Symfony

Getting Started

Version: 3.4

generated on November 14, 2019

Getting Started (3.4)

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

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

- **Attribution**: You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

- **Share Alike**: If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license. For any reuse or distribution, you must make clear to others the license terms of this work.

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

If you find typos or errors, feel free to report them by creating a ticket on the Symfony ticketing system (*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

This article explains how to install Symfony in different ways and how to solve the most common issues that may appear during the installation process.

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

Creating Symfony Applications

Symfony provides a dedicated application called the **Symfony Installer** to ease the creation of Symfony applications. This installer is a PHP 5.4 compatible executable that needs to be installed on your system only once:

Linux and macOS systems:

```
1  $ sudo mkdir -p /usr/local/bin
2  $ sudo curl -LsS https://symfony.com/installer -o /usr/local/bin/symfony
3  $ sudo chmod a+x /usr/local/bin/symfony
```

Windows systems:

```
1  > php -r "file_put_contents('symfony', file_get_contents('https://symfony.com/installer'));"
```

---

1. *https://symfonycasts.com/screencast/symfony3*

In Linux and macOS, a global `symfony` command is created. In Windows, move the `symfony` file to a directory that's included in the `PATH` environment variable and create a `symfony.bat` file to create the global command or move it to any other directory convenient for you:

```
1   # for example, if WAMP is used ...
2   > move symfony c:\wamp\bin\php
3   # create symfony.bat in the same folder
4   > cd c:\wamp\bin\php
5   > (echo @ECHO OFF & echo php "%~dp0symfony" %*) > symfony.bat
6   # ... then, execute the command as:
7   > symfony
8
9   # moving it to your projects folder ...
10  > move symfony c:\projects
11  # ... then, execute the command as
12  > cd projects
13  > php symfony
```

Once the Symfony Installer is installed, create your first Symfony application with the `new` command:

```
1   $ symfony new my_project_name --version=3.4
```

This command creates a new directory called `my_project_name/` that contains an empty project based on the most recent stable Symfony version available. In addition, the installer checks if your system meets the technical requirements to execute Symfony applications. If not, you'll see the list of changes needed to meet those requirements.

If the installer doesn't work for you or doesn't output anything, make sure that the PHP *Phar extension*[2] is installed and enabled on your computer.

If the SSL certificates are not properly installed in your system, you may get this error:

> cURL error 60: SSL certificate problem: unable to get local issuer certificate.

You can solve this issue as follows:

1. Download a file with the updated list of certificates from *https://curl.haxx.se/ca/cacert.pem*

2. Move the downloaded `cacert.pem` file to some safe location in your system

3. Update your `php.ini` file and configure the path to that file:

```
1   ; Linux and macOS systems
2   curl.cainfo = "/path/to/cacert.pem"
3
4   ; Windows systems
5   curl.cainfo = "C:\path\to\cacert.pem"
```

Basing your Project on a Specific Symfony Version

In case your project needs to be based on a specific Symfony version, use the `version` option of the `new` command:

---

2. `https://php.net/manual/en/intro.phar.php`

```
1   # use the most recent version in any Symfony branch
2   $ symfony new my_project_name --version=3.3
3   $ symfony new my_project_name --version=3.4
4
5   # use a specific Symfony version
6   $ symfony new my_project_name --version=3.3.10
7   $ symfony new my_project_name --version=3.4.5
8
9   # use the most recent 'lts' version (Long Term Support version)
10  $ symfony new my_project_name --version=lts
11
12  # use the 'next' Symfony version to be released (still in development)
13  $ symfony new my_project_name --version=next
```

Each version has its *own* documentation, which you can select on any documentation page.

> Read the *Symfony Release process* to better understand why there are several Symfony versions and which one to use for your projects.

Creating Symfony Applications with Composer

If you can't use the Symfony installer for any reason, you can create Symfony applications with *Composer*[3], the dependency manager used by modern PHP applications.

If you don't have Composer installed in your computer, start by *installing Composer*[4]. Then, execute the `create-project` command to create a new Symfony application based on its latest stable version:

```
1   $ composer create-project symfony/framework-standard-edition my_project_name
```

You can also install any other Symfony version by passing a second argument to the `create-project` command:

```
1   $ composer create-project symfony/framework-standard-edition my_project_name "2.8.*"
```

> If your Internet connection is slow, you may think that Composer is not doing anything. If that's your case, add the `-vvv` flag to the previous command to display a detailed output of everything that Composer is doing.

Running the Symfony Application

On production servers, Symfony applications use web servers such as Apache or Nginx (see *configuring a web server to run Symfony*). However, on your local development machine you can also use the web server provided by Symfony, which in turn uses the built-in web server provided by PHP.

First, *install the Symfony Web Server* and then, execute this command:

```
1   $ cd my_project_name/
2   $ php bin/console server:run
```

Open your browser and access the `http://localhost:8000/` URL to see the Welcome Page of Symfony:

---

3. https://getcomposer.org/
4. https://getcomposer.org/download/

If you see a blank page or an error page instead of the Welcome Page, there is a directory permission misconfiguration. The solution to this problem is explained in the *Setting up or Fixing File Permissions*.

When you are finished working on your Symfony application, stop the server by pressing `Ctrl+C` from the terminal or command console.

> Symfony's web server is great for developing, but should **not** be used on production. Instead, use Apache or Nginx. See *Configuring a Web Server*.

Checking Symfony Application Configuration and Setup

The Symfony Installer checks if your system is ready to run Symfony applications. However, the PHP configuration for the command console can be different from the PHP web configuration. For that reason, Symfony provides a visual configuration checker. Access the following URL to check your configuration and fix any issue before moving on:

Listing 1-10

```
1   http://localhost:8000/config.php
```

Fixing Permissions Problems

If you have any file permission errors or see a white screen, then read *Setting up or Fixing File Permissions* for more information.

Updating Symfony Applications

At this point, you've created a fully-functional Symfony application! Every Symfony app depends on a number of third-party libraries stored in the `vendor/` directory and managed by Composer.

Updating those libraries frequently is a good practice to prevent bugs and security vulnerabilities. Execute the `update` Composer command to update them all at once (this can take up to several minutes to complete depending on the complexity of your project):

Listing 1-11

```
1   $ cd my_project_name/
2   $ composer update
```

> Symfony provides a command to check whether your project's dependencies contain any known security vulnerability:
>
> *Listing 1-12*
>
> ```
> 1   $ php bin/console security:check
> ```
>
> A good security practice is to execute this command regularly to be able to update or replace compromised dependencies as soon as possible.

Installing the Symfony Demo or Other Distributions

You've already downloaded the *Symfony Standard Edition*[5]: the default starting project for all Symfony applications. You'll use this project throughout the documentation to build your application!

Symfony also provides some other projects and starting skeletons that you can use:

The Symfony Demo Application[6]

This is a fully-functional application that shows the recommended way to develop Symfony applications. The app has been conceived as a learning tool for Symfony newcomers and its source code contains tons of comments and helpful notes.

The Symfony CMF Standard Edition[7]

The *Symfony CMF*[8] is a project that helps make it easier for developers to add CMS functionality to their Symfony applications. This is a starting project containing the Symfony CMF.

The Symfony REST Edition[9]

Shows how to build an application that provides a RESTful API using the *FOSRestBundle*[10] and several other related Bundles.

Installing an Existing Symfony Application

When working collaboratively in a Symfony application, it's uncommon to create a new Symfony application as explained in the previous sections. Instead, someone else has already created and submitted it to a shared repository.

It's recommended to not submit some files (parameters.yml) and directories (`vendor/`, cache, logs) to the repository, so you'll have to do the following when installing an existing Symfony application:

```
1   # clone the project to download its contents
2   $ cd projects/
3   $ git clone ...
4
5   # make Composer install the project's dependencies into vendor/
6   $ cd my_project_name/
7   $ composer install
8
9   # now Composer will ask you for the values of any undefined parameter
10  $ ...
```

Keep Going!

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

Go Deeper with Setup

- Using Symfony with Homestead/Vagrant
- How to Create and Store a Symfony Project in Git
- How to Use PHP's built-in Web Server
- Configuring a Web Server

---

5. https://github.com/symfony/symfony-standard

6. **https://github.com/symfony/demo**

7. **https://github.com/symfony-cmf/standard-edition**

8. http://cmf.symfony.com/

9. **https://github.com/gimler/symfony-rest-edition**

10. https://github.com/FriendsOfSymfony/FOSRestBundle

- Upgrading a Third-Party Bundle for a Major Symfony Version
- Setting up or Fixing File Permissions
- Using Symfony Flex to Manage Symfony Applications
- How to Create and Store a Symfony Project in Subversion
- Symfony Local Web Server
- How to Install or Upgrade to the Latest, Unreleased Symfony Version
- Upgrading a Major Version (e.g. 2.7.0 to 3.0.0)
- Upgrading a Minor Version (e.g. 2.5.3 to 2.6.1)
- Upgrading a Patch Version (e.g. 2.6.0 to 2.6.1)

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 Joyful 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 that will be executed when someone goes to `/lucky/number`:

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

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

class LuckyController
```

---

1. *https://symfonycasts.com/screencast/symfony3*

```
 9  {
10      /**
11       * @Route("/lucky/number")
12       */
13      public function numberAction()
14      {
15          $number = random_int(0, 100);
16
17          return new Response(
18              '<html><body>Lucky number: '.$number.'</body></html>'
19          );
20      }
21  }
```

Before diving into this, test it out! If you are using PHP's internal web server go 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*: The `@Route` above `numberAction()` is the *route*: it defines the URL pattern for this page. You'll learn more about *routing* in its own section, including how to make *variable* URLs;
2. *Create a controller*: The method below the route - `numberAction()` - is called the *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.

The Web Debug Toolbar: Debugging Dream

If your page is working, then you should *also* see a bar along the bottom of your browser. This is called the Web Debug Toolbar: and it's your debugging best friend. 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.

Rendering a Template (with the Service Container)

If you're returning HTML from your controller, you'll probably want to render a template. Fortunately, Symfony comes with *Twig*[2]: a templating language that's easy, powerful and actually quite fun.

First, import the base **Controller**[3] class as shown on line 5 below. Then, let your `LuckyController` class extend the base class:

```
 1  // src/AppBundle/Controller/LuckyController.php
 2
 3  // ...
 4  // --> add this new use statement
 5  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
 6
 7  class LuckyController extends Controller
 8  {
 9      // ...
10  }
```

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

3. `https://github.com/symfony/symfony/blob/3.4/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php`

Now, use the handy `render()` function to render a template. Pass it our `number` variable so we can render that:

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

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

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

Finally, template files should live in the `app/Resources/views` directory. Create a new `app/Resources/views/lucky` directory with a new `number.html.twig` file inside:

```twig
{# app/Resources/views/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*

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.

Checking out the Project Structure

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

app/
> Contains things like configuration and templates. Basically, anything that is *not* PHP code goes here.

src/
> Your PHP code lives here.

99% of the time, you'll be working in `src/` (PHP files) or `app/` (everything else). 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).

tests/
> The automated tests (e.g. Unit tests) for your application live here.

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

vendor/

Third-party (i.e. "vendor") libraries live here! These are downloaded via the *Composer*[4] package manager.

web/

This is the document root for your project: put any publicly accessible files here (e.g. CSS, JS and images).

Bundles & Configuration

Your Symfony application comes pre-installed with a collection of *bundles*, like `FrameworkBundle` and `TwigBundle`. Bundles are similar to the idea of a *plugin*, but with one important difference: *all* functionality in a Symfony application comes from a bundle.

Bundles are registered in your `app/AppKernel.php` file (a rare PHP file in the `app/` directory) and each gives you more *tools*, sometimes called *services*:

```
1   class AppKernel extends Kernel
2   {
3       public function registerBundles()
4       {
5           $bundles = [
6               new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
7               new Symfony\Bundle\TwigBundle\TwigBundle(),
8               // ...
9           ];
10          // ...
11
12          return $bundles;
13      }
14
15      // ...
16  }
```

For example, `TwigBundle` is responsible for adding the Twig tool to your app!

Eventually, you'll download and add more third-party bundles to your app in order to get even more tools. Imagine a bundle that helps you create paginated lists. That exists!

You can control how your bundles behave via the `app/config/config.yml` file. That file - and other details like environments & parameters - are discussed in the *Configuring Symfony (and Environments)* article.

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 (and Environments)*

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

---

4. https://getcomposer.org

Have fun!

Go Deeper with HTTP & Framework Fundamentals

- Symfony versus Flat PHP
- Symfony and HTTP Fundamentals

Chapter 3

Routing

Beautiful URLs are a must for any serious web application. This means leaving behind ugly URLs like `index.php?article_id=57` in favor of something like `/read/intro-to-symfony`.

Having flexibility is even more important. What if you need to change the URL of a page from `/blog` to `/news`? How many links would you need to hunt down and update to make the change? If you're using Symfony's router, the change should be trivial.

The Symfony router lets you define creative URLs that you map to different areas of your application. By the end of this article, you'll be able to:

- Create complex routes that map to controllers
- Generate URLs inside templates and controllers
- Load routing resources from bundles (or anywhere else)
- Debug your routes

Routing Examples

A *route* is a map from a URL path to attributes (i.e a controller). Suppose you want one route that matches `/blog` exactly and another more dynamic route that can match *any* URL like `/blog/my-post` or `/blog/all-about-symfony`.

Routes can be configured in YAML, XML, PHP or annotations. All formats provide the same features and performance, so choose the one you prefer:

```
1  // src/AppBundle/Controller/BlogController.php
2  namespace AppBundle\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5  use Symfony\Component\Routing\Annotation\Route;
6
7  class BlogController extends Controller
8  {
9      /**
10      * Matches /blog exactly
11      *
12      * @Route("/blog", name="blog_list")
```

```
13       */
14      public function listAction()
15      {
16          // ...
17      }
18
19      /**
20       * Matches /blog/*
21       * but not /blog/slug/extra-part
22       *
23       * @Route("/blog/{slug}", name="blog_show")
24       */
25      public function showAction($slug)
26      {
27          // $slug will equal the dynamic part of the URL
28          // e.g. at /blog/yay-routing, then $slug='yay-routing'
29
30          // ...
31      }
32  }
```

Thanks to these two routes:

- If the user goes to /blog, the first route is matched and listAction() is executed;
- If the user goes to /blog/*, the second route is matched and showAction() is executed. Because the route path is /blog/{slug}, a $slug variable is passed to showAction() matching that value. For example, if the user goes to /blog/yay-routing, then $slug will equal yay-routing.

Whenever you have a {placeholder} in your route path, that portion becomes a wildcard: it matches *any* value. Your controller can now *also* have an argument called $placeholder (the wildcard and argument names *must* match).

⚠️ However, the slash / is ignored by default in placeholder values because the router uses it as separator between different placeholders. To learn more about this, you can read How to Allow a "/" Character in a Route Parameter.

Each route also has an internal name: blog_list and blog_show. These can be anything (as long as each is unique) and don't have any meaning yet. Later, you'll use it to generate URLs.

➕ Routing in Other Formats

The @Route above each method is called an *annotation*. If you'd rather configure your routes in YAML, XML or PHP, that's no problem!

In these formats, the _controller "defaults" value is a special key that tells Symfony which controller should be executed when a URL matches this route. The _controller string is called the logical name. It follows a pattern that points to a specific PHP class and method, in this case the AppBundle\Controller\BlogController::listAction and AppBundle\Controller\BlogController::showAction methods.

This is the goal of the Symfony router: to map the URL of a request to a controller. Along the way, you'll learn all sorts of tricks that make mapping even the most complex URLs easy.

Adding {wildcard} Requirements

Imagine the blog_list route will contain a paginated list of blog posts, with URLs like /blog/2 and /blog/3 for pages 2 and 3. If you change the route's path to /blog/{page}, you'll have a problem:

- blog_list: `/blog/{page}` will match `/blog/*`;
- blog_show: `/blog/{slug}` will *also* match `/blog/*`.

When two routes match the same URL, the *first* route that's loaded wins. Unfortunately, that means that `/blog/yay-routing` will match the `blog_list`. No good!

To fix this, add a *requirement* that the `{page}` wildcard can *only* match numbers (digits):

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

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

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

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

The `\d+` is a regular expression that matches a *digit* of any length. Now:

URL	Route	Parameters
`/blog/2`	`blog_list`	`$page = 2`
`/blog/yay-routing`	`blog_show`	`$slug = yay-routing`

To learn about other route requirements - like HTTP method, hostname and dynamic expressions - see *How to Define Route Requirements*.

Giving {placeholders} a Default Value

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

So how can you make `blog_list` once again match when the user visits `/blog`? By adding a *default* value:

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

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

class BlogController extends Controller
{
    /**
     * @Route("/blog/{page}", name="blog_list", requirements={"page"="\d+"})
```

```
11         */
12         public function listAction($page = 1)
13         {
14             // ...
15         }
16  }
```

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

Advanced Routing Example

With all of this in mind, check out this advanced example:

```
1   // src/AppBundle/Controller/ArticleController.php
2
3   // ...
4   class ArticleController extends Controller
5   {
6       /**
7        * @Route(
8        *     "/articles/{_locale}/{year}/{slug}.{_format}",
9        *     defaults={"_format": "html"},
10       *     requirements={
11       *         "_locale": "en|fr",
12       *         "_format": "html|rss",
13       *         "year": "\d+"
14       *     }
15       * )
16       */
17      public function showAction($_locale, $year, $slug)
18      {
19      }
20  }
```

As you've seen, this route will only match if the {_locale} portion of the URL is either en or fr and if
the {year} is a number. This route also shows how you can use a dot between placeholders instead of a
slash. URLs matching this route might look like:

- /articles/en/2010/my-post
- /articles/fr/2010/my-post.rss
- /articles/en/2013/my-latest-post.html

> **➕ The Special _format Routing Parameter**
>
> This example also highlights the special _format routing parameter. When using this parameter,
> the matched value becomes the "request format" of the Request object.
>
> Ultimately, the request format is used for such things as setting the Content-Type of the response
> (e.g. a json request format translates into a Content-Type of application/json). It can also
> be used in the controller to render a different template for each value of _format. The _format
> parameter is a very powerful way to render the same content in different formats.
>
> In Symfony versions previous to 3.0, it is possible to override the request format by adding a query
> parameter named _format (for example: /foo/bar?_format=json). Relying on this behavior
> not only is considered a bad practice but it will complicate the upgrade of your applications to
> Symfony 3.

Sometimes you want to make certain parts of your routes globally configurable. Symfony provides you with a way to do this by leveraging service container parameters. Read more about this in "*How to Use Service Container Parameters in your Routes*".

Special Routing Parameters

As you've seen, each routing parameter or default value is eventually available as an argument in the controller method. Additionally, there are four parameters that are special: each adds a unique piece of functionality inside your application:

`_controller`
> As you've seen, this parameter is used to determine which controller is executed when the route is matched.

`_format`
> Used to set the request format (read more).

`_fragment`
> Used to set the fragment identifier, the optional last part of a URL that starts with a `#` character and is used to identify a portion of a document.
>
> *New in version 3.2:* The `_fragment` parameter was introduced in Symfony 3.2.

`_locale`
> Used to set the locale on the request (read more).

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 path	If the requested URL is `/foo`	If the requested URL is `/foo/`
`/foo`	It matches (200 status response)	It doesn't match (404 status response)
`/foo/`	It makes a 301 redirect to `/foo/`	It matches (200 status response)

In summary, adding a trailing slash in the route path is the best way to ensure that both URLs work. Read the *Redirect URLs with a Trailing Slash* article to learn how to avoid the 404 error when the request URL contains a trailing slash and the route path does not.

Controller Naming Pattern

If you use YAML, XML or PHP route configuration, then each route must have a `_controller` parameter, which dictates which controller should be executed when that route is matched. This parameter uses a simple string pattern called the *logical controller name*, which Symfony maps to a specific PHP method and class. The pattern has three parts, each separated by a colon:

> **bundle**:**controller**:**action**

For example, a `_controller` value of `AppBundle:Blog:show` means:

Bundle	Controller Class	Method Name
AppBundle	BlogController	showAction()

The controller might look like this:

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

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function showAction($slug)
    {
        // ...
    }
}
```

Notice that Symfony adds the string `Controller` to the class name (`Blog => BlogController`) and `Action` to the method name (`show => showAction()`).

You could also refer to this controller using its fully-qualified class name and method: `AppBundle\Controller\BlogController::showAction`. But if you follow some simple conventions, the logical name is more concise and allows more flexibility.

> To refer to an action that is implemented as the `__invoke()` method of a controller class, you do not have to pass the method name, but can just use the fully qualified class name (e.g. `AppBundle\Controller\BlogController`).

> In addition to using the logical name or the fully-qualified class name, Symfony supports a third way of referring to a controller. This method uses just one colon separator (e.g. `service_name:indexAction`) and refers to the controller as a service (see *How to Define Controllers as Services*).

Loading Routes

Symfony loads all the routes for your application from a *single* routing configuration file: `app/config/routing.yml`. But from inside of this file, you can load any *other* routing files you want. In fact, by default, Symfony loads annotation route configuration from your AppBundle's `Controller/` directory, which is how Symfony sees our annotation routes:

```yaml
# app/config/routing.yml
app:
    resource: "@AppBundle/Controller/"
    type:     annotation
```

For more details on loading routes, including how to prefix the paths of loaded routes, see *How to Include External Routing Resources*.

Generating URLs

The routing system should also be used to generate URLs. In reality, routing is a bidirectional system: mapping the URL to a controller and a route back to a URL.

To generate a URL, you need to specify the name of the route (e.g. `blog_show`) and any wildcards (e.g. `slug = my-blog-post`) used in the path for that route:

```php
class MainController extends Controller
{
    public function showAction($slug)
    {
        // ...

        // /blog/my-blog-post
        $url = $this->generateUrl(
            'blog_show',
            ['slug' => 'my-blog-post']
        );
    }
}
```

> ✎ The `generateUrl()` method defined in the base *Controller*[1] class is just a shortcut for this code:
>
> ```php
> $url = $this->container->get('router')->generate(
>     'blog_show',
>     ['slug' => 'my-blog-post']
> );
> ```

Generating URLs with Query Strings

The `generate()` method takes an array of wildcard values to generate the URI. But if you pass extra ones, they will be added to the URI as a query string:

```php
$this->get('router')->generate('blog', [
    'page' => 2,
    'category' => 'Symfony',
]);
// /blog/2?category=Symfony
```

Generating URLs from a Template

To generate URLs inside Twig, see the templating article: Linking to Pages. If you also need to generate URLs in JavaScript, see *How to Generate Routing URLs in JavaScript*.

Generating Absolute URLs

By default, the router will generate relative URLs (e.g. `/blog`). From a controller, pass `UrlGeneratorInterface::ABSOLUTE_URL` to the third argument of the `generateUrl()` method:

```php
use Symfony\Component\Routing\Generator\UrlGeneratorInterface;

$this->generateUrl('blog_show', ['slug' => 'my-blog-post'], UrlGeneratorInterface::ABSOLUTE_URL);
// http://www.example.com/blog/my-blog-post
```

---

1. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php

The host that's used when generating an absolute URL is automatically detected using the current `Request` object. When generating absolute URLs from outside the web context (for instance in a console command) this doesn't work. See *How to Generate URLs from the Console* to learn how to solve this problem.

Troubleshooting

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

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

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

```php
public function showAction($slug)
{
    // ...
}
```

But your route path does *not* have a `{slug}` wildcard (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}`) wildcard in the route path. To fix this, pass a `slug` value when generating the route:

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

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

Translating Routes

Symfony doesn't support defining routes with different contents depending on the user language. In those cases, you can define multiple routes per controller, one for each supported language; or use any of the bundles created by the community to implement this feature, such as *JMSI18nRoutingBundle*[2] and *BeSimpleI18nRoutingBundle*[3].

Summary

Routing is a system for mapping the URL of incoming requests to the controller function that should be called to process the request. It both allows you to specify beautiful URLs and keeps the functionality of your application decoupled from those URLs. Routing is a bidirectional mechanism, meaning that it should also be used to generate URLs.

---

2. https://github.com/schmittjoh/JMSI18nRoutingBundle
3. https://github.com/BeSimple/BeSimpleI18nRoutingBundle

Keep Going!

Routing, check! Now, uncover the power of *controllers*.

Learn more about Routing

- How to Restrict Route Matching through Conditions
- How to Create a custom Route Loader
- How to Visualize And Debug Routes
- How to Include External Routing Resources
- How to Pass Extra Information from a Route to a Controller
- How to Generate Routing URLs in JavaScript
- How to Match a Route Based on the Host
- How to Define Optional Placeholders
- How to Configure a Redirect without a custom Controller
- Redirect URLs with a Trailing Slash
- How to Define Route Requirements
- Looking up Routes from a Database: Symfony CMF DynamicRouter
- How to Force Routes to Always Use HTTPS or HTTP
- How to Use Service Container Parameters in your Routes
- How to Allow a "/" Character in a Route Parameter

Chapter 4

Controller

A controller is a PHP function you create that reads information from the Symfony's `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 you can dream up. The controller executes whatever arbitrary logic *your application* needs to render the content of a page.

See how simple this is by looking at a Symfony controller in action. This renders a page that prints a lucky (random) number:

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

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

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

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

But in the real world, your controller will probably do a lot of work in order to create the response. It might read information from the request, load a database resource, send an email or set information on the user's session. But in all cases, the controller will eventually return the `Response` object that will be delivered back to the client.

> 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 (a function, method on an object, or a `Closure`), a controller is usually a method inside a controller class:

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

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

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

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

The controller is the `numberAction()` method, which lives inside a 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 should end in the word `Controller` (this isn't *required*, but some shortcuts rely on this).
- *line 12*: Each action method in a controller class is suffixed with `Action` (again, this isn't *required*, but some shortcuts rely on this). This 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}")` 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 Classes & Services

For convenience, Symfony comes with two optional base *Controller*[1] and *AbstractController*[2]

---

1. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php

classes. You can extend either to get access to a number of helper methods.

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

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

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class LuckyController extends Controller
{
    // ...
}
```

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

> You can extend either `Controller` or `AbstractController`. The difference is that when you extend `AbstractController`, you can't access your services via `$this->get()` or `$this->container->get()`, only to a set of common Symfony services. This forces you to write more robust code to access services.
>
> Moreover, in Symfony 4.2 `Controller` was deprecated in favor of `AbstractController`, so using the latter will make your applications future-proof.

New in version 3.3: The `AbstractController` class was introduced in Symfony 3.3.

Generating URLs

The **generateUrl()**[3] method is just a helper method that generates the URL for a given route:

```php
$url = $this->generateUrl('blog_show', ['slug' => 'slug-value']);
```

Redirecting

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

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

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

    // redirects to a route with parameters
    return $this->redirectToRoute('blog_show', ['slug' => 'my-page']);

    // 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');
}
```

For more information, see the *Routing article*.

---

2. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php

3. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php

> ⚠️ The `redirect()` method does not check its destination in any way. If you redirect to some URL provided by the end-users, your application may be open to the *unvalidated redirects security vulnerability*[4].

> 💡 The `redirectToRoute()` method is simply a shortcut that creates a `Response` object that specializes in redirecting the user. It's equivalent to:

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

public function indexAction()
{
    return new RedirectResponse($this->generateUrl('homepage'));
}
```

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:

```php
// renders app/Resources/views/lucky/number.html.twig
return $this->render('lucky/number.html.twig', ['number' => $number]);
```

Templates can also live in deeper sub-directories. Just try to avoid creating unnecessarily deep structures:

```php
// renders app/Resources/views/lottery/lucky/number.html.twig
return $this->render('lottery/lucky/number.html.twig', [
    'number' => $number,
]);
```

The Symfony templating system and Twig are explained more in the *Creating and Using Templates article*.

Fetching Services as Controller Arguments

New in version 3.3: The ability to type-hint a controller argument in order to receive a service was introduced in Symfony 3.3.

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

```php
use Psr\Log\LoggerInterface;
// ...

/**
 * @Route("/lucky/number/{max}")
 */
public function numberAction($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:

---

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

```
1   $ php bin/console debug:autowiring
```

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

```yaml
1   # app/config/services.yml
2   services:
3       # ...
4
5       # explicitly configure the service
6       AppBundle\Controller\LuckyController:
7           public: true
8           bind:
9               # for any $logger argument, pass this specific service
10              $logger: '@monolog.logger.doctrine'
```

You can also use normal constructor injection in your controllers.

> ⚠ You can *only* pass *services* to your controller arguments in this way. It's not possible, for example, to pass a service parameter as a controller argument, even by using `bind`. If you need a parameter, use the `$this->getParameter('kernel.debug')` shortcut or pass the value through your controller's `__construct()` method and specify its value with `bind`.

For more information about services, see the *Service Container* article.

> ✎ If this isn't working, make sure your controller is registered as a service, is autoconfigured and extends either *Controller*[5] or *AbstractController*[6]. If you use the services.yml configuration from the Symfony Standard Edition, then your controllers are already registered as services and autoconfigured.
>
> If you're not using the default configuration, you can tag your service manually with `controller.service_arguments`.

Accessing the Container Directly

If you extend the base `Controller` class, you can access any Symfony service via the *get()*[7] method. Here are several common services you might need:

```php
1   $templating = $this->get('templating');
2
3   $router = $this->get('router');
4
5   $mailer = $this->get('mailer');
6
7   // you can also fetch parameters
8   $someParameter = $this->getParameter('some_parameter');
```

If you receive an error like:

```
1   You have requested a non-existent service "my_service_id"
```

Check to make sure the service exists (use debug:container) and that it's public.

---

5. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php

6. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php

7. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php

Managing Errors and 404 Pages

When things are not found, you should play well with the HTTP protocol and return a 404 response. To do this, you'll throw a special type of exception. If you're extending the base `Controller` or the base `AbstractController` class, do the following:

```
 1  public function indexAction()
 2  {
 3      // retrieve the object from database
 4      $product = ...;
 5      if (!$product) {
 6          throw $this->createNotFoundException('The product does not exist');
 7      }
 8
 9      return $this->render(...);
10  }
```

The *createNotFoundException()*[8] method is just a shortcut to create a special *NotFoundHttpException*[9] object, which ultimately triggers a 404 HTTP response inside Symfony.

If you throw an exception that extends or is an instance of *HttpException*[10], Symfony will use the appropriate HTTP status code. Otherwise, the response will have a 500 HTTP status code:

```
// 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 using the `app_dev.php` front controller - see The imports Key: Loading other Configuration Files).

You'll want to customize the error page your user sees. To do that, 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? All of that information is stored in Symfony's `Request` object. To get it in your controller, just add it as an argument and **type-hint it with the Request class**:

```
 1  use Symfony\Component\HttpFoundation\Request;
 2
 3  public function indexAction(Request $request, $firstName, $lastName)
 4  {
 5      $page = $request->query->get('page', 1);
 6
 7      // ...
 8  }
```

Keep reading for more information about using the Request object.

Managing the Session

Symfony provides a nice session object that you can use to store information about the user between requests. By default, Symfony stores the token in a cookie and writes the attributes to a file by using

---

8. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php

9. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Component/HttpKernel/Exception/NotFoundHttpException.php

10. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Component/HttpKernel/Exception/HttpException.php

native PHP sessions.

New in version 3.3: The ability to request a `Session` instance in controllers was introduced in Symfony 3.3.

To retrieve the session, add the *SessionInterface*[11] type-hint to your argument and Symfony will provide you with a session:

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

public function indexAction(SessionInterface $session)
{
    // 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.

> Every `SessionInterface` implementation is supported. If you have your own implementation, type-hint this in the arguments instead.

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:

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

public function updateAction(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()`:

---

11. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Component/HttpFoundation/Session/SessionInterface.php

Listing 4-19

```twig
{# app/Resources/views/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 %}

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

New in version 3.3: The `app.flashes()` Twig function was introduced in Symfony 3.3. Prior, you had to use `app.session.flashBag()`.

> 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()*[12] method instead to retrieve the message while keeping it in the bag.

The Request and Response Object

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

Listing 4-20

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

public function indexAction(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
```

---

12. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.php

```
17        $request->files->get('foo');
18
19        // retrieves a COOKIE value
20        $request->cookies->get('PHPSESSID');
21
22        // retrieves an HTTP request header, with normalized, lowercase keys
23        $request->headers->get('host');
24        $request->headers->get('content-type');
25 }
```

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 also a public **headers** property. This is a *ResponseHeaderBag*[13] that has some nice methods for getting and setting response headers. The header names are normalized so that using **Content-Type** is equivalent to **content-type** or even **content_type**.

The only requirement for a controller is to return a **Response** object. The *Response*[14] class is an abstraction around the HTTP response - the text-based message filled with headers and content that's sent back to the client:

```
1 use Symfony\Component\HttpFoundation\Response;
2
3 // creates a simple Response with a 200 status code (the default)
4 $response = new Response('Hello '.$name, Response::HTTP_OK);
5
6 // creates a CSS-response with a 200 status code
7 $response = new Response('<style> ... </style>');
8 $response->headers->set('Content-Type', 'text/css');
```

There are special classes that make certain kinds of responses easier:

- For files, there is *BinaryFileResponse*[15]. See Serving Files.
- For streamed responses, there is *StreamedResponse*[16]. See Streaming a Response.

Now that you know the basics you can continue your research on Symfony Request and Response object in the HttpFoundation component documentation.

JSON Helper

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

```
1 // ...
2 public function indexAction()
3 {
4     // returns '{"username":"jane.doe"}' and sets the proper Content-Type header
5     return $this->json(['username' => 'jane.doe']);
6
7     // the shortcut defines three optional arguments
8     // return $this->json($data, $status = 200, $headers = [], $context = []);
9 }
```

If the *serializer service* is enabled in your application, contents passed to **json()** are encoded with it. Otherwise, the *json_encode*[17] function is used.

---

13. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Component/HttpFoundation/ResponseHeaderBag.php

14. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Component/HttpFoundation/Response.php

15. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Component/HttpFoundation/BinaryFileResponse.php

16. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Component/HttpFoundation/StreamedResponse.php

File helper

New in version 3.2: The `file()` helper was introduced in Symfony 3.2.

You can use the *`file()`*[18] helper to serve a file from inside a controller:

```php
public function fileAction()
{
    // 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:

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

public function fileAction()
{
    // 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

Whenever you create a page, you'll ultimately need to write some code that contains the logic for that page. In Symfony, this is called a controller, and it's a PHP function where you can do anything in order to return the final `Response` object that will be returned to the user.

To make life easier, you'll probably extend the base `Controller` class because this gives access to shortcut methods (like `render()` and `redirectToRoute()`).

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 Manually Validate a CSRF Token in a Controller
- How to Customize Error Pages
- How to Forward Requests to another Controller
- How to Define Controllers as Services

---

17. https://secure.php.net/manual/en/function.json-encode.php

18. https://github.com/symfony/symfony/blob/3.4/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php

- 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 simply 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:

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <title>Welcome to Symfony!</title>
5       </head>
6       <body>
7           <h1><?= $page_title ?></h1>
8
9           <ul id="navigation">
10              <?php foreach ($navigation as $item): ?>
11                  <li>
12                      <a href="<?= $item->getHref() ?>">
13                          <?= $item->getCaption() ?>
14                      </a>
15                  </li>
16              <?php endforeach ?>
17          </ul>
18      </body>
19  </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:

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <title>Welcome to Symfony!</title>
5       </head>
6       <body>
7           <h1>{{ page_title }}</h1>
8
9           <ul id="navigation">
10              {% for item in navigation %}
11                  <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
12              {% endfor %}
13          </ul>
14      </body>
15  </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:

```
1   {{ 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*.

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:

```
1   {% for i in 1..10 %}
2       <div class="{{ cycle(['even', 'odd'], i) }}">
3           <!-- some HTML here -->
4       </div>
5   {% 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 little features that make writing templates easier and more concise. Take the following example, which combines a loop with a logical `if` statement:

```
1   <ul>
2       {% for user in users if user.active %}
3           <li>{{ user.username }}</li>
4       {% else %}
5           <li>No users found</li>
6       {% endfor %}
7   </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 easy 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:

```
1   {# app/Resources/views/base.html.twig #}
2   <!DOCTYPE html>
3   <html>
4       <head>
5           <meta charset="UTF-8">
6           <title>{% block title %}Test Application{% endblock %}</title>
7       </head>
8       <body>
9           <div id="sidebar">
10              {% block sidebar %}
11                  <ul>
12                      <li><a href="/">Home</a></li>
13                      <li><a href="/blog">Blog</a></li>
14                  </ul>
15              {% endblock %}
16          </div>
17
18          <div id="content">
19              {% block body %}{% endblock %}
20          </div>
21      </body>
22  </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 simple 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 simply retain the default values used in this template.

A child template might look like this:

```
1   {# app/Resources/views/blog/index.html.twig #}
2   {% extends 'base.html.twig' %}
3
4   {% block title %}My cool blog posts{% endblock %}
5
6   {% block body %}
7       {% for entry in blog_entries %}
8           <h2>{{ entry.title }}</h2>
9           <p>{{ entry.body }}</p>
10      {% endfor %}
11  {% endblock %}
```

The parent template is stored in `app/Resources/views/`, so its path is simply `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:

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset="UTF-8">
5           <title>My cool blog posts</title>
6       </head>
7       <body>
8           <div id="sidebar">
9               <ul>
10                  <li><a href="/">Home</a></li>
11                  <li><a href="/blog">Blog</a></li>
12              </ul>
13          </div>
14
15          <div id="content">
16              <h2>My first post</h2>
17              <p>The body of the first post.</p>
18
19              <h2>Another post</h2>
20              <p>The body of the second post.</p>
21          </div>
22      </body>
23  </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 5-9
```
1  {% block sidebar %}
2      <h3>Table of Contents</h3>
3
4      {# ... #}
5
6      {{ parent() }}
7  {% endblock %}
```

Template Naming and Locations

By default, templates can live in two different locations:

`app/Resources/views/`
> 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 (see *How to Override Templates from Third-Party Bundles*).

`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 `app/` directory.

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

In the *How to Override Templates from Third-Party Bundles* section, you'll find out how each template living inside the AcmeBlogBundle, for example, can be overridden by placing a template of the same name in the **app/Resources/AcmeBlogBundle/views/** directory. This gives the power to override templates from any vendor 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 simply 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.

```
1   {# app/Resources/views/article/article_details.html.twig #}
2   <h2>{{ article.title }}</h2>
3   <h3 class="byline">by {{ article.authorName }}</h3>
4
5   <p>
6       {{ article.body }}
7   </p>
```

Including this template from any other template is simple:

```
1    {# app/Resources/views/article/list.html.twig #}
2    {% extends 'layout.html.twig' %}
3
4    {% block body %}
5        <h1>Recent Articles<h1>
6
7        {% for article in articles %}
8            {{ include('article/article_details.html.twig', { 'article': article }) }}
9        {% endfor %}
10   {% endblock %}
```

The template is included using the `{{ include() }}` function. 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:

```
1    // src/AppBundle/Controller/WelcomeController.php
2
3    // ...
4    use Symfony\Component\Routing\Annotation\Route;
5
6    class WelcomeController extends Controller
7    {
8        /**
9         * @Route("/", name="welcome")
10        */
11       public function indexAction()
12       {
13           // ...
14       }
```

---

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

```
15  }
```

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

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

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

```
1  // src/AppBundle/Controller/ArticleController.php
2
3  // ...
4  use Symfony\Component\Routing\Annotation\Route;
5
6  class ArticleController extends Controller
7  {
8      /**
9       * @Route("/article/{slug}", name="article_show")
10      */
11     public function showAction($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:

```
1  {# app/Resources/views/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:

```
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 path to these assets (e.g. **/images/logo.png**), but Symfony provides a more dynamic option via the **asset()** Twig function:

```
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:

```
1   <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. Symfony is also compatible with another library, called Assetic, which follows this philosophy but allows you to do much more interesting things with those assets. For more information on using Assetic see *How to Use Assetic for Asset Management*.

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:

```
1   {# app/Resources/views/base.html.twig #}
2   <html>
3       <head>
4           {# ... #}
5
6           {% block stylesheets %}
7               <link href="{{ asset('css/main.css') }}" rel="stylesheet"/>
8           {% endblock %}
9       </head>
10      <body>
11          {# ... #}
12
13          {% block javascripts %}
14              <script src="{{ asset('js/main.js') }}"></script>
15          {% endblock %}
16      </body>
17  </html>
```

That's easy enough! But what if 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:

```
1   {# app/Resources/views/contact/contact.html.twig #}
2   {% extends 'base.html.twig' %}
3
4   {% block stylesheets %}
5       {{ parent() }}
6
7       <link href="{{ asset('css/contact.css') }}" rel="stylesheet"/>
8   {% endblock %}
9
10  {# ... #}
```

In the child template, you simply 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

should 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 `web/` directory of your application.)

```
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`:

```
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 Override Templates from Third-Party Bundles
- 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 (and Environments)

Every Symfony application consists of a collection of bundles that add useful tools (*services*) to your project. Each bundle can be customized via configuration files that live - by default - in the `app/config` directory.

Configuration: config.yml

The main configuration file is called `config.yml`:

```
1  # app/config/config.yml
2  imports:
3      - { resource: parameters.yml }
4      - { resource: security.yml }
5      - { resource: services.yml }
6
7  framework:
8      secret:          '%secret%'
9      router:          { resource: '%kernel.project_dir%/app/config/routing.yml' }
10     # ...
11
12 # Twig Configuration
13 twig:
14     debug:           '%kernel.debug%'
15     strict_variables: '%kernel.debug%'
16
17 # ...
```

Most top-level keys - like `framework` and `twig` - are configuration for a specific bundle (i.e. `FrameworkBundle` and `TwigBundle`).

Configuration Formats

Throughout the documentation, all configuration examples will be shown in three formats (YAML, XML and PHP). YAML is used by default, but you can choose whatever you like best. There is no performance difference:

- *The YAML Format*: Simple, clean and readable;
- *XML*: More powerful than YAML at times & supports IDE autocompletion;
- *PHP*: Very powerful but less readable than standard configuration formats.

Configuration Reference & Dumping

There are *two* ways to know *what* keys you can configure:

1. Use the *Reference Section*;
2. Use the `config:dump-reference` command.

For example, if you want to configure something in Twig, you can see an example dump of all available configuration options by running:

```
1  $ php bin/console config:dump-reference twig
```

The imports Key: Loading other Configuration Files

Symfony's main configuration file is `app/config/config.yml`. But, for organization, it *also* loads other configuration files via its `imports` key:

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

The `imports` key works a lot like the PHP `include()` function: the contents of `parameters.yml`, `security.yml` and `services.yml` are read and loaded. You can also load XML files or PHP files.

If your application uses unconventional file extensions (for example, your YAML files have a `.res` extension) you can set the file type explicitly with the `type` option:

```
1  # app/config/config.yml
2  imports:
3      - { resource: parameters.res, type: yml }
4      # ...
```

The parameters Key: Parameters (Variables)

Another special key is called `parameters`: it's used to define *variables* that can be referenced in *any* other configuration file. For example, in `config.yml`, a `locale` parameter is defined and then

referenced below under the `framework` key:

```
1  # app/config/config.yml
2  # ...
3
4  parameters:
5      locale: en
6
7  framework:
8      # ...
9
10     # any string surrounded by two % is replaced by that parameter value
11     default_locale:  "%locale%"
12
13 # ...
```

You can define whatever parameter names you want under the `parameters` key of any configuration file. To reference a parameter, surround its name with two percent signs - e.g. `%locale%`.

You can also set parameters dynamically, like from environment variables. See How to Set external Parameters in the Service Container.

For more information about parameters - including how to reference them from inside a controller - see Service Parameters.

The Special parameters.yml File

On the surface, `parameters.yml` is just like any other configuration file: it is imported by `config.yml` and defines several parameters:

```
1  parameters:
2      # ...
3      database_user:      root
4      database_password:  ~
```

Not surprisingly, these are referenced from inside of `config.yml` and help to configure DoctrineBundle and other parts of Symfony:

```
1  # app/config/config.yml
2  doctrine:
3      dbal:
4          driver:    pdo_mysql
5          # ...
6          user:     '%database_user%'
7          password: '%database_password%'
```

But the `parameters.yml` file *is* special: it defines the values that usually change on each server. For example, the database credentials on your local development machine might be different from your workmates. That's why this file is not committed to the shared repository and is only stored on your machine.

Because of that, **parameters.yml is not committed to your version control**. In fact, the `.gitignore` file that comes with Symfony prevents it from being committed.

However, a `parameters.yml.dist` file *is* committed (with dummy values). This file isn't read by Symfony: it's just a reference so that Symfony knows which parameters need to be defined in the `parameters.yml` file. If you add or remove keys to `parameters.yml`, add or remove them from `parameters.yml.dist` too so both files are always in sync.

The Interactive Parameter Handler

When you install an existing Symfony project, you will need to create the `parameters.yml` file using the committed `parameters.yml.dist` file as a reference. To help with this, after you run `composer install`, a Symfony script will automatically create this file by interactively asking you to supply the value for each parameter defined in `parameters.yml.dist`. For more details - or to remove or control this behavior - see the *Incenteev Parameter Handler*[1] documentation.

Environments & the Other Config Files

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

- While **developing**, you want your app to log everything and expose nice debugging tools;
- After deploying to **production**, you want that *same* app to be optimized for speed and only log errors.

How can you make *one* application behave in two different ways? With *environments*.

You've probably already been using the `dev` environment without even knowing it. After you deploy, you'll use the `prod` environment.

To learn more about *how* to execute and control each environment, see *How to Master and Create new Environments*.

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*
- *How to Send an Email*
- *Logging*

And the many other topics.

Learn more

- How to Organize Configuration Files
- How to Master and Create new Environments
- How to Set external Parameters in the Service Container
- 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

---

1. `https://github.com/Incenteev/ParameterHandler`