



Symfony

Getting Started

Version: 4.3

generated on October 21, 2019

Getting Started (4.3)

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

Installing & Setting up the Symfony Framework	4
Create your First Page in Symfony	9
Routing	14
Controller.....	33



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 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

```
1 # run this if you are building a traditional web application
2 $ symfony new my_project_name --full
3
4 # run this if you are building a microservice, console application or API
5 $ 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

```
1 # run this if you are building a traditional web application
2 $ composer create-project symfony/website-skeleton my_project_name
3
4 # run this if you are building a microservice, console application or API
5 $ 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

```
1 $ cd my-project/
2 $ 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

```
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/
7 $ 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 an existing Symfony application for the first time, it may be useful to run this command which displays information about the project:

Listing 1-6

```
1 $ 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

```
1 $ cd my-project/
2 $ 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.

Symfony Packs

Sometimes a single feature requires installing several packages and bundles. Instead of installing them individually, Symfony provides **packs**, which are Composer metapackages that include several dependencies.

For example, to add debugging features in your application, you can run the `composer require --dev debug` command. This installs the `symfony/debug-pack`, which in turn installs several packages like `symfony/debug-bundle`, `symfony/monolog-bundle`, `symfony/var-dumper`, etc.

By default, when installing Symfony packs, your `composer.json` file shows the pack dependency (e.g. `"symfony/debug-pack": "^1.0"`) instead of the actual packages installed. To show the packages, add the `--unpack` option when installing a pack (e.g. `composer require debug --dev --unpack`) or run this command to unpack the already installed packs: `composer unpack PACK_NAME` (e.g. `composer unpack debug`).

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 1 `$ 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

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>

```

1 # use the most recent 'lts' version
2 $ symfony new my_project_name --version=lts
3
4 # use the 'next' Symfony version to be released (still in development)
5 $ symfony new my_project_name --version=next
6
7 # use a specific Symfony version
8 $ symfony new my_project_name --version=3.3.10
9 $ symfony new my_project_name --version=4.3.1

```

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 1 \$ symfony new my_project_name --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
- How to Use PHP's built-in 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)

19. <https://github.com/symfony/demo>



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

```
1 <?php
2 // src/Controller/LuckyController.php
3 namespace App\Controller;
4
5 use Symfony\Component\HttpFoundation\Response;
6
7 class LuckyController
8 {
9     public function number()
10     {
11         $number = random_int(0, 100);
12     }
13 }
```

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 Symfony pack 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

```
1 // src/Controller/LuckyController.php
2
3 // ...
4 + use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5
6 - class LuckyController
7 + class LuckyController extends AbstractController
8 {
9     // ...
10 }
```

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

```
1 // src/Controller/LuckyController.php
2 namespace App\Controller;
3
4 // ...
5 class LuckyController extends AbstractController
6 {
7     /**
8      * @Route("/lucky/number")
9      */
10    public function number()
11    {
12        $number = random_int(0, 100);
13
14        return $this->render('lucky/number.html.twig', [
15            'number' => $number,
16        ]);
17    }
18 }
```

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

```
1 {# templates/lucky/number.html.twig #}
2 <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 *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/4.3/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.

Creating Routes as Annotations

Run this command once in your application to add support for annotations:

Listing 3-1

```
1 $ composer require annotations
```

In addition to installing the needed dependencies, this command creates the following configuration file:

Listing 3-2

```
1 # config/routes.yaml
2 controllers:
3     resource: '../src/Controller/'
4     type:     annotation
```

This configuration tells Symfony to look for routes defined as annotations in any PHP class stored in the `src/Controller/` directory.

Suppose you want to define a route for the `/blog` URL in your application. To do so, create a *controller class* like the following:

Listing 3-3

```
1 // src/Controller/BlogController.php
2 namespace App\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
```

```

5 use Symfony\Component\Routing\Annotation\Route;
6
7 class BlogController extends AbstractController
8 {
9     /**
10      * @Route("/blog", name="blog_list")
11      */
12     public function list()
13     {
14         // ...
15     }
16 }

```

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.

Creating Routes in YAML, XML or PHP Files

Instead of defining routes in the controller classes, you can define them in a separate YAML, XML or PHP file. The main advantage is that they don't require any extra dependency. The main drawback is that you have to work with multiple files when checking the routing of some controller action.

The following example shows how to define in YAML/XML/PHP a route called **blog_list** that associates the **/blog** URL with the **list()** action of the **BlogController**:

Listing 3-4

```

1 # config/routes.yaml
2 blog_list:
3     path: /blog
4     # the controller value has the format 'controller_class::method_name'
5     controller: App\Controller\BlogController::list
6
7     # if the action is implemented as the __invoke() method of the
8     # controller class, you can skip the '::method_name' part:
9     # controller: App\Controller\BlogController

```

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 3-5

```

1 // src/Controller/BlogApiController.php
2 namespace App\Controller;
3
4 // ...
5
6 class BlogApiController extends AbstractController
7 {
8     /**
9      * @Route("/api/posts/{id}", methods={"GET", "HEAD"})
10     */
11     public function show(int $id)
12     {
13         // ... return a JSON response with the post
14     }
15 }

```

```

15
16     /**
17      * @Route("/api/posts/{id}", methods={"PUT"})
18      */
19     public function edit(int $id)
20     {
21         // ... edit a post
22     }
23 }

```



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 3-6

```

1  // src/Controller/DefaultController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\Routing\Annotation\Route;
6
7  class DefaultController extends AbstractController
8  {
9      /**
10       * @Route(
11       *     "/contact",
12       *     name="contact",
13       *     condition="context.getMethod() in ['GET', 'HEAD'] and request.headers.get('User-Agent') matches
14       *     '/firefox/i'"
15       * )
16       *
17       * expressions can also include config parameters:
18       * condition: "request.headers.get('User-Agent') matches '%app.allowed_browsers%'"
19       */
20     public function contact()
21     {
22         // ...
23     }
24 }

```

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.

1. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Routing/RequestContext.php>



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 3-7

```
1 $ php bin/console debug:router
2
3 -----
4 Name          Method  Scheme  Host  Path
5 -----
6 homepage      ANY     ANY     ANY   /
7 contact       GET     ANY     ANY   /contact
8 contact_process POST    ANY     ANY   /contact
9 article_show  ANY     ANY     ANY   /articles/{_locale}/{year}/{title}.{_format}
10 blog         ANY     ANY     ANY   /blog/{page}
11 blog_show    ANY     ANY     ANY   /blog/{slug}
12 -----
```

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

Listing 3-8

```
1 $ php bin/console debug:router app_lucky_number
2
3 +-----+
4 | Property | Value |
5 +-----+
6 | Route Name | app_lucky_number |
7 | Path      | /lucky/number/{max} |
8 | ...      | ... |
9 | Options   | compiler_class: Symfony\Component\Routing\RouteCompiler |
10 |          | utf8: true |
11 +-----+
```

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 3-9

```
1 $ php bin/console router:match /lucky/number/8
2
3 [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 3-10

```
1 // src/Controller/BlogController.php
2 namespace App\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5 use Symfony\Component\Routing\Annotation\Route;
6
```

```

7 class BlogController extends AbstractController
8 {
9     // ...
10
11     /**
12      * @Route("/blog/{slug}", name="blog_show")
13      */
14     public function show(string $slug)
15     {
16         // $slug will equal the dynamic part of the URL
17         // e.g. at /blog/yay-routing, then $slug='yay-routing'
18
19         // ...
20     }
21 }

```

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 3-11

```

1 // src/Controller/BlogController.php
2 namespace App\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5 use Symfony\Component\Routing\Annotation\Route;
6
7 class BlogController extends AbstractController
8 {
9     /**
10      * @Route("/blog/{page}", name="blog_list", requirements={"page"="\d+"})
11      */
12     public function list(int $page)
13     {
14         // ...
15     }
16
17     /**
18      * @Route("/blog/{slug}", name="blog_show")
19      */
20     public function show($slug)
21     {
22         // ...
23     }
24 }

```

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 3-12

```

1  // src/Controller/BlogController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\Routing\Annotation\Route;
6
7  class BlogController extends AbstractController
8  {
9      /**
10       * @Route("/blog/{page<\d+}&}", name="blog_list")
11       */
12       public function list(int $page)
13       {
14           // ...
15       }
16  }
```

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 3-13

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

```

1  // src/Controller/BlogController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\Routing\Annotation\Route;
6
7  class BlogController extends AbstractController
8  {
9      /**
10       * @Route("/blog/{page}", name="blog_list", requirements={"page"="\d+"})
11       */
12       public function list(int $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`.



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).

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}`

New in version 4.3: The feature to force the inclusion of default values in generated URLs was introduced in Symfony 4.3.

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 3-14

```

1  // src/Controller/BlogController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\Routing\Annotation\Route;
6
7  class BlogController extends AbstractController
8  {
9      /**
10       * @Route("/blog/{page<\d+>?1}", name="blog_list")
11       */
12       public function list(int $page)
13       {
14           // ...
15       }
16 }

```



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 3-15 1 `$ 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 3-16

```
1 // src/Controller/BlogController.php
2 namespace App\Controller;
3
4 use App\Entity\BlogPost;
5 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6 use Symfony\Component\Routing\Annotation\Route;
7
8 class BlogController extends AbstractController
9 {
10     // ...
11
12     /**
13      * @Route("/blog/{slug}", name="blog_show")
14      */
15     public function show(BlogPost $post)
16     {
17         // $post is the object whose slug matches the routing parameter
18
19         // ...
20     }
21 }
```

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.

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.

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

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 3-17 1 // src/Controller/ArticleController.php
2
3 // ...
4 class ArticleController extends AbstractController
5 {
6     /**
7      * @Route(
8      *     "/articles/{_locale}/search.{_format}",
9      *     locale="en",
10     *     format="html",
11     *     requirements={
12     *         "_locale": "en/fr",
13     *         "_format": "html/xml",
14     *     }
15     * )
16     */
17     public function search()
18     {
19     }
20 }
```

New in version 4.3: The special attributes were introduced in Symfony 4.3.

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 3-18 1 use Symfony\Component\Routing\Annotation\Route;
2
3 class BlogController
4 {
5     /**
6      * @Route("/blog/{page}", name="blog_index", defaults={"page": 1, "title": "Hello world!"})
7      */
8     public function index(int $page, string $title)
9     {
10         // ...
11     }
12 }
```

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 3-19 1 use Symfony\Component\Routing\Annotation\Route;
2
3 class DefaultController
4 {
5     /**
6      * @Route("/share/{token}", name="share", requirements={"token"=".+"})
7      */
8     public function share($token)
9     {
10         // ...
11     }
12 }
```

```
11     }
12 }
```



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 3-20

```
1 use Symfony\Component\Routing\Annotation\Route;
2
3 /**
4  * @Route("/blog", requirements={"_locale": "en|es|fr"}, name="blog_")
5  */
6 class BlogController
7 {
8     /**
9      * @Route("/{_locale}", name="index")
10     */
11     public function index()
12     {
13         // ...
14     }
15
16     /**
17      * @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 3-21

```
1 // src/Controller/BlogController.php
2 namespace App\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5 use Symfony\Component\HttpFoundation\Request;
6 use Symfony\Component\Routing\Annotation\Route;
7
8 class BlogController extends AbstractController
9 {
10     /**
11      * @Route("/blog", name="blog_list")
12      */
13     public function list(Request $request)
14     {
15         // ...
16
17         $routeName = $request->attributes->get('_route');
18         $routeParameters = $request->attributes->get('_route_params');
19
20         // use this to get all the available attributes (not only routing ones):
21         $allAttributes = $request->attributes->all();
22     }
23 }
```

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

Listing 3-22

```
1 {% set route_name = app.request.attributes.get('_route') %}
2 {% set route_parameters = app.request.attributes.get('_route_params') %}
3
4 {# use this to get all the available attributes (not only routing ones) #}
5 {% 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 a Template Directly from a Route

Read the section about rendering a template from a route in the main article about Symfony templates.

Redirecting to URLs and Routes Directly from a Route

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

Listing 3-23

```
1 # config/routes.yaml
2 doc_shortcut:
3     path: /doc
4     controller: Symfony\Bundle\FrameworkBundle\Controller\RedirectController::redirectAction
5     defaults:
6         route: 'doc_page'
```



```

7      # optionally you can define some arguments passed to the route
8      page: 'index'
9      version: 'current'
10     # redirections are temporary by default (code 302) but you can make them permanent (code 301)
11     permanent: true
12     # add this to keep the original query string parameters when redirecting
13     keepQueryParams: true
14     # add this to keep the HTTP method when redirecting. The redirect status changes
15     # * for temporary redirects, it uses the 307 status code instead of 302
16     # * for permanent redirects, it uses the 308 status code instead of 301
17     keepRequestMethod: true
18
19     legacy_doc:
20         path: /legacy/doc
21         controller: Symfony\Bundle\FrameworkBundle\Controller\RedirectController::urlRedirectAction
22         defaults:
23             # this value can be an absolute path or an absolute URL
24             path: 'https://legacy.example.com/doc'
25             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)

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 3-24

```

1  // src/Controller/MainController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\Routing\Annotation\Route;
6
7  class MainController extends AbstractController
8  {
9      /**
10       * @Route("/", name="mobile_homepage", host="m.example.com")
11       */
12     public function mobileHomepage()

```

```

13     {
14         // ...
15     }
16
17     /**
18      * @Route("/", name="homepage")
19      */
20     public function homepage()
21     {
22         // ...
23     }
24 }

```

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 3-25

```

1  // src/Controller/MainController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\Routing\Annotation\Route;
6
7  class MainController extends AbstractController
8  {
9      /**
10       * @Route(
11       *     "/",
12       *     name="mobile_homepage",
13       *     host="{subdomain}.example.com",
14       *     defaults={"subdomain"="m"},
15       *     requirements={"subdomain"="m|mobile"}
16       * )
17       */
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 **subdomain** 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 3-26

```
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 3-27

```
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 3-28

```
1 # config/routes/annotations.yaml
2 controllers:
3     resource: '../src/Controller/'
4     type: annotation
5     prefix:
6         en: '' # don't prefix URLs for English, the default locale
7         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 `` 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 3-29

```
1  // src/Controller/BlogController.php
2  namespace App\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5  use Symfony\Component\Routing\Annotation\Route;
6  use Symfony\Component\Routing\Generator\UrlGeneratorInterface;
7
8  class BlogController extends AbstractController
9  {
10     /**
11      * @Route("/blog", name="blog_list")
12      */
13     public function list()
14     {
15         // ...
16
17         // generate a URL with no route arguments
18         $signUpPage = $this->generateUrl('sign_up');
19
20         // generate a URL with route arguments
21         $userProfilePage = $this->generateUrl('user_profile', [
22             'username' => $user->getUsername(),
23         ]);
24
25         // generated URLs are "absolute paths" by default. Pass a third optional
26         // argument to generate different URLs (e.g. an "absolute URL")
27         $signUpPage = $this->generateUrl('sign_up', [], UrlGeneratorInterface::ABSOLUTE_URL);
28
29         // when a route is localized, Symfony uses by default the current request locale
30         // pass a different '_locale' value if you want to set the locale explicitly
31         $signUpPageInDutch = $this->generateUrl('sign_up', ['_locale' => 'nl']);
32     }
33 }
```



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 3-30

```
$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 3-31

```
1  // src/Service/SomeService.php
2  use Symfony\Component\Routing\Generator\UrlGeneratorInterface;
3
4  class SomeService
5  {
6      private $router;
7
8      public function __construct(UrlGeneratorInterface $router)
9      {
10         $this->router = $router;
11     }
12
13     public function someMethod()
14     {
15         // ...
16
17         // generate a URL with no route arguments
18         $signUpPage = $this->router->generate('sign_up');
19
20         // generate a URL with route arguments
21         $userProfilePage = $this->router->generate('user_profile', [
22             'username' => $user->getUsername(),
23         ]);
24
25         // generated URLs are "absolute paths" by default. Pass a third optional
26         // argument to generate different URLs (e.g. an "absolute URL")
27         $signUpPage = $this->router->generate('sign_up', [], UrlGeneratorInterface::ABSOLUTE_URL);
28
29         // when a route is localized, Symfony uses by default the current request locale
30         // pass a different '_locale' value if you want to set the locale explicitly
31         $signUpPageInDutch = $this->router->generate('sign_up', ['_locale' => 'nl']);
32     }
33 }
```

Generating URLs in Templates

Read the section about creating links between pages in the main article about Symfony templates.

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

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

5. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Routing/Generator/UrlGeneratorInterface.php>

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

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 the "request context" used by commands when they generate URLs. This context can be configured globally for all commands:

Listing 3-33

```
1 # config/services.yaml
2 parameters:
3     router.request_context.host: 'example.org'
4     router.request_context.base_url: 'my/path'
5     asset.request_context.base_path: '%router.request_context.base_url%'
```

This information can be configured per command too:

Listing 3-34

```
1 // src/Command/SomeCommand.php
2 use Symfony\Component\Routing\Generator\UrlGeneratorInterface;
3 use Symfony\Component\Routing\RouterInterface;
4 // ...
5
6 class SomeCommand extends Command
7 {
8     private $router;
9
10    public function __construct(RouterInterface $router)
11    {
12        parent::__construct();
13
14        $this->router = $router;
15    }
16
17    protected function execute(InputInterface $input, OutputInterface $output)
18    {
19        // these values override any global configuration
20        $context = $this->router->getContext();
21        $context->setHost('example.com');
22        $context->setBaseUrl('my/path');
23
24        // generate a URL with no route arguments
25        $signUpPage = $this->router->generate('sign_up');
26
27        // generate a URL with route arguments
28        $userProfilePage = $this->router->generate('user_profile', [
29            'username' => $user->getUsername(),
30        ]);
31
32        // generated URLs are "absolute paths" by default. Pass a third optional
33        // argument to generate different URLs (e.g. an "absolute URL")
34        $signUpPage = $this->router->generate('sign_up', [], UrlGeneratorInterface::ABSOLUTE_URL);
35
36        // when a route is localized, Symfony uses by default the current request locale
37        // pass a different '_locale' value if you want to set the locale explicitly
38        $signUpPageInDutch = $this->router->generate('sign_up', ['_locale' => 'nl']);
39
40        // ...
41    }
42 }
```

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

```
1 use Symfony\Component\Routing\Exception\RouteNotFoundException;
2
3 // ...
4
5 try {
6     $url = $this->router->generate($routeName, $routeParameters);
7 } catch (RouteNotFoundException $e) {
8     // the route is not defined...
9 }
```

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:

Listing 3-36

```
1 # config/services.yaml
2 parameters:
3     router.request_context.scheme: 'https'
4     asset.request_context.secure: true
```

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

Listing 3-37

```
1 // src/Controller/MainController.php
2 namespace App\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5 use Symfony\Component\Routing\Annotation\Route;
6
7 class SecurityController extends AbstractController
8 {
9     /**
10      * @Route("/login", name="login", schemes={"https"})
11      */
12     public function login()
13     {
14         // ...
15     }
16 }
```

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

```
1 {# if the current scheme is HTTPS, generates a relative URL: /login #}
2 {{ path('login') }}
3
4 {# if the current scheme is HTTP, generates an absolute URL to change
5    the scheme: https://example.com/login #}
6 {{ path('login') }}
```

7. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Routing/Router.php>

8. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/Routing/Exception/RouteNotFoundException.php>

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

```
1 # config/routes/annotations.yaml
2 controllers:
3     resource: '../src/Controller/'
4     type: annotation
5     defaults:
6         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

```
1  // src/Controller/LuckyController.php
2  namespace App\Controller;
3
4  use Symfony\Component\HttpFoundation\Response;
5  use Symfony\Component\Routing\Annotation\Route;
6
7  class LuckyController
8  {
9      /**
10       * @Route("/lucky/number/{max}", name="app_lucky_number")
11       */
12       public function number($max)
13       {
14           $number = random_int(0, $max);
15
16           return new Response(
17               '<html><body>Lucky number: '.$number.'</body></html>'
18           );
19     }
20 }
```

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

```
1 // src/Controller/LuckyController.php
2 namespace App\Controller;
3
4 + use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
5
6 - class LuckyController
7 + class LuckyController extends AbstractController
8 {
9     // ...
10 }
```

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/4.3/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php>
2. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php>

Listing 4-4

```

1 use Symfony\Component\HttpFoundation\RedirectResponse;
2
3 // ...
4 public function index()
5 {
6     // redirects to the "homepage" route
7     return $this->redirectToRoute('homepage');
8
9     // redirectToRoute is a shortcut for:
10    // return new RedirectResponse($this->generateUrl('homepage'));
11
12    // does a permanent - 301 redirect
13    return $this->redirectToRoute('homepage', [], 301);
14
15    // redirect to a route with parameters
16    return $this->redirectToRoute('app_lucky_number', ['max' => 10]);
17
18    // redirects to a route and maintains the original query string parameters
19    return $this->redirectToRoute('blog_show', $request->query->all());
20
21    // redirects externally
22    return $this->redirect('http://symfony.com/doc');
23 }

```



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

```

// 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

```

1 use Psr\Log\LoggerInterface;
2 // ...
3
4 /**
5  * @Route("/lucky/number/{max}")
6  */
7 public function number($max, LoggerInterface $logger)
8 {
9     $logger->info('We are logging!');
10    // ...
11 }

```

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 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:

Listing 4-8

```
1 # config/services.yaml
2 services:
3     # ...
4
5     # explicitly configure the service
6     App\Controller\LuckyController:
7         public: true
8         bind:
9             # for any $logger argument, pass this specific service
10            $logger: '@monolog.logger.doctrine'
11            # for any $projectDir argument, pass this parameter value
12            $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

```
1 $ php bin/console make:controller BrandNewController
2
3 created: src/Controller/BrandNewController.php
4 created: templates/brandnew/index.html.twig
```

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

Listing 4-10

```
1 $ php bin/console make:crud Product
2
3 created: src/Controller/ProductController.php
4 created: src/Form/ProductType.php
5 created: templates/product/_delete_form.html.twig
6 created: templates/product/_form.html.twig
7 created: templates/product/edit.html.twig
8 created: templates/product/index.html.twig
9 created: templates/product/new.html.twig
10 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

```
1 use Symfony\Component\HttpFoundation\Exception\NotFoundHttpException;
2
3 // ...
4 public function index()
5 {
6     // retrieve the object from database
7     $product = ...;
8     if (!$product) {
```

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

```

9         throw $this->createNotFoundException('The product does not exist');
10
11         // the above is just a shortcut for:
12         // throw new NotFoundException('The product does not exist');
13     }
14
15     return $this->render(...);
16 }

```

The `createNotFoundException()`⁵ method is just a shortcut to create a special `NotFoundException`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

```

1 use Symfony\Component\HttpFoundation\Request;
2
3 public function index(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 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

```

1 use Symfony\Component\HttpFoundation\Session\SessionInterface;
2
3 public function index(SessionInterface $session)
4 {

```

5. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php>

6. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Exception/NotFoundException.php>

7. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpKernel/Exception/HttpException.php>

8. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/Session/SessionInterface.php>

```

5      // stores an attribute for reuse during a later user request
6      $session->set('foo', 'bar');
7
8      // gets the attribute set by another controller in another request
9      $foobar = $session->get('foobar');
10
11     // uses a default value if the attribute doesn't exist
12     $filters = $session->get('filters', []);
13 }

```

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

```

1 use Symfony\Component\HttpFoundation\Request;
2
3 public function update(Request $request)
4 {
5     // ...
6
7     if ($form->isSubmitted() && $form->isValid()) {
8         // do some sort of processing
9
10        $this->addFlash(
11            'notice',
12            'Your changes were saved!'
13        );
14        // $this->addFlash() is equivalent to $request->getSession()->getFlashBag()->add()
15
16        return $this->redirectToRoute(...);
17    }
18
19    return $this->render(...);
20 }

```

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 the **flashes()** method provided by the Twig global app variable:

Listing 4-16

```

1 {# templates/base.html.twig #}
2
3 {# read and display just one flash message type #}
4 {% for message in app.flashes('notice') %}
5     <div class="flash-notice">
6         {{ message }}
7     </div>
8 {% endfor %}
9
10 {# read and display several types of flash messages #}
11 {% for label, messages in app.flashes(['success', 'warning']) %}
12     {% for message in messages %}
13         <div class="flash-{{ label }}">
14             {{ message }}
15         </div>
16     {% endfor %}
17 {% endfor %}

```

```

18
19 {# read and display all flash messages #}
20 {% for label, messages in app.flashes %}
21     {% for message in messages %}
22         <div class="flash-{{ label }}">
23             {{ message }}
24         </div>
25     {% endfor %}
26 {% 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

```

1 use Symfony\Component\HttpFoundation\Request;
2
3 public function index(Request $request)
4 {
5     $request->isXmlHttpRequest(); // is it an Ajax request?
6
7     $request->getPreferredLanguage(['en', 'fr']);
8
9     // retrieves GET and POST variables respectively
10    $request->query->get('page');
11    $request->request->get('page');
12
13    // retrieves SERVER variables
14    $request->server->get('HTTP_HOST');
15
16    // retrieves an instance of UploadedFile identified by foo
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 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/4.3/src/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.php>

10. <https://github.com/symfony/symfony/blob/4.3/src/Symfony/Component/HttpFoundation/ResponseHeaderBag.php>

Listing 4-18

```

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');
```

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

```

1 // ...
2 public function index()
3 {
4     $contentsDir = $this->getParameter('kernel.project_dir').'/contents';
5     // ...
6 }
```

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

```

1 // ...
2 public function index()
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, 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

```

1 public function download()
2 {
3     // send the file contents and force the browser to download it
4     return $this->file('/path/to/some_file.pdf');
5 }
```

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/4.3/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php>


```

1 use Symfony\Component\HttpFoundation\File\File;
2 use Symfony\Component\HttpFoundation\ResponseHeaderBag;
3
4 public function download()
5 {
6     // load the file from the filesystem
7     $file = new File('/path/to/some_file.pdf');
8
9     return $this->file($file);
10
11     // rename the downloaded file
12     return $this->file($file, 'custom_name.pdf');
13
14     // display the file contents in the browser instead of downloading it
15     return $this->file('invoice_3241.pdf', 'my_invoice.pdf', ResponseHeaderBag::DISPOSITION_INLINE);
16 }

```

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

