Symfony

Getting Started

Version: master

generated on August 31, 2019

Getting Started (master)

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

Contents at a Glance

Chapter 1
Installing & Setting up the Symfony Framework

Do you prefer video tutorials? Check out the Stellar Development with Symfony[1] *screencast series.*

Technical Requirements

Before creating your first Symfony application you must:

- Install PHP 7.1 or higher and these PHP extensions (which are installed and enabled by default in most PHP 7 installations): *Ctype*[2], *iconv*[3], *JSON*[4], *PCRE*[5], *Session*[6], *SimpleXML*[7], and *Tokenizer*[8];
- *Install Composer*[9], which is used to install PHP packages;
- *Install Symfony*[10], which creates in your computer a binary called `symfony` that provides all the tools you need to develop your application locally.

The `symfony` binary provides a tool to check if your computer meets these requirements. Open your console terminal and run this command:

Listing 1-1
```
$ symfony check:requirements
```

1. *http://symfonycasts.com/screencast/symfony*
2. https://php.net/book.ctype
3. https://php.net/book.iconv
4. https://php.net/book.json
5. https://php.net/book.pcre
6. https://php.net/book.session
7. https://php.net/book.simplexml
8. https://php.net/book.tokenizer
9. https://getcomposer.org/download/
10. https://symfony.com/download

Creating Symfony Applications

Open your console terminal and run any of these commands to create a new Symfony application:

Listing 1-2
```
# run this if you are building a traditional web application
$ symfony new --full my_project_name

# run this if you are building a microservice, console application or API
$ symfony new my_project_name
```

The only difference between these two commands is the number of packages installed by default. The `--full` option installs all the packages that you usually need to build web applications, so the installation size will be bigger.

If you can't or don't want to *install Symfony*[11] for any reason, run these commands to create the new Symfony application using Composer:

Listing 1-3
```
# run this if you are building a traditional web application
$ composer create-project symfony/website-skeleton my_project_name

# run this if you are building a microservice, console application or API
$ composer create-project symfony/skeleton my_project_name
```

No matter which command you run to create the Symfony application. All of them will create a new `my_project_name/` directory, download some dependencies into it and even generate the basic directories and files you'll need to get started. In other words, your new application is ready!

The project's cache and logs directory (by default, `<project>/var/cache/` and `<project>/var/log/`) must be writable by the web server. If you have any issue, read how to *set up permissions for Symfony applications*.

Running Symfony Applications

On production, you should use a web server like Nginx or Apache (see *configuring a web server to run Symfony*). But for development, it's more convenient to use the *local web server* provided by Symfony.

This local server provides support for HTTP/2, TLS/SSL, automatic generation of security certificates and many other features. It works with any PHP application, not only Symfony projects, so it's a very useful development tool.

Open your console terminal, move into your new project directory and start the local web server as follows:

Listing 1-4
```
$ cd my-project/
$ symfony server:start
```

Open your browser and navigate to `http://localhost:8000/`. If everything is working, you'll see a welcome page. Later, when you are finished working, stop the server by pressing `Ctrl+C` from your terminal.

11. `https://symfony.com/download`

Setting up an Existing Symfony Project

In addition to creating new Symfony projects, you will also work on projects already created by other developers. In that case, you only need to get the project code and install the dependencies with Composer. Assuming your team uses Git, setup your project with the following commands:

Listing 1-5
```
# clone the project to download its contents
$ cd projects/
$ git clone ...

# make Composer install the project's dependencies into vendor/
$ cd my-project/
$ composer install
```

You'll probably also need to customize your .env file and do a few other project-specific tasks (e.g. creating a database). When working on a existing Symfony application for the first time, it may be useful to run this command which displays information about the project:

Listing 1-6
```
$ php bin/console about
```

Installing Packages

A common practice when developing Symfony applications is to install packages (Symfony calls them *bundles*) that provide ready-to-use features. Packages usually require some setup before using them (editing some file to enable the bundle, creating some file to add some initial config, etc.)

Most of the time this setup can be automated and that's why Symfony includes *Symfony Flex*[12], a tool to simplify the installation/removal of packages in Symfony applications. Technically speaking, Symfony Flex is a Composer plugin that is installed by default when creating a new Symfony application and which **automates the most common tasks of Symfony applications**.

You can also *add Symfony Flex to an existing project*.

Symfony Flex modifies the behavior of the `require`, `update`, and `remove` Composer commands to provide advanced features. Consider the following example:

Listing 1-7
```
$ cd my-project/
$ composer require logger
```

If you execute that command in a Symfony application which doesn't use Flex, you'll see a Composer error explaining that `logger` is not a valid package name. However, if the application has Symfony Flex installed, that command installs and enables all the packages needed to use the official Symfony logger.

This is possible because lots of Symfony packages/bundles define **"recipes"**, which are a set of automated instructions to install and enable packages into Symfony applications. Flex keeps tracks of the recipes it installed in a `symfony.lock` file, which must be committed to your code repository.

Symfony Flex recipes are contributed by the community and they are stored in two public repositories:

- *Main recipe repository*[13], is a curated list of recipes for high quality and maintained packages. Symfony Flex only looks in this repository by default.

12. https://github.com/symfony/flex
13. https://github.com/symfony/recipes

- *Contrib recipe repository*[14], contains all the recipes created by the community. All of them are guaranteed to work, but their associated packages could be unmaintained. Symfony Flex will ask your permission before installing any of these recipes.

Read the *Symfony Recipes documentation*[15] to learn everything about how to create recipes for your own packages.

Checking Security Vulnerabilities

The `symfony` binary created when you *install Symfony*[16] provides a command to check whether your project's dependencies contain any known security vulnerability:

Listing 1-8
```
$ symfony check:security
```

A good security practice is to execute this command regularly to be able to update or replace compromised dependencies as soon as possible. The security check is done locally by cloning the public *PHP security advisories database*[17], so your `composer.lock` file is not sent on the network.

The `check:security` command terminates with a non-zero exit code if any of your dependencies is affected by a known security vulnerability. This way you can add it to your project build process and your continuous integration workflows to make them fail when there are vulnerabilities.

Symfony LTS Versions

According to the *Symfony release process*, "long-term support" (or LTS for short) versions are published every two years. Check out the *Symfony roadmap*[18] to know which is the latest LTS version.

By default, the command that creates new Symfony applications uses the latest stable version. If you want to use an LTS version, add the `--version` option:

Listing 1-9
```
# find the latest LTS version at https://symfony.com/roadmap
$ symfony new --version=3.4 my_project_name_name

# you can also base your project on development versions
$ symfony new --version=4.4.x-dev my_project_name
$ symfony new --version=dev-master my_project_name
```

The Symfony Demo application

The Symfony Demo Application[19] is a fully-functional application that shows the recommended way to develop Symfony applications. It's a great learning tool for Symfony newcomers and its code contains tons of comments and helpful notes.

Run this command to create a new project based on the Symfony Demo application:

Listing 1-10
```
$ symfony new --demo my_project_name
```

14. https://github.com/symfony/recipes-contrib
15. https://github.com/symfony/recipes/blob/master/README.rst
16. https://symfony.com/download
17. https://github.com/FriendsOfPHP/security-advisories
18. https://symfony.com/roadmap
19. https://github.com/symfony/demo

Start Coding!

With setup behind you, it's time to *Create your first page in Symfony*.

Learn More

- Using Symfony with Homestead/Vagrant
- Configuring a Web Server
- Upgrading a Third-Party Bundle for a Major Symfony Version
- Setting up or Fixing File Permissions
- Upgrading Existing Applications to Symfony Flex
- Symfony Local Web Server
- How to Install or Upgrade to the Latest, Unreleased Symfony Version
- Upgrading a Major Version (e.g. 3.4.0 to 4.1.0)
- Upgrading a Minor Version (e.g. 4.0.0 to 4.1.0)
- Upgrading a Patch Version (e.g. 4.1.0 to 4.1.1)

Chapter 2
Create your First Page in Symfony

Creating a new page - whether it's an HTML page or a JSON endpoint - is a two-step process:

1. **Create a route**: A route is the URL (e.g. `/about`) to your page and points to a controller;
2. **Create a controller**: A controller is the PHP function you write that builds the page. You take the incoming request information and use it to create a Symfony `Response` object, which can hold HTML content, a JSON string or even a binary file like an image or PDF.

Do you prefer video tutorials? Check out the Stellar Development with Symfony[1] *screencast series.*

Symfony embraces the HTTP Request-Response lifecycle. To find out more, see Symfony and HTTP Fundamentals.

Creating a Page: Route and Controller

Before continuing, make sure you've read the *Setup* article and can access your new Symfony app in the browser.

Suppose you want to create a page - `/lucky/number` - that generates a lucky (well, random) number and prints it. To do that, create a "Controller" class and a "controller" method inside of it:

Listing 2-1

```php
<?php
// src/Controller/LuckyController.php
namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;

class LuckyController
{
    public function number()
    {
        $number = random_int(0, 100);

```

1. *https://symfonycasts.com/screencast/symfony/setup*

```
13          return new Response(
14              '<html><body>Lucky number: '.$number.'</body></html>'
15          );
16      }
17  }
```

Now you need to associate this controller function with a public URL (e.g. `/lucky/number`) so that the `number()` method is executed when a user browses to it. This association is defined by creating a **route** in the `config/routes.yaml` file:

Listing 2-2
```
1  # config/routes.yaml
2
3  # the "app_lucky_number" route name is not important yet
4  app_lucky_number:
5      path: /lucky/number
6      controller: App\Controller\LuckyController::number
```

That's it! If you are using Symfony web server, try it out by going to:

> *http://localhost:8000/lucky/number*

If you see a lucky number being printed back to you, congratulations! But before you run off to play the lottery, check out how this works. Remember the two steps to creating a page?

1. ***Create a route*: In `config/routes.yaml`, the route defines the URL to your** page (`path`) and what `controller` to call. You'll learn more about *routing* in its own section, including how to make *variable* URLs;

2. *Create a controller*: This is a function where *you* build the page and ultimately return a `Response` object. You'll learn more about *controllers* in their own section, including how to return JSON responses.

Annotation Routes

Instead of defining your route in YAML, Symfony also allows you to use *annotation* routes. To do this, install the annotations package:

Listing 2-3
```
1  $ composer require annotations
```

You can now add your route directly *above* the controller:

Listing 2-4
```
1   // src/Controller/LuckyController.php
2
3   // ...
4   + use Symfony\Component\Routing\Annotation\Route;
5
6   class LuckyController
7   {
8   +     /**
9   +      * @Route("/lucky/number")
10  +      */
11      public function number()
12      {
13          // this looks exactly the same
14      }
15  }
```

That's it! The page - `http://localhost:8000/lucky/number` will work exactly like before! Annotations are the recommended way to configure routes.

Auto-Installing Recipes with Symfony Flex

You may not have noticed, but when you ran `composer require annotations`, two special things happened, both thanks to a powerful Composer plugin called Flex.

First, `annotations` isn't a real package name: it's an *alias* (i.e. shortcut) that Flex resolves to `sensio/framework-extra-bundle`.

Second, after this package was downloaded, Flex executed a *recipe*, which is a set of automated instructions that tell Symfony how to integrate an external package. *Flex recipes*[2] exist for many packages and have the ability to do a lot, like adding configuration files, creating directories, updating `.gitignore` and adding new config to your `.env` file. Flex *automates* the installation of packages so you can get back to coding.

The bin/console Command

Your project already has a powerful debugging tool inside: the `bin/console` command. Try running it:

Listing 2-5
```
1 $ php bin/console
```

You should see a list of commands that can give you debugging information, help generate code, generate database migrations and a lot more. As you install more packages, you'll see more commands.

To get a list of *all* of the routes in your system, use the `debug:router` command:

Listing 2-6
```
1 $ php bin/console debug:router
```

You should see your `app_lucky_number` route at the very top:

Name	Method	Scheme	Host	Path
app_lucky_number	ANY	ANY	ANY	/lucky/number

You will also see debugging routes below `app_lucky_number` -- more on the debugging routes in the next section.

You'll learn about many more commands as you continue!

The Web Debug Toolbar: Debugging Dream

One of Symfony's *killer* features is the Web Debug Toolbar: a bar that displays a *huge* amount of debugging information along the bottom of your page while developing. This is all included out of the box using a package called `symfony/profiler-pack`.

You will see a black bar along the bottom of the page. You'll learn more about all the information it holds along the way, but feel free to experiment: hover over and click the different icons to get information about routing, performance, logging and more.

2. `https://flex.symfony.com`

Rendering a Template

If you're returning HTML from your controller, you'll probably want to render a template. Fortunately, Symfony comes with *Twig*[3]: a templating language that's easy, powerful and actually quite fun.

Make sure that `LuckyController` extends Symfony's base *`AbstractController`*[4] class:

Listing 2-7

```
// src/Controller/LuckyController.php

// ...
+ use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

- class LuckyController
+ class LuckyController extends AbstractController
{
    // ...
}
```

Now, use the handy `render()` function to render a template. Pass it a `number` variable so you can use it in Twig:

Listing 2-8

```
// src/Controller/LuckyController.php

// ...
class LuckyController extends AbstractController
{
    /**
     * @Route("/lucky/number")
     */
    public function number()
    {
        $number = random_int(0, 100);

        return $this->render('lucky/number.html.twig', [
            'number' => $number,
        ]);
    }
}
```

Template files live in the `templates/` directory, which was created for you automatically when you installed Twig. Create a new `templates/lucky` directory with a new `number.html.twig` file inside:

Listing 2-9

```
{# templates/lucky/number.html.twig #}
<h1>Your lucky number is {{ number }}</h1>
```

The `{{ number }}` syntax is used to *print* variables in Twig. Refresh your browser to get your *new* lucky number!

> *http://localhost:8000/lucky/number*

Now you may wonder where the Web Debug Toolbar has gone: that's because there is no `</body>` tag in the current template. You can add the body element yourself, or extend `base.html.twig`, which contains all default HTML elements.

In the *Creating and Using Templates* article, you'll learn all about Twig: how to loop, render other templates and leverage its powerful layout inheritance system.

---

3. https://twig.symfony.com
4. https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php

Checking out the Project Structure

Great news! You've already worked inside the most important directories in your project:

`config/`
: Contains... configuration!. You will configure routes, *services* and packages.

`src/`
: All your PHP code lives here.

`templates/`
: All your Twig templates live here.

Most of the time, you'll be working in `src/`, `templates/` or `config/`. As you keep reading, you'll learn what can be done inside each of these.

So what about the other directories in the project?

`bin/`
: The famous `bin/console` file lives here (and other, less important executable files).

`var/`
: This is where automatically-created files are stored, like cache files (`var/cache/`) and logs (`var/log/`).

`vendor/`
: Third-party (i.e. "vendor") libraries live here! These are downloaded via the *Composer*[5] package manager.

`public/`
: This is the document root for your project: you put any publicly accessible files here.

And when you install new packages, new directories will be created automatically when needed.

What's Next?

Congrats! You're already starting to master Symfony and learn a whole new way of building beautiful, functional, fast and maintainable applications.

Ok, time to finish mastering the fundamentals by reading these articles:

- *Routing*
- *Controller*
- *Creating and Using Templates*
- *Configuring Symfony*

Then, learn about other important topics like the *service container*, the *form system*, using *Doctrine* (if you need to query a database) and more!

Have fun!

Go Deeper with HTTP & Framework Fundamentals

- Symfony versus Flat PHP
- Symfony and HTTP Fundamentals

5. `https://getcomposer.org`

Chapter 3
Routing

When your application receives a request, it executes a *controller action* to generate the response. The routing configuration defines which action to run for each incoming URL. It also provides other useful features, like generating SEO-friendly URLs (e.g. `/read/intro-to-symfony` instead of `index.php?article_id=57`).

Creating Routes

Routes can be configured in YAML, XML, PHP or using annotations. All formats provide the same features and performance, so choose your favorite. *Symfony recommends annotations* because it's convenient to put the route and controller in the same place instead of dealing with multiple files.

If you choose annotations, run this command once in your application to add support for them:

Listing 3-1
```
$ composer require annotations
```

Suppose you want to define a route for the `/blog` URL in your application:

Listing 3-2
```
// src/Controller/BlogController.php
namespace App\Controller;

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

class BlogController extends AbstractController
{
    /**
     * @Route("/blog", name="blog_list")
     */
    public function list()
    {
        // ...
    }
}
```

This configuration defines a route called `blog_list` that matches when the user requests the `/blog` URL. When the match occurs, the application runs the `list()` method of the `BlogController` class.

The query string of a URL is not considered when matching routes. In this example, URLs like `/blog?foo=bar` and `/blog?foo=bar&bar=foo` will also match the `blog_list` route.

The route name (`blog_list`) is not important for now, but it will be essential later when generating URLs. You only have to keep in mind that each route name must be unique in the application.

Matching HTTP Methods

By default, routes match any HTTP verb (`GET`, `POST`, `PUT`, etc.) Use the `methods` option to restrict the verbs each route should respond to:

Listing 3-3

```
// src/Controller/BlogApiController.php
namespace App\Controller;

// ...

class BlogApiController extends AbstractController
{
    /**
     * @Route("/api/posts/{id}", methods={"GET","HEAD"})
     */
    public function show(int $id)
    {
        // ... return a JSON response with the post
    }

    /**
     * @Route("/api/posts/{id}", methods={"PUT"})
     */
    public function edit(int $id)
    {
        // ... edit a post
    }
}
```

HTML forms only support `GET` and `POST` methods. If you're calling a route with a different method from an HTML form, add a hidden field called `_method` with the method to use (e.g. `<input type="hidden" name="_method" value="PUT"/>`). If you create your forms with *Symfony Forms* this is done automatically for you.

Matching Expressions

Use the `condition` option if you need some route to match based on some arbitrary matching logic:

Listing 3-4

```
// src/Controller/DefaultController.php
namespace App\Controller;

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

class DefaultController extends AbstractController
{
    /**
     * @Route(
     *     "/contact",
     *     name="contact",
     *     condition="context.getMethod() in ['GET', 'HEAD'] and request.headers.get('User-Agent') matches
'/firefox/i'"
     * )
     *
```

```
        * expressions can also include config parameters:
        * condition: "request.headers.get('User-Agent') matches '%app.allowed_browsers%'"
        */
       public function contact()
       {
           // ...
       }
   }
```

The value of the `condition` option is any valid *ExpressionLanguage expression* and can use any of these variables created by Symfony:

`context`
: An instance of *`RequestContext`*[1], which holds the most fundamental information about the route being matched.

`request`
: The Symfony Request object that represents the current request.

Behind the scenes, expressions are compiled down to raw PHP. Because of this, using the `condition` key causes no extra overhead beyond the time it takes for the underlying PHP to execute.

Conditions are *not* taken into account when generating URLs (which is explained later in this article).

Debugging Routes

As your application grows, you'll eventually have a *lot* of routes. Symfony includes some commands to help you debug routing issues. First, the `debug:router` command lists all your application routes in the same order in which Symfony evaluates them:

Listing 3-5

```
$ php bin/console debug:router

----------------  -------  -------  -----  ------------------------------------------
Name              Method   Scheme   Host   Path
----------------  -------  -------  -----  ------------------------------------------
homepage          ANY      ANY      ANY    /
contact           GET      ANY      ANY    /contact
contact_process   POST     ANY      ANY    /contact
article_show      ANY      ANY      ANY    /articles/{_locale}/{year}/{title}.{_format}
blog              ANY      ANY      ANY    /blog/{page}
blog_show         ANY      ANY      ANY    /blog/{slug}
----------------  -------  -------  -----  ------------------------------------------
```

Pass the name (or part of the name) of some route to this argument to print the route details:

Listing 3-6

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

+-------------+-------------------------------------------------------+
| Property    | Value                                                 |
+-------------+-------------------------------------------------------+
| Route Name  | app_lucky_number                                      |
| Path        | /lucky/number/{max}                                   |
| ...         | ...                                                   |
| Options     | compiler_class: Symfony\Component\Routing\RouteCompiler |
|             | utf8: true                                            |
+-------------+-------------------------------------------------------+
```

1. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Routing/RequestContext.php

The other command is called `router:match` and it shows which route will match the given URL. It's useful to find out why some URL is not executing the controller action that you expect:

Listing 3-7
```
$ php bin/console router:match /lucky/number/8

  [OK] Route "app_lucky_number" matches
```

Route Parameters

The previous examples defined routes where the URL never changes (e.g. `/blog`). However, it's common to define routes where some parts are variable. For example, the URL to display some blog post will probably include the title or slug (e.g. `/blog/my-first-post` or `/blog/all-about-symfony`).

In Symfony routes, variable parts are wrapped in `{ ... }` and they must have a unique name. For example, the route to display the blog post contents is defined as `/blog/{slug}`:

Listing 3-8
```
// src/Controller/BlogController.php
namespace App\Controller;

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

class BlogController extends AbstractController
{
    // ...

    /**
     * @Route("/blog/{slug}", name="blog_show")
     */
    public function show(string $slug)
    {
        // $slug will equal the dynamic part of the URL
        // e.g. at /blog/yay-routing, then $slug='yay-routing'

        // ...
    }
}
```

The name of the variable part (`{slug}` in this example) is used to create a PHP variable where that route content is stored and passed to the controller. If a user visits the `/blog/my-first-post` URL, Symfony executes the `show()` method in the `BlogController` class and passes a `$slug = 'my-first-post'` argument to the `show()` method.

Routes can define any number of parameters, but each of them can only be used once on each route (e.g. `/blog/posts-about-{category}/page/{pageNumber}`).

Parameters Validation

Imagine that your application has a `blog_show` route (URL: `/blog/{slug}`) and a `blog_list` route (URL: `/blog/{page}`). Given that route parameters accept any value, there's no way to differentiate both routes.

If the user requests `/blog/my-first-post`, both routes will match and Symfony will use the route which was defined first. To fix this, add some validation to the `{page}` parameter using the `requirements` option:

Listing 3-9
```
// src/Controller/BlogController.php
namespace App\Controller;

```

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

class BlogController extends AbstractController
{
    /**
     * @Route("/blog/{page}", name="blog_list", requirements={"page"="\d+"})
     */
    public function list(int $page)
    {
        // ...
    }

    /**
     * @Route("/blog/{slug}", name="blog_show")
     */
    public function show($slug)
    {
        // ...
    }
}
```

The `requirements` option defines the *PHP regular expressions*[2] that route parameters must match for the entire route to match. In this example, `\d+` is a regular expression that matches a *digit* of any length. Now:

URL	Route	Parameters
/blog/2	blog_list	$page = 2
/blog/my-first-post	blog_show	$slug = my-first-post

Route requirements (and route paths too) can include container parameters, which is useful to define complex regular expressions once and reuse them in multiple routes.

Parameters also support *PCRE Unicode properties*[3], which are escape sequences that match generic character types. For example, `\p{Lu}` matches any uppercase character in any language, `\p{Greek}` matches any Greek character, etc.

When using regular expressions in route parameters, you can set the `utf8` route option to `true` to make any `.` character match any UTF-8 characters instead of just a single byte.

If you prefer, requirements can be inlined in each parameter using the syntax `{parameter_name<requirements>}`. This feature makes configuration more concise, but it can decrease route readability when requirements are complex:

Listing 3-10

```php
// src/Controller/BlogController.php
namespace App\Controller;

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

class BlogController extends AbstractController
{
    /**
```

2. https://www.php.net/manual/en/book.pcre.php
3. http://php.net/manual/en/regexp.reference.unicode.php

```
     * @Route("/blog/{page<\d+>}", name="blog_list")
     */
    public function list(int $page)
    {
        // ...
    }
}
```

Optional Parameters

In the previous example, the URL of `blog_list` is `/blog/{page}`. If users visit `/blog/1`, it will match. But if they visit `/blog`, it will **not** match. As soon as you add a parameter to a route, it must have a value.

You can make `blog_list` once again match when the user visits `/blog` by adding a default value for the `{page}` parameter. When using annotations, default values are defined in the arguments of the controller action. In the other configuration formats they are defined with the `defaults` option:

Listing 3-11
```
// src/Controller/BlogController.php
namespace App\Controller;

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

class BlogController extends AbstractController
{
    /**
     * @Route("/blog/{page}", name="blog_list", requirements={"page"="\d+"})
     */
    public function list(int $page = 1)
    {
        // ...
    }
}
```

Now, when the user visits `/blog`, the `blog_list` route will match and `$page` will default to a value of `1`.

You can have more than one optional parameter (e.g. `/blog/{slug}/{page}`), but everything after an optional parameter must be optional. For example, `/{page}/blog` is a valid path, but `page` will always be required (i.e. `/blog` will not match this route).

Routes with optional parameters at the end will not match on requests with a trailing slash (i.e. `/blog/` will not match, `/blog` will match).

If you want to always include some default value in the generated URL (for example to force the generation of `/blog/1` instead of `/blog` in the previous example) add the `!` character before the parameter name: `/blog/{!page}`

As it happens with requirements, default values can also be inlined in each parameter using the syntax `{parameter_name?default_value}`. This feature is compatible with inlined requirements, so you can inline both in a single parameter:

Listing 3-12
```
// src/Controller/BlogController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
```

```
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController
{
    /**
     * @Route("/blog/{page<\d+>?1}", name="blog_list")
     */
    public function list(int $page)
    {
        // ...
    }
}
```

To give a `null` default value to any parameter, add nothing after the `?` character (e.g. `/blog/{page?}`).

Parameter Conversion

A common routing need is to convert the value stored in some parameter (e.g. an integer acting as the user ID) into another value (e.g. the object that represents the user). This feature is called "param converter" and is only available when using annotations to define routes.

In case you didn't run this command before, run it now to add support for annotations and "param converters":

Listing 3-13
```
$ composer require annotations
```

Now, keep the previous route configuration, but change the arguments of the controller action. Instead of `string $slug`, add `BlogPost $post`:

Listing 3-14
```
// src/Controller/BlogController.php
namespace App\Controller;

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

class BlogController extends AbstractController
{
    // ...

    /**
     * @Route("/blog/{slug}", name="blog_show")
     */
    public function show(BlogPost $post)
    {
        // $post is the object whose slug matches the routing parameter

        // ...
    }
}
```

If your controller arguments include type-hints for objects (`BlogPost` in this case), the "param converter" makes a database request to find the object using the request parameters (`slug` in this case). If no object is found, Symfony generates a 404 response automatically.

Read the *full param converter documentation*[4] to learn about the converters provided by Symfony and how to configure them.

4. https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html

Special Parameters

In addition to your own parameters, routes can include any of the following special parameters created by Symfony:

`_controller`
: This parameter is used to determine which controller and action is executed when the route is matched.

`_format`
: The matched value is used to set the "request format" of the `Request` object. This is used for such things as setting the `Content-Type` of the response (e.g. a `json` format translates into a `Content-Type` of `application/json`).

`_fragment`
: Used to set the fragment identifier, which is the optional last part of a URL that starts with a `#` character and is used to identify a portion of a document.

`_locale`
: Used to set the locale on the request.

You can include these attributes (except `_fragment`) both in individual routes and in route imports. Symfony defines some special attributes with the same name (except for the leading underscore) so you can define them easier:

Listing 3-15
```php
// src/Controller/ArticleController.php

// ...
class ArticleController extends AbstractController
{
    /**
     * @Route(
     *     "/articles/{_locale}/search.{_format}",
     *     locale="en",
     *     format="html",
     *     requirements={
     *         "_locale": "en|fr",
     *         "_format": "html|xml",
     *     }
     * )
     */
    public function search()
    {
    }
}
```

Extra Parameters

In the `defaults` option of a route you can optionally define parameters not included in the route configuration. This is useful to pass extra arguments to the controllers of the routes:

Listing 3-16
```php
use Symfony\Component\Routing\Annotation\Route;

class BlogController
{
    /**
     * @Route("/blog/{page}", name="blog_index", defaults={"page": 1, "title": "Hello world!"})
     */
    public function index(int $page, string $title)
    {
        // ...
    }
}
```

Slash Characters in Route Parameters

Route parameters can contain any values except the `/` slash character, because that's the character used to separate the different parts of the URLs. For example, if the `token` value in the `/share/{token}` route contains a `/` character, this route won't match.

A possible solution is to change the parameter requirements to be more permissive:

Listing 3-17

```
use Symfony\Component\Routing\Annotation\Route;

class DefaultController
{
    /**
     * @Route("/share/{token}", name="share", requirements={"token"=".+"})
     */
    public function share($token)
    {
        // ...
    }
}
```

If the route defines several parameter and you apply this permissive regular expression to all of them, the results won't be the expected. For example, if the route definition is `/share/{path}/{token}` and both `path` and `token` accept `/`, then `path` will contain its contents and the token, and `token` will be empty.

If the route includes the special `{_format}` parameter, you shouldn't use the `.+` requirement for the parameters that allow slashes. For example, if the pattern is `/share/{token}.{_format}` and `{token}` allows any character, the `/share/foo/bar.json` URL will consider `foo/bar.json` as the token and the format will be empty. This can be solved by replacing the `.+` requirement by `[^.]+` to allow any character except dots.

Route Groups and Prefixes

It's common for a group of routes to share some options (e.g. all routes related to the blog start with `/blog`) That's why Symfony includes a feature to share route configuration.

When defining routes as annotations, put the common configuration in the `@Route` annotation of the controller class. In other routing formats, define the common configuration using options when importing the routes.

Listing 3-18

```
use Symfony\Component\Routing\Annotation\Route;

/**
 * @Route("/blog", requirements={"locale": "en|es|fr"}, name="blog_")
 */
class BlogController
{
    /**
     * @Route("/{_locale}", name="index")
     */
    public function index()
    {
        // ...
    }

    /**
     * @Route("/{_locale}/posts/{slug}", name="post")
```

```
18       */
19      public function show(Post $post)
20      {
21          // ...
22      }
23  }
```

In this example, the route of the `index()` action will be called `blog_index` and its URL will be `/blog/`. The route of the `show()` action will be called `blog_post` and its URL will be `/blog/{_locale}/posts/{slug}`. Both routes will also validate that the `_locale` parameter matches the regular expression defined in the class annotation.

Symfony can import routes from different sources and you can even create your own route loader.

Getting the Route Name and Parameters

The `Request` object created by Symfony stores all the route configuration (such as the name and parameters) in the "request attributes". You can get this information in a controller via the `Request` object:

Listing 3-19

```
// src/Controller/BlogController.php
namespace App\Controller;

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

class BlogController extends AbstractController
{
    /**
     * @Route("/blog", name="blog_list")
     */
    public function list(Request $request)
    {
        // ...

        $routeName = $request->attributes->get('_route');
        $routeParameters = $request->attributes->get('_route_params');

        // use this to get all the available attributes (not only routing ones):
        $allAttributes = $request->attributes->all();
    }
}
```

You can get this information in services too injecting the `request_stack` service to *get the Request object in a service*. In Twig templates, use the global app object to get the request and its attributes:

Listing 3-20

```
{% set route_name = app.request.attributes.get('_route') %}
{% set route_parameters = app.request.attributes.get('_route_params') %}

{# use this to get all the available attributes (not only routing ones) #}
{% set all_attributes = app.request.attributes.all %}
```

Special Routes

Symfony defines some special controllers to render templates and redirect to other routes from the route configuration so you don't have to create a controller action.

Rendering Templates

Use the `TemplateController` to render the template whose path is defined in the `template` option:

Listing 3-21
```yaml
# config/routes.yaml
about_us:
    path: /site/about-us
    controller: Symfony\Bundle\FrameworkBundle\Controller\TemplateController::templateAction
    defaults:
        template: 'static_pages/about_us.html.twig'

        # optionally you can define some arguments passed to the template
        site_name: 'ACME'
        theme: 'dark'
```

Redirecting to URLs and Routes

Use the `RedirectController` to redirect to other routes (`redirectAction`) and URLs (`urlRedirectAction`):

Listing 3-22
```yaml
# config/routes.yaml
doc_shortcut:
    path: /doc
    controller: Symfony\Bundle\FrameworkBundle\Controller\RedirectController::redirectAction
    defaults:
        route: 'doc_page'
        # optionally you can define some arguments passed to the route
        page: 'index'
        version: 'current'
        # redirections are temporary by default (code 302) but you can make them permanent (code 301)
        permanent: true
        # add this to keep the original query string parameters when redirecting
        keepQueryParams: true
        # add this to keep the HTTP method when redirecting. The redirect status changes
        # * for temporary redirects, it uses the 307 status code instead of 302
        # * for permanent redirects, it uses the 308 status code instead of 301
        keepRequestMethod: true

legacy_doc:
    path: /legacy/doc
    controller: Symfony\Bundle\FrameworkBundle\Controller\RedirectController::urlRedirectAction
    defaults:
        # this value can be an absolute path or an absolute URL
        path: 'https://legacy.example.com/doc'
        permanent: true
```

Symfony also provides some utilities to redirect inside controllers

Redirecting URLs with Trailing Slashes

Historically, URLs have followed the UNIX convention of adding trailing slashes for directories (e.g. `https://example.com/foo/`) and removing them to refer to files (`https://example.com/foo`). Although serving different contents for both URLs is OK, nowadays it's common to treat both URLs as the same URL and redirect between them.

Symfony follows this logic to redirect between URLs with and without trailing slashes (but only for `GET` and `HEAD` requests):

Route URL	If the requested URL is `/foo`	If the requested URL is `/foo/`
`/foo`	It matches (200 status response)	It makes a 301 redirect to `/foo`
`/foo/`	It makes a 301 redirect to `/foo/`	It matches (200 status response)

If your application defines different routes for each path (`/foo` and `/foo/`) this automatic redirection doesn't take place and the right route is always matched.

Sub-Domain Routing

Routes can configure a `host` option to require that the HTTP host of the incoming requests matches some specific value. In the following example, both routes match the same path (`/`) but one of them only responds to a specific host name:

Listing 3-23

```php
// src/Controller/MainController.php
namespace App\Controller;

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

class MainController extends AbstractController
{
    /**
     * @Route("/", name="mobile_homepage", host="m.example.com")
     */
    public function mobileHomepage()
    {
        // ...
    }

    /**
     * @Route("/", name="homepage")
     */
    public function homepage()
    {
        // ...
    }
}
```

The value of the `host` option can include parameters (which is useful in multi-tenant applications) and these parameters can be validated too with `requirements`:

Listing 3-24

```php
// src/Controller/MainController.php
namespace App\Controller;

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

class MainController extends AbstractController
{
    /**
     * @Route(
     *     "/",
     *     name="mobile_homepage",
     *     host="{subdomain}.example.com",
     *     defaults={"subdomain"="m"},
     *     requirements={"subdomain"="m|mobile"}
     * )
     */
```

```
18      public function mobileHomepage()
19      {
20          // ...
21      }
22
23      /**
24       * @Route("/", name="homepage")
25       */
26      public function homepage()
27      {
28          // ...
29      }
30  }
```

In the above example, the `domain` parameter defines a default value because otherwise you need to include a domain value each time you generate a URL using these routes.

You can also set the `host` option when importing routes to make all of them require that host name.

When using sub-domain routing, you must set the `Host` HTTP headers in *functional tests* or routes won't match:

Listing 3-25
```
1  $crawler = $client->request(
2      'GET',
3      '/',
4      [],
5      [],
6      ['HTTP_HOST' => 'm.example.com']
7      // or get the value from some container parameter:
8      // ['HTTP_HOST' => 'm.' . $client->getContainer()->getParameter('domain')]
9  );
```

Localized Routes (i18n)

If your application is translated into multiple languages, each route can define a different URL per each *translation locale*. This avoids the need for duplicating routes, which also reduces the potential bugs:

Listing 3-26
```
1  // src/Controller/CompanyController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\Routing\Annotation\Route;
6
7  class CompanyController extends AbstractController
8  {
9      /**
10      * @Route({
11      *     "en": "/about-us",
12      *     "nl": "/over-ons"
13      * }, name="about_us")
14      */
15     public function about()
16     {
17         // ...
18     }
19 }
```

When a localized route is matched, Symfony uses the same locale automatically during the entire request.

When the application uses full "language + territory" locales (e.g. `fr_FR`, `fr_BE`), if the URLs are the same in all related locales, routes can use only the language part (e.g. `fr`) to avoid repeating the same URLs.

A common requirement for internationalized applications is to prefix all routes with a locale. This can be done by defining a different prefix for each locale (and setting an empty prefix for your default locale if you prefer it):

Listing 3-27
```yaml
# config/routes/annotations.yaml
controllers:
    resource: '../src/Controller/'
    type: annotation
    prefix:
        en: '' # don't prefix URLs for English, the default locale
        nl: '/nl'
```

Generating URLs

Routing systems are bidirectional: 1) they associate URLs with controllers (as explained in the previous sections); 2) they generate URLs for a given route. Generating URLs from routes allows you to not write the `<a href="...">` values manually in your HTML templates. Also, if the URL of some route changes, you only have to update the route configuration and all links will be updated.

To generate a URL, you need to specify the name of the route (e.g. `blog_show`) and the values of the parameters defined by the route (e.g. `slug = my-blog-post`).

For that reason each route has an internal name that must be unique in the application. If you don't set the route name explicitly with the `name` option, Symfony generates an automatic name based on the controller and action.

Generating URLs in Controllers

If your controller extends from the AbstractController, use the `generateUrl()` helper:

Listing 3-28
```php
// src/Controller/BlogController.php
namespace App\Controller;

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

class BlogController extends AbstractController
{
    /**
     * @Route("/blog", name="blog_list")
     */
    public function list()
    {
        // ...

        // generate a URL with no route arguments
        $signUpPage = $this->generateUrl('sign_up');

        // generate a URL with route arguments
        $userProfilePage = $this->generateUrl('user_profile', [
            'username' => $user->getUsername(),
        ]);

        // generated URLs are "absolute paths" by default. Pass a third optional
        // argument to generate different URLs (e.g. an "absolute URL")
```

```
        $signUpPage = $this->generateUrl('sign_up', [], UrlGeneratorInterface::ABSOLUTE_URL);

        // when a route is localized, Symfony uses by default the current request locale
        // pass a different '_locale' value if you want to set the locale explicitly
        $signUpPageInDutch = $this->generateUrl('sign_up', ['_locale' => 'nl']);
    }
}
```

If you pass to the `generateUrl()` method some parameters that are not part of the route definition, they are included in the generated URL as a query string::

Listing 3-29

```
$this->generateUrl('blog', ['page' => 2, 'category' => 'Symfony']);
// the 'blog' route only defines the 'page' parameter; the generated URL is:
// /blog/2?category=Symfony
```

If your controller does not extend from `AbstractController`, you'll need to fetch services in your controller and follow the instructions of the next section.

Generating URLs in Services

Inject the `router` Symfony service into your own services and use its `generate()` method. When using *service autowiring* you only need to add an argument in the service constructor and type-hint it with the *`UrlGeneratorInterface`*[5] class:

Listing 3-30

```
// src/Service/SomeService.php
use Symfony\Component\Routing\Generator\UrlGeneratorInterface;

class SomeService
{
    private $router;

    public function __construct(UrlGeneratorInterface $router)
    {
        $this->router = $router;
    }

    public function someMethod()
    {
        // ...

        // generate a URL with no route arguments
        $signUpPage = $this->router->generate('sign_up');

        // generate a URL with route arguments
        $userProfilePage = $this->router->generate('user_profile', [
            'username' => $user->getUsername(),
        ]);

        // generated URLs are "absolute paths" by default. Pass a third optional
        // argument to generate different URLs (e.g. an "absolute URL")
        $signUpPage = $this->router->generate('sign_up', [], UrlGeneratorInterface::ABSOLUTE_URL);

        // when a route is localized, Symfony uses by default the current request locale
        // pass a different '_locale' value if you want to set the locale explicitly
        $signUpPageInDutch = $this->router->generate('sign_up', ['_locale' => 'nl']);
    }
}
```

5. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Routing/Generator/UrlGeneratorInterface.php

Generating URLs in Templates

The Twig template language used in *Symfony templates* provides some functions to generate both relative and absolute URLs:

Listing 3-31
```twig
{# generates relative URLs #}
<a href="{{ path('sign_up') }}">Sign up</a>
<a href="{{ path('user_profile', {username: app.user.username}) }}">
    View your profile
</a>
<a href="{{ path('user_profile', {username: app.user.username, _locale: 'es'}) }}">
    Ver mi perfil
</a>

{# generates absolute URLs #}
<a href="{{ url('sign_up') }}">Sign up</a>
<a href="{{ url('user_profile', {username: app.user.username}) }}">
    View your profile
</a>
<a href="{{ url('user_profile', {username: app.user.username, _locale: 'es'}) }}">
    Ver mi perfil
</a>
```

Generating URLs in JavaScript

If your JavaScript code is included in a Twig template, you can use the `path()` and `url()` Twig functions to generate the URLs and store them in JavaScript variables. The `escape()` function is needed to escape any non-JavaScript-safe values:

Listing 3-32
```twig
<script>
    const route = "{{ path('blog_show', {slug: 'my-blog-post'})|escape('js') }}";
</script>
```

If you need to generate URLs dynamically or if you are using pure JavaScript code, this solution doesn't work. In those cases, consider using the *FOSJsRoutingBundle*[6].

Generating URLs in Commands

Generating URLs in commands works the same as generating URLs in services. The only difference is that commands are not executed in the HTTP context, so they don't have access to HTTP requests. In practice, this means that if you generate absolute URLs, you'll get `http://localhost/` as the host name instead of your real host name.

The solution is to configure "request context" used by commands when they generate URLs. This context can be configured globally for all commands:

Listing 3-33
```yaml
# config/services.yaml
parameters:
    router.request_context.host: 'example.org'
    router.request_context.base_url: 'my/path'
    asset.request_context.base_path: '%router.request_context.base_url%'
```

This information can be configured per command too:

Listing 3-34
```php
// src/Command/SomeCommand.php
use Symfony\Component\Routing\Generator\UrlGeneratorInterface;
use Symfony\Component\Routing\RouterInterface;
// ...

class SomeCommand extends Command
```

6. https://github.com/FriendsOfSymfony/FOSJsRoutingBundle

```
{
    private $router;

    public function __construct(RouterInterface $router)
    {
        parent::__construct();

        $this->router = $router;
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        // these values override any global configuration
        $context = $this->router->getContext();
        $context->setHost('example.com');
        $context->setBaseUrl('my/path');

        // generate a URL with no route arguments
        $signUpPage = $this->router->generate('sign_up');

        // generate a URL with route arguments
        $userProfilePage = $this->router->generate('user_profile', [
            'username' => $user->getUsername(),
        ]);

        // generated URLs are "absolute paths" by default. Pass a third optional
        // argument to generate different URLs (e.g. an "absolute URL")
        $signUpPage = $this->router->generate('sign_up', [], UrlGeneratorInterface::ABSOLUTE_URL);

        // when a route is localized, Symfony uses by default the current request locale
        // pass a different '_locale' value if you want to set the locale explicitly
        $signUpPageInDutch = $this->router->generate('sign_up', ['_locale' => 'nl']);

        // ...
    }
}
```

Checking if a Route Exists

In highly dynamic applications, it may be necessary to check whether a route exists before using it to generate a URL. In those cases, don't use the *`getRouteCollection()`*[7] method because that regenerates the routing cache and slows down the application.

Instead, try to generate the URL and catch the *`RouteNotFoundException`*[8] thrown when the route doesn't exist:

Listing 3-35
```
use Symfony\Component\Routing\Exception\RouteNotFoundException;

// ...

try {
    $url = $this->router->generate($routeName, $routeParameters);
} catch (RouteNotFoundException $e) {
    // the route is not defined...
}
```

Forcing HTTPS on Generated URLs

By default, generated URLs use the same HTTP scheme as the current request. In console commands, where there is no HTTP request, URLs use `http` by default. You can change this per command (via the router's `getContext()` method) or globally with these configuration parameters:

7. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Routing/Router.php
8. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Routing/Exception/RouteNotFoundException.php

Listing 3-36
```yaml
# config/services.yaml
parameters:
    router.request_context.scheme: 'https'
    asset.request_context.secure: true
```

Outside of console commands, use the `schemes` option to define the scheme of each route explicitly:

Listing 3-37
```php
// src/Controller/MainController.php
namespace App\Controller;

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

class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="login", schemes={"https"})
     */
    public function login()
    {
        // ...
    }
}
```

The URL generated for the `login` route will always use HTTPS. This means that when using the `path()` Twig function to generate URLs, you may get an absolute URL instead of a relative URL if the HTTP scheme of the original request is different from the scheme used by the route:

Listing 3-38
```twig
{# if the current scheme is HTTPS, generates a relative URL: /login #}
{{ path('login') }}

{# if the current scheme is HTTP, generates an absolute URL to change
   the scheme: https://example.com/login #}
{{ path('login') }}
```

The scheme requirement is also enforced for incoming requests. If you try to access the `/login` URL with HTTP, you will automatically be redirected to the same URL, but with the HTTPS scheme.

If you want to force a group of routes to use HTTPS, you can define the default scheme when importing them. The following example forces HTTPS on all routes defined as annotations:

Listing 3-39
```yaml
# config/routes/annotations.yaml
controllers:
    resource: '../src/Controller/'
    type: annotation
    defaults:
        schemes: [https]
```

The Security component provides *another way to enforce HTTP or HTTPS* via the `requires_channel` setting.

Troubleshooting

Here are some common errors you might see while working with routing:

> Controller "App\Controller\BlogController::show()" requires that you provide a value for the "$slug" argument.

This happens when your controller method has an argument (e.g. `$slug`):

Listing 3-40
```
public function show($slug)
{
    // ...
}
```

But your route path does *not* have a `{slug}` parameter (e.g. it is `/blog/show`). Add a `{slug}` to your route path: `/blog/show/{slug}` or give the argument a default value (i.e. `$slug = null`).

> Some mandatory parameters are missing ("slug") to generate a URL for route "blog_show".

This means that you're trying to generate a URL to the `blog_show` route but you are *not* passing a `slug` value (which is required, because it has a `{slug}` parameter in the route path). To fix this, pass a `slug` value when generating the route:

Listing 3-41
```
$this->generateUrl('blog_show', ['slug' => 'slug-value']);

// or, in Twig
// {{ path('blog_show', {slug: 'slug-value'}) }}
```

Learn more about Routing

- How to Create a custom Route Loader
- Looking up Routes from a Database: Symfony CMF DynamicRouter

Chapter 4
Controller

A controller is a PHP function you create that reads information from the `Request` object and creates and returns a `Response` object. The response could be an HTML page, JSON, XML, a file download, a redirect, a 404 error or anything else. The controller executes whatever arbitrary logic *your application* needs to render the content of a page.

If you haven't already created your first working page, check out *Create your First Page in Symfony* and then come back!

A Simple Controller

While a controller can be any PHP callable (function, method on an object, or a `Closure`), a controller is usually a method inside a controller class:

Listing 4-1

```
// src/Controller/LuckyController.php
namespace App\Controller;

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

class LuckyController
{
    /**
     * @Route("/lucky/number/{max}", name="app_lucky_number")
     */
    public function number($max)
    {
        $number = random_int(0, $max);

        return new Response(
            '<html><body>Lucky number: '.$number.'</body></html>'
        );
    }
}
```

The controller is the `number()` method, which lives inside the controller class `LuckyController`.

This controller is pretty straightforward:

- *line 2*: Symfony takes advantage of PHP's namespace functionality to namespace the entire controller class.
- *line 4*: Symfony again takes advantage of PHP's namespace functionality: the `use` keyword imports the `Response` class, which the controller must return.
- *line 7*: The class can technically be called anything, but it's suffixed with `Controller` by convention.
- *line 12*: The action method is allowed to have a `$max` argument thanks to the `{max}` *wildcard in the route*.
- *line 16*: The controller creates and returns a `Response` object.

Mapping a URL to a Controller

In order to *view* the result of this controller, you need to map a URL to it via a route. This was done above with the `@Route("/lucky/number/{max}")` route annotation.

To see your page, go to this URL in your browser:

> *http://localhost:8000/lucky/number/100*

For more information on routing, see *Routing*.

The Base Controller Class & Services

To aid development, Symfony comes with an optional base controller class called *`AbstractController`*[1]. It can be extended to gain access to helper methods.

Add the `use` statement atop your controller class and then modify `LuckyController` to extend it:

Listing 4-2
```
// src/Controller/LuckyController.php
namespace App\Controller;

+ use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

- class LuckyController
+ class LuckyController extends AbstractController
{
    // ...
}
```

That's it! You now have access to methods like $this->render() and many others that you'll learn about next.

Generating URLs

The *`generateUrl()`*[2] method is just a helper method that generates the URL for a given route:

Listing 4-3
```
$url = $this->generateUrl('app_lucky_number', ['max' => 10]);
```

Redirecting

If you want to redirect the user to another page, use the `redirectToRoute()` and `redirect()` methods:

1. https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php
2. https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php

Listing 4-4
```php
use Symfony\Component\HttpFoundation\RedirectResponse;

// ...
public function index()
{
    // redirects to the "homepage" route
    return $this->redirectToRoute('homepage');

    // redirectToRoute is a shortcut for:
    // return new RedirectResponse($this->generateUrl('homepage'));

    // does a permanent - 301 redirect
    return $this->redirectToRoute('homepage', [], 301);

    // redirect to a route with parameters
    return $this->redirectToRoute('app_lucky_number', ['max' => 10]);

    // redirects to a route and maintains the original query string parameters
    return $this->redirectToRoute('blog_show', $request->query->all());

    // redirects externally
    return $this->redirect('http://symfony.com/doc');
}
```

The `redirect()` method does not check its destination in any way. If you redirect to a URL provided by end-users, your application may be open to the *unvalidated redirects security vulnerability*[3].

Rendering Templates

If you're serving HTML, you'll want to render a template. The `render()` method renders a template **and** puts that content into a `Response` object for you:

Listing 4-5
```php
// renders templates/lucky/number.html.twig
return $this->render('lucky/number.html.twig', ['number' => $number]);
```

Templating and Twig are explained more in the *Creating and Using Templates article*.

Fetching Services

Symfony comes *packed* with a lot of useful objects, called *services*. These are used for rendering templates, sending emails, querying the database and any other "work" you can think of.

If you need a service in a controller, type-hint an argument with its class (or interface) name. Symfony will automatically pass you the service you need:

Listing 4-6
```php
use Psr\Log\LoggerInterface;
// ...

/**
 * @Route("/lucky/number/{max}")
 */
public function number($max, LoggerInterface $logger)
{
    $logger->info('We are logging!');
    // ...
}
```

Awesome!

What other services can you type-hint? To see them, use the `debug:autowiring` console command:

3. https://www.owasp.org/index.php/Open_redirect

Listing 4-7
```
$ php bin/console debug:autowiring
```

If you need control over the *exact* value of an argument, you can bind the argument by its name:

Listing 4-8
```
# config/services.yaml
services:
    # ...

    # explicitly configure the service
    App\Controller\LuckyController:
        public: true
        bind:
            # for any $logger argument, pass this specific service
            $logger: '@monolog.logger.doctrine'
            # for any $projectDir argument, pass this parameter value
            $projectDir: '%kernel.project_dir%'
```

Like with all services, you can also use regular constructor injection in your controllers.
For more information about services, see the *Service Container* article.

Generating Controllers

To save time, you can install *Symfony Maker*[4] and tell Symfony to generate a new controller class:

Listing 4-9
```
$ php bin/console make:controller BrandNewController

created: src/Controller/BrandNewController.php
created: templates/brandnew/index.html.twig
```

If you want to generate an entire CRUD from a Doctrine *entity*, use:

Listing 4-10
```
$ php bin/console make:crud Product

created: src/Controller/ProductController.php
created: src/Form/ProductType.php
created: templates/product/_delete_form.html.twig
created: templates/product/_form.html.twig
created: templates/product/edit.html.twig
created: templates/product/index.html.twig
created: templates/product/new.html.twig
created: templates/product/show.html.twig
```

New in version 1.2: The `make:crud` command was introduced in MakerBundle 1.2.

Managing Errors and 404 Pages

When things are not found, you should return a 404 response. To do this, throw a special type of exception:

Listing 4-11
```
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

// ...
public function index()
{
    // retrieve the object from database
    $product = ...;
    if (!$product) {
```

4. https://symfony.com/doc/current/bundles/SymfonyMakerBundle/index.html

```
        throw $this->createNotFoundException('The product does not exist');

        // the above is just a shortcut for:
        // throw new NotFoundHttpException('The product does not exist');
    }

    return $this->render(...);
}
```

The *createNotFoundException()*[5] method is just a shortcut to create a special *NotFoundHttpException*[6] object, which ultimately triggers a 404 HTTP response inside Symfony.

If you throw an exception that extends or is an instance of *HttpException*[7], Symfony will use the appropriate HTTP status code. Otherwise, the response will have a 500 HTTP status code:

Listing 4-12
```
// this exception ultimately generates a 500 status error
throw new \Exception('Something went wrong!');
```

In every case, an error page is shown to the end user and a full debug error page is shown to the developer (i.e. when you're in "Debug" mode - see Configuration Environments).

To customize the error page that's shown to the user, see the *How to Customize Error Pages* article.

The Request object as a Controller Argument

What if you need to read query parameters, grab a request header or get access to an uploaded file? That information is stored in Symfony's `Request` object. To access it in your controller, add it as an argument and **type-hint it with the Request class**:

Listing 4-13
```
use Symfony\Component\HttpFoundation\Request;

public function index(Request $request, $firstName, $lastName)
{
    $page = $request->query->get('page', 1);

    // ...
}
```

Keep reading for more information about using the Request object.

Managing the Session

Symfony provides a session service that you can use to store information about the user between requests. Session is enabled by default, but will only be started if you read or write from it.

Session storage and other configuration can be controlled under the framework.session configuration in `config/packages/framework.yaml`.

To get the session, add an argument and type-hint it with *SessionInterface*[8]:

Listing 4-14
```
use Symfony\Component\HttpFoundation\Session\SessionInterface;

public function index(SessionInterface $session)
{
```

5. https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php
6. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/HttpKernel/Exception/NotFoundHttpException.php
7. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/HttpKernel/Exception/HttpException.php
8. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/HttpFoundation/Session/SessionInterface.php

```
    // stores an attribute for reuse during a later user request
    $session->set('foo', 'bar');

    // gets the attribute set by another controller in another request
    $foobar = $session->get('foobar');

    // uses a default value if the attribute doesn't exist
    $filters = $session->get('filters', []);
}
```

Stored attributes remain in the session for the remainder of that user's session.

For more info, see *Sessions*.

Flash Messages

You can also store special messages, called "flash" messages, on the user's session. By design, flash messages are meant to be used exactly once: they vanish from the session automatically as soon as you retrieve them. This feature makes "flash" messages particularly great for storing user notifications.

For example, imagine you're processing a *form* submission:

Listing 4-15
```
use Symfony\Component\HttpFoundation\Request;

public function update(Request $request)
{
    // ...

    if ($form->isSubmitted() && $form->isValid()) {
        // do some sort of processing

        $this->addFlash(
            'notice',
            'Your changes were saved!'
        );
        // $this->addFlash() is equivalent to $request->getSession()->getFlashBag()->add()

        return $this->redirectToRoute(...);
    }

    return $this->render(...);
}
```

After processing the request, the controller sets a flash message in the session and then redirects. The message key (`notice` in this example) can be anything: you'll use this key to retrieve the message.

In the template of the next page (or even better, in your base layout template), read any flash messages from the session using `app.flashes()`:

Listing 4-16
```
{# templates/base.html.twig #}

{# read and display just one flash message type #}
{% for message in app.flashes('notice') %}
    <div class="flash-notice">
        {{ message }}
    </div>
{% endfor %}

{# read and display several types of flash messages #}
{% for label, messages in app.flashes(['success', 'warning']) %}
    {% for message in messages %}
        <div class="flash-{{ label }}">
            {{ message }}
        </div>
    {% endfor %}
{% endfor %}
```

```twig

{# read and display all flash messages #}
{% for label, messages in app.flashes %}
    {% for message in messages %}
        <div class="flash-{{ label }}">
            {{ message }}
        </div>
    {% endfor %}
{% endfor %}
```

It's common to use `notice`, `warning` and `error` as the keys of the different types of flash messages, but you can use any key that fits your needs.

You can use the *peek()*[9] method instead to retrieve the message while keeping it in the bag.

The Request and Response Object

As mentioned earlier, Symfony will pass the `Request` object to any controller argument that is type-hinted with the `Request` class:

Listing 4-17

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

public function index(Request $request)
{
    $request->isXmlHttpRequest(); // is it an Ajax request?

    $request->getPreferredLanguage(['en', 'fr']);

    // retrieves GET and POST variables respectively
    $request->query->get('page');
    $request->request->get('page');

    // retrieves SERVER variables
    $request->server->get('HTTP_HOST');

    // retrieves an instance of UploadedFile identified by foo
    $request->files->get('foo');

    // retrieves a COOKIE value
    $request->cookies->get('PHPSESSID');

    // retrieves an HTTP request header, with normalized, lowercase keys
    $request->headers->get('host');
    $request->headers->get('content-type');
}
```

The `Request` class has several public properties and methods that return any information you need about the request.

Like the `Request`, the `Response` object has a public `headers` property. This object is of the type *ResponseHeaderBag*[10] and provides methods for getting and setting response headers. The header names are normalized. As a result, the name `Content-Type` is equivalent to the name `content-type` or `content_type`.

In Symfony, a controller is required to return a `Response` object:

9. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.php
10. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/HttpFoundation/ResponseHeaderBag.php

Listing 4-18
```php
use Symfony\Component\HttpFoundation\Response;

// creates a simple Response with a 200 status code (the default)
$response = new Response('Hello '.$name, Response::HTTP_OK);

// creates a CSS-response with a 200 status code
$response = new Response('<style> ... </style>');
$response->headers->set('Content-Type', 'text/css');
```

To facilitate this, different response objects are included to address different response types. Some of these are mentioned below. To learn more about the `Request` and `Response` (and different `Response` classes), see the HttpFoundation component documentation.

Accessing Configuration Values

To get the value of any configuration parameter from a controller, use the `getParameter()` helper method:

Listing 4-19
```php
// ...
public function index()
{
    $contentsDir = $this->getParameter('kernel.project_dir').'/contents';
    // ...
}
```

Returning JSON Response

To return JSON from a controller, use the `json()` helper method. This returns a `JsonResponse` object that encodes the data automatically:

Listing 4-20
```php
// ...
public function index()
{
    // returns '{"username":"jane.doe"}' and sets the proper Content-Type header
    return $this->json(['username' => 'jane.doe']);

    // the shortcut defines three optional arguments
    // return $this->json($data, $status = 200, $headers = [], $context = []);
}
```

If the *serializer service* is enabled in your application, it will be used to serialize the data to JSON. Otherwise, the *`json_encode`*[11] function is used.

Streaming File Responses

You can use the *`file()`*[12] helper to serve a file from inside a controller:

Listing 4-21
```php
public function download()
{
    // send the file contents and force the browser to download it
    return $this->file('/path/to/some_file.pdf');
}
```

The `file()` helper provides some arguments to configure its behavior:

Listing 4-22

11. https://secure.php.net/manual/en/function.json-encode.php
12. https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php

```php
use Symfony\Component\HttpFoundation\File\File;
use Symfony\Component\HttpFoundation\ResponseHeaderBag;

public function download()
{
    // load the file from the filesystem
    $file = new File('/path/to/some_file.pdf');

    return $this->file($file);

    // rename the downloaded file
    return $this->file($file, 'custom_name.pdf');

    // display the file contents in the browser instead of downloading it
    return $this->file('invoice_3241.pdf', 'my_invoice.pdf', ResponseHeaderBag::DISPOSITION_INLINE);
}
```

Final Thoughts

In Symfony, a controller is usually a class method which is used to accept requests, and return a `Response` object. When mapped with a URL, a controller becomes accessible and its response can be viewed.

To facilitate the development of controllers, Symfony provides an `AbstractController`. It can be used to extend the controller class allowing access to some frequently used utilities such as `render()` and `redirectToRoute()`. The `AbstractController` also provides the `createNotFoundException()` utility which is used to return a page not found response.

In other articles, you'll learn how to use specific services from inside your controller that will help you persist and fetch objects from a database, process form submissions, handle caching and more.

Keep Going!

Next, learn all about *rendering templates with Twig*.

Learn more about Controllers

- Extending Action Argument Resolving
- How to Customize Error Pages
- How to Forward Requests to another Controller
- How to Define Controllers as Services
- How to Create a SOAP Web Service in a Symfony Controller
- How to Upload Files

Chapter 5
Creating and Using Templates

As explained in *the previous article*, controllers are responsible for handling each request that comes into a Symfony application and they usually end up rendering a template to generate the response contents.

In reality, the controller delegates most of the heavy work to other places so that code can be tested and reused. When a controller needs to generate HTML, CSS or any other content, it hands the work off to the templating engine.

In this article, you'll learn how to write powerful templates that can be used to return content to the user, populate email bodies, and more. You'll learn shortcuts, clever ways to extend templates and how to reuse template code.

Templates

A template is a text file that can generate any text-based format (HTML, XML, CSV, LaTeX ...). The most familiar type of template is a *PHP* template - a text file parsed by PHP that contains a mix of text and PHP code:

Listing 5-1

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1><?= $page_title ?></h1>

        <ul id="navigation">
            <?php foreach ($navigation as $item): ?>
                <li>
                    <a href="<?= $item->getHref() ?>">
                        <?= $item->getCaption() ?>
                    </a>
                </li>
            <?php endforeach ?>
        </ul>
    </body>
</html>
```

But Symfony packages an even more powerful templating language called *Twig*[1]. Twig allows you to write concise, readable templates that are more friendly to web designers and, in several ways, more powerful than PHP templates:

Listing 5-2
```twig
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>

        <ul id="navigation">
            {% for item in navigation %}
                <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Twig defines three types of special syntax:

`{{ ... }}`
: "Says something": prints a variable or the result of an expression to the template.

`{% ... %}`
: "Does something": a **tag** that controls the logic of the template; it is used to execute statements such as for-loops for example.

`{# ... #}`
: "Comment something": it's the equivalent of the PHP `/* comment */` syntax. It's used to add single or multi-line comments. The content of the comments isn't included in the rendered pages.

Twig also contains **filters**, which modify content before being rendered. The following makes the `title` variable all uppercase before rendering it:

Listing 5-3
```twig
{{ title|upper }}
```

Twig comes with a long list of *tags*[2], *filters*[3] and *functions*[4] that are available by default. You can even add your own *custom* filters, functions (and more) via a *Twig Extension*. Run the following command to list them all:

Listing 5-4
```
$ php bin/console debug:twig
```

Twig code will look similar to PHP code, with subtle, nice differences. The following example uses a standard `for` tag and the `cycle()` function to print ten div tags, with alternating `odd`, `even` classes:

Listing 5-5
```twig
{% for i in 1..10 %}
    <div class="{{ cycle(['even', 'odd'], i) }}">
        <!-- some HTML here -->
    </div>
{% endfor %}
```

Throughout this article, template examples will be shown in both Twig and PHP.

1. https://twig.symfony.com
2. https://twig.symfony.com/doc/2.x/tags/index.html
3. https://twig.symfony.com/doc/2.x/filters/index.html
4. https://twig.symfony.com/doc/2.x/functions/index.html

Why Twig?

Twig templates are meant to be simple and won't process PHP tags. This is by design: the Twig template system is meant to express presentation, not program logic. The more you use Twig, the more you'll appreciate and benefit from this distinction. And you'll be loved by web designers everywhere.

Twig can also do things that PHP can't, such as whitespace control, sandboxing, automatic HTML escaping, manual contextual output escaping, and the inclusion of custom functions and filters that only affect templates. Twig contains a lot of features that make writing templates easier and more concise. Take the following example, which combines a loop with a logical `if` statement:

Listing 5-6

```
<ul>
    {% for user in users if user.active %}
        <li>{{ user.username }}</li>
    {% else %}
        <li>No users found</li>
    {% endfor %}
</ul>
```

Twig Template Caching

Twig is fast because each template is compiled to a native PHP class and cached. But don't worry: this happens automatically and doesn't require *you* to do anything. And while you're developing, Twig is smart enough to re-compile your templates after you make any changes. That means Twig is fast in production, but convenient to use while developing.

Template Inheritance and Layouts

More often than not, templates in a project share common elements, like the header, footer, sidebar or more. In Symfony, this problem is thought about differently: a template can be decorated by another one. This works exactly the same as PHP classes: template inheritance allows you to build a base "layout" template that contains all the common elements of your site defined as **blocks** (think "PHP class with base methods"). A child template can extend the base layout and override any of its blocks (think "PHP subclass that overrides certain methods of its parent class").

First, build a base layout file:

Listing 5-7

```
{# templates/base.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Test Application{% endblock %}</title>
    </head>
    <body>
        <div id="sidebar">
            {% block sidebar %}
                <ul>
                    <li><a href="/">Home</a></li>
                    <li><a href="/blog">Blog</a></li>
                </ul>
            {% endblock %}
        </div>

        <div id="content">
            {% block body %}{% endblock %}
        </div>
    </body>
</html>
```

Though the discussion about template inheritance will be in terms of Twig, the philosophy is the same between Twig and PHP templates.

This template defines the base HTML skeleton document of a two-column page. In this example, three `{% block %}` areas are defined (`title`, `sidebar` and `body`). Each block may be overridden by a child template or left with its default implementation. This template could also be rendered directly. In that case the `title`, `sidebar` and `body` blocks would retain the default values used in this template.

A child template might look like this:

Listing 5-8

```twig
{# templates/blog/index.html.twig #}
{% extends 'base.html.twig' %}

{% block title %}My cool blog posts{% endblock %}

{% block body %}
    {% for entry in blog_entries %}
        <h2>{{ entry.title }}</h2>
        <p>{{ entry.body }}</p>
    {% endfor %}
{% endblock %}
```

The parent template is stored in `templates/`, so its path is `base.html.twig`. The template naming conventions are explained fully in Template Naming and Locations.

The key to template inheritance is the `{% extends %}` tag. This tells the templating engine to first evaluate the base template, which sets up the layout and defines several blocks. The child template is then rendered, at which point the `title` and `body` blocks of the parent are replaced by those from the child. Depending on the value of `blog_entries`, the output might look like this:

Listing 5-9

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>My cool blog posts</title>
    </head>
    <body>
        <div id="sidebar">
            <ul>
                <li><a href="/">Home</a></li>
                <li><a href="/blog">Blog</a></li>
            </ul>
        </div>

        <div id="content">
            <h2>My first post</h2>
            <p>The body of the first post.</p>

            <h2>Another post</h2>
            <p>The body of the second post.</p>
        </div>
    </body>
</html>
```

Notice that since the child template didn't define a `sidebar` block, the value from the parent template is used instead. Content within a `{% block %}` tag in a parent template is always used by default.

You can use as many levels of inheritance as you want! See *How to Organize Your Twig Templates Using Inheritance* for more info.

When working with template inheritance, here are some tips to keep in mind:

- If you use `{% extends %}` in a template, it must be the first tag in that template;
- The more `{% block %}` tags you have in your base templates, the better. Remember, child templates don't have to define all parent blocks, so create as many blocks in your base templates as you want and give each a sensible default. The more blocks your base templates have, the more flexible your layout will be;
- If you find yourself duplicating content in a number of templates, it probably means you should move that content to a `{% block %}` in a parent template. In some cases, a better solution may be to move the content to a new template and `include` it (see Including other Templates);
- If you need to get the content of a block from the parent template, you can use the `{{ parent() }}` function. This is useful if you want to add to the contents of a parent block instead of completely overriding it:

Listing 5-10
```
{% block sidebar %}
    <h3>Table of Contents</h3>

    {# ... #}

    {{ parent() }}
{% endblock %}
```

Template Naming and Locations

By default, templates can live in two different locations:

`templates/`
: The application's `views` directory can contain application-wide base templates (i.e. your application's layouts and templates of the application bundle) as well as templates that override third party bundle templates.

`vendor/path/to/CoolBundle/Resources/views/`
: Each third party bundle houses its templates in its `Resources/views/` directory (and subdirectories). When you plan to share your bundle, you should put the templates in the bundle instead of the `templates/` directory.

Most of the templates you'll use live in the `templates/` directory. The path you'll use will be relative to this directory. For example, to render/extend `templates/base.html.twig`, you'll use the `base.html.twig` path and to render/extend `templates/blog/index.html.twig`, you'll use the `blog/index.html.twig` path.

Referencing Templates in a Bundle

If you need to refer to a template that lives in a bundle, Symfony uses the Twig namespaced syntax (`@BundleName/directory/filename.html.twig`). This allows for several types of templates, each which lives in a specific location:

- `@AcmeBlog/Blog/index.html.twig`: This syntax is used to specify a template for a specific page. The three parts of the string, each separated by a slash (`/`), mean the following:

- `@AcmeBlog`: is the bundle name without the `Bundle` suffix. This template lives in the AcmeBlogBundle (e.g. `src/Acme/BlogBundle`);
- `Blog`: (*directory*) indicates that the template lives inside the `Blog` subdirectory of `Resources/views/`;
- `index.html.twig`: (*filename*) the actual name of the file is `index.html.twig`.

Assuming that the AcmeBlogBundle lives at `src/Acme/BlogBundle`, the final path to the layout would be `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`.

- `@AcmeBlog/layout.html.twig`: This syntax refers to a base template that's specific to the AcmeBlogBundle. Since the middle, "directory", portion is missing (e.g. `Blog`), the template lives at `Resources/views/layout.html.twig` inside AcmeBlogBundle.

Using this namespaced syntax instead of the real file paths allows applications to override templates that live inside any bundle.

Template Suffix

Every template name also has two extensions that specify the *format* and *engine* for that template.

Filename	Format	Engine
`blog/index.html.twig`	HTML	Twig
`blog/index.html.php`	HTML	PHP
`blog/index.css.twig`	CSS	Twig

By default, any Symfony template can be written in either Twig or PHP, and the last part of the extension (e.g. `.twig` or `.php`) specifies which of these two *engines* should be used. The first part of the extension, (e.g. `.html`, `.css`, etc) is the final format that the template will generate. Unlike the engine, which determines how Symfony parses the template, this is an organizational tactic used in case the same resource needs to be rendered as HTML (`index.html.twig`), XML (`index.xml.twig`), or any other format. For more information, read the *How to Work with Different Output Formats in Templates* section.

Tags and Helpers

You already understand the basics of templates, how they're named and how to use template inheritance. The hardest parts are already behind you. In this section, you'll learn about a large group of tools available to help perform the most common template tasks such as including other templates, linking to pages and including images.

Symfony comes bundled with several specialized Twig tags and functions that ease the work of the template designer. In PHP, the templating system provides an extensible *helper* system that provides useful features in a template context.

You've already seen a few built-in Twig tags like `{% block %}` and `{% extends %}`. Here you will learn a few more.

Including other Templates

You'll often want to include the same template or code fragment on several pages. For example, in an application with "news articles", the template code displaying an article might be used on the article detail page, on a page displaying the most popular articles, or in a list of the latest articles.

When you need to reuse a chunk of PHP code, you typically move the code to a new PHP class or function. The same is true for templates. By moving the reused template code into its own template, it can be included from any other template. First, create the template that you'll need to reuse.

Listing 5-11
```twig
{# templates/article/article_details.html.twig #}
<h2>{{ article.title }}</h2>
<h3 class="byline">by {{ article.authorName }}</h3>

<p>
    {{ article.body }}
</p>
```

Including this template from any other template is achieved with the `{{ include() }}` function:

Listing 5-12
```twig
{# templates/article/list.html.twig #}
{% extends 'layout.html.twig' %}

{% block body %}
    <h1>Recent Articles<h1>

    {% for article in articles %}
        {{ include('article/article_details.html.twig', { 'article': article }) }}
    {% endfor %}
{% endblock %}
```

Notice that the template name follows the same typical convention. The `article_details.html.twig` template uses an `article` variable, which we pass to it. In this case, you could avoid doing this entirely, as all of the variables available in `list.html.twig` are also available in `article_details.html.twig` (unless you set *with_context*[5] to false).

The `{'article': article}` syntax is the standard Twig syntax for hash maps (i.e. an array with named keys). If you needed to pass in multiple elements, it would look like this: `{'foo': foo, 'bar': bar}`.

Linking to Pages

Creating links to other pages in your application is one of the most common jobs for a template. Instead of hardcoding URLs in templates, use the `path` Twig function (or the `router` helper in PHP) to generate URLs based on the routing configuration. Later, if you want to modify the URL of a particular page, all you'll need to do is change the routing configuration: the templates will automatically generate the new URL.

First, link to the "welcome" page, which is accessible via the following routing configuration:

Listing 5-13
```php
// src/Controller/WelcomeController.php

// ...
use Symfony\Component\Routing\Annotation\Route;

class WelcomeController extends AbstractController
{
    /**
     * @Route("/", name="welcome", methods={"GET"})
     */
    public function index()
    {
        // ...
    }
}
```

To link to the page, use the `path()` Twig function and refer to the route:

Listing 5-14

5. https://twig.symfony.com/doc/2.x/functions/include.html

```
1  <a href="{{ path('welcome') }}">Home</a>
```

As expected, this will generate the URL `/`. Now, for a more complicated route:

Listing 5-15
```
1  // src/Controller/ArticleController.php
2
3  // ...
4  use Symfony\Component\Routing\Annotation\Route;
5
6  class ArticleController extends AbstractController
7  {
8      /**
9       * @Route("/article/{slug}", name="article_show", methods={"GET"})
10      */
11     public function show($slug)
12     {
13         // ...
14     }
15 }
```

In this case, you need to specify both the route name (`article_show`) and a value for the `{slug}` parameter. Using this route, revisit the `recent_list.html.twig` template from the previous section and link to the articles correctly:

Listing 5-16
```
1  {# templates/article/recent_list.html.twig #}
2  {% for article in articles %}
3      <a href="{{ path('article_show', {'slug': article.slug}) }}">
4          {{ article.title }}
5      </a>
6  {% endfor %}
```

You can also generate an absolute URL by using the `url()` Twig function:

Listing 5-17
```
1  <a href="{{ url('welcome') }}">Home</a>
```

Linking to Assets

Templates also commonly refer to images, JavaScript, stylesheets and other assets. You could hard-code the web path to these assets (e.g. `/images/logo.png`), but Symfony provides a more dynamic option via the `asset()` Twig function.

To use this function, install the *asset* package:

Listing 5-18
```
1  $ composer require symfony/asset
```

You can now use the `asset()` function:

Listing 5-19
```
1  <img src="{{ asset('images/logo.png') }}" alt="Symfony!"/>
2
3  <link href="{{ asset('css/blog.css') }}" rel="stylesheet"/>
```

The `asset()` function's main purpose is to make your application more portable. If your application lives at the root of your host (e.g. `http://example.com`), then the rendered paths should be `/images/logo.png`. But if your application lives in a subdirectory (e.g. `http://example.com/my_app`), each asset path should render with the subdirectory (e.g. `/my_app/images/logo.png`).

The `asset()` function takes care of this by determining how your application is being used and generating the correct paths accordingly.

The `asset()` function supports various cache busting techniques via the version, version_format, and json_manifest_path configuration options.

If you need absolute URLs for assets, use the `absolute_url()` Twig function as follows:

Listing 5-20
```
<img src="{{ absolute_url(asset('images/logo.png')) }}" alt="Symfony!"/>
```

Including Stylesheets and JavaScripts in Twig

No site would be complete without including JavaScript files and stylesheets. In Symfony, the inclusion of these assets is handled elegantly by taking advantage of Symfony's template inheritance.

This section will teach you the philosophy behind including stylesheet and JavaScript assets in Symfony. If you are interested in compiling and creating those assets, check out the *Webpack Encore documentation* a tool that seamlessly integrates Webpack and other modern JavaScript tools into Symfony applications.

Start by adding two blocks to your base template that will hold your assets: one called `stylesheets` inside the `head` tag and another called `javascripts` just above the closing `body` tag. These blocks will contain all of the stylesheets and JavaScripts that you'll need throughout your site:

Listing 5-21
```
{# templates/base.html.twig #}
<html>
    <head>
        {# ... #}

        {% block stylesheets %}
            <link href="{{ asset('css/main.css') }}" rel="stylesheet"/>
        {% endblock %}
    </head>
    <body>
        {# ... #}

        {% block javascripts %}
            <script src="{{ asset('js/main.js') }}"></script>
        {% endblock %}
    </body>
</html>
```

This looks almost like regular HTML, but with the addition of the `{% block %}`. Those are useful when you need to include an extra stylesheet or JavaScript from a child template. For example, suppose you have a contact page and you need to include a `contact.css` stylesheet *just* on that page. From inside that contact page's template, do the following:

Listing 5-22
```
{# templates/contact/contact.html.twig #}
{% extends 'base.html.twig' %}

{% block stylesheets %}
    {{ parent() }}

    <link href="{{ asset('css/contact.css') }}" rel="stylesheet"/>
{% endblock %}

{# ... #}
```

In the child template, you override the `stylesheets` block and put your new stylesheet tag inside of that block. Since you want to add to the parent block's content (and not actually *replace* it), you also use the `parent()` Twig function to include everything from the `stylesheets` block of the base template.

You can also include assets located in your bundles' `Resources/public/` folder. You will need to run the `php bin/console assets:install target [--symlink]` command, which copies (or symlinks) files into the correct location. (By default, the target is the `public/` directory of your application.)

Listing 5-23
```
1 <link href="{{ asset('bundles/acmedemo/css/contact.css') }}" rel="stylesheet"/>
```

The end result is a page that includes `main.js` and both the `main.css` and `contact.css` stylesheets.

Referencing the Request, User or Session

Symfony also gives you a global `app` variable in Twig that can be used to access the current user, the Request and more.

See *How to Access the User, Request, Session & more in Twig via the app Variable* for details.

Output Escaping

Twig performs automatic "output escaping" when rendering any content in order to protect you from Cross Site Scripting (XSS) attacks.

Suppose `description` equals `I <3 this product`:

Listing 5-24
```
1 {# output escaping is on automatically #}
2 {{ description }} {# I &lt;3 this product #}
3
4 {# disable output escaping with the raw filter #}
5 {{ description|raw }} {# I <3 this product #}
```

PHP templates do not automatically escape content.

For more details, see *How to Escape Output in Templates*.

Final Thoughts

The templating system is just *one* of the many tools in Symfony. And its job is simple: allow us to render dynamic & complex HTML output so that this can ultimately be returned to the user, sent in an email or something else.

Keep Going!

Before diving into the rest of Symfony, check out the *configuration system*.

Learn more

- How to Use PHP instead of Twig for Templates
- How to Access the User, Request, Session & more in Twig via the `app` Variable
- How to Dump Debug Information in Twig Templates
- How to Embed Controllers in a Template
- How to Escape Output in Templates
- How to Work with Different Output Formats in Templates
- How to Inject Variables into all Templates (i.e. global Variables)
- How to Embed Asynchronous Content with hinclude.js
- How to Organize Your Twig Templates Using Inheritance
- How to Use and Register Namespaced Twig Paths
- How to Render a Template without a custom Controller
- How to Check the Syntax of Your Twig Templates
- How to Write a custom Twig Extension

Chapter 6
Configuring Symfony

Configuration Files

Symfony applications are configured with the files stored in the `config/` directory, which has this default structure:

Listing 6-1
```
your-project/
├─ config/
│  ├─ packages/
│  ├─ bundles.php
│  ├─ routes.yaml
│  └─ services.yaml
├─ ...
```

The `routes.yaml` file defines the *routing configuration*; the `services.yaml` file configures the services of the *service container*; the `bundles.php` file enables/ disables packages in your application.

You'll be working most in the `config/packages/` directory. This directory stores the configuration of every package installed in your application. Packages (also called "bundles" in Symfony and "plugins/ modules" in other projects) add ready-to-use features to your projects.

When using Symfony Flex, which is enabled by default in Symfony applications, packages update the `bundles.php` file and create new files in `config/packages/` automatically during their installation. For example, this is the default file created by the "API Platform" package:

Listing 6-2
```
# config/packages/api_platform.yaml
api_platform:
    mapping:
        paths: ['%kernel.project_dir%/src/Entity']
```

Splitting the configuration into lots of small files is intimidating for some Symfony newcomers. However, you'll get used to them quickly and you rarely need to change these files after package installation

To learn about all the available configuration options, check out the *Symfony Configuration Reference* or run the `config:dump-reference` command.

Configuration Formats

Unlike other frameworks, Symfony doesn't impose you a specific format to configure your applications. Symfony lets you choose between YAML, XML and PHP and throughout the Symfony documentation, all configuration examples will be shown in these three formats.

There isn't any practical difference between formats. In fact, Symfony transforms and caches all of them into PHP before running the application, so there's not even any performance difference between them.

YAML is used by default when installing packages because it's concise and very readable. These are the main advantages and disadvantages of each format:

- **YAML**: simple, clean and readable, but not all IDEs support autocompletion and validation for it. *Learn the YAML syntax*;
- **XML**:autocompleted/validated by most IDEs and is parsed natively by PHP, but sometimes it generates too verbose configuration. *Learn the XML syntax*[1];
- **PHP**: very powerful and it allows to create dynamic configuration, but the resulting configuration is less readable than the other formats.

Importing Configuration Files

Symfony loads configuration files using the *Config component*, which provides advanced features such as importing other configuration files, even if they use a different format:

Listing 6-3
```yaml
# config/services.yaml
imports:
    - { resource: 'legacy_config.php' }
    # ignore_errors silently discards errors if the loaded file doesn't exist
    - { resource: 'my_config_file.xml', ignore_errors: true }
    # glob expressions are also supported to load multiple files
    - { resource: '/etc/myapp/*.yaml' }

# ...
```

Configuration Parameters

Sometimes the same configuration value is used in several configuration files. Instead of repeating it, you can define it as a "parameter", which is like a reusable configuration value. By convention, parameters are defined under the `parameters` key in the `config/services.yaml` file:

Listing 6-4
```yaml
# config/services.yaml
parameters:
    # the parameter name is an arbitrary string (the 'app.' prefix is recommended
    # to better differentiate your parameters from Symfony parameters).
    app.admin_email: 'something@example.com'

    # boolean parameters
    app.enable_v2_protocol: true

    # array/collection parameters
    app.supported_locales: ['en', 'es', 'fr']

```

1. https://en.wikipedia.org/wiki/XML

```yaml
    # binary content parameters (encode the contents with base64_encode())
    app.some_parameter: !!binary VGhpcyBpcyBhIEJlbGwgY2hhciAH

    # PHP constants as parameter values
    app.some_constant: !php/const GLOBAL_CONSTANT
    app.another_constant: !php/const App\Entity\BlogPost::MAX_ITEMS

# ...
```

When using XML configuration, the values between `<parameter>` tags are not trimmed. This means that the value of the following parameter will be `'\n something@example.com\n'`:

Listing 6-5
```xml
<parameter key="app.admin_email">
    something@example.com
</parameter>
```

Once defined, you can reference this parameter value from any other configuration file using a special syntax: wrap the parameter name in two `%` (e.g. `%app.admin_email%`):

Listing 6-6
```yaml
# config/packages/some_package.yaml
some_package:
    # any string surrounded by two % is replaced by that parameter value
    email_address: '%app.admin_email%'

    # ...
```

If some parameter value includes the `%` character, you need to escape it by adding another `%` so Symfony doesn't consider it a reference to a parameter name:

Listing 6-7
```yaml
# config/services.yaml
parameters:
    # Parsed as 'https://symfony.com/?foo=%s&amp;bar=%d'
    url_pattern: 'https://symfony.com/?foo=%%s&amp;bar=%%d'
```

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 6-8
```yaml
# config/services.yaml
imports:
    - { resource: '%kernel.project_dir%/somefile.yaml' }
```

Configuration parameters are very common in Symfony applications. Some packages even define their own parameters (e.g. when installing the translation package, a new `locale` parameter is added to the `config/services.yaml` file).

> *Read the Accessing Configuration Values section of this article to learn about how to use these configuration parameters in services and controllers.*

Configuration Environments

You have just one application, but whether you realize it or not, you need it to behave differently at different times:

- While **developing**, you want to log everything and expose nice debugging tools;

- After deploying to **production**, you want that same application to be optimized for speed and only log errors.

The files stored in `config/packages/` are used by Symfony to configure the *application services*. In other words, you can change the application behavior by changing which configuration files are loaded. That's the idea of Symfony's **configuration environments**.

A typical Symfony application begins with three environments: `dev` (for local development), `prod` (for production servers) and `test` (for *automated tests*). When running the application, Symfony loads the configuration files in this order (the last files can override the values set in the previous ones):

1. `config/packages/*.yaml` (and `.xml` and `*.php` files too);
2. `config/packages/<environment-name>/*.yaml` (and `.xml` and `*.php` files too);
3. `config/packages/services.yaml` (and `services.xml` and `services.php` files too);

Take the `framework` package, installed by default, as an example:

- First, `config/packages/framework.yaml` is loaded in all environments and it configures the framework with some options;
- In the **prod** environment, nothing extra will be set as there is no `config/packages/prod/framework.yaml` file;
- In the **dev** environment, there is no file either (`config/packages/dev/framework.yaml` does not exist).
- In the **test** environment, the `config/packages/test/framework.yaml` file is loaded to override some of the settings previously configured in `config/packages/framework.yaml`.

In reality, each environment differs only somewhat from others. This means that all environments share a large base of common configurations, which is put in files directly in the `config/packages/` directory.

> *See the `configureContainer()` method of the Kernel class to learn everything about the loading order of configuration files.*

Selecting the Active Environment

Symfony applications come with a file called `.env` located at the project root directory. This file is used to define the value of environment variables and it's explained in detail later in this article.

Open the `.env` file (or better, the `.env.local` file if you created one) and edit the value of the `APP_ENV` variable to change the environment in which the application runs. For example, to run the application in production:

Listing 6-9
```
# .env (or .env.local)
APP_ENV=prod
```

This value is used both for the web and for the console commands. However, you can override it for commands by setting the `APP_ENV` value before running them:

Listing 6-10
```
# Use the environment defined in the .env file
$ php bin/console command_name

# Ignore the .env file and run this command in production
$ APP_ENV=prod php bin/console command_name
```

Creating a New Environment

The default three environments provided by Symfony are enough for most projects, but you can define your own environments too. For example, this is how you can define a `staging` environment where the client can test the project before going to production:

1. Create a configuration directory with the same name as the environment (in this case, `config/packages/staging/`);
2. Add the needed configuration files in `config/packages/staging/` to define the behavior of the new environment. Symfony loads first the files in `config/packages/*.yaml`, so you must only configure the differences with those files;
3. Select the `staging` environment using the `APP_ENV` env var as explained in the previous section.

It's common for environments to be similar between each other, so you can use *symbolic links*[2] between `config/packages/<environment-name>/` directories to reuse the same configuration.

Configuration Based on Environment Variables

Using *environment variables*[3] (or "env vars" for short) is a common practice to configure options that depend on where the application is run (e.g. the database credentials are usually different in production and in your local machine).

Instead of defining those as regular options, you can define them as environment variables and reference them in the configuration files using the special syntax `%env(ENV_VAR_NAME)%`. The values of these options are resolved at runtime (only once per request, to not impact performance).

This example shows how to configure the database connection using an env var:

Listing 6-11
```yaml
# config/packages/doctrine.yaml
doctrine:
    dbal:
        # by convention the env var names are always uppercase
        url: '%env(DATABASE_URL)%'
    # ...
```

The next step is to define the value of those env vars in your shell, your web server, etc. This is explained in the following sections, but to protect your application from undefined env vars, you can give them a default value using the `.env` file:

Listing 6-12
```
# .env
DATABASE_URL=sqlite:///%kernel.project_dir%/var/data.db
```

> *The values of env vars can only be strings, but Symfony includes some env var processors to transform their contents (e.g. to turn a string value into an integer).*

In order to define the actual values of env vars, Symfony proposes different solutions depending if the application is running in production or in your local development machine.

Independent from the way you set environmnet variables, you may need to run the `debug:container` command with the `--env-vars` option to verify that they are defined and have the expected values:

Listing 6-13
```
$ php bin/console debug:container --env-vars

---------------- ----------------- ------------------------------------------
 Name             Default value     Real value
---------------- ----------------- ------------------------------------------
 APP_SECRET       n/a               "471a62e2d601a8952deb186e44186cb3"
 FOO              "[1, "2.5", 3]"   n/a
 BAR              null              n/a
---------------- ----------------- ------------------------------------------
```

2. https://en.wikipedia.org/wiki/Symbolic_link
3. https://en.wikipedia.org/wiki/Environment_variable

```
10
11  # you can also filter the list of env vars by name:
12  $ php bin/console debug:container --env-vars foo
13
14  # run this command to show all the details for a specific env var:
15  $ php bin/console debug:container --env-var=FOO
```

Configuring Environment Variables in Development

Instead of defining env vars in your shell or your web server, Symfony proposes a convenient way of defining them in your local machine based on a file called `.env` (with a leading dot) located at the root of your project.

The `.env` file is read and parsed on every request and its env vars are added to the `$_ENV` PHP variable. The existing env vars are never overwritten by the values defined in `.env`, so you can combine both.

This is for example the content of the `.env` file to define the value of the `DATABASE_URL` env var shown earlier in this article:

Listing 6-14

```
# .env
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name"
```

In addition to your own env vars, this `.env` file also contains the env vars defined by the third-party packages installed in your application (they are added automatically by Symfony Flex when installing packages).

Configuring Environment Variables in Production

In production, the `.env` files are also parsed and loaded on each request so you can override the env vars already defined in the server. In order to improve performance, you can run the `dump-env` command (available when using Symfony Flex 1.2 or later).

This command parses all the `.env` files once and compiles their contents into a new PHP-optimized file called `.env.local.php`. From that moment, Symfony will load the parsed file instead of parsing the `.env` files again:

Listing 6-15

```
$ composer dump-env prod
```

Update your deployment tools/workflow to run the `dump-env` command after each deploy to improve the application performance.

Creating `.env` files is the easiest way of using env vars in Symfony applications. However, you can also configure real env vars in your servers and operating systems.

SymfonyCloud, the cloud service optimized for Symfony applications, defines some *utilities to manage env vars*[4] in production.

4. https://symfony.com/doc/master/cloud/cookbooks/env.html

Beware that dumping the contents of the `$_SERVER` and `$_ENV` variables or outputting the `phpinfo()` contents will display the values of the environment variables, exposing sensitive information such as the database credentials.

The values of the env vars are also exposed in the web interface of the *Symfony profiler*. In practice this shouldn't be a problem because the web profiler must **never** be enabled in production.

Managing Multiple .env Files

The `.env` file defines the default values for all env vars. However, it's common to override some of those values depending on the environment (e.g. to use a different database for tests) or depending on the machine (e.g. to use a different OAuth token on your local machine while developing).

That's why you can define multiple `.env` files to override env vars. The following list shows the files loaded in all environments. The `.env` file is the only mandatory file and each file content overrides the previous one:

- `.env`: defines the default values of the env vars needed by the application;
- `.env.local`: defines machine-specific overrides for env vars on all environments. This file is not committed to the repository, so these overrides only apply to the machine which contains the file (your local computer, production server, etc.);
- `.env.<environment>` (e.g. `.env.test`): overrides env vars only for some environment but for all machines;
- `.env.<environment>.local` (e.g. `.env.test.local`): defines machine-specific env vars overrides only for some environment. It's similar to `.env.local`, but the overrides only apply to some particular environment.

The real environment variables defined in the server always win over the env vars created by the `.env` files.

The `.env` and `.env.<environment>` files should be committed to the shared repository because they are the same for all developers and machines. However, the env files ending in `.local` (`.env.local` and `.env.<environment>.local`) **should not be committed** because only you will use them. In fact, the `.gitignore` file that comes with Symfony prevents them from being committed.

Applications created before November 2018 had a slightly different system, involving a `.env.dist` file. For information about upgrading, see: *Nov 2018 Changes to .env & How to Update*.

Accessing Configuration Values

Controllers and services can access all the configuration parameters. This includes both the parameters defined by yourself and the parameters created by packages/bundles. Run the following command to see all the parameters that exist in your application:

Listing 6-16
```
$ php bin/console debug:container --parameters
```

Parameters are injected in services as arguments to their constructors. *Service autowiring* doesn't work for parameters. Instead, inject them explicitly:

Listing 6-17
```
# config/services.yaml
parameters:
    app.contents_dir: '...'
```

```yaml

services:
    App\Service\MessageGenerator:
        arguments:
            $contentsDir: '%app.contents_dir%'
```

If you inject the same parameters over and over again, use instead the `services._defaults.bind` option. The arguments defined in that option are injected automatically whenever a service constructor or controller action define an argument with that exact name. For example, to inject the value of the kernel.project_dir parameter whenever a service/controller defines a `$projectDir` argument, use this:

Listing 6-18

```yaml
# config/services.yaml
services:
    _defaults:
        bind:
            # pass this value to any $projectDir argument for any service
            # that's created in this file (including controller arguments)
            $projectDir: '%kernel.project_dir%'

    # ...
```

Read the article about binding arguments by name and/or type to learn more about this powerful feature.

Finally, if some service needs to access to lots of parameters, instead of injecting each of them individually, you can inject all the application parameters at once by type-hinting any of its constructor arguments with the *`ContainerBagInterface`*[5]:

Listing 6-19

```php
// src/Service/MessageGenerator.php
// ...

use Symfony\Component\DependencyInjection\ParameterBag\ContainerBagInterface;

class MessageGenerator
{
    private $params;

    public function __construct(ContainerBagInterface $params)
    {
        $this->params = $params;
    }

    public function someMethod()
    {
        // get any container parameter from $this->params, which stores all of them
        $sender = $this->params->get('mailer_sender');
        // ...
    }
}
```

Keep Going!

Congratulations! You've tackled the basics in Symfony. Next, learn about *each* part of Symfony individually by following the guides. Check out:

- *Forms*
- *Databases and the Doctrine ORM*
- *Service Container*
- *Security*

5. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/DependencyInjection/ParameterBag/ContainerBagInterface.php

- *Swift Mailer*
- *Logging*

And all the other topics related to configuration:

- Nov 2018 Changes to .env & How to Update
- Environment Variable Processors
- Understanding how the Front Controller, Kernel and Environments Work together
- Building your own Framework with the MicroKernelTrait
- How To Create Symfony Applications with Multiple Kernels
- How to Override Symfony's default Directory Structure
- Using Parameters within a Dependency Injection Class