/security.yml
2  security:
3      # ...
4
5      firewalls:
6          secured_area:
7              pattern: ^/api
8              stateless: false
9              simple_preauth:
10                 authenticator: apikey_authenticator
11                 provider: api_key_user_provider
12
13     providers:
14         api_key_user_provider:
15             id: api_key_user_provider

```

Even though the token is being stored in the session, the credentials - in this case the API key (i.e. `$token->getCredentials()`) - are not stored in the session for security reasons. To take advantage of the session, update **ApiKeyAuthenticator** to see if the stored token has a valid User object that can be used:

Listing 109-9

```

1  // src/AppBundle/Security/ApiKeyAuthenticator.php
2
3  // ...

```



```

4 class ApiKeyAuthenticator implements SimplePreAuthenticatorInterface
5 {
6     // ...
7     public function authenticateToken(TokenInterface $token, UserProviderInterface $userProvider,
8     $providerKey)
9     {
10         if (!$userProvider instanceof ApiKeyUserProvider) {
11             throw new \InvalidArgumentException(
12                 sprintf(
13                     'The user provider must be an instance of ApiKeyUserProvider (%s was given).',
14                     get_class($userProvider)
15                 )
16             );
17         }
18
19         $apiKey = $token->getCredentials();
20         $username = $userProvider->getUsernameForApiKey($apiKey);
21
22         // User is the Entity which represents your user
23         $user = $token->getUser();
24         if ($user instanceof User) {
25             return new PreAuthenticatedToken(
26                 $user,
27                 $apiKey,
28                 $providerKey,
29                 $user->getRoles()
30             );
31         }
32
33         if (!$username) {
34             // this message will be returned to the client
35             throw new CustomUserMessageAuthenticationException(
36                 sprintf('API Key "%s" does not exist.', $apiKey)
37             );
38         }
39
40         $user = $userProvider->loadUserByUsername($username);
41
42         return new PreAuthenticatedToken(
43             $user,
44             $apiKey,
45             $providerKey,
46             $user->getRoles()
47         );
48     }
49     // ...
50 }

```

Storing authentication information in the session works like this:

1. At the end of each request, Symfony serializes the token object (returned from `authenticateToken()`), which also serializes the user object (since it's set on a property on the token);
2. On the next request the token is deserialized and the deserialized user object is passed to the `refreshUser()` function of the user provider.

The second step is the important one: Symfony calls `refreshUser()` and passes you the user object that was serialized in the session. If your users are stored in the database, then you may want to re-query for a fresh version of the user to make sure it's not out-of-date. But regardless of your requirements, `refreshUser()` should now return the User object:

Listing 109-10

```

1 // src/AppBundle/Security/ApiKeyUserProvider.php
2
3 // ...
4 class ApiKeyUserProvider implements UserProviderInterface
5 {
6     // ...
7
8     public function refreshUser(UserInterface $user)
9     {

```

```

10     // $user is the User that you set in the token inside authenticateToken()
11     // after it has been deserialized from the session
12
13     // you might use $user to query the database for a fresh user
14     // $id = $user->getId();
15     // use $id to make a query
16
17     // if you are *not* reading from a database and are just creating
18     // a User object (like in this example), you can just return it
19     return $user;
20 }
21 }

```



You'll also want to make sure that your **User** object is being serialized correctly. If your **User** object has private properties, PHP can't serialize those. In this case, you may get back a User object that has a **null** value for each property. For an example, see *How to Load Security Users from the Database (the Entity Provider)*.

Only Authenticating for Certain URLs

This entry has assumed that you want to look for the **apikey** authentication on *every* request. But in some situations (like an OAuth flow), you only really need to look for authentication information once the user has reached a certain URL (e.g. the redirect URL in OAuth).

Fortunately, handling this situation is easy: just check to see what the current URL is before creating the token in **createToken()**:

Listing 109-11

```

1  // src/AppBundle/Security/ApiKeyAuthenticator.php
2
3  // ...
4  use Symfony\Component\Security\Http\HttpUtils;
5  use Symfony\Component\HttpFoundation\Request;
6
7  class ApiKeyAuthenticator implements SimplePreAuthenticatorInterface
8  {
9      protected $httpUtils;
10
11      public function __construct(HttpUtils $httpUtils)
12      {
13          $this->httpUtils = $httpUtils;
14      }
15
16      public function createToken(Request $request, $providerKey)
17      {
18          // set the only URL where we should look for auth information
19          // and only return the token if we're at that URL
20          $targetUrl = '/login/check';
21          if (!$this->httpUtils->checkRequestPath($request, $targetUrl)) {
22              return;
23          }
24
25          // ...
26      }
27  }

```

This uses the handy *HttpUtils*⁵ class to check if the current URL matches the URL you're looking for. In this case, the URL (**/login/check**) has been hardcoded in the class, but you could also inject it as the second constructor argument.

Next, just update your service configuration to inject the **security.http_utils** service:

5. <http://api.symfony.com/3.0/Symfony/Component/Security/Http/HttpUtils.html>

Listing 109-12

```
1 # app/config/config.yml
2 services:
3   # ...
4
5   apikey_authenticator:
6     class: AppBundle\Security\ApiKeyAuthenticator
7     arguments: ["@security.http_utils"]
8     public: false
```

That's it! Have fun!



Chapter 110

How to Create a custom Authentication Provider



Creating a custom authentication system is hard, and this entry will walk you through that process. But depending on your needs, you may be able to solve your problem in a simpler, or via a community bundle:

- *How to Create a Custom Authentication System with Guard*
- *How to Create a Custom Form Password Authenticator*
- *How to Authenticate Users with API Keys*
- To authenticate via OAuth using a third-party service such as Google, Facebook or Twitter, try using the *HWIOAuthBundle*¹ community bundle.

If you have read the chapter on *Security*, you understand the distinction Symfony makes between authentication and authorization in the implementation of security. This chapter discusses the core classes involved in the authentication process, and how to implement a custom authentication provider. Because authentication and authorization are separate concepts, this extension will be user-provider agnostic, and will function with your application's user providers, may they be based in memory, a database, or wherever else you choose to store them.

Meet WSSE

The following chapter demonstrates how to create a custom authentication provider for WSSE authentication. The security protocol for WSSE provides several security benefits:

1. Username / Password encryption
2. Safe guarding against replay attacks
3. No web server configuration required

WSSE is very useful for the securing of web services, may they be SOAP or REST.

1. <https://github.com/hwi/HWIOAuthBundle>

There is plenty of great documentation on WSSE², but this article will focus not on the security protocol, but rather the manner in which a custom protocol can be added to your Symfony application. The basis of WSSE is that a request header is checked for encrypted credentials, verified using a timestamp and *nonce*³, and authenticated for the requested user using a password digest.



WSSE also supports application key validation, which is useful for web services, but is outside the scope of this chapter.

The Token

The role of the token in the Symfony security context is an important one. A token represents the user authentication data present in the request. Once a request is authenticated, the token retains the user's data, and delivers this data across the security context. First, you'll create your token class. This will allow the passing of all relevant information to your authentication provider.

Listing 110-1

```
1  // src/AppBundle/Security/Authentication/Token/WsseUserToken.php
2  namespace AppBundle\Security\Authentication\Token;
3
4  use Symfony\Component\Security\Core\Authentication\Token\AbstractToken;
5
6  class WsseUserToken extends AbstractToken
7  {
8      public $created;
9      public $digest;
10     public $nonce;
11
12     public function __construct(array $roles = array())
13     {
14         parent::__construct($roles);
15
16         // If the user has roles, consider it authenticated
17         $this->setAuthenticated(count($roles) > 0);
18     }
19
20     public function getCredentials()
21     {
22         return '';
23     }
24 }
```



The `WsseUserToken` class extends the Security component's *AbstractToken*⁴ class, which provides basic token functionality. Implement the *TokenInterface*⁵ on any class to use as a token.

The Listener

Next, you need a listener to listen on the firewall. The listener is responsible for fielding requests to the firewall and calling the authentication provider. A listener must be an instance of

2. <http://www.xml.com/pub/a/2003/12/17/dive.html>

3. https://en.wikipedia.org/wiki/Cryptographic_nonce

4. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Token/AbstractToken.html>

5. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Token/TokenInterface.html>

*ListenerInterface*⁶. A security listener should handle the *GetResponseEvent*⁷ event, and set an authenticated token in the token storage if successful.

```
Listing 110-2 1 // src/AppBundle\Security\Firewall\WsseListener.php
2 namespace AppBundle\Security\Firewall;
3
4 use Symfony\Component\HttpFoundation\Response;
5 use Symfony\Component\HttpFoundation\Event\GetResponseEvent;
6 use Symfony\Component\Security\Core\Authentication\AuthenticationManagerInterface;
7 use Symfony\Component\Security\Core\Authentication\Token\Storage\TokenStorageInterface;
8 use Symfony\Component\Security\Core\Exception\AuthenticationException;
9 use Symfony\Component\Security\Http\Firewall\ListenerInterface;
10 use AppBundle\Security\Authentication\Token\WsseUserToken;
11
12 class WsseListener implements ListenerInterface
13 {
14     protected $tokenStorage;
15     protected $authenticationManager;
16
17     public function __construct(TokenStorageInterface $tokenStorage, AuthenticationManagerInterface
18 $authenticationManager)
19     {
20         $this->tokenStorage = $tokenStorage;
21         $this->authenticationManager = $authenticationManager;
22     }
23
24     public function handle(GetResponseEvent $event)
25     {
26         $request = $event->getRequest();
27
28         $wsseRegex = '/UsernameToken Username="([^"]+)", PasswordDigest="([^"]+)", Nonce="([a-zA-Z0-9+\\
29 ]+={0,2})", Created="([^"]+)"/';
30         if (!$request->headers->has('x-wsse') || 1 !== preg_match($wsseRegex,
31 $request->headers->get('x-wsse'), $matches)) {
32             return;
33         }
34
35         $token = new WsseUserToken();
36         $token->setUser($matches[1]);
37
38         $token->digest = $matches[2];
39         $token->nonce = $matches[3];
40         $token->created = $matches[4];
41
42         try {
43             $authToken = $this->authenticationManager->authenticate($token);
44             $this->tokenStorage->setToken($authToken);
45
46             return;
47         } catch (AuthenticationException $failed) {
48             // ... you might log something here
49
50             // To deny the authentication clear the token. This will redirect to the login page.
51             // Make sure to only clear your token, not those of other authentication listeners.
52             // $token = $this->tokenStorage->getToken();
53             // if ($token instanceof WsseUserToken && $this->providerKey === $token->getProviderKey()) {
54             //     $this->tokenStorage->setToken(null);
55             // }
56             // return;
57         }
58
59         // By default deny authorization
60         $response = new Response();
61         $response->setStatusCode(Response::HTTP_FORBIDDEN);
62         $event->setResponse($response);
63     }
64 }
```

6. <http://api.symfony.com/3.0/Symfony/Component/Security/Http/Firewall/ListenerInterface.html>

7. <http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Event/GetResponseEvent.html>

This listener checks the request for the expected **X-WSSE** header, matches the value returned for the expected WSSE information, creates a token using that information, and passes the token on to the authentication manager. If the proper information is not provided, or the authentication manager throws an *AuthenticationException*⁸, a 403 Response is returned.



A class not used above, the *AbstractAuthenticationListener*⁹ class, is a very useful base class which provides commonly needed functionality for security extensions. This includes maintaining the token in the session, providing success / failure handlers, login form URLs, and more. As WSSE does not require maintaining authentication sessions or login forms, it won't be used for this example.



Returning prematurely from the listener is relevant only if you want to chain authentication providers (for example to allow anonymous users). If you want to forbid access to anonymous users and have a nice 403 error, you should set the status code of the response before returning.

The Authentication Provider

The authentication provider will do the verification of the **WsseUserToken**. Namely, the provider will verify the **Created** header value is valid within five minutes, the **Nonce** header value is unique within five minutes, and the **PasswordDigest** header value matches with the user's password.

Listing 110-3

```
1 // src/AppBundle\Security/Authentication/Provider/WsseProvider.php
2 namespace AppBundle\Security\Authentication\Provider;
3
4 use Symfony\Component\Security\Core\Authentication\Provider\AuthenticationProviderInterface;
5 use Symfony\Component\Security\Core\User\UserProviderInterface;
6 use Symfony\Component\Security\Core\Exception\AuthenticationException;
7 use Symfony\Component\Security\Core\Exception\NonceExpiredException;
8 use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
9 use AppBundle\Security\Authentication\Token\WsseUserToken;
10
11 class WsseProvider implements AuthenticationProviderInterface
12 {
13     private $userProvider;
14     private $cacheDir;
15
16     public function __construct(UserProviderInterface $userProvider, $cacheDir)
17     {
18         $this->userProvider = $userProvider;
19         $this->cacheDir = $cacheDir;
20     }
21
22     public function authenticate(TokenInterface $token)
23     {
24         $user = $this->userProvider->loadUserByUsername($token->getUsername());
25
26         if ($user && $this->validateDigest($token->digest, $token->nonce, $token->created,
27 $user->getPassword())) {
28             $authenticatedToken = new WsseUserToken($user->getRoles());
29             $authenticatedToken->setUser($user);
30
31             return $authenticatedToken;
32         }
33
34         throw new AuthenticationException('The WSSE authentication failed.');
```

8. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Exception/AuthenticationException.html>

9. <http://api.symfony.com/3.0/Symfony/Component/Security/Http/Firewall/AbstractAuthenticationListener.html>

```

37  /**
38   * This function is specific to Wsse authentication and is only used to help this example
39   *
40   * For more information specific to the logic here, see
41   * https://github.com/symfony/symfony-docs/pull/3134#issuecomment-27699129
42   */
43  protected function validateDigest($digest, $nonce, $created, $secret)
44  {
45      // Check created time is not in the future
46      if (strtotime($created) > time()) {
47          return false;
48      }
49
50      // Expire timestamp after 5 minutes
51      if (time() - strtotime($created) > 300) {
52          return false;
53      }
54
55      // Validate that the nonce is not used in the last 5 minutes
56      // if it has, this could be a replay attack
57      if (
58          file_exists($this->cacheDir.'/'.$md5($nonce))
59          && file_get_contents($this->cacheDir.'/'.$md5($nonce)) + 300 > time()
60      ) {
61          throw new NonceExpiredException('Previously used nonce detected');
62      }
63      // If cache directory does not exist we create it
64      if (!is_dir($this->cacheDir)) {
65          mkdir($this->cacheDir, 0777, true);
66      }
67      file_put_contents($this->cacheDir.'/'.$md5($nonce), time());
68
69      // Validate Secret
70      $expected = base64_encode(sha1(base64_decode($nonce).$created.$secret, true));
71
72      return hash_equals($expected, $digest);
73  }
74
75  public function supports(TokenInterface $token)
76  {
77      return $token instanceof WsseUserToken;
78  }

```



The *AuthenticationProviderInterface*¹⁰ requires an **authenticate** method on the user token, and a **supports** method, which tells the authentication manager whether or not to use this provider for the given token. In the case of multiple providers, the authentication manager will then move to the next provider in the list.



While the *hash_equals*¹¹ function was introduced in PHP 5.6, you are safe to use it with any PHP version in your Symfony application. In PHP versions prior to 5.6, *Symfony Polyfill*¹² (which is included in Symfony) will define the function for you.

The Factory

You have created a custom token, custom listener, and custom provider. Now you need to tie them all together. How do you make a unique provider available for every firewall? The answer is by using a

10. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Provider/AuthenticationProviderInterface.html>

11. <http://php.net/manual/en/function.hash-equals.php>

12. <https://github.com/symfony/polyfill>

factory. A factory is where you hook into the Security component, telling it the name of your provider and any configuration options available for it. First, you must create a class which implements *SecurityFactoryInterface*¹³.

Listing 110-4

```
1 // src/AppBundle/DependencyInjection/Security/Factory/WsseFactory.php
2 namespace AppBundle\DependencyInjection\Security\Factory;
3
4 use Symfony\Component\DependencyInjection\ContainerBuilder;
5 use Symfony\Component\DependencyInjection\Reference;
6 use Symfony\Component\DependencyInjection\DefinitionDecorator;
7 use Symfony\Component\Config\Definition\Builder\NodeDefinition;
8 use Symfony\Bundle\SecurityBundle\DependencyInjection\Security\Factory\SecurityFactoryInterface;
9
10 class WsseFactory implements SecurityFactoryInterface
11 {
12     public function create(ContainerBuilder $container, $id, $config, $userProvider, $defaultEntryPoint)
13     {
14         $providerId = 'security.authentication.provider.wsse.'.$id;
15         $container
16             ->setDefinition($providerId, new DefinitionDecorator('wsse.security.authentication.provider'))
17             ->replaceArgument(0, new Reference($userProvider))
18         ;
19
20         $listenerId = 'security.authentication.listener.wsse.'.$id;
21         $listener = $container->setDefinition($listenerId, new
22 DefinitionDecorator('wsse.security.authentication.listener'));
23
24         return array($providerId, $listenerId, $defaultEntryPoint);
25     }
26
27     public function getPosition()
28     {
29         return 'pre_auth';
30     }
31
32     public function getKey()
33     {
34         return 'wsse';
35     }
36
37     public function addConfiguration(NodeDefinition $node)
38     {
39     }
40 }
```

The *SecurityFactoryInterface*¹⁴ requires the following methods:

create

Method which adds the listener and authentication provider to the DI container for the appropriate security context.

getPosition

Returns when the provider should be called. This can be one of `pre_auth`, `form`, `http` or `remember_me`.

getKey

Method which defines the configuration key used to reference the provider in the firewall configuration.

addConfiguration

Method which is used to define the configuration options underneath the configuration key in your security configuration. Setting configuration options are explained later in this chapter.

13. <http://api.symfony.com/3.0/Symfony/Bundle/SecurityBundle/DependencyInjection/Security/Factory/SecurityFactoryInterface.html>

14. <http://api.symfony.com/3.0/Symfony/Bundle/SecurityBundle/DependencyInjection/Security/Factory/SecurityFactoryInterface.html>



A class not used in this example, *AbstractFactory*¹⁵, is a very useful base class which provides commonly needed functionality for security factories. It may be useful when defining an authentication provider of a different type.

Now that you have created a factory class, the **wsse** key can be used as a firewall in your security configuration.



You may be wondering "why do you need a special factory class to add listeners and providers to the dependency injection container?". This is a very good question. The reason is you can use your firewall multiple times, to secure multiple parts of your application. Because of this, each time your firewall is used, a new service is created in the DI container. The factory is what creates these new services.

Configuration

It's time to see your authentication provider in action. You will need to do a few things in order to make this work. The first thing is to add the services above to the DI container. Your factory class above makes reference to service ids that do not exist yet: **wsse.security.authentication.provider** and **wsse.security.authentication.listener**. It's time to define those services.

Listing 110-5

```

1  # app/config/services.yml
2  services:
3      wsse.security.authentication.provider:
4          class: AppBundle\Security\Authentication\Provider\WsseProvider
5          arguments:
6              - '' # User Provider
7              - '%kernel.cache_dir%/security/nonces'
8          public: false
9
10     wsse.security.authentication.listener:
11         class: AppBundle\Security\Firewall\WsseListener
12         arguments: ['@security.token_storage', '@security.authentication.manager']
13         public: false

```

Now that your services are defined, tell your security context about your factory in your bundle class:

Listing 110-6

```

1  // src/AppBundle/AppBundle.php
2  namespace AppBundle;
3
4  use AppBundle\DependencyInjection\Security\Factory\WsseFactory;
5  use Symfony\Component\HttpKernel\Bundle\Bundle;
6  use Symfony\Component\DependencyInjection\ContainerBuilder;
7
8  class AppBundle extends Bundle
9  {
10     public function build(ContainerBuilder $container)
11     {
12         parent::build($container);
13
14         $extension = $container->getExtension('security');
15         $extension->addSecurityListenerFactory(new WsseFactory());
16     }
17 }

```

You are finished! You can now define parts of your app as under WSSE protection.

Listing 110-7

15. <http://api.symfony.com/3.0/Symfony/Bundle/SecurityBundle/DependencyInjection/Security/Factory/AbstractFactory.html>

```

1 # app/config/security.yml
2 security:
3   # ...
4
5   firewalls:
6     wsse_secured:
7       pattern: ^/api/
8       stateless: true
9       wsse: true

```

Congratulations! You have written your very own custom security authentication provider!

A little Extra

How about making your WSSE authentication provider a bit more exciting? The possibilities are endless. Why don't you start by adding some sparkle to that shine?

Configuration

You can add custom options under the **wsse** key in your security configuration. For instance, the time allowed before expiring the **Created** header item, by default, is 5 minutes. Make this configurable, so different firewalls can have different timeout lengths.

You will first need to edit **WsseFactory** and define the new option in the **addConfiguration** method.

Listing 110-8

```

1 class WsseFactory implements SecurityFactoryInterface
2 {
3   // ...
4
5   public function addConfiguration(NodeDefinition $node)
6   {
7     $node
8       ->children()
9       ->scalarNode('lifetime')->defaultValue(300)
10      ->end();
11  }
12 }

```

Now, in the **create** method of the factory, the **\$config** argument will contain a **lifetime** key, set to 5 minutes (300 seconds) unless otherwise set in the configuration. Pass this argument to your authentication provider in order to put it to use.

Listing 110-9

```

1 class WsseFactory implements SecurityFactoryInterface
2 {
3   public function create(ContainerBuilder $container, $id, $config, $userProvider, $defaultEntryPoint)
4   {
5     $providerId = 'security.authentication.provider.wsse.'.$id;
6     $container
7       ->setDefinition($providerId,
8         new DefinitionDecorator('wsse.security.authentication.provider'))
9       ->replaceArgument(0, new Reference($userProvider))
10      ->replaceArgument(2, $config['lifetime']);
11     // ...
12   }
13
14   // ...
15 }

```



You'll also need to add a third argument to the `wsse.security.authentication.provider` service configuration, which can be blank, but will be filled in with the lifetime in the factory. The `WsseProvider` class will also now need to accept a third constructor argument - the lifetime - which it should use instead of the hard-coded 300 seconds. These two steps are not shown here.

The lifetime of each WSSE request is now configurable, and can be set to any desirable value per firewall.

Listing 110-10

```
1 # app/config/security.yml
2 security:
3   # ...
4
5   firewalls:
6     wsse_secured:
7       pattern: ^/api/
8       stateless: true
9       wsse: { lifetime: 30 }
```

The rest is up to you! Any relevant configuration items can be defined in the factory and consumed or passed to the other classes in the container.



Chapter 111

Using pre Authenticated Security Firewalls

A lot of authentication modules are already provided by some web servers, including Apache. These modules generally set some environment variables that can be used to determine which user is accessing your application. Out of the box, Symfony supports most authentication mechanisms. These requests are called *pre authenticated* requests because the user is already authenticated when reaching your application.



User impersonation is not compatible with pre-authenticated firewalls. The reason is that impersonation requires the authentication state to be maintained server-side, but pre-authenticated information (`SSL_CLIENT_S_DN_Email`, `REMOTE_USER` or other) is sent in each request.

X.509 Client Certificate Authentication

When using client certificates, your webserver is doing all the authentication process itself. With Apache, for example, you would use the `SSLVerifyClient Require` directive.

Enable the x509 authentication for a particular firewall in the security configuration:

Listing 111-1

```
1 # app/config/security.yml
2 security:
3     # ...
4
5     firewalls:
6         secured_area:
7             pattern: ^/
8             x509:
9                 provider: your_user_provider
```

By default, the firewall provides the `SSL_CLIENT_S_DN_Email` variable to the user provider, and sets the `SSL_CLIENT_S_DN` as credentials in the *PreAuthenticatedToken*¹. You can override these by setting the `user` and the `credentials` keys in the x509 firewall configuration respectively.

1. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authentication/Token/PreAuthenticatedToken.html>



An authentication provider will only inform the user provider of the username that made the request. You will need to create (or use) a "user provider" that is referenced by the **provider** configuration parameter (**your_user_provider** in the configuration example). This provider will turn the username into a User object of your choice. For more information on creating or configuring a user provider, see:

- *How to Create a custom User Provider*
- *How to Load Security Users from the Database (the Entity Provider)*

REMOTE_USER Based Authentication

A lot of authentication modules, like **auth_kerb** for Apache provide the username using the **REMOTE_USER** environment variable. This variable can be trusted by the application since the authentication happened before the request reached it.

To configure Symfony using the **REMOTE_USER** environment variable, simply enable the corresponding firewall in your security configuration:

Listing 111-2

```
1 # app/config/security.yml
2 security:
3     firewalls:
4         secured_area:
5             pattern: ^/
6             remote_user:
7                 provider: your_user_provider
```

The firewall will then provide the **REMOTE_USER** environment variable to your user provider. You can change the variable name used by setting the **user** key in the **remote_user** firewall configuration.



Just like for X509 authentication, you will need to configure a "user provider". See the previous note for more information.



Chapter 112

How to Change the default Target Path Behavior

By default, the Security component retains the information of the last request URI in a session variable named `_security.main.target_path` (with `main` being the name of the firewall, defined in `security.yml`). Upon a successful login, the user is redirected to this path, as to help them continue from the last known page they visited.

In some situations, this is not ideal. For example, when the last request URI was an XMLHttpRequest which returned a non-HTML or partial HTML response, the user is redirected back to a page which the browser cannot render.

To get around this behavior, you would simply need to extend the `ExceptionListener` class and override the default method named `setTargetPath()`.

First, override the `security.exception_listener.class` parameter in your configuration file. This can be done from your main configuration file (in `app/config`) or from a configuration file being imported from a bundle:

Listing 112-1

```
1 # app/config/services.yml
2 parameters:
3     # ...
4     security.exception_listener.class: AppBundle\Security\Firewall\ExceptionListener
```

Next, create your own `ExceptionListener`:

Listing 112-2

```
1 // src/AppBundle/Security/Firewall/ExceptionListener.php
2 namespace AppBundle\Security\Firewall;
3
4 use Symfony\Component\HttpFoundation\Request;
5 use Symfony\Component\Security\Http\Firewall\ExceptionListener as BaseExceptionListener;
6
7 class ExceptionListener extends BaseExceptionListener
8 {
9     protected function setTargetPath(Request $request)
10     {
11         // Do not save target path for XHR requests
12         // You can add any more logic here you want
13     }
14 }
```

```
13      // Note that non-GET requests are already ignored
14      if ($request->isXmlHttpRequest()) {
15          return;
16      }
17
18      parent::setTargetPath($request);
19  }
20 }
```

Add as much or as little logic here as required for your scenario!



Chapter 113

Using CSRF Protection in the Login Form

When using a login form, you should make sure that you are protected against CSRF (*Cross-site request forgery*¹). The Security component already has built-in support for CSRF. In this article you'll learn how you can use it in your login form.



Login CSRF attacks are a bit less well-known. See *Forging Login Requests*² if you're curious about more details.

Configuring CSRF Protection

First, configure the Security component so it can use CSRF protection. The Security component needs a CSRF token provider. You can set this to use the default provider available in the Security component:

Listing 113-1

```
1 # app/config/security.yml
2 security:
3   # ...
4
5   firewalls:
6     secured_area:
7       # ...
8       form_login:
9         # ...
10        csrf_token_generator: security.csrf.token_manager
```

The Security component can be configured further, but this is all information it needs to be able to use CSRF in the login form.

1. https://en.wikipedia.org/wiki/Cross-site_request_forgery

2. https://en.wikipedia.org/wiki/Cross-site_request_forgery#Forging_login_requests

Rendering the CSRF field

Now that Security component will check for the CSRF token, you have to add a *hidden* field to the login form containing the CSRF token. By default, this field is named `_csrf_token`. That hidden field must contain the CSRF token, which can be generated by using the `csrf_token` function. That function requires a token ID, which must be set to `authenticate` when using the login form:

Listing 113-2

```
1  {% src/AppBundle/Resources/views/Security/login.html.twig %}  
2  
3  {% ... %}  
4  <form action="{{ path('login') }}" method="post">  
5      {% ... the login fields %}  
6  
7      <input type="hidden" name="_csrf_token"  
8          value="{{ csrf_token('authenticate') }}">  
9      >  
10  
11     <button type="submit">login</button>  
12 </form>
```

After this, you have protected your login form against CSRF attacks.



You can change the name of the field by setting `csrf_parameter` and change the token ID by setting `csrf_token_id` in your configuration:

Listing 113-3

```
1  # app/config/security.yml  
2  security:  
3      # ...  
4  
5      firewalls:  
6          secured_area:  
7              # ...  
8              form_login:  
9                  # ...  
10                 csrf_parameter: _csrf_security_token  
11                 csrf_token_id: a_private_string
```



Chapter 114

How to Choose the Password Encoder Algorithm Dynamically

Usually, the same password encoder is used for all users by configuring it to apply to all instances of a specific class:

Listing 114-1

```
1 # app/config/security.yml
2 security:
3     # ...
4     encoders:
5         Symfony\Component\Security\Core\User\User: sha512
```

Another option is to use a "named" encoder and then select which encoder you want to use dynamically. In the previous example, you've set the **sha512** algorithm for **Acme\UserBundle\Entity\User**. This may be secure enough for a regular user, but what if you want your admins to have a stronger algorithm, for example **bcrypt**. This can be done with named encoders:

Listing 114-2

```
1 # app/config/security.yml
2 security:
3     # ...
4     encoders:
5         harsh:
6             algorithm: bcrypt
7             cost: 15
```

This creates an encoder named **harsh**. In order for a **User** instance to use it, the class must implement *EncoderAwareInterface*¹. The interface requires one method - **getEncoderName** - which should return the name of the encoder to use:

Listing 114-3

```
1 // src/Acme/UserBundle/Entity/User.php
2 namespace Acme\UserBundle\Entity;
3
4 use Symfony\Component\Security\Core\User\UserInterface;
5 use Symfony\Component\Security\Core\Encoder\EncoderAwareInterface;
6
```

1. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Encoder/EncoderAwareInterface.html>

```
7 class User implements UserInterface, EncoderAwareInterface
8 {
9     public function getEncoderName()
10    {
11        if ($this->isAdmin()) {
12            return 'harsh';
13        }
14
15        return null; // use the default encoder
16    }
17 }
```



Chapter 115

How to Use multiple User Providers

Each authentication mechanism (e.g. HTTP Authentication, form login, etc) uses exactly one user provider, and will use the first declared user provider by default. But what if you want to specify a few users via configuration and the rest of your users in the database? This is possible by creating a new provider that chains the two together:

Listing 115-1

```
1 # app/config/security.yml
2 security:
3   providers:
4     chain_provider:
5       chain:
6         providers: [in_memory, user_db]
7     in_memory:
8       memory:
9         users:
10          foo: { password: test }
11     user_db:
12       entity: { class: AppBundle\Entity\User, property: username }
```

Now, all authentication mechanisms will use the **chain_provider**, since it's the first specified. The **chain_provider** will, in turn, try to load the user from both the **in_memory** and **user_db** providers.

You can also configure the firewall or individual authentication mechanisms to use a specific provider. Again, unless a provider is specified explicitly, the first provider is always used:

Listing 115-2

```
1 # app/config/security.yml
2 security:
3   firewalls:
4     secured_area:
5       # ...
6       pattern: ^/
7       provider: user_db
8       http_basic:
9         realm: 'Secured Demo Area'
10      provider: in_memory
11     form_login: ~
```

In this example, if a user tries to log in via HTTP authentication, the authentication system will use the **in_memory** user provider. But if the user tries to log in via the form login, the **user_db** provider will be used (since it's the default for the firewall as a whole).

For more information about user provider and firewall configuration, see the *SecurityBundle Configuration* ("security").



Chapter 116

How to Use Multiple Guard Authenticators

The Guard authentication component allows you to easily use many different authenticators at a time.

An entry point is a service id (of one of your authenticators) whose `start()` method is called to start the authentication process.

Multiple Authenticators with Shared Entry Point

Sometimes you want to offer your users different authentication mechanisms like a form login and a Facebook login while both entry points redirect the user to the same login page. However, in your configuration you have to explicitly say which entry point you want to use.

This is how your security configuration can look in action:

```
Listing 116-1 1 # app/config/security.yml
                2 security:
                3     # ...
                4     firewalls:
                5         default:
                6             anonymous: ~
                7             guard:
                8                 authenticators:
                9                     - app.form_login_authenticator
               10                     - app.facebook_connect_authenticator
               11             entry_point: app.form_login_authenticator
```

There is one limitation with this approach - you have to use exactly one entry point.

Multiple Authenticators with Separate Entry Points

However, there are use cases where you have authenticators that protect different parts of your application. For example, you have a login form that protects the secured area of your application front-end and API end points that are protected with API tokens. As you can only configure one entry point per firewall, the solution is to split the configuration into two separate firewalls:

Listing 116-2

```
1 # app/config/security.yml
2 security:
3     # ...
4     firewalls:
5         api:
6             pattern: ^/api/
7             guard:
8                 authenticators:
9                     - app.api_token_authenticator
10        default:
11            anonymous: ~
12            guard:
13                authenticators:
14                    - app.form_login_authenticator
15    access_control:
16        - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
17        - { path: ^/api, roles: ROLE_API_USER }
18        - { path: ^/, roles: ROLE_USER }
```




Chapter 117

How to Restrict Firewalls to a Specific Request

When using the Security component, you can create firewalls that match certain request options. In most cases, matching against the URL is sufficient, but in special cases you can further restrict the initialization of a firewall against other options of the request.



You can use any of these restrictions individually or mix them together to get your desired firewall configuration.

Restricting by Pattern

This is the default restriction and restricts a firewall to only be initialized if the request URL matches the configured **pattern**.

Listing 117-1

```
1 # app/config/security.yml
2
3 # ...
4 security:
5     firewalls:
6         secured_area:
7             pattern: ^/admin
8             # ...
```

The **pattern** is a regular expression. In this example, the firewall will only be activated if the URL starts (due to the **^** regex character) with **/admin**. If the URL does not match this pattern, the firewall will not be activated and subsequent firewalls will have the opportunity to be matched for this request.

Restricting by Host

If matching against the **pattern** only is not enough, the request can also be matched against **host**. When the configuration option **host** is set, the firewall will be restricted to only initialize if the host from the request matches against the configuration.

Listing 117-2

```
1 # app/config/security.yml
2
3 # ...
4 security:
5     firewalls:
6         secured_area:
7             host: ^admin\.example\.com$
8             # ...
```

The **host** (like the **pattern**) is a regular expression. In this example, the firewall will only be activated if the host is equal exactly (due to the **^** and **\$** regex characters) to the hostname **admin.example.com**. If the hostname does not match this pattern, the firewall will not be activated and subsequent firewalls will have the opportunity to be matched for this request.

Restricting by HTTP Methods

The configuration option **methods** restricts the initialization of the firewall to the provided HTTP methods.

Listing 117-3

```
1 # app/config/security.yml
2
3 # ...
4 security:
5     firewalls:
6         secured_area:
7             methods: [GET, POST]
8             # ...
```

In this example, the firewall will only be activated if the HTTP method of the request is either **GET** or **POST**. If the method is not in the array of the allowed methods, the firewall will not be activated and subsequent firewalls will again have the opportunity to be matched for this request.



Chapter 118

How to Restrict Firewalls to a Specific Host

As of Symfony 2.5, more possibilities to restrict firewalls have been added. You can read everything about all the possibilities (including `host`) in "*How to Restrict Firewalls to a Specific Request*".



Chapter 119

How to Create and Enable Custom User Checkers

During the authentication of a user, additional checks might be required to verify if the identified user is allowed to log in. By defining a custom user checker, you can define per firewall which checker should be used.

Creating a Custom User Checker

User checkers are classes that must implement the *UserCheckerInterface*¹. This interface defines two methods called `checkPreAuth()` and `checkPostAuth()` to perform checks before and after user authentication. If one or more conditions are not met, an exception should be thrown which extends the *AccountStatusException*².

Listing 119-1

```
1 namespace AppBundle\Security;
2
3 use AppBundle\Exception\AccountDeletedException;
4 use AppBundle\Security\User as AppUser;
5 use Symfony\Component\Security\Core\Exception\AccountExpiredException;
6 use Symfony\Component\Security\Core\User\UserCheckerInterface;
7 use Symfony\Component\Security\Core\User\UserInterface;
8
9 class UserChecker implements UserCheckerInterface
10 {
11     public function checkPreAuth(UserInterface $user)
12     {
13         if (!$user instanceof AppUser) {
14             return;
15         }
16
17         // user is deleted, show a generic Account Not Found message.
18         if ($user->isDeleted()) {
19             throw new AccountDeletedException('...');
```

1. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/UserCheckerInterface.html>

2. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Exception/AccountStatusException.html>

```

20     }
21 }
22
23 public function checkPostAuth(UserInterface $user)
24 {
25     if (!$user instanceof AppUser) {
26         return;
27     }
28
29     // user account is expired, the user may be notified
30     if ($user->isExpired()) {
31         throw new AccountExpiredException('...');
32     }
33 }
34 }

```

Enabling the Custom User Checker

All that's left to be done is creating a service definition and configuring this in the firewall configuration. Configuring the service is done like any other service:

Listing 119-2

```

1 # app/config/services.yml
2 services:
3     app.user_checker:
4         class: AppBundle\Security\UserChecker

```

All that's left to do is add the checker to the desired firewall where the value is the service id of your user checker:

Listing 119-3

```

1 # app/config/security.yml
2
3 # ...
4 security:
5     firewalls:
6         secured_area:
7             pattern: ^/
8             user_checker: app.user_checker
9             # ...

```

Additional Configurations

It's possible to have a different user checker per firewall.

Listing 119-4

```

1 # app/config/security.yml
2
3 # ...
4 security:
5     firewalls:
6         admin:
7             pattern: ^/admin
8             user_checker: app.admin_user_checker
9             # ...
10     secured_area:
11         pattern: ^/
12         user_checker: app.user_checker

```



Internally the user checkers are aliased per firewall. For `secured_area` the alias `security.user_checker.secured_area` would point to `app.user_checker`.



Chapter 120

How to Use Voters to Check User Permissions

In Symfony, you can check the permission to access data by using the *ACL module*, which is a bit overwhelming for many applications. A much easier solution is to work with custom voters, which are like simple conditional statements.



Take a look at the *authorization* chapter for an even deeper understanding on voters.

How Symfony Uses Voters

In order to use voters, you have to understand how Symfony works with them. All voters are called each time you use the `isGranted()` method on Symfony's authorization checker (i.e. the `security.authorization_checker` service). Each one decides if the current user should have access to some resource.

Ultimately, Symfony takes the responses from all voters and makes the final decision (to allow or deny access to the resource) according to the strategy defined in the application, which can be: affirmative, consensus or unanimous.

For more information take a look at the section about access decision managers.

The Voter Interface

A custom voter needs to implement *VoterInterface*¹ or extend *Voter*², which makes creating a voter even easier.

Listing 120-1

```
1 abstract class Voter implements VoterInterface
2 {
3     abstract protected function supports($attribute, $subject);
```

-
1. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/Voter/VoterInterface.html>
 2. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/Voter/Voter.html>

```

4     abstract protected function voteOnAttribute($attribute, $subject, TokenInterface $token);
5 }

```

Setup: Checking for Access in a Controller

Suppose you have a **Post** object and you need to decide whether or not the current user can *edit* or *view* the object. In your controller, you'll check access with code like this:

Listing 120-2

```

1  // src/AppBundle/Controller/PostController.php
2  // ...
3
4  class PostController extends Controller
5  {
6      /**
7       * @Route("/posts/{id}", name="post_show")
8       */
9      public function showAction($id)
10     {
11         // get a Post object - e.g. query for it
12         $post = ...;
13
14         // check for "view" access: calls all voters
15         $this->denyAccessUnlessGranted('view', $post);
16
17         // ...
18     }
19
20     /**
21      * @Route("/posts/{id}/edit", name="post_edit")
22      */
23     public function editAction($id)
24     {
25         // get a Post object - e.g. query for it
26         $post = ...;
27
28         // check for "edit" access: calls all voters
29         $this->denyAccessUnlessGranted('edit', $post);
30
31         // ...
32     }
33 }

```

The `denyAccessUnlessGranted()` method (and also, the simpler `isGranted()` method) calls out to the "voter" system. Right now, no voters will vote on whether or not the user can "view" or "edit" a **Post**. But you can create your *own* voter that decides this using whatever logic you want.



The `denyAccessUnlessGranted()` function and the `isGranted()` functions are both just shortcuts to call `isGranted()` on the `security.authorization_checker` service.

Creating the custom Voter

Suppose the logic to decide if a user can "view" or "edit" a **Post** object is pretty complex. For example, a **User** can always edit or view a **Post** they created. And if a **Post** is marked as "public", anyone can view it. A voter for this situation would look like this:

Listing 120-3

```

1  // src/AppBundle/Security/PostVoter.php
2  namespace AppBundle\Security;

```

```

3
4 use AppBundle\Entity\Post;
5 use AppBundle\Entity\User;
6 use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
7 use Symfony\Component\Security\Core\Authorization\Voter\Voter;
8
9 class PostVoter extends Voter
10 {
11     // these strings are just invented: you can use anything
12     const VIEW = 'view';
13     const EDIT = 'edit';
14
15     protected function supports($attribute, $subject)
16     {
17         // if the attribute isn't one we support, return false
18         if (!in_array($attribute, array(self::VIEW, self::EDIT))) {
19             return false;
20         }
21
22         // only vote on Post objects inside this voter
23         if (!$subject instanceof Post) {
24             return false;
25         }
26
27         return true;
28     }
29
30     protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
31     {
32         $user = $token->getUser();
33
34         if (!$user instanceof User) {
35             // the user must be logged in; if not, deny access
36             return false;
37         }
38
39         // you know $subject is a Post object, thanks to supports
40         /** @var Post $post */
41         $post = $subject;
42
43         switch ($attribute) {
44             case self::VIEW:
45                 return $this->canView($post, $user);
46             case self::EDIT:
47                 return $this->canEdit($post, $user);
48         }
49
50         throw new \LogicException('This code should not be reached!');
51     }
52
53     private function canView(Post $post, User $user)
54     {
55         // if they can edit, they can view
56         if ($this->canEdit($post, $user)) {
57             return true;
58         }
59
60         // the Post object could have, for example, a method isPrivate()
61         // that checks a boolean $private property
62         return !$post->isPrivate();
63     }
64
65     private function canEdit(Post $post, User $user)
66     {
67         // this assumes that the data object has a getOwner() method
68         // to get the entity of the user who owns this data object
69         return $user === $post->getOwner();
70     }
71 }

```

That's it! The voter is done! Next, configure it.

To recap, here's what's expected from the two abstract methods:

Voter::supports(\$attribute, \$subject)

When `isGranted()` (or `denyAccessUnlessGranted()`) is called, the first argument is passed here as `$attribute` (e.g. `ROLE_USER`, `edit`) and the second argument (if any) is passed as `$subject` (e.g. `null`, a `Post` object). Your job is to determine if your voter should vote on the attribute/subject combination. If you return `true`, `voteOnAttribute()` will be called. Otherwise, your voter is done: some other voter should process this. In this example, you return `true` if the attribute is `view` or `edit` and if the object is a `Post` instance.

voteOnAttribute(\$attribute, \$subject, TokenInterface \$token)

If you return `true` from `supports()`, then this method is called. Your job is simple: return `true` to allow access and `false` to deny access. The `$token` can be used to find the current user object (if any). In this example, all of the complex business logic is included to determine access.

Configuring the Voter

To inject the voter into the security layer, you must declare it as a service and tag it with `security.voter`:

Listing 120-4

```
1 # app/config/services.yml
2 services:
3     app.post_voter:
4         class: AppBundle\Security\PostVoter
5         tags:
6             - { name: security.voter }
7         # small performance boost
8         public: false
```

You're done! Now, when you call `isGranted()` with `view/edit` and a `Post` object, your voter will be executed and you can control access.

Checking for Roles inside a Voter

What if you want to call `isGranted()` from *inside* your voter - e.g. you want to see if the current user has `ROLE_SUPER_ADMIN`. That's possible by injecting the *AccessDecisionManager*³ into your voter. You can use this to, for example, *always* allow access to a user with `ROLE_SUPER_ADMIN`:

Listing 120-5

```
1 // src/AppBundle/Security/PostVoter.php
2
3 // ...
4 use Symfony\Component\Security\Core\Authorization\AccessDecisionManagerInterface;
5
6 class PostVoter extends Voter
7 {
8     // ...
9
10    private $decisionManager;
11
12    public function __construct(AccessDecisionManagerInterface $decisionManager)
13    {
14        $this->decisionManager = $decisionManager;
15    }
16
17    protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
18    {
19        // ...
```

3. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Authorization/AccessDecisionManager.html>

```

20
21     // ROLE_SUPER_ADMIN can do anything! The power!
22     if ($this->decisionManager->decide($token, array('ROLE_SUPER_ADMIN'))) {
23         return true;
24     }
25
26     // ... all the normal voter logic
27 }
28 }

```

Next, update `services.yml` to inject the `security.access.decision_manager` service:

Listing 120-6

```

1 # app/config/services.yml
2 services:
3     app.post_voter:
4         class: AppBundle\Security\PostVoter
5         arguments: ['@security.access.decision_manager']
6         public: false
7         tags:
8             - { name: security.voter }

```

That's it! Calling `decide()` on the `AccessDecisionManager` is essentially the same as calling `isGranted()` from a controller or other places (it's just a little lower-level, which is necessary for a voter).



The `security.access.decision_manager` is private. This means you can't access it directly from a controller: you can only inject it into other services. That's ok: use `security.authorization_checker` instead in all cases except for voters.

Changing the Access Decision Strategy

Normally, only one voter will vote at any given time (the rest will "abstain", which means they return `false` from `supports()`). But in theory, you could make multiple voters vote for one action and object. For instance, suppose you have one voter that checks if the user is a member of the site and a second one that checks if the user is older than 18.

To handle these cases, the access decision manager uses an access decision strategy. You can configure this to suit your needs. There are three strategies available:

affirmative (default)

This grants access as soon as there is *one* voter granting access;

consensus

This grants access if there are more voters granting access than denying;

unanimous

This only grants access once *all* voters grant access.

In the above scenario, both voters should grant access in order to grant access to the user to read the post. In this case, the default strategy is no longer valid and **unanimous** should be used instead. You can set this in the security configuration:

Listing 120-7

```

1 # app/config/security.yml
2 security:
3     access_decision_manager:
4         strategy: unanimous

```



Chapter 121

How to Use Access Control Lists (ACLs)

In complex applications, you will often face the problem that access decisions cannot only be based on the person (**Token**) who is requesting access, but also involve a domain object that access is being requested for. This is where the ACL system comes in.



Alternatives to ACLs

Using ACL's isn't trivial, and for simpler use cases, it may be overkill. If your permission logic could be described by just writing some code (e.g. to check if a **Blog** is owned by the current **User**), then consider using *voters*. A voter is passed the object being voted on, which you can use to make complex decisions and effectively implement your own ACL. Enforcing authorization (e.g. the **isGranted** part) will look similar to what you see in this entry, but your voter class will handle the logic behind the scenes, instead of the ACL system.

Imagine you are designing a blog system where your users can comment on your posts. Now, you want a user to be able to edit their own comments, but not those of other users; besides, you yourself want to be able to edit all comments. In this scenario, **Comment** would be the domain object that you want to restrict access to. You could take several approaches to accomplish this using Symfony, two basic approaches are (non-exhaustive):

- *Enforce security in your business methods*: Basically, that means keeping a reference inside each **Comment** to all users who have access, and then compare these users to the provided **Token**.
- *Enforce security with roles*: In this approach, you would add a role for each **Comment** object, i.e. `ROLE_COMMENT_1`, `ROLE_COMMENT_2`, etc.

Both approaches are perfectly valid. However, they couple your authorization logic to your business code which makes it less reusable elsewhere, and also increases the difficulty of unit testing. Besides, you could run into performance issues if many users would have access to a single domain object.

Fortunately, there is a better way, which you will find out about now.

Bootstrapping

Now, before you can finally get into action, you need to do some bootstrapping. First, you need to configure the connection the ACL system is supposed to use:

Listing 121-1

```
1 # app/config/security.yml
2 security:
3     # ...
4
5     acl:
6         connection: default
```



The ACL system requires a connection from either Doctrine DBAL (usable by default) or Doctrine MongoDB (usable with *MongoDBAclBundle*¹). However, that does not mean that you have to use Doctrine ORM or ODM for mapping your domain objects. You can use whatever mapper you like for your objects, be it Doctrine ORM, MongoDB ODM, Propel, raw SQL, etc. The choice is yours.

After the connection is configured, you have to import the database structure. Fortunately, there is a task for this. Simply run the following command:

Listing 121-2

```
1 $ php bin/console init:acl
```

Getting Started

Coming back to the small example from the beginning, you can now implement ACL for it.

Once the ACL is created, you can grant access to objects by creating an Access Control Entry (ACE) to solidify the relationship between the entity and your user.

Creating an ACL and Adding an ACE

Listing 121-3

```
1 // src/AppBundle/Controller/BlogController.php
2 namespace AppBundle\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5 use Symfony\Component\Security\Core\Exception\AccessDeniedException;
6 use Symfony\Component\Security\Acl\Domain\ObjectIdentity;
7 use Symfony\Component\Security\Acl\Domain\UserSecurityIdentity;
8 use Symfony\Component\Security\Acl\Permission\MaskBuilder;
9
10 class BlogController extends Controller
11 {
12     // ...
13
14     public function addCommentAction(Post $post)
15     {
16         $comment = new Comment();
17
18         // ... setup $form, and submit data
19
20         if ($form->isValid()) {
21             $entityManager = $this->getDoctrine()->getManager();
22             $entityManager->persist($comment);
23             $entityManager->flush();
24
25             // creating the ACL
26             $aclProvider = $this->get('security.acl.provider');
```

1. <https://github.com/IamPersistent/MongoDBAclBundle>

```

27     $objectIdentity = ObjectIdentity::fromDomainObject($comment);
28     $acl = $aclProvider->createAcl($objectIdentity);
29
30     // retrieving the security identity of the currently logged-in user
31     $tokenStorage = $this->get('security.token_storage');
32     $user = $tokenStorage->getToken()->getUser();
33     $securityIdentity = UserSecurityIdentity::fromAccount($user);
34
35     // grant owner access
36     $acl->insertObjectAce($securityIdentity, MaskBuilder::MASK_OWNER);
37     $aclProvider->updateAcl($acl);
38 }
39 }
40 }

```

There are a couple of important implementation decisions in this code snippet. For now, I only want to highlight two:

First, you may have noticed that `->createAcl()` does not accept domain objects directly, but only implementations of the `ObjectIdentityInterface`. This additional step of indirection allows you to work with ACLs even when you have no actual domain object instance at hand. This will be extremely helpful if you want to check permissions for a large number of objects without actually hydrating these objects.

The other interesting part is the `->insertObjectAce()` call. In the example, you are granting the user who is currently logged in owner access to the Comment. The `MaskBuilder::MASK_OWNER` is a pre-defined integer bitmask; don't worry the mask builder will abstract away most of the technical details, but using this technique you can store many different permissions in one database row which gives a considerable boost in performance.



The order in which ACEs are checked is significant. As a general rule, you should place more specific entries at the beginning.

Checking Access

Listing 121-4

```

1  // src/AppBundle/Controller/BlogController.php
2
3  // ...
4
5  class BlogController
6  {
7      // ...
8
9      public function editCommentAction(Comment $comment)
10     {
11         $authorizationChecker = $this->get('security.authorization_checker');
12
13         // check for edit access
14         if (false === $authorizationChecker->isGranted('EDIT', $comment)) {
15             throw new AccessDeniedException();
16         }
17
18         // ... retrieve actual comment object, and do your editing here
19     }
20 }

```

In this example, you check whether the user has the `EDIT` permission. Internally, Symfony maps the permission to several integer bitmasks, and checks whether the user has any of them.



You can define up to 32 base permissions (depending on your OS PHP might vary between 30 to 32). In addition, you can also define cumulative permissions.

Cumulative Permissions

In the first example above, you only granted the user the **OWNER** base permission. While this effectively also allows the user to perform any operation such as view, edit, etc. on the domain object, there are cases where you may want to grant these permissions explicitly.

The **MaskBuilder** can be used for creating bit masks easily by combining several base permissions:

```
Listing 121-5 1 $builder = new MaskBuilder();
2 $builder
3     ->add('view')
4     ->add('edit')
5     ->add('delete')
6     ->add('undelete')
7 ;
8 $mask = $builder->get(); // int(29)
```

This integer bitmask can then be used to grant a user the base permissions you added above:

```
Listing 121-6 1 $identity = new UserSecurityIdentity('johannes', 'AppBundle\Entity\User');
2 $acl->insertObjectAce($identity, $mask);
```

The user is now allowed to view, edit, delete, and un-delete objects.



Chapter 122

How to Use advanced ACL Concepts

The aim of this chapter is to give a more in-depth view of the ACL system, and also explain some of the design decisions behind it.

Design Concepts

Symfony's object instance security capabilities are based on the concept of an Access Control List. Every domain object **instance** has its own ACL. The ACL instance holds a detailed list of Access Control Entries (ACEs) which are used to make access decisions. Symfony's ACL system focuses on two main objectives:

- providing a way to efficiently retrieve a large amount of ACLs/ACEs for your domain objects, and to modify them;
- providing a way to easily make decisions of whether a person is allowed to perform an action on a domain object or not.

As indicated by the first point, one of the main capabilities of Symfony's ACL system is a high-performance way of retrieving ACLs/ACEs. This is extremely important since each ACL might have several ACEs, and inherit from another ACL in a tree-like fashion. Therefore, no ORM is leveraged, instead the default implementation interacts with your connection directly using Doctrine's DBAL.

Object Identities

The ACL system is completely decoupled from your domain objects. They don't even have to be stored in the same database, or on the same server. In order to achieve this decoupling, in the ACL system your objects are represented through object identity objects. Every time you want to retrieve the ACL for a domain object, the ACL system will first create an object identity from your domain object, and then pass this object identity to the ACL provider for further processing.

Security Identities

This is analog to the object identity, but represents a user, or a role in your application. Each role, or user has its own security identity.



For users, the security identity is based on the username. This means that, if for any reason, a user's username was to change, you must ensure its security identity is updated too. The `MutableAclProvider::updateUserSecurityIdentity()`¹ method is there to handle the update.

Database Table Structure

The default implementation uses five database tables as listed below. The tables are ordered from least rows to most rows in a typical application:

- `acl_security_identities`: This table records all security identities (SID) which hold ACEs. The default implementation ships with two security identities: `RoleSecurityIdentity`² and `UserSecurityIdentity`³.
- `acl_classes`: This table maps class names to a unique ID which can be referenced from other tables.
- `acl_object_identities`: Each row in this table represents a single domain object instance.
- `acl_object_identity_ancestors`: This table allows all the ancestors of an ACL to be determined in a very efficient way.
- `acl_entries`: This table contains all ACEs. This is typically the table with the most rows. It can contain tens of millions without significantly impacting performance.

Scope of Access Control Entries

Access control entries can have different scopes in which they apply. In Symfony, there are basically two different scopes:

- Class-Scope: These entries apply to all objects with the same class.
- Object-Scope: This was the scope solely used in the previous chapter, and it only applies to one specific object.

Sometimes, you will find the need to apply an ACE only to a specific field of the object. Suppose you want the ID only to be viewable by an administrator, but not by your customer service. To solve this common problem, two more sub-scopes have been added:

- Class-Field-Scope: These entries apply to all objects with the same class, but only to a specific field of the objects.
- Object-Field-Scope: These entries apply to a specific object, and only to a specific field of that object.

Pre-Authorization Decisions

For pre-authorization decisions, that is decisions made before any secure method (or secure action) is invoked, the proven `AccessDecisionManager` service is used. The `AccessDecisionManager` is also used for reaching authorization decisions based on roles. Just like roles, the ACL system adds several new attributes which may be used to check for different permissions.

1. http://api.symfony.com/3.0/Symfony/Component/Security/Acl/Dbal/MutableAclProvider.html#method_updateUserSecurityIdentity
2. <http://api.symfony.com/3.0/Symfony/Component/Security/Acl/Domain/RoleSecurityIdentity.html>
3. <http://api.symfony.com/3.0/Symfony/Component/Security/Acl/Domain/UserSecurityIdentity.html>

Built-in Permission Map

Attribute	Intended Meaning	Integer Bitmasks
VIEW	Whether someone is allowed to view the domain object.	VIEW, EDIT, OPERATOR, MASTER, or OWNER
EDIT	Whether someone is allowed to make changes to the domain object.	EDIT, OPERATOR, MASTER, or OWNER
CREATE	Whether someone is allowed to create the domain object.	CREATE, OPERATOR, MASTER, or OWNER
DELETE	Whether someone is allowed to delete the domain object.	DELETE, OPERATOR, MASTER, or OWNER
UNDELETE	Whether someone is allowed to restore a previously deleted domain object.	UNDELETE, OPERATOR, MASTER, or OWNER
OPERATOR	Whether someone is allowed to perform all of the above actions.	OPERATOR, MASTER, or OWNER
MASTER	Whether someone is allowed to perform all of the above actions, and in addition is allowed to grant any of the above permissions to others.	MASTER, or OWNER
OWNER	Whether someone owns the domain object. An owner can perform any of the above actions <i>and</i> grant master and owner permissions.	OWNER

Permission Attributes vs. Permission Bitmasks

Attributes are used by the AccessDecisionManager, just like roles. Often, these attributes represent in fact an aggregate of integer bitmasks. Integer bitmasks on the other hand, are used by the ACL system internally to efficiently store your users' permissions in the database, and perform access checks using extremely fast bitmask operations.

Extensibility

The above permission map is by no means static, and theoretically could be completely replaced at will. However, it should cover most problems you encounter, and for interoperability with other bundles, you are encouraged to stick to the meaning envisaged for them.

Post Authorization Decisions

Post authorization decisions are made after a secure method has been invoked, and typically involve the domain object which is returned by such a method. After invocation providers also allow to modify, or filter the domain object before it is returned.

Due to current limitations of the PHP language, there are no post-authorization capabilities build into the core Security component. However, there is an experimental *JMSSecurityExtraBundle*⁴ which adds these capabilities. See its documentation for further information on how this is accomplished.

Process for Reaching Authorization Decisions

The ACL class provides two methods for determining whether a security identity has the required bitmasks, **isGranted** and **isFieldGranted**. When the ACL receives an authorization request through one of these methods, it delegates this request to an implementation of *PermissionGrantingStrategy*⁵. This allows you to replace the way access decisions are reached without actually modifying the ACL class itself.

The **PermissionGrantingStrategy** first checks all your object-scope ACEs. If none is applicable, the class-scope ACEs will be checked. If none is applicable, then the process will be repeated with the ACEs of the parent ACL. If no parent ACL exists, an exception will be thrown.

4. <https://github.com/schmittjoh/JMSSecurityExtraBundle>

5. <http://api.symfony.com/3.0/Symfony/Component/Security/Acl/Domain/PermissionGrantingStrategy.html>



Chapter 123

How to Force HTTPS or HTTP for different URLs

You can force areas of your site to use the HTTPS protocol in the security config. This is done through the `access_control` rules using the `requires_channel` option. For example, if you want to force all URLs starting with `/secure` to use HTTPS then you could use the following configuration:

Listing 123-1

```
1 # app/config/security.yml
2 security:
3   # ...
4
5   access_control:
6     - { path: ^/secure, roles: ROLE_ADMIN, requires_channel: https }
```

The login form itself needs to allow anonymous access, otherwise users will be unable to authenticate. To force it to use HTTPS you can still use `access_control` rules by using the `IS_AUTHENTICATED_ANONYMOUSLY` role:

Listing 123-2

```
1 # app/config/security.yml
2 security:
3   # ...
4
5   access_control:
6     - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY, requires_channel: https }
```

It is also possible to specify using HTTPS in the routing configuration, see *How to Force Routes to always Use HTTPS or HTTP* for more details.



Chapter 124

How to Secure any Service or Method in your Application

In the security chapter, you can see how to secure a controller by requesting the `security.authorization_checker` service from the Service Container and checking the current user's role:

Listing 124-1

```
1 // ...
2 use Symfony\Component\Security\Core\Exception\AccessDeniedException;
3
4 public function helloAction($name)
5 {
6     $this->denyAccessUnlessGranted('ROLE_ADMIN');
7
8     // ...
9 }
```

You can also secure *any* service by injecting the `security.authorization_checker` service into it. For a general introduction to injecting dependencies into services see the *Service Container* chapter of the book. For example, suppose you have a `NewsletterManager` class that sends out emails and you want to restrict its use to only users who have some `ROLE_NEWSLETTER_ADMIN` role. Before you add security, the class looks something like this:

Listing 124-2

```
1 // src/AppBundle/Newsletter/NewsletterManager.php
2 namespace AppBundle\Newsletter;
3
4 class NewsletterManager
5 {
6     public function sendNewsletter()
7     {
8         // ... where you actually do the work
9     }
10
11     // ...
12 }
```

Your goal is to check the user's role when the `sendNewsletter()` method is called. The first step towards this is to inject the `security.authorization_checker` service into the object. Since it

won't make sense *not* to perform the security check, this is an ideal candidate for constructor injection, which guarantees that the authorization checker object will be available inside the **NewsletterManager** class:

```
Listing 124-3 1 // src/AppBundle/Newsletter/NewsletterManager.php
2
3 // ...
4 use Symfony\Component\Security\Core\Authorization\AuthorizationCheckerInterface;
5
6 class NewsletterManager
7 {
8     protected $authorizationChecker;
9
10    public function __construct(AuthorizationCheckerInterface $authorizationChecker)
11    {
12        $this->authorizationChecker = $authorizationChecker;
13    }
14
15    // ...
16 }
```

Then in your service configuration, you can inject the service:

```
Listing 124-4 1 # app/config/services.yml
2 services:
3     newsletter_manager:
4         class: AppBundle\Newsletter\NewsletterManager
5         arguments: ['@security.authorization_checker']
```

The injected service can then be used to perform the security check when the **sendNewsletter()** method is called:

```
Listing 124-5 1 namespace AppBundle\Newsletter;
2
3 use Symfony\Component\Security\Core\Authorization\AuthorizationCheckerInterface;
4 use Symfony\Component\Security\Core\Exception\AccessDeniedException;
5 // ...
6
7 class NewsletterManager
8 {
9     protected $authorizationChecker;
10
11    public function __construct(AuthorizationCheckerInterface $authorizationChecker)
12    {
13        $this->authorizationChecker = $authorizationChecker;
14    }
15
16    public function sendNewsletter()
17    {
18        if (false === $this->authorizationChecker->isGranted('ROLE_NEWSLETTER_ADMIN')) {
19            throw new AccessDeniedException();
20        }
21
22        // ...
23    }
24
25    // ...
26 }
```

If the current user does not have the **ROLE_NEWSLETTER_ADMIN**, they will be prompted to log in.

Securing Methods Using Annotations

You can also secure method calls in any service with annotations by using the optional *JMSSecurityExtraBundle*¹ bundle. This bundle is not included in the Symfony Standard Distribution, but you can choose to install it.

To enable the annotations functionality, tag the service you want to secure with the **security.secure_service** tag (you can also automatically enable this functionality for all services, see the sidebar below):

Listing 124-6

```
1 # app/config/services.yml
2 services:
3     newsletter_manager:
4         class: AppBundle\Newsletter\NewsletterManager
5         tags:
6             - { name: security.secure_service }
```

You can then achieve the same results as above using an annotation:

Listing 124-7

```
1 namespace AppBundle\Newsletter;
2
3 use JMS\SecurityExtraBundle\Annotation\Secure;
4 // ...
5
6 class NewsletterManager
7 {
8
9     /**
10      * @Secure(roles="ROLE_NEWSLETTER_ADMIN")
11      */
12     public function sendNewsletter()
13     {
14         // ...
15     }
16
17     // ...
18 }
```



The annotations work because a proxy class is created for your class which performs the security checks. This means that, whilst you can use annotations on public and protected methods, you cannot use them with private methods or methods marked final.

The JMSSecurityExtraBundle also allows you to secure the parameters and return values of methods. For more information, see the *JMSSecurityExtraBundle*² documentation.



Activating the Annotations Functionality for all Services

When securing the method of a service (as shown above), you can either tag each service individually, or activate the functionality for *all* services at once. To do so, set the **secure_all_services** configuration option to true:

Listing 124-8

```
1 # app/config/config.yml
2 jms_security_extra:
3     # ...
4     secure_all_services: true
```

The disadvantage of this method is that, if activated, the initial page load may be very slow depending on how many services you have defined.

1. <https://github.com/schmittjoh/JMSSecurityExtraBundle>

2. <https://github.com/schmittjoh/JMSSecurityExtraBundle>



Chapter 125

How Does the Security access_control Work?

For each incoming request, Symfony checks each `access_control` entry to find *one* that matches the current request. As soon as it finds a matching `access_control` entry, it stops - only the **first** matching `access_control` is used to enforce access.

Each `access_control` has several options that configure two different things:

1. should the incoming request match this access control entry
2. once it matches, should some sort of access restriction be enforced:

1. Matching Options

Symfony creates an instance of *RequestMatcher*¹ for each `access_control` entry, which determines whether or not a given access control should be used on this request. The following `access_control` options are used for matching:

- path
- ip Or ips
- host
- methods

Take the following `access_control` entries as an example:

```
Listing 125-1 1 # app/config/security.yml
                2 security:
                3     # ...
                4     access_control:
                5         - { path: ^/admin, roles: ROLE_USER_IP, ip: 127.0.0.1 }
                6         - { path: ^/admin, roles: ROLE_USER_HOST, host: symfony\.com$ }
                7         - { path: ^/admin, roles: ROLE_USER_METHOD, methods: [POST, PUT] }
                8         - { path: ^/admin, roles: ROLE_USER }
```

For each incoming request, Symfony will decide which `access_control` to use based on the URI, the client's IP address, the incoming host name, and the request method. Remember, the first rule that

1. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/RequestMatcher.html>

matches is used, and if `ip`, `host` or `method` are not specified for an entry, that `access_control` will match any `ip`, `host` or `method`:

URI	IP	HOST	METHOD	access_control	Why?
/admin/ user	127.0.0.1	example.com	GET	rule #1 (ROLE_USER_IP)	The URI matches path and the IP matches ip.
/admin/ user	127.0.0.1	symfony.com	GET	rule #1 (ROLE_USER_IP)	The path and ip still match. This would also match the <code>ROLE_USER_HOST</code> entry, but <i>only</i> the first <code>access_control</code> match is used.
/admin/ user	168.0.0.1	symfony.com	GET	rule #2 (ROLE_USER_HOST)	The ip doesn't match the first rule, so the second rule (which matches) is used.
/admin/ user	168.0.0.1	symfony.com	POST	rule #2 (ROLE_USER_HOST)	The second rule still matches. This would also match the third rule (<code>ROLE_USER_METHOD</code>), but only the first matched <code>access_control</code> is used.
/admin/ user	168.0.0.1	example.com	POST	rule #3 (ROLE_USER_METHOD)	The ip and host don't match the first two entries, but the third - <code>ROLE_USER_METHOD</code> - matches and is used.
/admin/ user	168.0.0.1	example.com	GET	rule #4 (ROLE_USER)	The ip, host and method prevent the first three entries from matching. But since the URI matches the path pattern of the <code>ROLE_USER</code> entry, it is used.
/foo	127.0.0.1	symfony.com	POST	matches no entries	This doesn't match any <code>access_control</code> rules, since its URI doesn't match any of the path values.

2. Access Enforcement

Once Symfony has decided which `access_control` entry matches (if any), it then *enforces* access restrictions based on the `roles`, `allow_if` and `requires_channel` options:

- `role` If the user does not have the given role(s), then access is denied (internally, an `AccessDeniedException`² is thrown);
- `allow_if` If the expression returns false, then access is denied;
- `requires_channel` If the incoming request's channel (e.g. http) does not match this value (e.g. https), the user will be redirected (e.g. redirected from http to https, or vice versa).



If access is denied, the system will try to authenticate the user if not already (e.g. redirect the user to the login page). If the user is already logged in, the 403 "access denied" error page will be shown. See *How to Customize Error Pages* for more information.

2. <http://api.symfony.com/3.0/Symfony/Component/Security/Core/Exception/AccessDeniedException.html>

Matching access_control By IP

Certain situations may arise when you need to have an `access_control` entry that *only* matches requests coming from some IP address or range. For example, this *could* be used to deny access to a URL pattern to all requests *except* those from a trusted, internal server.



As you'll read in the explanation below the example, the `ips` option does not restrict to a specific IP address. Instead, using the `ips` key means that the `access_control` entry will only match this IP address, and users accessing it from a different IP address will continue down the `access_control` list.

Here is an example of how you configure some example `/internal*` URL pattern so that it is only accessible by requests from the local server itself:

Listing 125-2

```
1 # app/config/security.yml
2 security:
3     # ...
4     access_control:
5         #
6         - { path: ^/internal, roles: IS_AUTHENTICATED_ANONYMOUSLY, ips: [127.0.0.1, ::1] }
7         - { path: ^/internal, roles: ROLE_NO_ACCESS }
```

Here is how it works when the path is `/internal/something` coming from the external IP address `10.0.0.1`:

- The first access control rule is ignored as the `path` matches but the IP address does not match either of the IPs listed;
- The second access control rule is enabled (the only restriction being the `path`) and so it matches. If you make sure that no users ever have `ROLE_NO_ACCESS`, then access is denied (`ROLE_NO_ACCESS` can be anything that does not match an existing role, it just serves as a trick to always deny access).

But if the same request comes from `127.0.0.1` or `::1` (the IPv6 loopback address):

- Now, the first access control rule is enabled as both the `path` and the `ip` match: access is allowed as the user always has the `IS_AUTHENTICATED_ANONYMOUSLY` role.
- The second access rule is not examined as the first rule matched.

Securing by an Expression

Once an `access_control` entry is matched, you can deny access via the `roles` key or use more complex logic with an expression in the `allow_if` key:

Listing 125-3

```
1 # app/config/security.yml
2 security:
3     # ...
4     access_control:
5         -
6             path: ^/_internal/secure
7             allow_if: "'127.0.0.1' == request.getClientIp() or has_role('ROLE_ADMIN')"
```

In this case, when the user tries to access any URL starting with `/_internal/secure`, they will only be granted access if the IP address is `127.0.0.1` or if the user has the `ROLE_ADMIN` role.

Inside the expression, you have access to a number of different variables and functions including `request`, which is the Symfony *Request*³ object (see Request).

3. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html>

For a list of the other functions and variables, see functions and variables.

Forcing a Channel (http, https)

You can also require a user to access a URL via SSL; just use the **requires_channel** argument in any **access_control** entries. If this **access_control** is matched and the request is using the **http** channel, the user will be redirected to **https**:

```
Listing 125-4 1 # app/config/security.yml
                2 security:
                3   # ...
                4   access_control:
                5     - { path: ^/cart/checkout, roles: IS_AUTHENTICATED_ANONYMOUSLY, requires_channel: https }
```



Chapter 126

How to Use the Serializer

Serializing and deserializing to and from objects and different formats (e.g. JSON or XML) is a very complex topic. Symfony comes with a *Serializer Component*, which gives you some tools that you can leverage for your solution.

In fact, before you start, get familiar with the serializer, normalizers and encoders by reading the *Serializer Component*.

Activating the Serializer

The **serializer** service is not available by default. To turn it on, activate it in your configuration:

Listing 126-1

```
1 # app/config/config.yml
2 framework:
3     # ...
4     serializer:
5         enabled: true
```

Using the Serializer Service

Once enabled, the **serializer** service can be injected in any service where you need it or it can be used in a controller like the following:

Listing 126-2

```
1 // src/AppBundle/Controller/DefaultController.php
2 namespace AppBundle\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5
6 class DefaultController extends Controller
7 {
8     public function indexAction()
9     {
10         $serializer = $this->get('serializer');
11
12         // ...
```

```
13     }
14 }
```

Adding Normalizers and Encoders

Once enabled, the `serializer` service will be available in the container and will be loaded with two encoders (*JsonEncoder*¹ and *XmlEncoder*²) and the `ObjectNormalizer` normalizer.

You can load normalizers and/or encoders by tagging them as `serializer.normalizer` and `serializer.encoder`. It's also possible to set the priority of the tag in order to decide the matching order.

Here is an example on how to load the *GetSetMethodNormalizer*³:

```
Listing 126-3 1 # app/config/services.yml
2 services:
3     get_set_method_normalizer:
4         class: Symfony\Component\Serializer\Normalizer\GetSetMethodNormalizer
5         tags:
6             - { name: serializer.normalizer }
```

Using Serialization Groups Annotations

Enable serialization groups annotation with the following configuration:

```
Listing 126-4 1 # app/config/config.yml
2 framework:
3     # ...
4     serializer:
5         enable_annotations: true
```

Next, add the `@Groups` annotations to your class and choose which groups to use when serializing:

```
Listing 126-5 1 $serializer = $this->get('serializer');
2 $json = $serializer->serialize(
3     $someObject,
4     'json', array('groups' => array('group1'))
5 );
```

In addition to the `@Groups` annotation, the `Serializer` component also supports `Yaml` or `XML` files. These files are automatically loaded when being stored in one of the following locations:

- The `serialization.yml` or `serialization.xml` file in the `Resources/config/` directory of a bundle;
- All `*.yaml` and `*.xml` files in the `Resources/config/serialization/` directory of a bundle.

Enabling the Metadata Cache

Metadata used by the `Serializer` component such as groups can be cached to enhance application performance. Any service implementing the `Doctrine\Common\Cache\Cache` interface can be used.

A service leveraging *APCu*⁴ (and *APC* for `PHP < 5.5`) is built-in.

1. <http://api.symfony.com/3.0/Symfony/Component/Serializer/Encoder/JsonEncoder.html>

2. <http://api.symfony.com/3.0/Symfony/Component/Serializer/Encoder/XmlEncoder.html>

3. <http://api.symfony.com/3.0/Symfony/Component/Serializer/Normalizer/GetSetMethodNormalizer.html>

4. <https://github.com/kraekjoe/apcu>

Listing 126-6

```
1 # app/config/config_prod.yml
2 framework:
3   # ...
4   serializer:
5     cache: serializer.mapping.cache.apc
```

Going Further with the Serializer Component

*ApiPlatform*⁵ provides an API system supporting *JSON-LD*⁶ and *Hydra Core Vocabulary*⁷ hypermedia formats. It is built on top of the Symfony Framework and its Serializer component. It provides custom normalizers and a custom encoder, custom metadata and a caching system.

If you want to leverage the full power of the Symfony Serializer component, take a look at how this bundle works.

5. <https://github.com/api-platform/core>

6. <http://json-ld.org>

7. <http://hydra-cg.com>



Chapter 127

How to Define Non Shared Services

In the service container, all services are shared by default. This means that each time you retrieve the service, you'll get the *same* instance. This is often the behavior you want, but in some cases, you might want to always get a *new* instance.

In order to always get a new instance, set the **shared** setting to **false** in your service definition:

Listing 127-1

```
1 # app/config/services.yml
2 services:
3     app.some_not_shared_service:
4         class: ...
5         shared: false
6         # ...
```

Now, whenever you call `$container->get('app.some_not_shared_service')` or inject this service, you'll receive a new instance.



Chapter 128

How to Work with Compiler Passes in Bundles

Compiler passes give you an opportunity to manipulate other service definitions that have been registered with the service container. You can read about how to create them in the components section "Execute Code During Compilation".

When using separate compiler passes, you need to register them in the **build()** method of the bundle class (this is not needed when implementing the **process()** method in the extension):

Listing 128-1

```
1  // src/AppBundle/AppBundle.php
2  namespace AppBundle;
3
4  use Symfony\Component\HttpKernel\Bundle\Bundle;
5  use Symfony\Component\DependencyInjection\ContainerBuilder;
6  use AppBundle\DependencyInjection\Compiler\CustomPass;
7
8  class AppBundle extends Bundle
9  {
10     public function build(ContainerBuilder $container)
11     {
12         parent::build($container);
13
14         $container->addCompilerPass(new CustomPass());
15     }
16 }
```

One of the most common use-cases of compiler passes is to work with tagged services (read more about tags in the components section "*Working with Tagged Services*"). If you are using custom tags in a bundle then by convention, tag names consist of the name of the bundle (lowercase, underscores as separators), followed by a dot and finally the "real" name. For example, if you want to introduce some sort of "mail_transport" tag in your AppBundle, you should call it **app.mail_transport**.



Chapter 129

Session Proxy Examples

The session proxy mechanism has a variety of uses and this article demonstrates two common uses. Rather than using the regular session handler, you can create a custom save handler just by defining a class that extends the *SessionHandlerProxy*¹ class.

Then, define a new service related to the custom session handler:

```
Listing 129-1 1 # app/config/services.yml
2 services:
3     app.session_handler:
4         class: AppBundle\Session\CustomSessionHandler
```

Finally, use the `framework.session.handler_id` configuration option to tell Symfony to use your own session handler instead of the default one:

```
Listing 129-2 1 # app/config/config.yml
2 framework:
3     session:
4         # ...
5         handler_id: app.session_handler
```

Keep reading the next sections to learn how to use the session handlers in practice to solve two common use cases: encrypt session information and define readonly guest sessions.

Encryption of Session Data

If you wanted to encrypt the session data, you could use the proxy to encrypt and decrypt the session as required:

```
Listing 129-3 1 // src/AppBundle/Session/EncryptedSessionProxy.php
2 namespace AppBundle\Session;
3
4 use Symfony\Component\HttpFoundation\Session\Storage\Proxy\SessionHandlerProxy;
5
6 class EncryptedSessionProxy extends SessionHandlerProxy
```

1. <http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Storage/Proxy/SessionHandlerProxy.html>


```

7 {
8     private $key;
9
10    public function __construct(\SessionHandlerInterface $handler, $key)
11    {
12        $this->key = $key;
13
14        parent::__construct($handler);
15    }
16
17    public function read($id)
18    {
19        $data = parent::read($id);
20
21        return mcrypt_decrypt(\MCRYPT_3DES, $this->key, $data);
22    }
23
24    public function write($id, $data)
25    {
26        $data = mcrypt_encrypt(\MCRYPT_3DES, $this->key, $data);
27
28        return parent::write($id, $data);
29    }
30 }

```

Readonly Guest Sessions

There are some applications where a session is required for guest users, but where there is no particular need to persist the session. In this case you can intercept the session before it is written:

Listing 129-4

```

1  // src/AppBundle/Session/ReadOnlySessionProxy.php
2  namespace AppBundle\Session;
3
4  use AppBundle\Entity\User;
5  use Symfony\Component\HttpFoundation\Session\Storage\Proxy\SessionHandlerProxy;
6
7  class ReadOnlySessionProxy extends SessionHandlerProxy
8  {
9      private $user;
10
11     public function __construct(\SessionHandlerInterface $handler, User $user)
12     {
13         $this->user = $user;
14
15         parent::__construct($handler);
16     }
17
18     public function write($id, $data)
19     {
20         if ($this->user->isGuest()) {
21             return;
22         }
23
24         return parent::write($id, $data);
25     }
26 }

```



Chapter 130

Making the Locale "Sticky" during a User's Session

Symfony stores the locale setting in the Request, which means that this setting is not available in subsequent requests. In this article, you'll learn how to store the locale in the session, so that it'll be the same for every subsequent request.

Creating a LocaleListener

To simulate that the locale is stored in a session, you need to create and register a *new event listener*. The listener will look something like this. Typically, `_locale` is used as a routing parameter to signify the locale, though it doesn't really matter how you determine the desired locale from the request:

```
Listing 130-1 1 // src/AppBundle/EventListener/LocaleListener.php
2 namespace AppBundle\EventListener;
3
4 use Symfony\Component\HttpFoundation\Event\GetResponseEvent;
5 use Symfony\Component\HttpFoundation\KernelEvents;
6 use Symfony\Component\EventDispatcher\EventSubscriberInterface;
7
8 class LocaleListener implements EventSubscriberInterface
9 {
10     private $defaultLocale;
11
12     public function __construct($defaultLocale = 'en')
13     {
14         $this->defaultLocale = $defaultLocale;
15     }
16
17     public function onKernelRequest(GetResponseEvent $event)
18     {
19         $request = $event->getRequest();
20         if (!$request->hasPreviousSession()) {
21             return;
22         }
23
24         // try to see if the locale has been set as a _locale routing parameter
25         if ($locale = $request->attributes->get('_locale')) {
```

```

26         $request->getSession()->set('_locale', $locale);
27     } else {
28         // if no explicit locale has been set on this request, use one from the session
29         $request->setLocale($request->getSession()->get('_locale', $this->defaultLocale));
30     }
31 }
32
33 public static function getSubscribedEvents()
34 {
35     return array(
36         // must be registered after the default Locale listener
37         KernelEvents::REQUEST => array(array('onKernelRequest', 15)),
38     );
39 }
40 }

```

Then register the listener:

Listing 130-2

```

1 services:
2     app.locale_listener:
3         class: AppBundle\EventListener\LocaleListener
4         arguments: ['%kernel.default_locale%']
5         tags:
6             - { name: kernel.event_subscriber }

```

That's it! Now celebrate by changing the user's locale and seeing that it's sticky throughout the request. Remember, to get the user's locale, always use the *Request::getLocale*¹ method:

Listing 130-3

```

1 // from a controller...
2 use Symfony\Component\HttpFoundation\Request;
3
4 public function indexAction(Request $request)
5 {
6     $locale = $request->getLocale();
7 }

```

Setting the Locale Based on the User's Preferences

You might want to improve this technique even further and define the locale based on the user entity of the logged in user. However, since the **LocaleListener** is called before the **FirewallListener**, which is responsible for handling authentication and setting the user token on the **TokenStorage**, you have no access to the user which is logged in.

Suppose you have defined a **locale** property on your **User** entity and you want to use this as the locale for the given user. To accomplish this, you can hook into the login process and update the user's session with this locale value before they are redirected to their first page.

To do this, you need an event listener for the **security.interactive_login** event:

Listing 130-4

```

1 // src/AppBundle/EventListener/UserLocaleListener.php
2 namespace AppBundle\EventListener;
3
4 use Symfony\Component\HttpFoundation\Session\Session;
5 use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;
6
7 /**
8  * Stores the locale of the user in the session after the
9  * login. This can be used by the LocaleListener afterwards.
10  */
11 class UserLocaleListener

```

1. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getLocale

```

12 {
13     /**
14      * @var Session
15      */
16     private $session;
17
18     public function __construct(Session $session)
19     {
20         $this->session = $session;
21     }
22
23     /**
24      * @param InteractiveLoginEvent $event
25      */
26     public function onInteractiveLogin(InteractiveLoginEvent $event)
27     {
28         $user = $event->getAuthenticationToken()->getUser();
29
30         if (null !== $user->getLocale()) {
31             $this->session->set('_locale', $user->getLocale());
32         }
33     }
34 }

```

Then register the listener:

Listing 130-5

```

1 # app/config/services.yml
2 services:
3     app.user_locale_listener:
4         class: AppBundle\EventListener\UserLocaleListener
5         arguments: ['@session']
6         tags:
7             - { name: kernel.event_listener, event: security.interactive_login, method: onInteractiveLogin }

```



In order to update the language immediately after a user has changed their language preferences, you need to update the session after an update to the **User** entity.



Chapter 131

Configuring the Directory where Session Files are Saved

By default, the Symfony Standard Edition uses the global `php.ini` values for `session.save_handler` and `session.save_path` to determine where to store session data. This is because of the following configuration:

Listing 131-1

```
1 # app/config/config.yml
2 framework:
3     session:
4         # handler_id set to null will use default session handler from php.ini
5         handler_id: ~
```

With this configuration, changing *where* your session metadata is stored is entirely up to your `php.ini` configuration.

However, if you have the following configuration, Symfony will store the session data in files in the cache directory `%kernel.cache_dir%/sessions`. This means that when you clear the cache, any current sessions will also be deleted:

Listing 131-2

```
1 # app/config/config.yml
2 framework:
3     session: ~
```

Using a different directory to save session data is one method to ensure that your current sessions aren't lost when you clear Symfony's cache.



Using a different session save handler is an excellent (yet more complex) method of session management available within Symfony. See *Configuring Sessions and Save Handlers* for a discussion of session save handlers. There are also entries in the cookbook about storing sessions in a *relational database* or a *NoSQL database*.

To change the directory in which Symfony saves session data, you only need change the framework configuration. In this example, you will change the session directory to `app/sessions`:

Listing 131-3

```
1 # app/config/config.yml
2 framework:
3     session:
4         handler_id: session.handler.native_file
5         save_path: '%kernel.root_dir%/sessions'
```



Chapter 132

Bridge a legacy Application with Symfony Sessions

If you're integrating the Symfony full-stack Framework into a legacy application that starts the session with `session_start()`, you may still be able to use Symfony's session management by using the PHP Bridge session.

If the application has its own PHP save handler, you can specify null for the `handler_id`:

Listing 132-1

```
1 framework:
2     session:
3         storage_id: session.storage.php_bridge
4         handler_id: ~
```

Otherwise, if the problem is simply that you cannot avoid the application starting the session with `session_start()`, you can still make use of a Symfony based session save handler by specifying the save handler as in the example below:

Listing 132-2

```
1 framework:
2     session:
3         storage_id: session.storage.php_bridge
4         handler_id: session.handler.native_file
```



If the legacy application requires its own session save handler, do not override this. Instead set `handler_id: ~`. Note that a save handler cannot be changed once the session has been started. If the application starts the session before Symfony is initialized, the save handler will have already been set. In this case, you will need `handler_id: ~`. Only override the save handler if you are sure the legacy application can use the Symfony save handler without side effects and that the session has not been started before Symfony is initialized.

For more details, see *Integrating with Legacy Sessions*.



Chapter 133

Limit Session Metadata Writes

The default behavior of PHP session is to persist the session regardless of whether the session data has changed or not. In Symfony, each time the session is accessed, metadata is recorded (session created/last used) which can be used to determine session age and idle time.

If for performance reasons you wish to limit the frequency at which the session persists, this feature can adjust the granularity of the metadata updates and persist the session less often while still maintaining relatively accurate metadata. If other session data is changed, the session will always persist.

You can tell Symfony not to update the metadata "session last updated" time until a certain amount of time has passed, by setting `framework.session.metadata_update_threshold` to a value in seconds greater than zero:

Listing 133-1

```
1 framework:
2   session:
3     metadata_update_threshold: 120
```



PHP default's behavior is to save the session whether it has been changed or not. When using `framework.session.metadata_update_threshold` Symfony will wrap the session handler (configured at `framework.session.handler_id`) into the `WriteCheckSessionHandler`. This will prevent any session write if the session was not modified.



Be aware that if the session is not written at every request, it may be garbage collected sooner than usual. This means that your users may be logged out sooner than expected.



Chapter 134

Avoid Starting Sessions for Anonymous Users

Sessions are automatically started whenever you read, write or even check for the existence of data in the session. This means that if you need to avoid creating a session cookie for some users, it can be difficult: you must *completely* avoid accessing the session.

For example, one common problem in this situation involves checking for flash messages, which are stored in the session. The following code would guarantee that a session is *always* started:

Listing 134-1

```
1 {% for flashMessage in app.session.flashBag.get('notice') %}
2     <div class="flash-notice">
3         {{ flashMessage }}
4     </div>
5 {% endfor %}
```

Even if the user is not logged in and even if you haven't created any flash messages, just calling the `get()` (or even `has()`) method of the `flashBag` will start a session. This may hurt your application performance because all users will receive a session cookie. To avoid this behavior, add a check before trying to access the flash messages:

Listing 134-2

```
1 {% if app.request.hasPreviousSession %}
2     {% for flashMessage in app.session.flashBag.get('notice') %}
3         <div class="flash-notice">
4             {{ flashMessage }}
5         </div>
6     {% endfor %}
7 {% endif %}
```



Chapter 135

How to Inject Variables into all Templates (i.e. global Variables)

Sometimes you want a variable to be accessible to all the templates you use. This is possible inside your `app/config/config.yml` file:

```
Listing 135-1 1 # app/config/config.yml
                2 twig:
                3   # ...
                4   globals:
                5     ga_tracking: UA-xxxxx-x
```

Now, the variable `ga_tracking` is available in all Twig templates:

```
Listing 135-2 1 <p>The google tracking code is: {{ ga_tracking }}</p>
```

It's that easy!

Using Service Container Parameters

You can also take advantage of the built-in Service Parameters system, which lets you isolate or reuse the value:

```
Listing 135-3 1 # app/config/parameters.yml
                2 parameters:
                3   ga_tracking: UA-xxxxx-x
```

```
Listing 135-4 1 # app/config/config.yml
                2 twig:
                3   globals:
                4     ga_tracking: '%ga_tracking%'
```

The same variable is available exactly as before.

Referencing Services

Instead of using static values, you can also set the value to a service. Whenever the global variable is accessed in the template, the service will be requested from the service container and you get access to that object.



The service is not loaded lazily. In other words, as soon as Twig is loaded, your service is instantiated, even if you never use that global variable.

To define a service as a global Twig variable, prefix the string with `@`. This should feel familiar, as it's the same syntax you use in service configuration.

Listing 135-5

```
1 # app/config/config.yml
2 twig:
3     # ...
4     globals:
5         user_management: '@app.user_management'
```

Using a Twig Extension

If the global variable you want to set is more complicated - say an object - then you won't be able to use the above method. Instead, you'll need to create a Twig Extension and return the global variable as one of the entries in the `getGlobals` method.



Chapter 136

How to Use and Register Namespaced Twig Paths

Usually, when you refer to a template, you'll use the `MyBundle:Subdir:filename.html.twig` format (see Template Naming and Locations).

Twig also natively offers a feature called "namespaced paths", and support is built-in automatically for all of your bundles.

Take the following paths as an example:

```
Listing 136-1 1 {% extends "AppBundle::layout.html.twig" %}  
2 {{ include('AppBundle:Foo:bar.html.twig') }}
```

With namespaced paths, the following works as well:

```
Listing 136-2 1 {% extends "@App/layout.html.twig" %}  
2 {{ include('@App/Foo/bar.html.twig') }}
```

Both paths are valid and functional by default in Symfony.



As an added bonus, the namespaced syntax is faster.

Registering your own Namespaces

You can also register your own custom namespaces. Suppose that you're using some third-party library that includes Twig templates that live in `vendor/acme/foo-bar/templates`. First, register a namespace for this directory:

```
Listing 136-3 1 # app/config/config.yml  
2 twig:  
3     # ...
```

```
4     paths:
5         "%kernel.root_dir%../vendor/acme/foo-bar/templates": foo_bar
```

The registered namespace is called **foo_bar**, which refers to the **vendor/acme/foo-bar/templates** directory. Assuming there's a file called **sidebar.twig** in that directory, you can use it easily:

Listing 136-4 1 `{{ include('@foo_bar/sidebar.twig') }}`

Multiple Paths per Namespace

You can also assign several paths to the same template namespace. The order in which paths are configured is very important, because Twig will always load the first template that exists, starting from the first configured path. This feature can be used as a fallback mechanism to load generic templates when the specific template doesn't exist.

Listing 136-5 1 `# app/config/config.yml`
2 `twig:`
3 `# ...`
4 `paths:`
5 `"%kernel.root_dir%../vendor/acme/themes/theme1": theme`
6 `"%kernel.root_dir%../vendor/acme/themes/theme2": theme`
7 `"%kernel.root_dir%../vendor/acme/themes/common": theme`

Now, you can use the same **@theme** namespace to refer to any template located in the previous three directories:

Listing 136-6 1 `{{ include('@theme/header.twig') }}`



Chapter 137

How to Use PHP instead of Twig for Templates

Symfony defaults to Twig for its template engine, but you can still use plain PHP code if you want. Both templating engines are supported equally in Symfony. Symfony adds some nice features on top of PHP to make writing templates with PHP more powerful.

Rendering PHP Templates

If you want to use the PHP templating engine, first, make sure to enable it in your application configuration file:

Listing 137-1

```
1 # app/config/config.yml
2 framework:
3     # ...
4     templating:
5         engines: ['twig', 'php']
```

You can now render a PHP template instead of a Twig one simply by using the `.php` extension in the template name instead of `.twig`. The controller below renders the `index.html.php` template:

Listing 137-2

```
1 // src/AppBundle/Controller/HelloController.php
2
3 // ...
4 public function indexAction($name)
5 {
6     return $this->render(
7         'AppBundle:Hello:index.html.php',
8         array('name' => $name)
9     );
10 }
```

You can also use the `@Template`¹ shortcut to render the default `AppBundle:Hello:index.html.php` template:

Listing 137-3

```
1 // src/AppBundle/Controller/HelloController.php
2 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
```

1. <https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/view>

```

3
4 // ...
5
6 /**
7  * @Template(engine="php")
8  */
9 public function indexAction($name)
10 {
11     return array('name' => $name);
12 }

```



Enabling the **php** and **twig** template engines simultaneously is allowed, but it will produce an undesirable side effect in your application: the **@** notation for Twig namespaces will no longer be supported for the **render()** method:

Listing 137-4

```

1 public function indexAction()
2 {
3     // ...
4
5     // namespaced templates will no longer work in controllers
6     $this->render('@App/Default/index.html.twig');
7
8     // you must use the traditional template notation
9     $this->render('AppBundle:Default:index.html.twig');
10 }

```

Listing 137-5

```

1 {# inside a Twig template, namespaced templates work as expected #}
2 {{ include('@App/Default/index.html.twig') }}
3
4 {# traditional template notation will also work #}
5 {{ include('AppBundle:Default:index.html.twig') }}

```

Decorating Templates

More often than not, templates in a project share common elements, like the well-known header and footer. In Symfony, this problem is thought about differently: a template can be decorated by another one.

The **index.html.php** template is decorated by **layout.html.php**, thanks to the **extend()** call:

Listing 137-6

```

1 <!-- app/Resources/views/Hello/index.html.php -->
2 <?php $view->extend('AppBundle:layout.html.php') ?>
3
4 Hello <?php echo $name ?>!

```

The **AppBundle::layout.html.php** notation sounds familiar, doesn't it? It is the same notation used to reference a template. The **::** part simply means that the controller element is empty, so the corresponding file is directly stored under **views/**.

Now, have a look at the **layout.html.php** file:

Listing 137-7

```

1 <!-- app/Resources/views/layout.html.php -->
2 <?php $view->extend('::base.html.php') ?>
3
4 <h1>Hello Application</h1>
5
6 <?php $view['slots']->output('_content') ?>

```

The layout is itself decorated by another one (`::base.html.php`). Symfony supports multiple decoration levels: a layout can itself be decorated by another one. When the bundle part of the template name is empty, views are looked for in the `app/Resources/views/` directory. This directory stores global views for your entire project:

```
Listing 137-8 1 <!-- app/Resources/views/base.html.php -->
2 <!DOCTYPE html>
3 <html>
4   <head>
5     <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
6     <title><?php $view['slots']->output('title', 'Hello Application') ?></title>
7   </head>
8   <body>
9     <?php $view['slots']->output('_content') ?>
10  </body>
11 </html>
```

For both layouts, the `$view['slots']->output('_content')` expression is replaced by the content of the child template, `index.html.php` and `layout.html.php` respectively (more on slots in the next section).

As you can see, Symfony provides methods on a mysterious `$view` object. In a template, the `$view` variable is always available and refers to a special object that provides a bunch of methods that makes the template engine tick.

Working with Slots

A slot is a snippet of code, defined in a template, and reusable in any layout decorating the template. In the `index.html.php` template, define a `title` slot:

```
Listing 137-9 1 <!-- app/Resources/views/Hello/index.html.php -->
2 <?php $view->extend('AppBundle:layout.html.php') ?>
3
4 <?php $view['slots']->set('title', 'Hello World Application') ?>
5
6 Hello <?php echo $name ?>!
```

The base layout already has the code to output the title in the header:

```
Listing 137-10 1 <!-- app/Resources/views/base.html.php -->
2 <head>
3   <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
4   <title><?php $view['slots']->output('title', 'Hello Application') ?></title>
5 </head>
```

The `output()` method inserts the content of a slot and optionally takes a default value if the slot is not defined. And `_content` is just a special slot that contains the rendered child template.

For large slots, there is also an extended syntax:

```
Listing 137-11 1 <?php $view['slots']->start('title') ?>
2   Some large amount of HTML
3 <?php $view['slots']->stop() ?>
```

Including other Templates

The best way to share a snippet of template code is to define a template that can then be included into other templates.

Create a `hello.html.php` template:

```
Listing 137-12 1 <!-- app/Resources/views/Hello/hello.html.php -->
                2 Hello <?php echo $name ?>!
```

And change the `index.html.php` template to include it:

```
Listing 137-13 1 <!-- app/Resources/views/Hello/index.html.php -->
                2 <?php $view->extend('AppBundle::layout.html.php') ?>
                3
                4 <?php echo $view->render('AppBundle:Hello:hello.html.php', array('name' => $name)) ?>
```

The `render()` method evaluates and returns the content of another template (this is the exact same method as the one used in the controller).

Embedding other Controllers

And what if you want to embed the result of another controller in a template? That's very useful when working with Ajax, or when the embedded template needs some variable not available in the main template.

If you create a **fancy** action, and want to include it into the `index.html.php` template, simply use the following code:

```
Listing 137-14 1 <!-- app/Resources/views/Hello/index.html.php -->
                2 <?php echo $view['actions']->render(
                3     new \Symfony\Component\HttpKernel\Controller\ControllerReference('AppBundle:Hello:fancy', array(
                4         'name' => $name,
                5         'color' => 'green',
                6     ))
                7 ) ?>
```

Here, the `AppBundle:Hello:fancy` string refers to the **fancy** action of the **Hello** controller:

```
Listing 137-15 1 // src/AppBundle/Controller/HelloController.php
                2
                3 class HelloController extends Controller
                4 {
                5     public function fancyAction($name, $color)
                6     {
                7         // create some object, based on the $color variable
                8         $object = ...;
                9
                10     return $this->render('AppBundle:Hello:fancy.html.php', array(
                11         'name' => $name,
                12         'object' => $object
                13     ));
                14 }
                15
                16 // ...
                17 }
```

But where is the `$view['actions']` array element defined? Like `$view['slots']`, it's called a template helper, and the next section tells you more about those.

Using Template Helpers

The Symfony templating system can be easily extended via helpers. Helpers are PHP objects that provide features useful in a template context. **actions** and **slots** are two of the built-in Symfony helpers.

Creating Links between Pages

Speaking of web applications, creating links between pages is a must. Instead of hardcoding URLs in templates, the **router** helper knows how to generate URLs based on the routing configuration. That way, all your URLs can be easily updated by changing the configuration:

```
Listing 137-16 1 <a href="<?php echo $view['router']->path('hello', array('name' => 'Thomas')) ?>">
2     Greet Thomas!
3 </a>
```

The **path()** method takes the route name and an array of parameters as arguments. The route name is the main key under which routes are referenced and the parameters are the values of the placeholders defined in the route pattern:

```
Listing 137-17 1 # src/AppBundle/Resources/config/routing.yml
2 hello: # The route name
3     path: /hello/{name}
4     defaults: { _controller: AppBundle:Hello:index }
```

Using Assets: Images, JavaScripts and Stylesheets

What would the Internet be without images, JavaScripts, and stylesheets? Symfony provides the **assets** tag to deal with them easily:

```
Listing 137-18 1 <link href="<?php echo $view['assets']->getUrl('css/blog.css') ?>" rel="stylesheet" type="text/css" />
2
3 
```

The **assets** helper's main purpose is to make your application more portable. Thanks to this helper, you can move the application root directory anywhere under your web root directory without changing anything in your template's code.

Profiling Templates

By using the **stopwatch** helper, you are able to time parts of your template and display it on the timeline of the WebProfilerBundle:

```
Listing 137-19 <?php $view['stopwatch']->start('foo') ?>
... things that get timed
<?php $view['stopwatch']->stop('foo') ?>
```



If you use the same name more than once in your template, the times are grouped on the same line in the timeline.

Output Escaping

When using PHP templates, escape variables whenever they are displayed to the user:

```
Listing 137-20 <?php echo $view->escape($var) ?>
```

By default, the **escape()** method assumes that the variable is outputted within an HTML context. The second argument lets you change the context. For instance, to output something in a JavaScript script, use the **js** context:

```
Listing 137-21 <?php echo $view->escape($var, 'js') ?>
```



Chapter 138

How to Write a custom Twig Extension

The main motivation for writing an extension is to move often used code into a reusable class like adding support for internationalization. An extension can define tags, filters, tests, operators, global variables, functions, and node visitors.

Creating an extension also makes for a better separation of code that is executed at compilation time and code needed at runtime. As such, it makes your code faster.



Before writing your own extensions, have a look at the *Twig official extension repository*¹.

Create the Extension Class



This cookbook describes how to write a custom Twig extension as of Twig 1.12. If you are using an older version, please read *Twig extensions documentation legacy*².

To get your custom functionality you must first create a Twig Extension class. As an example you'll create a price filter to format a given number into price:

```
Listing 138-1 1 // src/AppBundle/Twig/AppExtension.php
2 namespace AppBundle\Twig;
3
4 class AppExtension extends \Twig_Extension
5 {
6     public function getFilters()
7     {
8         return array(
9             new \Twig_SimpleFilter('price', array($this, 'priceFilter')),
10        );
11    }
12 }
```

1. <https://github.com/twigphp/Twig-extensions>

2. http://twig.sensiolabs.org/doc/advanced_legacy.html#creating-an-extension

```

13     public function priceFilter($number, $decimals = 0, $decPoint = '.', $thousandsSep = ',')
14     {
15         $price = number_format($number, $decimals, $decPoint, $thousandsSep);
16         $price = '$'.$price;
17
18         return $price;
19     }
20
21     public function getName()
22     {
23         return 'app_extension';
24     }
25 }

```



Along with custom filters, you can also add custom *functions* and register *global variables*.

Register an Extension as a Service

Now you must let the Service Container know about your newly created Twig Extension:

Listing 138-2

```

1 # app/config/services.yml
2 services:
3     app.twig_extension:
4         class: AppBundle\Twig\AppExtension
5         public: false
6         tags:
7             - { name: twig.extension }

```

Using the custom Extension

Using your newly created Twig Extension is no different than any other:

Listing 138-3

```

1 {# outputs $5,500.00 #}
2 {{ '5500'|price }}

```

Passing other arguments to your filter:

Listing 138-4

```

1 {# outputs $5500,2516 #}
2 {{ '5500.2515'|price(4, ',', '') }}

```

Learning further

For a more in-depth look into Twig Extensions, please take a look at the *Twig extensions documentation*³.

3. <http://twig.sensiolabs.org/doc/advanced.html#creating-an-extension>



Chapter 139

How to Render a Template without a custom Controller

Usually, when you need to create a page, you need to create a controller and render a template from within that controller. But if you're rendering a simple template that doesn't need any data passed into it, you can avoid creating the controller entirely, by using the built-in `FrameworkBundle:Template:template` controller.

For example, suppose you want to render a `static/privacy.html.twig` template, which doesn't require that any variables are passed to it. You can do this without creating a controller:

```
Listing 139-1 1 acme_privacy:
                2     path: /privacy
                3     defaults:
                4         _controller: FrameworkBundle:Template:template
                5         template:    static/privacy.html.twig
```

The `FrameworkBundle:Template:template` controller will simply render whatever template you've passed as the `template` default value.

You can of course also use this trick when rendering embedded controllers from within a template. But since the purpose of rendering a controller from within a template is typically to prepare some data in a custom controller, this is probably only useful if you'd like to cache this page partial (see Caching the static Template).

```
Listing 139-2 1 {{ render(url('acme_privacy')) }}
```

Caching the static Template

Since templates that are rendered in this way are typically static, it might make sense to cache them. Fortunately, this is easy! By configuring a few other variables in your route, you can control exactly how your page is cached:

```
Listing 139-3
```

```
1  acme_privacy:
2    path: /privacy
3    defaults:
4      _controller: FrameworkBundle:Template:template
5      template:    'static/privacy.html.twig'
6      maxAge:      86400
7      sharedAge:   86400
```

The **maxAge** and **sharedAge** values are used to modify the Response object created in the controller. For more information on caching, see *HTTP Cache*.

There is also a **private** variable (not shown here). By default, the Response will be made public, as long as **maxAge** or **sharedAge** are passed. If set to **true**, the Response will be marked as private.



Chapter 140

How to Simulate HTTP Authentication in a Functional Test

If your application needs HTTP authentication, pass the username and password as server variables to `createClient()`:

Listing 140-1

```
$client = static::createClient(array(), array(  
    'PHP_AUTH_USER' => 'username',  
    'PHP_AUTH_PW'   => 'pa$$word',  
));
```

You can also override it on a per request basis:

Listing 140-2

```
$client->request('DELETE', '/post/12', array(), array(), array(  
    'PHP_AUTH_USER' => 'username',  
    'PHP_AUTH_PW'   => 'pa$$word',  
));
```

When your application is using a `form_login`, you can simplify your tests by allowing your test configuration to make use of HTTP authentication. This way you can use the above to authenticate in tests, but still have your users log in via the normal `form_login`. The trick is to include the `http_basic` key in your firewall, along with the `form_login` key:

Listing 140-3

```
1 # app/config/config_test.yml  
2 security:  
3     firewalls:  
4         your_firewall_name:  
5             http_basic: ~
```



Chapter 141

How to Simulate Authentication with a Token in a Functional Test

Authenticating requests in functional tests might slow down the suite. It could become an issue especially when `form_login` is used, since it requires additional requests to fill in and submit the form.

One of the solutions is to configure your firewall to use `http_basic` in the test environment as explained in *How to Simulate HTTP Authentication in a Functional Test*. Another way would be to create a token yourself and store it in a session. While doing this, you have to make sure that an appropriate cookie is sent with a request. The following example demonstrates this technique:

```
Listing 141-1 1 // tests/AppBundle/Controller/DefaultControllerTest.php
2 namespace Tests\Appbundle\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;
5 use Symfony\Component\BrowserKit\Cookie;
6 use Symfony\Component\Security\Core\Authentication\Token\UsernamePasswordToken;
7
8 class DefaultControllerTest extends WebTestCase
9 {
10     private $client = null;
11
12     public function setUp()
13     {
14         $this->client = static::createClient();
15     }
16
17     public function testSecuredHello()
18     {
19         $this->login();
20
21         $crawler = $this->client->request('GET', '/admin');
22
23         $this->assertTrue($this->client->getResponse()->isSuccessful());
24         $this->assertGreaterThan(0, $crawler->filter('html:contains("Admin Dashboard")')->count());
25     }
26
27     private function login()
28     {
29         $session = $this->client->getContainer()->get('session');
```



```

31     // the firewall context (defaults to the firewall name)
32     $firewall = 'secured_area';
33
34     $token = new UsernamePasswordToken('admin', null, $firewall, array('ROLE_ADMIN'));
35     $session->set('_security_'. $firewall, serialize($token));
36     $session->save();
37
38     $cookie = new Cookie($session->getName(), $session->getId());
39     $this->client->getCookieJar()->set($cookie);
40 }
41 }

```



The technique described in *How to Simulate HTTP Authentication in a Functional Test* is cleaner and therefore the preferred way.



Chapter 142

How to Test the Interaction of several Clients

If you need to simulate an interaction between different clients (think of a chat for instance), create several clients:

Listing 142-1

```
1 // ...
2
3 $harry = static::createClient();
4 $sally = static::createClient();
5
6 $harry->request('POST', '/say/sally/Hello');
7 $sally->request('GET', '/messages');
8
9 $this->assertEquals(Response::HTTP_CREATED, $harry->getResponse()->getStatusCode());
10 $this->assertRegExp('/Hello/', $sally->getResponse()->getContent());
```

This works except when your code maintains a global state or if it depends on a third-party library that has some kind of global state. In such a case, you can insulate your clients:

Listing 142-2

```
1 // ...
2
3 $harry = static::createClient();
4 $sally = static::createClient();
5
6 $harry->insulate();
7 $sally->insulate();
8
9 $harry->request('POST', '/say/sally/Hello');
10 $sally->request('GET', '/messages');
11
12 $this->assertEquals(Response::HTTP_CREATED, $harry->getResponse()->getStatusCode());
13 $this->assertRegExp('/Hello/', $sally->getResponse()->getContent());
```

Insulated clients transparently execute their requests in a dedicated and clean PHP process, thus avoiding any side-effects.



As an insulated client is slower, you can keep one client in the main process, and insulate the other ones.



Chapter 143

How to Use the Profiler in a Functional Test

It's highly recommended that a functional test only tests the Response. But if you write functional tests that monitor your production servers, you might want to write tests on the profiling data as it gives you a great way to check various things and enforce some metrics.

The Symfony Profiler gathers a lot of data for each request. Use this data to check the number of database calls, the time spent in the framework, etc. But before writing assertions, enable the profiler and check that the profiler is indeed available (it is enabled by default in the **test** environment):

```
Listing 143-1 1 class LuckyControllerTest extends WebTestCase
2 {
3     public function testNumberAction()
4     {
5         $client = static::createClient();
6
7         // Enable the profiler for the next request
8         // (it does nothing if the profiler is not available)
9         $client->enableProfiler();
10
11         $crawler = $client->request('GET', '/lucky/number');
12
13         // ... write some assertions about the Response
14
15         // Check that the profiler is enabled
16         if ($profile = $client->getProfile()) {
17             // check the number of requests
18             $this->assertLessThan(
19                 10,
20                 $profile->getCollector('db')->getQueryCount()
21             );
22
23             // check the time spent in the framework
24             $this->assertLessThan(
25                 500,
26                 $profile->getCollector('time')->getDuration()
27             );
28         }
29     }
30 }
```

If a test fails because of profiling data (too many DB queries for instance), you might want to use the Web Profiler to analyze the request after the tests finish. It's easy to achieve if you embed the token in the error message:

Listing 143-2

```

1 $this->assertLessThan(
2     30,
3     $profile->getCollector('db')->getQueryCount(),
4     sprintf(
5         'Checks that query count is less than 30 (token %s)',
6         $profile->getToken()
7     )
8 );

```



The profiler store can be different depending on the environment (especially if you use the SQLite store, which is the default configured one).



The profiler information is available even if you insulate the client or if you use an HTTP layer for your tests.



Read the API for built-in *data collectors* to learn more about their interfaces.

Speeding up Tests by not Collecting Profiler Data

To avoid collecting data in each test you can set the **collect** parameter to false:

Listing 143-3

```

1 # app/config/config_test.yml
2
3 # ...
4 framework:
5     profiler:
6         enabled: true
7         collect: false

```

In this way only tests that call `$client->enableProfiler()` will collect data.



Chapter 144

How to Test Code that Interacts with the Database

If your code interacts with the database, e.g. reads data from or stores data into it, you need to adjust your tests to take this into account. There are many ways how to deal with this. In a unit test, you can create a mock for a **Repository** and use it to return expected objects. In a functional test, you may need to prepare a test database with predefined values to ensure that your test always has the same data to work with.



If you want to test your queries directly, see *How to Test Doctrine Repositories*.

Mocking the Repository in a Unit Test

If you want to test code which depends on a Doctrine repository in isolation, you need to mock the **Repository**. Normally you inject the **EntityManager** into your class and use it to get the repository. This makes things a little more difficult as you need to mock both the **EntityManager** and your repository class.



It is possible (and a good idea) to inject your repository directly by registering your repository as a *factory service*. This is a little bit more work to setup, but makes testing easier as you only need to mock the repository.

Suppose the class you want to test looks like this:

```
Listing 144-1 1 // src/AppBundle/Salary/SalaryCalculator.php
2 namespace AppBundle\Salary;
3
4 use Doctrine\Common\Persistence\ObjectManager;
5
6 class SalaryCalculator
```

```

7 {
8     private $entityManager;
9
10    public function __construct(ObjectManager $entityManager)
11    {
12        $this->entityManager = $entityManager;
13    }
14
15    public function calculateTotalSalary($id)
16    {
17        $employeeRepository = $this->entityManager
18            ->getRepository('AppBundle:Employee');
19        $employee = $employeeRepository->find($id);
20
21        return $employee->getSalary() + $employee->getBonus();
22    }
23 }

```

Since the **ObjectManager** gets injected into the class through the constructor, it's easy to pass a mock object within a test:

Listing 144-2

```

1 // tests/AppBundle/Salary/SalaryCalculatorTest.php
2 namespace Tests\AppBundle\Salary;
3
4 use AppBundle\Salary\SalaryCalculator;
5 use AppBundle\Entity\Employee;
6 use Doctrine\ORM\EntityRepository;
7 use Doctrine\Common\Persistence\ObjectManager;
8
9 class SalaryCalculatorTest extends \PHPUnit_Framework_TestCase
10 {
11     public function testCalculateTotalSalary()
12     {
13         // First, mock the object to be used in the test
14         $employee = $this->getMock(Employee::class);
15         $employee->expects($this->once())
16             ->method('getSalary')
17             ->will($this->returnValue(1000));
18         $employee->expects($this->once())
19             ->method('getBonus')
20             ->will($this->returnValue(1100));
21
22         // Now, mock the repository so it returns the mock of the employee
23         $employeeRepository = $this
24             ->getMockBuilder(EntityRepository::class)
25             ->disableOriginalConstructor()
26             ->getMock();
27         $employeeRepository->expects($this->once())
28             ->method('find')
29             ->will($this->returnValue($employee));
30
31         // Last, mock the EntityManager to return the mock of the repository
32         $entityManager = $this
33             ->getMockBuilder(ObjectManager::class)
34             ->disableOriginalConstructor()
35             ->getMock();
36         $entityManager->expects($this->once())
37             ->method('getRepository')
38             ->will($this->returnValue($employeeRepository));
39
40         $salaryCalculator = new SalaryCalculator($entityManager);
41         $this->assertEquals(2100, $salaryCalculator->calculateTotalSalary(1));
42     }
43 }

```

In this example, you are building the mocks from the inside out, first creating the employee which gets returned by the **Repository**, which itself gets returned by the **EntityManager**. This way, no real class is involved in testing.

Changing Database Settings for Functional Tests

If you have functional tests, you want them to interact with a real database. Most of the time you want to use a dedicated database connection to make sure not to overwrite data you entered when developing the application and also to be able to clear the database before every test.

To do this, you can specify a database configuration which overwrites the default configuration:

Listing 144-3

```
1 # app/config/config_test.yml
2 doctrine:
3     # ...
4     dbal:
5         host:      localhost
6         dbname:    testdb
7         user:      testdb
8         password: testdb
```

Make sure that your database runs on localhost and has the defined database and user credentials set up.



Chapter 145

How to Test Doctrine Repositories

Unit testing Doctrine repositories in a Symfony project is not recommended. When you're dealing with a repository, you're really dealing with something that's meant to be tested against a real database connection.

Fortunately, you can easily test your queries against a real database, as described below.

Functional Testing

If you need to actually execute a query, you will need to boot the kernel to get a valid connection. In this case, you'll extend the `KernelTestCase`, which makes all of this quite easy:

```
Listing 145-1 1 // tests/AppBundle/Entity/ProductRepositoryTest.php
2 namespace Tests\AppBundle\Entity;
3
4 use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
5
6 class ProductRepositoryTest extends KernelTestCase
7 {
8     /**
9      * @var \Doctrine\ORM\EntityManager
10     */
11     private $em;
12
13     /**
14      * {@inheritdoc}
15     */
16     protected function setUp()
17     {
18         self::bootKernel();
19
20         $this->em = static::$kernel->getContainer()
21             ->get('doctrine')
22             ->getManager();
23     }
24
25     public function testSearchByCategoryName()
26     {
27         $products = $this->em
28             ->getRepository('AppBundle:Product')
```



```

29         ->searchByCategoryName('foo')
30     ;
31
32     $this->assertCount(1, $products);
33 }
34
35 /**
36  * {@inheritdoc}
37  */
38 protected function tearDown()
39 {
40     parent::tearDown();
41
42     $this->em->close();
43     $this->em = null; // avoid memory leaks
44 }
45 }

```



Chapter 146

How to Customize the Bootstrap Process before Running Tests

Sometimes when running tests, you need to do additional bootstrap work before running those tests. For example, if you're running a functional test and have introduced a new translation resource, then you will need to clear your cache before running those tests. This cookbook covers how to do that.

First, add the following file:

```
Listing 146-1 1 // app/tests.bootstrap.php
2 if (isset($_ENV['BOOTSTRAP_CLEAR_CACHE_ENV'])) {
3     passthru(sprintf(
4         'php "%s/console" cache:clear --env=%s --no-warmup',
5         __DIR__,
6         $_ENV['BOOTSTRAP_CLEAR_CACHE_ENV']
7     ));
8 }
9
10 require __DIR__.'/autoload.php';
```

Replace the test bootstrap file `autoload.php` in `phpunit.xml.dist` with `tests.bootstrap.php`:

```
Listing 146-2 1 <!-- phpunit.xml.dist -->
2
3 <!-- ... -->
4 <phpunit
5     bootstrap = "tests.bootstrap.php"
6 >
```

Now, you can define in your `phpunit.xml.dist` file which environment you want the cache to be cleared:

```
Listing 146-3 1 <!-- phpunit.xml.dist -->
2 <php>
3     <env name="BOOTSTRAP_CLEAR_CACHE_ENV" value="test"/>
4 </php>
```

This now becomes an environment variable (i.e. `$_ENV`) that's available in the custom bootstrap file (`tests.bootstrap.php`).



Chapter 147

Upgrading a Patch Version (e.g. 2.6.0 to 2.6.1)

When a new patch version is released (only the last number changed), it is a release that only contains bug fixes. This means that upgrading to a new patch version is *really* easy:

Listing 147-1 1 \$ composer update symfony/symfony

That's it! You should not encounter any backwards-compatibility breaks or need to change anything else in your code. That's because when you started your project, your **composer.json** included Symfony using a constraint like **2.6.***, where only the *last* version number will change when you update.



It is recommended to update to a new patch version as soon as possible, as important bugs and security leaks may be fixed in these new releases.

Upgrading other Packages

You may also want to upgrade the rest of your libraries. If you've done a good job with your *version constraints*¹ in **composer.json**, you can do this safely by running:

Listing 147-2 1 \$ composer update



Beware, if you have some unspecific *version constraints*² in your **composer.json** (e.g. **dev-master**), this could upgrade some non-Symfony libraries to new versions that contain backwards-compatibility breaking changes.

1. <https://getcomposer.org/doc/01-basic-usage.md#package-versions>

2. <https://getcomposer.org/doc/01-basic-usage.md#package-versions>



Chapter 148

Upgrading a Minor Version (e.g. 2.5.3 to 2.6.1)

If you're upgrading a minor version (where the middle number changes), then you should *not* encounter significant backwards compatibility changes. For details, see the *Symfony backwards compatibility promise*.

However, some backwards-compatibility breaks *are* possible and you'll learn in a second how to prepare for them.

There are two steps to upgrading a minor version:

1. Update the Symfony library via Composer;
2. Update your code to work with the new version.

1) Update the Symfony Library via Composer

First, you need to update Symfony by modifying your `composer.json` file to use the new version:

```
Listing 148-1 1 {  
2     "...": "...",  
3  
4     "require": {  
5         "symfony/symfony": "2.6.*",  
6     },  
7     "...": "...",  
8 }
```

Next, use Composer to download new versions of the libraries:

```
Listing 148-2 1 $ composer update symfony/symfony
```

Dependency Errors

If you get a dependency error, it may simply mean that you need to upgrade other Symfony dependencies too. In that case, try the following command:

```
Listing 148-3 1 $ composer update symfony/symfony --with-dependencies
```

This updates **symfony/symfony** and *all* packages that it depends on, which will include several other packages. By using tight version constraints in **composer.json**, you can control what versions each library upgrades to.

If this still doesn't work, your **composer.json** file may specify a version for a library that is not compatible with the newer Symfony version. In that case, updating that library to a newer version in **composer.json** may solve the issue.

Or, you may have deeper issues where different libraries depend on conflicting versions of other libraries. Check your error message to debug.

Upgrading other Packages

You may also want to upgrade the rest of your libraries. If you've done a good job with your *version constraints*¹ in **composer.json**, you can do this safely by running:

Listing 148-4 1 \$ composer update



Beware, if you have some unspecific *version constraints*² in your **composer.json** (e.g. **dev-master**), this could upgrade some non-Symfony libraries to new versions that contain backwards-compatibility breaking changes.

2) Updating your Code to Work with the new Version

In theory, you should be done! However, you *may* need to make a few changes to your code to get everything working. Additionally, some features you're using might still work, but might now be deprecated. While that's just fine, if you know about these deprecations, you can start to fix them over time.

Every version of Symfony comes with an UPGRADE file (e.g. *UPGRADE-2.7.md*³) included in the Symfony directory that describes these changes. If you follow the instructions in the document and update your code accordingly, it should be safe to update in the future.

These documents can also be found in the *Symfony Repository*⁴.

1. <https://getcomposer.org/doc/01-basic-usage.md#package-versions>

2. <https://getcomposer.org/doc/01-basic-usage.md#package-versions>

3. <https://github.com/symfony/symfony/blob/2.7/UPGRADE-2.7.md>

4. <https://github.com/symfony/symfony>



Chapter 149

Upgrading a Major Version (e.g. 2.7.0 to 3.0.0)

Every few years, Symfony releases a new major version release (the first number changes). These releases are the trickiest to upgrade, as they are allowed to contain BC breaks. However, Symfony tries to make this upgrade process as smooth as possible.

This means that you can update most of your code before the major release is actually released. This is called making your code *future compatible*.

There are a couple of steps to upgrading a major version:

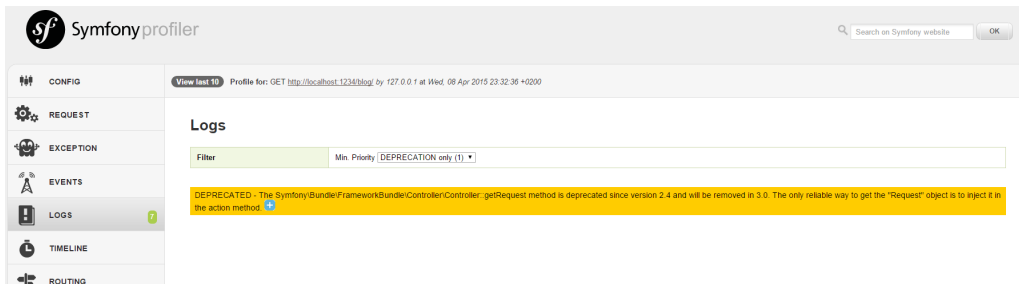
1. Make your code deprecation free;
2. Update to the new major version via Composer;
3. Update your code to work with the new version.

1) Make your Code Deprecation Free

During the lifecycle of a major release, new features are added and method signatures and public API usages are changed. However, *minor versions* should not contain any backwards incompatible changes. To accomplish this, the "old" (e.g. functions, classes, etc) code still works, but is marked as *deprecated*, indicating that it will be removed/changed in the future and that you should stop using it.

When the major version is released (e.g. 3.0.0), all deprecated features and functionality are removed. So, as long as you've updated your code to stop using these deprecated features in the last version before the major (e.g. 2.8.*), you should be able to upgrade without a problem.

To help you with this, deprecation notices are triggered whenever you end up using a deprecated feature. When visiting your application in the *dev environment* in your browser, these notices are shown in the web dev toolbar:



Of course ultimately, you want to stop using the deprecated functionality. Sometimes, this is easy: the warning might tell you exactly what to change.

But other times, the warning might be unclear: a setting somewhere might cause a class deeper to trigger the warning. In this case, Symfony does its best to give a clear message, but you may need to research that warning further.

And sometimes, the warning may come from a third-party library or bundle that you're using. If that's true, there's a good chance that those deprecations have already been updated. In that case, upgrade the library to fix them.

Once all the deprecation warnings are gone, you can upgrade with a lot more confidence.

Deprecations in PHPUnit

When you run your tests using PHPUnit, no deprecation notices are shown. To help you here, Symfony provides a PHPUnit bridge. This bridge will show you a nice summary of all deprecation notices at the end of the test report.

All you need to do is install the PHPUnit bridge:

Listing 149-1 `$ composer require --dev symfony/phpunit-bridge`

Now, you can start fixing the notices:

Listing 149-2

```

1  $ phpunit
2  ...
3
4  OK (10 tests, 20 assertions)
5
6  Remaining deprecation notices (6)
7
8  The "request" service is deprecated and will be removed in 3.0. Add a typehint for
9  Symfony\Component\HttpFoundation\Request to your controller parameters to retrieve the
10 request instead: 6x
11   3x in PageAdminTest::testPageShow from Symfony\Cmf\SimpleCmsBundle\Tests\WebTest\Admin
12   2x in PageAdminTest::testPageList from Symfony\Cmf\SimpleCmsBundle\Tests\WebTest\Admin
13   1x in PageAdminTest::testPageEdit from Symfony\Cmf\SimpleCmsBundle\Tests\WebTest\Admin

```

Once you fixed them all, the command ends with **0** (success) and you're done!



Using the Weak Deprecations Mode

Sometimes, you can't fix all deprecations (e.g. something was deprecated in 2.8 and you still need to support 2.7). In these cases, you can still use the bridge to fix as many deprecations as possible and then switch to the weak test mode to make your tests pass again. You can do this by using the `SYMFONY_DEPRECATIONS_HELPER` env variable:

```
Listing 149-3 1 <!-- phpunit.xml.dist -->
2 <phpunit>
3   <!-- ... -->
4
5   <php>
6     <env name="SYMFONY_DEPRECATIONS_HELPER" value="weak"/>
7   </php>
8 </phpunit>
```

(you can also execute the command like `SYMFONY_DEPRECATIONS_HELPER=weak phpunit`).

2) Update to the New Major Version via Composer

Once your code is deprecation free, you can update the Symfony library via Composer by modifying your `composer.json` file:

```
Listing 149-4 1 {
2   "...": "...",
3
4   "require": {
5     "symfony/symfony": "3.0.*",
6   },
7   "...": "..."
8 }
```

Next, use Composer to download new versions of the libraries:

```
Listing 149-5 1 $ composer update symfony/symfony
```

Dependency Errors

If you get a dependency error, it may simply mean that you need to upgrade other Symfony dependencies too. In that case, try the following command:

```
Listing 149-6 1 $ composer update symfony/symfony --with-dependencies
```

This updates `symfony/symfony` and *all* packages that it depends on, which will include several other packages. By using tight version constraints in `composer.json`, you can control what versions each library upgrades to.

If this still doesn't work, your `composer.json` file may specify a version for a library that is not compatible with the newer Symfony version. In that case, updating that library to a newer version in `composer.json` may solve the issue.

Or, you may have deeper issues where different libraries depend on conflicting versions of other libraries. Check your error message to debug.

Upgrading other Packages

You may also want to upgrade the rest of your libraries. If you've done a good job with your *version constraints*¹ in **composer.json**, you can do this safely by running:

Listing 149-7 1 \$ composer update



Beware, if you have some unspecific *version constraints*² in your **composer.json** (e.g. **dev-master**), this could upgrade some non-Symfony libraries to new versions that contain backwards-compatibility breaking changes.

3) Update your Code to Work with the New Version

There is a good chance that you're done now! However, the next major version *may* also contain new BC breaks as a BC layer is not always a possibility. Make sure you read the **UPGRADE-X.0.md** (where X is the new major version) included in the Symfony repository for any BC break that you need to be aware of.

1. <https://getcomposer.org/doc/01-basic-usage.md#package-versions>

2. <https://getcomposer.org/doc/01-basic-usage.md#package-versions>



Chapter 150

Upgrading a Third-Party Bundle for a Major Symfony Version

Symfony 3 was released on November 2015. Although this version doesn't contain any new feature, it removes all the backwards compatibility layers included in the previous 2.8 version. If your bundle uses any deprecated feature and it's published as a third-party bundle, applications upgrading to Symfony 3 will no longer be able to use it.

Allowing to Install Symfony 3 Components

Most third-party bundles define their Symfony dependencies using the `~2.N` or `^2.N` constraints in the `composer.json` file. For example:

```
Listing 150-1 1 {
2     "require": {
3         "symfony/framework-bundle": "~2.7",
4         "symfony/finder": "~2.7",
5         "symfony/validator": "~2.7"
6     }
7 }
```

These constraints prevent the bundle from using Symfony 3 components, so it makes it impossible to install it in a Symfony 3 based application. This issue is very easy to solve thanks to the flexibility of Composer dependencies constraints. Just replace `~2.N` by `~2.N|~3.0` (or `^2.N` by `^2.N|~3.0`).

The above example can be updated to work with Symfony 3 as follows:

```
Listing 150-2 1 {
2     "require": {
3         "symfony/framework-bundle": "~2.7|~3.0",
4         "symfony/finder": "~2.7|~3.0",
5         "symfony/validator": "~2.7|~3.0"
6     }
7 }
```



Another common version constraint found on third-party bundles is `>=2.N`. You should avoid using that constraint because it's too generic (it means that your bundle is compatible with any future Symfony version). Use instead `~2.N|~3.0` or `^2.N|~3.0` to make your bundle future-proof.

Looking for Deprecations and Fix Them

Besides allowing users to use your bundle with Symfony 3, your bundle must stop using any feature deprecated by the 2.8 version because they are removed in 3.0 (you'll get exceptions or PHP errors). The easiest way to detect deprecations is to install the *symfony/phpunit-bridge* package¹ and then run the test suite.

First, install the component as a **dev** dependency of your bundle:

Listing 150-3 1 `$ composer require --dev symfony/phpunit-bridge`

Then, run your test suite and look for the deprecation list displayed after the PHPUnit test report:

Listing 150-4

```
1 $ phpunit
2
3 # ... PHPUnit output
4
5 Remaining deprecation notices (3)
6
7 The "pattern" option in file ... is deprecated since version 2.2 and will be
8 removed in 3.0. Use the "path" option in the route definition instead ...
9
10 Twig Function "form_encytype" is deprecated. Use "form_start" instead in ...
11
12 The Symfony\Component\Security\Core\SecurityContext class is deprecated since
13 version 2.6 and will be removed in 3.0. Use ...
```

Fix the reported deprecations, run the test suite again and repeat the process until no deprecation usage is reported.

Useful Resources

There are several resources that can help you detect, understand and fix the use of deprecated features:

Official Symfony Guide to Upgrade from 2.x to 3.0²

The full list of changes required to upgrade to Symfony 3.0 and grouped by component.

SensioLabs DeprecationDetector³

It runs a static code analysis against your project's source code to find usages of deprecated methods, classes and interfaces. It works for any PHP application, but it includes special detectors for Symfony applications, where it can also detect usages of deprecated services.

Symfony Upgrade Fixer⁴

It analyzes Symfony projects to find deprecations. In addition it solves automatically some of them thanks to the growing list of supported "fixers".

1. <https://github.com/symfony/phpunit-bridge>

2. <https://github.com/symfony/symfony/blob/2.8/UPGRADE-3.0.md>

3. <https://github.com/sensiolabs-de/deprecation-detector>

4. <https://github.com/umpirsky/Symfony-Upgrade-Fixer>

Testing your Bundle in Symfony 3

Now that your bundle has removed all deprecations, it's time to test it for real in a Symfony 3 application. Assuming that you already have a Symfony 3 application, you can test the updated bundle locally without having to install it through Composer.

If your operating system supports symbolic links, just point the appropriate vendor directory to your local bundle root directory:

```
Listing 150-5 1 $ ln -s /path/to/your/local/bundle/ vendor/you-vendor-name/your-bundle-name
```

If your operating system doesn't support symbolic links, you'll need to copy your local bundle directory into the appropriate directory inside **vendor/**.

Update the Travis CI Configuration

In addition to running tools locally, it's recommended to set-up Travis CI service to run the tests of your bundle using different Symfony configurations. Use the following recommended configuration as the starting point of your own configuration:

```
Listing 150-6 1 language: php
2 sudo: false
3 php:
4   - 5.3
5   - 5.6
6   - 7.0
7
8 matrix:
9   include:
10    - php: 5.3.3
11      env: COMPOSER_FLAGS='--prefer-lowest --prefer-stable' SYMFONY_DEPRECATIONS_HELPER=weak
12    - php: 5.6
13      env: SYMFONY_VERSION='2.7.*'
14    - php: 5.6
15      env: SYMFONY_VERSION='2.8.*'
16    - php: 5.6
17      env: SYMFONY_VERSION='3.0.*'
18    - php: 5.6
19      env: SYMFONY_VERSION='3.1.*'
20    - php: 5.6
21      env: DEPENDENCIES='dev' SYMFONY_VERSION='3.2.*@dev'
22
23 before_install:
24   - composer self-update
25   - if [ "$DEPENDENCIES" == "dev" ]; then perl -pi -e 's/^\s*, "minimum-stability": "dev"/' composer.json;
26 fi;
27   - if [ "$SYMFONY_VERSION" != "" ]; then composer --no-update require symfony/symfony:${SYMFONY_VERSION};
28 fi;
29
30 install: composer update $COMPOSER_FLAGS
31
32 script: phpunit
```

Updating your Code to Support Symfony 2.x and 3.x at the Same Time

The real challenge of adding Symfony 3 support for your bundles is when you want to support both Symfony 2.x and 3.x simultaneously using the same code. There are some edge cases where you'll need to deal with the API differences.

Before diving into the specifics of the most common edge cases, the general recommendation is to **not rely on the Symfony Kernel version** to decide which code to use:

Listing 150-7

```
1 if (Kernel::VERSION_ID <= 20800) {
2     // code for Symfony 2.x
3 } else {
4     // code for Symfony 3.x
5 }
```

Instead of checking the Symfony Kernel version, check the version of the specific component. For example, the OptionsResolver API changed in its 2.6 version by adding a **setDefined()** method. The recommended check in this case would be:

Listing 150-8

```
1 if (!method_exists('Symfony\Component\OptionsResolver\OptionsResolver', 'setDefined')) {
2     // code for the old OptionsResolver API
3 } else {
4     // code for the new OptionsResolver API
5 }
```



Chapter 151

How to Create a custom Validation Constraint

You can create a custom constraint by extending the base constraint class, *Constraint*¹. As an example you're going to create a simple validator that checks if a string contains only alphanumeric characters.

Creating the Constraint Class

First you need to create a Constraint class and extend *Constraint*²:

Listing 151-1

```
1 // src/AppBundle/Validator/Constraints/ContainsAlphanumeric.php
2 namespace AppBundle\Validator\Constraints;
3
4 use Symfony\Component\Validator\Constraint;
5
6 /**
7  * @Annotation
8  */
9 class ContainsAlphanumeric extends Constraint
10 {
11     public $message = 'The string "%string%" contains an illegal character: it can only contain letters or
12     numbers.';
13 }
```



The **@Annotation** annotation is necessary for this new constraint in order to make it available for use in classes via annotations. Options for your constraint are represented as public properties on the constraint class.

1. <http://api.symfony.com/3.0/Symfony/Component/Validator/Constraint.html>
2. <http://api.symfony.com/3.0/Symfony/Component/Validator/Constraint.html>

Creating the Validator itself

As you can see, a constraint class is fairly minimal. The actual validation is performed by another "constraint validator" class. The constraint validator class is specified by the constraint's `validatedBy()` method, which includes some simple default logic:

```
Listing 151-2 1 // in the base Symfony\Component\Validator\Constraint class
2 public function validatedBy()
3 {
4     return get_class($this).'Validator';
5 }
```

In other words, if you create a custom **Constraint** (e.g. **MyConstraint**), Symfony will automatically look for another class, **MyConstraintValidator** when actually performing the validation.

The validator class is also simple, and only has one required method `validate()`:

```
Listing 151-3 1 // src/AppBundle\Validator/Constraints/ContainsAlphanumericValidator.php
2 namespace AppBundle\Validator\Constraints;
3
4 use Symfony\Component\Validator\Constraint;
5 use Symfony\Component\Validator\ConstraintValidator;
6
7 class ContainsAlphanumericValidator extends ConstraintValidator
8 {
9     public function validate($value, Constraint $constraint)
10    {
11        if (!preg_match('/^[a-zA-Z0-9]+$/', $value, $matches)) {
12            $this->context->buildViolation($constraint->message)
13                ->setParameter('%string%', $value)
14                ->addViolation();
15        }
16    }
17 }
```

Inside `validate`, you don't need to return a value. Instead, you add violations to the validator's `context` property and a value will be considered valid if it causes no violations. The `buildViolation` method takes the error message as its argument and returns an instance of *ConstraintViolationBuilderInterface*³. The `addViolation` method call finally adds the violation to the context.

Using the new Validator

Using custom validators is very easy, just as the ones provided by Symfony itself:

```
Listing 151-4 1 // src/AppBundle/Entity/AcmeEntity.php
2 use Symfony\Component\Validator\Constraints as Assert;
3 use AppBundle\Validator\Constraints as AcmeAssert;
4
5 class AcmeEntity
6 {
7     // ...
8
9     /**
10      * @Assert\NotBlank
11      * @AcmeAssert\ContainsAlphanumeric
12      */
13     protected $name;
14 }
```

3. <http://api.symfony.com/3.0/Symfony/Component/Validator/Violation/ConstraintViolationBuilderInterface.html>

```

15     // ...
16 }

```

If your constraint contains options, then they should be public properties on the custom Constraint class you created earlier. These options can be configured like options on core Symfony constraints.

Constraint Validators with Dependencies

If your constraint validator has dependencies, such as a database connection, it will need to be configured as a service in the Dependency Injection Container. This service must include the `validator.constraint_validator` tag and should include an `alias` attribute to be used in the `validatedBy` method of your validator class:

Listing 151-5

```

1  # app/config/services.yml
2  services:
3      validator.unique.your_validator_name:
4          class: Fully\Qualified\Validator\Class\Name
5          tags:
6              - { name: validator.constraint_validator, alias: alias_name }

```

As mentioned above, Symfony will automatically look for a class named after the constraint, with **Validator** appended. You can override this in your constraint class:

Listing 151-6

```

public function validatedBy()
{
    return 'Fully\Qualified\ConstraintValidator\Class\Name'; // or 'alias_name' if provided
}

```

Make sure to use the 'alias_name' when you have configured your validator as a service. Otherwise your validator class will be simply instantiated without your dependencies.

Class Constraint Validator

Beside validating a class property, a constraint can have a class scope by providing a target in its **Constraint** class:

Listing 151-7

```

public function getTargets()
{
    return self::CLASS_CONSTRAINT;
}

```

With this, the validator `validate()` method gets an object as its first argument:

Listing 151-8

```

1  class ProtocolClassValidator extends ConstraintValidator
2  {
3      public function validate($protocol, Constraint $constraint)
4      {
5          if ($protocol->getFoo() != $protocol->getBar()) {
6              $this->context->buildViolation($constraint->message)
7                  ->atPath('foo')
8                  ->addViolation();
9          }
10     }
11 }

```

Note that a class constraint validator is applied to the class itself, and not to the property:

Listing 151-9

```

1  /**
2   * @AcmeAssert\ContainsAlphanumeric
3   */
4  class AcmeEntity

```



```
5 {  
6 // ...  
7 }
```



Chapter 152

How to Handle Different Error Levels

Sometimes, you may want to display constraint validation error messages differently based on some rules. For example, you have a registration form for new users where they enter some personal information and choose their authentication credentials. They would have to choose a username and a secure password, but providing bank account information would be optional. Nonetheless, you want to make sure that these optional fields, if entered, are still valid, but display their errors differently.

The process to achieve this behavior consists of two steps:

1. Apply different error levels to the validation constraints;
2. Customize your error messages depending on the configured error level.

1. Assigning the Error Level

Use the `payload` option to configure the error level for each constraint:

```
Listing 152-1 1 // src/AppBundle/Entity/User.php
2 namespace AppBundle\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class User
7 {
8     /**
9      * @Assert\NotBlank(payload = {"severity" = "error"})
10     */
11     protected $username;
12
13     /**
14      * @Assert\NotBlank(payload = {"severity" = "error"})
15     */
16     protected $password;
17
18     /**
19      * @Assert\NotBlank(payload = {"severity" = "warning"})
20     */
21     protected $bankAccountNumber;
22 }
```

2. Customize the Error Message Template

When validation of the `User` object fails, you can retrieve the constraint that caused a particular failure using the `getConstraint()`¹ method. Each constraint exposes the attached payload as a public property:

```
Listing 152-2 1 // a constraint validation failure, instance of
                2 // Symfony\Component\Validator\ConstraintViolation
                3 $constraintViolation = ...;
                4 $constraint = $constraintViolation->getConstraint();
                5 $severity = isset($constraint->payload['severity']) ? $constraint->payload['severity'] : null;
```

For example, you can leverage this to customize the `form_errors` block so that the severity is added as an additional HTML class:

```
Listing 152-3 1 {%- block form_errors -%}
                2     {%- if errors|length > 0 -%}
                3     <ul>
                4         {%- for error in errors -%}
                5             {% if error.cause.constraint.payload.severity is defined %}
                6                 {% set severity = error.cause.constraint.payload.severity %}
                7                 {% endif %}
                8                 <li{% if severity is defined %} class="{{ severity }}"{% endif %}>{{ error.message }}</li>
                9             {%- endfor -%}
                10    </ul>
                11    {%- endif -%}
                12 {%- endblock form_errors -%}
```

For more information on customizing form rendering, see *How to Customize Form Rendering*.

1. http://api.symfony.com/3.0/Symfony/Component/Validator/ConstraintViolation.html#method_getConstraint



Chapter 153

How to Dynamically Configure Validation Groups

Sometimes you need advanced logic to determine the validation groups. If they can't be determined by a simple callback, you can use a service. Create a service that implements `__invoke` which accepts a `FormInterface` as a parameter.

```
Listing 153-1 1 // src/AppBundle/Validation/ValidationGroupResolver.php
2 namespace AppBundle\Validation;
3
4 use Symfony\Component\Form\FormInterface;
5
6 class ValidationGroupResolver
7 {
8     private $service1;
9
10    private $service2;
11
12    public function __construct($service1, $service2)
13    {
14        $this->service1 = $service1;
15        $this->service2 = $service2;
16    }
17
18    /**
19     * @param FormInterface $form
20     * @return array
21     */
22    public function __invoke(FormInterface $form)
23    {
24        $groups = array();
25
26        // ... determine which groups to apply and return an array
27
28        return $groups;
29    }
30 }
```

Then in your form, inject the resolver and set it as the `validation_groups`.

Listing 153-2

```

1  // src/AppBundle/Form/MyClassType.php;
2  namespace AppBundle\Form;
3
4  use AppBundle\Validator\ValidationGroupResolver;
5  use Symfony\Component\Form\AbstractType
6  use Symfony\Component\OptionsResolver\OptionsResolver;
7
8  class MyClassType extends AbstractType
9  {
10     private $groupResolver;
11
12     public function __construct(ValidationGroupResolver $groupResolver)
13     {
14         $this->groupResolver = $groupResolver;
15     }
16
17     // ...
18     public function configureOptions(OptionsResolver $resolver)
19     {
20         $resolver->setDefaults(array(
21             'validation_groups' => $this->groupResolver,
22         ));
23     }
24 }

```

This will result in the form validator invoking your group resolver to set the validation groups returned when validating.



Chapter 154

How to Use PHP's built-in Web Server

Since PHP 5.4 the CLI SAPI comes with a *built-in web server*¹. It can be used to run your PHP applications locally during development, for testing or for application demonstrations. This way, you don't have to bother configuring a full-featured web server such as *Apache* or *Nginx*.



The built-in web server is meant to be run in a controlled environment. It is not designed to be used on public networks.

Starting the Web Server

Running a Symfony application using PHP's built-in web server is as easy as executing the **server:start** command:

Listing 154-1 1 \$ php bin/console server:start

This starts the web server at **localhost:8000** in the background that serves your Symfony application. By default, the web server listens on port 8000 on the loopback device. You can change the socket passing an IP address and a port as a command-line argument:

Listing 154-2 1 \$ php bin/console server:start 192.168.0.1:8080



You can use the **--force** option to force the web server start if the process wasn't correctly stopped (without using the **server:stop** command).

Listing 154-3 1 \$ php bin/console server:start --force

1. <http://www.php.net/manual/en/features.commandline.webserver.php>



Listing 154-4

You can use the **server:status** command to check if a web server is listening on a certain socket:

```

1 $ php bin/console server:status
2
3 $ php bin/console server:status 192.168.0.1:8080

```

The first command shows if your Symfony application will be server through **localhost:8000**, the second one does the same for **192.168.0.1:8080**.



Some systems do not support the **server:start** command, in these cases you can execute the **server:run** command. This command behaves slightly different. Instead of starting the server in the background, it will block the current terminal until you terminate it (this is usually done by pressing Ctrl and C).



Using the built-in Web Server from inside a Virtual Machine

If you want to use the built-in web server from inside a virtual machine and then load the site from a browser on your host machine, you'll need to listen on the **0.0.0.0:8000** address (i.e. on all IP addresses that are assigned to the virtual machine):

Listing 154-5

```

1 $ php bin/console server:start 0.0.0.0:8000

```



You should **NEVER** listen to all interfaces on a computer that is directly accessible from the Internet. The built-in web server is not designed to be used on public networks.

Command Options

The built-in web server expects a "router" script (read about the "router" script on *php.net*²) as an argument. Symfony already passes such a router script when the command is executed in the **prod** or in the **dev** environment. Use the **--router** option in any other environment or to use another router script:

Listing 154-6

```

1 $ php bin/console server:start --env=test --router=app/config/router_test.php

```

If your application's document root differs from the standard directory layout, you have to pass the correct location using the **--docroot** option:

Listing 154-7

```

1 $ php bin/console server:start --docroot=public_html

```

Stopping the Server

When you are finished, you can simply stop the web server using the **server:stop** command:

Listing 154-8

```

1 $ php bin/console server:stop

```

2. <http://php.net/manual/en/features.commandline.webserver.php#example-411>

Like with the start command, if you omit the socket information, Symfony will stop the web server bound to **localhost:8000**. Just pass the socket information when the web server listens to another IP address or to another port:

Listing 154-9 1 \$ php bin/console server:stop 192.168.0.1:8080



Chapter 155

How to Create a SOAP Web Service in a Symfony Controller

Setting up a controller to act as a SOAP server is simple with a couple tools. You must, of course, have the *PHP SOAP*¹ extension installed. As the PHP SOAP extension can not currently generate a WSDL, you must either create one from scratch or use a 3rd party generator.



There are several SOAP server implementations available for use with PHP. *Zend SOAP*² and *NuSOAP*³ are two examples. Although the PHP SOAP extension is used in these examples, the general idea should still be applicable to other implementations.

SOAP works by exposing the methods of a PHP object to an external entity (i.e. the person using the SOAP service). To start, create a class - **HelloService** - which represents the functionality that you'll expose in your SOAP service. In this case, the SOAP service will allow the client to call a method called **hello**, which happens to send an email:

```
Listing 155-1 1 // src/Acme/SoapBundle/Services/HelloService.php
2 namespace Acme\SoapBundle\Services;
3
4 class HelloService
5 {
6     private $mailer;
7
8     public function __construct(\Swift_Mailer $mailer)
9     {
10         $this->mailer = $mailer;
11     }
12
13     public function hello($name)
14     {
15
16         $message = \Swift_Message::newInstance()
17             ->setTo('me@example.com')
```

-
1. <http://php.net/manual/en/book.soap.php>
 2. <http://framework.zend.com/manual/current/en/modules/zend.soap.server.html>
 3. <http://sourceforge.net/projects/nusoap>

```

18             ->setSubject('Hello Service')
19             ->setBody($name . ' says hi!');
20
21     $this->mailer->send($message);
22
23     return 'Hello, '.$name;
24 }
25 }

```

Next, you can train Symfony to be able to create an instance of this class. Since the class sends an email, it's been designed to accept a **Swift_Mailer** instance. Using the Service Container, you can configure Symfony to construct a **HelloService** object properly:

Listing 155-2

```

1 # app/config/services.yml
2 services:
3     hello_service:
4         class: Acme\SoapBundle\Services\HelloService
5         arguments: ['@mailer']

```

Below is an example of a controller that is capable of handling a SOAP request. If **indexAction()** is accessible via the route **/soap**, then the WSDL document can be retrieved via **/soap?wsdl**.

Listing 155-3

```

1 namespace Acme\SoapBundle\Controller;
2
3 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
4 use Symfony\Component\HttpFoundation\Response;
5
6 class HelloServiceController extends Controller
7 {
8     public function indexAction()
9     {
10         $server = new \SoapServer('/path/to/hello.wsdl');
11         $server->setObject($this->get('hello_service'));
12
13         $response = new Response();
14         $response->headers->set('Content-Type', 'text/xml; charset=ISO-8859-1');
15
16         ob_start();
17         $server->handle();
18         $response->setContent(ob_get_clean());
19
20         return $response;
21     }
22 }

```

Take note of the calls to **ob_start()** and **ob_get_clean()**. These methods control *output buffering*⁴ which allows you to "trap" the echoed output of **\$server->handle()**. This is necessary because Symfony expects your controller to return a **Response** object with the output as its "content". You must also remember to set the "Content-Type" header to "text/xml", as this is what the client will expect. So, you use **ob_start()** to start buffering the STDOUT and use **ob_get_clean()** to dump the echoed output into the content of the Response and clear the output buffer. Finally, you're ready to return the **Response**.

Below is an example calling the service using a **NuSOAP**⁵ client. This example assumes that the **indexAction** in the controller above is accessible via the route **/soap**:

Listing 155-4

```

$client = new \Soapclient('http://example.com/app.php/soap?wsdl');
$result = $client->call('hello', array('name' => 'Scott'));

```

4. <http://php.net/manual/en/book.outcontrol.php>
5. <http://sourceforge.net/projects/nusoap>

An example WSDL is below.

Listing 155-5

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <definitions xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
3   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
6   xmlns:tns="urn:arnleadservicewsd1"
7   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
8   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
9   xmlns="http://schemas.xmlsoap.org/wsdl/"
10  targetNamespace="urn:hellowsdl">
11
12  <types>
13    <xsd:schema targetNamespace="urn:hellowsdl">
14      <xsd:import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
15      <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/" />
16    </xsd:schema>
17  </types>
18
19  <message name="helloRequest">
20    <part name="name" type="xsd:string" />
21  </message>
22
23  <message name="helloResponse">
24    <part name="return" type="xsd:string" />
25  </message>
26
27  <portType name="hellowsdlPortType">
28    <operation name="hello">
29      <documentation>Hello World</documentation>
30      <input message="tns:helloRequest"/>
31      <output message="tns:helloResponse"/>
32    </operation>
33  </portType>
34
35  <binding name="hellowsdlBinding" type="tns:hellowsdlPortType">
36    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
37    <operation name="hello">
38      <soap:operation soapAction="urn:arnleadservicewsd1#hello" style="rpc"/>
39
40      <input>
41        <soap:body use="encoded" namespace="urn:hellowsdl"
42          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
43      </input>
44
45      <output>
46        <soap:body use="encoded" namespace="urn:hellowsdl"
47          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
48      </output>
49    </operation>
50  </binding>
51
52  <service name="hellowsdl">
53    <port name="hellowsdlPort" binding="tns:hellowsdlBinding">
54      <soap:address location="http://example.com/app.php/soap" />
55    </port>
56  </service>
57 </definitions>
```



Chapter 156

How to Create and Store a Symfony Project in Git



Though this entry is specifically about Git, the same generic principles will apply if you're storing your project in Subversion.

Once you've read through *Create your First Page in Symfony* and become familiar with using Symfony, you'll no-doubt be ready to start your own project. In this cookbook article, you'll learn the best way to start a new Symfony project that's stored using the *Git*¹ source control management system.

Initial Project Setup

To get started, you'll need to download Symfony and get things running. See the *Installing and Configuring Symfony* chapter for details.

Once your project is running, just follow these simple steps:

1. Initialize your Git repository:

Listing 156-1 1 \$ git init

2. Add all of the initial files to Git:

Listing 156-2 1 \$ git add .

1. <http://git-scm.com/>



As you might have noticed, not all files that were downloaded by Composer in step 1, have been staged for commit by Git. Certain files and folders, such as the project's dependencies (which are managed by Composer), **parameters.yml** (which contains sensitive information such as database credentials), log and cache files and dumped assets (which are created automatically by your project), should not be committed in Git. To help you prevent committing those files and folders by accident, the Standard Distribution comes with a file called **.gitignore**, which contains a list of files and folders that Git should ignore.



You may also want to create a **.gitignore** file that can be used system-wide. This allows you to exclude files/folders for all your projects that are created by your IDE or operating system. For details, see *GitHub .gitignore*².

3. Create an initial commit with your started project:

Listing 156-3 1 \$ git commit -m "Initial commit"

At this point, you have a fully-functional Symfony project that's correctly committed to Git. You can immediately begin development, committing the new changes to your Git repository.

You can continue to follow along with the *Create your First Page in Symfony* chapter to learn more about how to configure and develop inside your application.



The Symfony Standard Edition comes with some example functionality. To remove the sample code, follow the instructions in the *"How to Remove the AcmeDemoBundle"* article.

Managing Vendor Libraries with **composer.json**

How Does it Work?

Every Symfony project uses a group of third-party "vendor" libraries. One way or another the goal is to download these files into your **vendor/** directory and, ideally, to give you some sane way to manage the exact version you need for each.

By default, these libraries are downloaded by running a **composer install** "downloader" binary. This **composer** file is from a library called *Composer*³ and you can read more about installing it in the Installation chapter.

The **composer** command reads from the **composer.json** file at the root of your project. This is an JSON-formatted file, which holds a list of each of the external packages you need, the version to be downloaded and more. **composer** also reads from a **composer.lock** file, which allows you to pin each library to an **exact** version. In fact, if a **composer.lock** file exists, the versions inside will override those in **composer.json**. To upgrade your libraries to new versions, run **composer update**.



If you want to add a new package to your application, run the **composer require** command:

Listing 156-4 1 \$ composer require doctrine/doctrine-fixtures-bundle

2. <https://help.github.com/articles/ignoring-files>

3. <https://getcomposer.org/>

To learn more about Composer, see *GetComposer.org*⁴:

It's important to realize that these vendor libraries are *not* actually part of *your* repository. Instead, they're simply un-tracked files that are downloaded into the **vendor/**. But since all the information needed to download these files is saved in **composer.json** and **composer.lock** (which *are* stored in the repository), any other developer can use the project, run **composer install**, and download the exact same set of vendor libraries. This means that you're controlling exactly what each vendor library looks like, without needing to actually commit them to *your* repository.

So, whenever a developer uses your project, they should run the **composer install** script to ensure that all of the needed vendor libraries are downloaded.



Upgrading Symfony

Since Symfony is just a group of third-party libraries and third-party libraries are entirely controlled through **composer.json** and **composer.lock**, upgrading Symfony means simply upgrading each of these files to match their state in the latest Symfony Standard Edition.

Of course, if you've added new entries to **composer.json**, be sure to replace only the original parts (i.e. be sure not to also delete any of your custom entries).

Storing your Project on a remote Server

You now have a fully-functional Symfony project stored in Git. However, in most cases, you'll also want to store your project on a remote server both for backup purposes, and so that other developers can collaborate on the project.

The easiest way to store your project on a remote server is via a web-based hosting service like *GitHub*⁵ or *Bitbucket*⁶. Of course, there are more services out there, you can start your research with a *comparison of hosting services*⁷.

Alternatively, you can store your Git repository on any server by creating a *barebones repository*⁸ and then pushing to it. One library that helps manage this is *Gitolite*⁹.

4. <https://getcomposer.org/>

5. <https://github.com/>

6. <https://bitbucket.org/>

7. https://en.wikipedia.org/wiki/Comparison_of_open-source_software_hosting_facilities

8. <http://git-scm.com/book/en/Git-Basics-Getting-a-Git-Repository>

9. <https://github.com/sitaramc/gitolite>



Chapter 157

How to Create and Store a Symfony Project in Subversion



This entry is specifically about Subversion, and based on principles found in *How to Create and Store a Symfony Project in Git*.

Once you've read through *Create your First Page in Symfony* and become familiar with using Symfony, you'll no-doubt be ready to start your own project. The preferred method to manage Symfony projects is using *Git*¹ but some prefer to use *Subversion*² which is totally fine!. In this cookbook article, you'll learn how to manage your project using *SVN*³ in a similar manner you would do with *Git*⁴.



This is **a** method to tracking your Symfony project in a Subversion repository. There are several ways to do and this one is simply one that works.

The Subversion Repository

For this article it's assumed that your repository layout follows the widespread standard structure:

Listing 157-1

```
1 myproject/  
2   branches/  
3   tags/  
4   trunk/
```

1. <http://git-scm.com/>

2. <http://subversion.apache.org/>

3. <http://subversion.apache.org/>

4. <http://git-scm.com/>



Most Subversion hosting should follow this standard practice. This is the recommended layout in *Version Control with Subversion*⁵ and the layout used by most free hosting (see Subversion Hosting Solutions).

Initial Project Setup

To get started, you'll need to download Symfony and get the basic Subversion setup. First, download and get your Symfony project running by following the *Installation* chapter.

Once you have your new project directory and things are working, follow along with these steps:

1. Checkout the Subversion repository that will host this project. Suppose it is hosted on *Google code*⁶ and called **myproject**:

Listing 157-2 1 `$ svn checkout http://myproject.googlecode.com/svn/trunk myproject`

2. Copy the Symfony project files in the Subversion folder:

Listing 157-3 1 `$ mv Symfony/* myproject/`

3. Now, set the ignore rules. Not everything *should* be stored in your Subversion repository. Some files (like the cache) are generated and others (like the database configuration) are meant to be customized on each machine. This makes use of the **svn:ignore** property, so that specific files can be ignored.

Listing 157-4

```
1 $ cd myproject/
2 $ svn add --depth=empty app var var/cache var/logs app/config web
3
4 $ svn propset svn:ignore "vendor" .
5 $ svn propset svn:ignore "bootstrap*" var/
6 $ svn propset svn:ignore "parameters.yml" app/config/
7 $ svn propset svn:ignore "*" var/cache/
8 $ svn propset svn:ignore "*" var/logs/
9 $ svn propset svn:ignore "*" var/sessions/
10
11 $ svn propset svn:ignore "bundles" web
12
13 $ svn ci -m "commit basic Symfony ignore list (vendor, var/bootstrap*, app/config/parameters.yml,
    var/cache/*, var/logs/*, web/bundles)"
```

4. The rest of the files can now be added and committed to the project:

Listing 157-5

```
1 $ svn add --force .
2 $ svn ci -m "add basic Symfony Standard 3.X.Y"
```

That's it! Since the `app/config/parameters.yml` file is ignored, you can store machine-specific settings like database passwords here without committing them. The `parameters.yml.dist` file is committed, but is not read by Symfony. And by adding any new keys you need to both files, new developers can quickly clone the project, copy this file to `parameters.yml`, customize it, and start developing.

At this point, you have a fully-functional Symfony project stored in your Subversion repository. The development can start with commits in the Subversion repository.

You can continue to follow along with the *Create your First Page in Symfony* chapter to learn more about how to configure and develop inside your application.

5. <http://svnbook.red-bean.com/>

6. <http://code.google.com/hosting/>



The Symfony Standard Edition comes with some example functionality. To remove the sample code, follow the instructions in the "How to Remove the AcmeDemoBundle" article.

Managing Vendor Libraries with `composer.json`

How Does it Work?

Every Symfony project uses a group of third-party "vendor" libraries. One way or another the goal is to download these files into your `vendor/` directory and, ideally, to give you some sane way to manage the exact version you need for each.

By default, these libraries are downloaded by running a `composer install` "downloader" binary. This `composer` file is from a library called *Composer*⁷ and you can read more about installing it in the Installation chapter.

The `composer` command reads from the `composer.json` file at the root of your project. This is an JSON-formatted file, which holds a list of each of the external packages you need, the version to be downloaded and more. `composer` also reads from a `composer.lock` file, which allows you to pin each library to an **exact** version. In fact, if a `composer.lock` file exists, the versions inside will override those in `composer.json`. To upgrade your libraries to new versions, run `composer update`.



If you want to add a new package to your application, run the `composer require` command:

Listing 157-6 1 \$ `composer require doctrine/doctrine-fixtures-bundle`

To learn more about Composer, see *GetComposer.org*⁸:

It's important to realize that these vendor libraries are *not* actually part of *your* repository. Instead, they're simply un-tracked files that are downloaded into the `vendor/`. But since all the information needed to download these files is saved in `composer.json` and `composer.lock` (which *are* stored in the repository), any other developer can use the project, run `composer install`, and download the exact same set of vendor libraries. This means that you're controlling exactly what each vendor library looks like, without needing to actually commit them to *your* repository.

So, whenever a developer uses your project, they should run the `composer install` script to ensure that all of the needed vendor libraries are downloaded.



Upgrading Symfony

Since Symfony is just a group of third-party libraries and third-party libraries are entirely controlled through `composer.json` and `composer.lock`, upgrading Symfony means simply upgrading each of these files to match their state in the latest Symfony Standard Edition.

Of course, if you've added new entries to `composer.json`, be sure to replace only the original parts (i.e. be sure not to also delete any of your custom entries).

7. <https://getcomposer.org/>

8. <https://getcomposer.org/>

Subversion Hosting Solutions

The biggest difference between *Git*⁹ and *SVN*¹⁰ is that Subversion *needs* a central repository to work. You then have several solutions:

- Self hosting: create your own repository and access it either through the filesystem or the network. To help in this task you can read Version Control with Subversion.
- Third party hosting: there are a lot of serious free hosting solutions available like *GitHub*¹¹, *Google code*¹², *SourceForge*¹³ or *Gna*¹⁴. Some of them offer Git hosting as well.

9. <http://git-scm.com/>

10. <http://subversion.apache.org/>

11. <https://github.com/>

12. <http://code.google.com/hosting/>

13. <http://sourceforge.net/>

14. <http://gna.org/>



Chapter 158

Using Symfony with Homestead/Vagrant

In order to develop a Symfony application, you might want to use a virtual development environment instead of the built-in server or WAMP/LAMP. *Homestead*¹ is an easy-to-use *Vagrant*² box to get a virtual environment up and running quickly.



Due to the amount of filesystem operations in Symfony (e.g. updating cache files and writing to log files), Symfony can slow down significantly. To improve the speed, consider overriding the cache and log directories to a location outside the NFS share (for instance, by using `sys_get_temp_dir`³). You can read *this blog post*⁴ for more tips to speed up Symfony on Vagrant.

Install Vagrant and Homestead

Before you can use Homestead, you need to install and configure Vagrant and Homestead as explained in *the Homestead documentation*⁵.

Setting Up a Symfony Application

Imagine you've installed your Symfony application in `~/projects/symfony_demo` on your local system. You first need Homestead to sync your files in this project. Execute `homestead edit` to edit the Homestead configuration and configure the `~/projects` directory:

Listing 158-1

```
1 # ...
2 folders:
3     - map: ~/projects
4       to: /home/vagrant/projects
```

-
1. <http://laravel.com/docs/homestead>
 2. <https://www.vagrantup.com/>
 3. <http://php.net/manual/en/function.sys-get-temp-dir.php>
 4. http://www.whitewashing.de/2013/08/19/speedup_symfony2_on_vagrant_boxes.html
 5. <http://laravel.com/docs/homestead#installation-and-setup>

The **projects/** directory on your PC is now accessible at **/home/vagrant/projects** in the Homestead environment.

After you've done this, configure the Symfony application in the Homestead configuration:

Listing 158-2

```
1 # ...
2 sites:
3     - map: symfony-demo.dev
4       to: /home/vagrant/projects/symfony_demo/web
5       type: symfony
```

The **type** option tells Homestead to use the Symfony nginx configuration.

At last, edit the hosts file on your local machine to map **symfony-demo.dev** to **192.168.10.10** (which is the IP used by Homestead):

Listing 158-3

```
# /etc/hosts (unix) or C:\Windows\System32\drivers\etc\hosts (Windows)
192.168.10.10 symfony-demo.dev
```

Now, navigate to **http://symfony-demo.dev** in your web browser and enjoy developing your Symfony application!

To learn more features of Homestead, including Blackfire Profiler integration, automatic creation of MySQL databases and more, read the Daily Usage⁶ section of the Homestead documentation.

6. <http://laravel.com/docs/5.1/homestead#daily-usage>

