Symfony

Getting Started

Version: 4.1

generated on January 24, 2019

Getting Started (4.1)

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

Contents at a Glance

Chapter 1
Installing & Setting up the Symfony Framework

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

To create your new Symfony application, first make sure you're using PHP 7.1 or higher and have *Composer*[2] installed. If you don't, start by *installing Composer globally* on your system. If you want to use a virtual machine (VM), check out *Homestead*.

Create your new project by running:

Listing 1-1
```
$ composer create-project symfony/website-skeleton my-project
```

This will create a new `my-project` 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 app is ready!

The `website-skeleton` is optimized for traditional web applications. If you are building microservices, console applications or APIs, consider using the much simpler `skeleton` project:

Listing 1-2
```
$ composer create-project symfony/skeleton my-project

# optional: install the web server bundle (explained next)
$ cd my-project
$ composer require symfony/web-server-bundle --dev
```

Running your Symfony Application

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 convenient to use the *Symfony PHP web server*.

1. *http://symfonycasts.com/screencast/symfony*
2. https://getcomposer.org/

Move into your new project and start the server:

Listing 1-3
```
$ cd my-project
$ php bin/console server:run
```

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.

If you're having any problems running Symfony, your system may be missing some technical requirements. Use the *Symfony Requirements Checker* tool to make sure your system is set up.

If you're using a VM, you may need to tell the server to bind to all IP addresses:

Listing 1-4
```
$ php bin/console server:start 0.0.0.0:8000
```

You should **NEVER** listen to all interfaces on a computer that is directly accessible from the Internet.

Storing your Project in git

Storing your project in services like GitHub, GitLab and Bitbucket works like with any other code project! Init a new repository with `Git` and you are ready to push to your remote:

Listing 1-5
```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

Your project already has a sensible `.gitignore` file. And as you install more packages, a system called Flex will add more lines to that file when needed.

Setting up an Existing Symfony Project

If you're working on an existing Symfony application, 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 1-6
```
# clone the project to download its contents
$ cd projects/
$ git clone ...

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

You'll probably also need to customize your .env and do a few other project-specific tasks (e.g. creating database schema).

Checking for Security Vulnerabilities

Symfony provides a utility called the "Security Checker" to check whether your project's dependencies contain any known security vulnerability. Check out the integration instructions for *the Security Checker*[3] to set it up.

The Symfony Demo application

The Symfony Demo Application[4] 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.

To check out its code and install it locally, see *symfony/symfony-demo*[5].

Start Coding!

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 Use PHP's built-in Web Server
- Configuring a Web Server
- Installing Composer
- 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 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)

3. `https://github.com/sensiolabs/security-checker#integration`
4. `https://github.com/symfony/demo`
5. `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

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

use Symfony\Component\HttpFoundation\Response;

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

```

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

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

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

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

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

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

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

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

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

Annotation Routes

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

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

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

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

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

Auto-Installing Recipes with Symfony Flex

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

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

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

You can learn more about Flex by reading "*Using Symfony Flex to Manage Symfony Applications*". But that's not necessary: Flex works automatically in the background when you add packages.

The bin/console Command

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

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

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

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

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

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

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

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

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

The Web Debug Toolbar: Debugging Dream

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

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

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.

Rendering a Template

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

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

Listing 2-7
```
// src/Controller/LuckyController.php

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

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

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

Listing 2-8
```
// src/Controller/LuckyController.php

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

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

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

Listing 2-9
```
{# templates/lucky/number.html.twig #}

<h1>Your lucky number is {{ number }}</h1>
```

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

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

3. https://twig.symfony.com
4. https://github.com/symfony/symfony/blob/4.1/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php

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

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

Checking out the Project Structure

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

`config/`
: Contains... configuration of course!. 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 apps.

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!

Have fun!

5. `https://getcomposer.org`

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 is simple.

Creating Routes

A *route* is a map from a URL path to 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 and PHP. All formats provide the same features and performance, so choose the one you prefer. If you choose PHP annotations, run this command once in your app to add support for them:

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

Now you can configure the routes:

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

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

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

```
        // ...
    }

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

        // ...
    }
}
```

Thanks to these two routes:

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

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. You'll use them later 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! Create a new routing file (e.g. `routes.xml`) in the `config/` directory and Symfony will automatically use it.

Localized Routing (i18n)

New in version 4.1: The feature to localize routes was introduced in Symfony 4.1.

Routes can be localized to provide unique paths per *locale*. Symfony provides a handy way to declare localized routes without duplication.

Listing 3-3

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

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

class CompanyController extends AbstractController
{
    /**
     * @Route({
     *     "nl": "/over-ons",
     *     "en": "/about-us"
     * }, name="about_us")
     */
    public function about()
    {
        // ...
```

```
18      }
19  }
```

When a localized route is matched Symfony automatically knows which locale should be used during the request. Defining routes this way also eliminated the need for duplicate registration of routes which minimizes the risk for any bugs caused by definition inconsistency.

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

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):

Listing 3-5
```php
// src/Controller/BlogController.php
namespace App\Controller;

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

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

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

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

Listing 3-6

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

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

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

New in version 4.1: The feature to inline requirements was introduced in Symfony 4.1.

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:

Listing 3-7

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

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

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

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

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

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

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

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

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

New in version 4.1: The feature to inline default values was introduced in Symfony 4.1.

Listing all of your Routes

As your app grows, you'll eventually have a *lot* of routes! To see them all, run:

Listing 3-9
```
$ php bin/console debug:router

------------------------------ -------- ------------------------------------
 Name                           Method   Path
------------------------------ -------- ------------------------------------
 app_lucky_number               ANY      /lucky/number/{max}
 ...
------------------------------ -------- ------------------------------------
```

Advanced Routing Example

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

Listing 3-10
```php
// src/Controller/ArticleController.php

// ...
class ArticleController extends AbstractController
{
    /**
     * @Route(
     *     "/articles/{_locale}/{year}/{slug}.{_format}",
     *     defaults={"_format": "html"},
     *     requirements={
     *         "_locale": "en|fr",
     *         "_format": "html|rss",
     *         "year": "\d+"
     *     }
     * )
     */
```

```
17      public function show($_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`).

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.

`_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 makes a `301` redirect to `/foo`
`/foo/`	It makes a `301` redirect to `/foo/`	It matches (`200` status response)

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

New in version 4.1: The automatic `301` redirection from `/foo/` to `/foo` was introduced in Symfony 4.1. In previous Symfony versions this results in a `404` response.

Controller Naming Pattern

The `controller` value in your routes has the format `CONTROLLER_CLASS::METHOD`.

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, you can also use the fully qualified class name (e.g. `App\Controller\BlogController`).

Generating URLs

The routing system can also generate URLs. In reality, routing is a bidirectional system: mapping the URL to a controller and also 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. With this information, an URL can be generated in a controller:

Listing 3-11
```php
class MainController extends AbstractController
{
    public function show($slug)
    {
        // ...

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

If you need to generate a URL from a service, type-hint the *`UrlGeneratorInterface`*[1] service:

Listing 3-12
```php
// src/Service/SomeService.php

use Symfony\Component\Routing\Generator\UrlGeneratorInterface;

class SomeService
{
    private $router;
```

1. https://github.com/symfony/symfony/blob/4.1/src/Symfony/Component/Routing/Generator/UrlGeneratorInterface.php

```
8
9       public function __construct(UrlGeneratorInterface $router)
10      {
11          $this->router = $router;
12      }
13
14      public function someMethod()
15      {
16          $url = $this->router->generate(
17              'blog_show',
18              ['slug' => 'my-blog-post']
19          );
20          // ...
21      }
22  }
```

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:

Listing 3-13
```
1  $this->router->generate('blog', [
2      'page' => 2,
3      'category' => 'Symfony',
4  ]);
5  // /blog/2?category=Symfony
```

Generating Localized URLs

When a route is localized, Symfony uses by default the current request locale to generate the URL. In order to generate the URL for a different locale you must pass the `_locale` in the parameters array:

Listing 3-14
```
$this->router->generate('about_us', [
    '_locale' => 'nl',
]);
// generates: /over-ons
```

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:

Listing 3-15
```
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
```

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 "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 3-16
```
public function show($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:

Listing 3-17
```
$this->generateUrl('blog_show', ['slug' => 'slug-value']);

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

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 `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.

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:

Listing 4-1

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

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

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

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

The controller is the `number()` method, which lives inside 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 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 make life nicer, Symfony comes with an optional base controller class called *`AbstractController`*[1]. You can extend it to get access to some *helper methods*[2].

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

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

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

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

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

Generating URLs

The *`generateUrl()`*[3] 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]);
```

1. https://github.com/symfony/symfony/blob/4.1/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php
2. https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/FrameworkBundle/Controller/ControllerTrait.php
3. https://github.com/symfony/symfony/blob/4.1/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php

Redirecting

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

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

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

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

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

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

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

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

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

Rendering Templates

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

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

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

Fetching Services

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

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

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

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

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

```
    // ...
}
```

Awesome!

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

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

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

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

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

Like with all services, you can also use regular constructor injection in your controllers.

New in version 4.1: The ability to bind scalar values to controller arguments was introduced in Symfony 4.1. Previously you could only bind services.

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

Generating Controllers

To save time, you can install *Symfony Maker*[5] and tell Symfony to generate a new controller class:

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

created: src/Controller/BrandNewController.php
```

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

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

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

Managing Errors and 404 Pages

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

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

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

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

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

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

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

The *createNotFoundException()*[6] method is just a shortcut to create a special *NotFoundHttpException*[7] object, which ultimately triggers a 404 HTTP response inside Symfony.

If you throw an exception that extends or is an instance of *HttpException*[8], 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 The parameters Key: Parameters (Variables)).

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? All of that information is stored in Symfony's `Request` object. To get it in your controller, add it as an argument and **type-hint it with the Request class**:

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

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

    // ...
}
```

Keep reading for more information about using the Request object.

Managing the Session

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

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

To get the session, add an argument and type-hint it with *SessionInterface*[9]:

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

public function index(SessionInterface $session)
```

6. https://github.com/symfony/symfony/blob/4.1/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php
7. https://github.com/symfony/symfony/blob/4.1/src/Symfony/Component/HttpKernel/Exception/NotFoundHttpException.php
8. https://github.com/symfony/symfony/blob/4.1/src/Symfony/Component/HttpKernel/Exception/HttpException.php
9. https://github.com/symfony/symfony/blob/4.1/src/Symfony/Component/HttpFoundation/Session/SessionInterface.php

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

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

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

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

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

For more info, see *Sessions*.

Flash Messages

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

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

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

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

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

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

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

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

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

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

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

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

{# ...or you can read and display every flash message available #}
```

```twig
{% for label, messages in app.flashes %}
    {% for message in messages %}
        <div class="flash-{{ label }}">
            {{ message }}
        </div>
    {% endfor %}
{% endfor %}
```

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

You can use the *peek()*[10] method instead to retrieve the message while keeping it in the bag.

The Request and Response Object

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

Listing 4-17

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

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

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

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

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

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

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

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

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

Like the `Request`, the `Response` object has also a public `headers` property. This is a *ResponseHeaderBag*[11] 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:

Listing 4-18

10. https://github.com/symfony/symfony/blob/4.1/src/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.php
11. https://github.com/symfony/symfony/blob/4.1/src/Symfony/Component/HttpFoundation/ResponseHeaderBag.php

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

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

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

There are special classes that make certain kinds of responses easier. Some of these are mentioned below. To learn more about the `Request` and `Response` (and special `Response` classes), see the HttpFoundation component documentation.

Returning JSON Response

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

Listing 4-19
```php
// ...
public function index()
{
    // returns '{"username":"jane.doe"}' and sets the proper Content-Type header
    return $this->json(['username' => 'jane.doe']);

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

If the *serializer service* is enabled in your application, it will be used to serialize the data to JSON. Otherwise, the *`json_encode`*[12] function is used.

Streaming File Responses

You can use the *`file()`*[13] helper to serve a file from inside a controller:

Listing 4-20
```php
public function download()
{
    // send the file contents and force the browser to download it
    return $this->file('/path/to/some_file.pdf');
}
```

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

Listing 4-21
```php
use Symfony\Component\HttpFoundation\File\File;
use Symfony\Component\HttpFoundation\ResponseHeaderBag;

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

    return $this->file($file);

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

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

12. https://secure.php.net/manual/en/function.json-encode.php
13. https://github.com/symfony/symfony/blob/4.1/src/Symfony/Bundle/FrameworkBundle/Controller/AbstractController.php

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 `AbstractController` 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 Customize Error Pages
- How to Forward Requests to another Controller
- How to Define Controllers as Services
- How to Create a SOAP Web Service in a Symfony Controller
- How to Upload Files

Chapter 5
Creating and Using Templates

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

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

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

Templates

A template is 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:

Listing 5-1

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

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

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

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

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

Twig defines three types of special syntax:

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

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

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

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

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

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

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

Listing 5-4
```twig
{% for i in 1..10 %}
    <div class="{{ cycle(['even', 'odd'], i) }}">
      <!-- some HTML here -->
    </div>
{% endfor %}
```

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

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

Why Twig?

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

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

Listing 5-5

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

Twig Template Caching

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

Template Inheritance and Layouts

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

First, build a base layout file:

Listing 5-6

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

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

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

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

A child template might look like this:

Listing 5-7

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

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

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

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

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

Listing 5-8

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

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

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

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

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

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

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

Listing 5-9
```
{% block sidebar %}
    <h3>Table of Contents</h3>

    {# ... #}

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

Template Naming and Locations

By default, templates can live in two different locations:

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

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

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

Referencing Templates in a Bundle

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

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

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

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

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

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

Template Suffix

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

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

By default, any Symfony template can be written in either Twig or PHP, and the last part of the extension (e.g. `.twig` or `.php`) specifies which of these two *engines* should be used. The first part of the extension, (e.g. `.html`, `.css`, etc) is the final format that the template will generate. Unlike the engine, which determines how Symfony parses the template, this is 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.

Listing 5-10
```twig
{# templates/article/article_details.html.twig #}
<h2>{{ article.title }}</h2>
<h3 class="byline">by {{ article.authorName }}</h3>

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

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

Listing 5-11
```twig
{# templates/article/list.html.twig #}
{% extends 'layout.html.twig' %}

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

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

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

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

Linking to Pages

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

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

Listing 5-12
```php
// src/Controller/WelcomeController.php

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

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

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

Listing 5-13

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

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

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

Listing 5-14
```php
// src/Controller/ArticleController.php

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

class ArticleController extends AbstractController
{
    /**
     * @Route("/article/{slug}", name="article_show")
     */
    public function show($slug)
    {
        // ...
    }
}
```

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

Listing 5-15
```twig
{# templates/article/recent_list.html.twig #}
{% for article in articles %}
    <a href="{{ path('article_show', {'slug': article.slug}) }}">
        {{ article.title }}
    </a>
{% endfor %}
```

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

Listing 5-16
```twig
<a href="{{ url('welcome') }}">Home</a>
```

Linking to Assets

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

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

Listing 5-17
```
$ composer require symfony/asset
```

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

Listing 5-18
```twig
<img src="{{ asset('images/logo.png') }}" alt="Symfony!" />

<link href="{{ asset('css/blog.css') }}" rel="stylesheet" />
```

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

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

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

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

Listing 5-19
```twig
<img src="{{ absolute_url(asset('images/logo.png')) }}" alt="Symfony!" />
```

Including Stylesheets and JavaScripts in Twig

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

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

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

Listing 5-20
```twig
{# templates/base.html.twig #}
<html>
    <head>
        {# ... #}

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

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

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

Listing 5-21
```twig
{# templates/contact/contact.html.twig #}
{% extends 'base.html.twig' %}

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

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

{# ... #}
```

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

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

Listing 5-22
```
1 <link href="{{ asset('bundles/acmedemo/css/contact.css') }}" rel="stylesheet" />
```

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

Referencing the Request, User or Session

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

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

Output Escaping

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

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

Listing 5-23
```
1 <!-- output escaping is on automatically -->
2 {{ description }} <!-- I &lt;3 this product -->
3
4 <!-- disable output escaping with the raw filter -->
5 {{ description|raw }} <!-- I <3 this product -->
```

PHP templates do not automatically escape content.

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

Final Thoughts

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

Keep Going!

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

Learn more

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

Chapter 6
Configuring Symfony (and Environments)

Symfony applications can install third-party packages (bundles, libraries, etc.) to bring in new features (*services*) to your project. Each package can be customized via configuration files that live - by default - in the `config/` directory.

Configuration: config/packages/

The configuration for each package can be found in `config/packages/`. For instance, the framework bundle is configured in `config/packages/framework.yaml`:

Listing 6-1
```yaml
# config/packages/framework.yaml
framework:
    secret: '%env(APP_SECRET)%'
    #default_locale: en
    #csrf_protection: true
    #http_method_override: true

    # Enables session support. Note that the session will ONLY be started if you read or write from it.
    # Remove or comment this section to explicitly disable session support.
    session:
        handler_id: ~

    #esi: true
    #fragments: true
    php_errors:
        log: true
```

The top-level key (here `framework`) references configuration for a specific bundle (*FrameworkBundle* in this case).

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 related to the framework bundle, you can see an example dump of all available configuration options by running:

Listing 6-2
```
$ php bin/console config:dump-reference framework
```

The parameters Key: Parameters (Variables)

The configuration has some special top-level keys. One of them is called `parameters`: it's used to define *variables* that can be referenced in *any* other configuration file. For example, when you install the *translation* package, a `locale` parameter is added to `config/services.yaml`:

Listing 6-3
```
# config/services.yaml
parameters:
    locale: en

# ...
```

This parameter is then referenced in the framework config in `config/packages/translation.yaml`:

Listing 6-4
```
# config/packages/translation.yaml
framework:
    # any string surrounded by two % is replaced by that parameter value
    default_locale: '%locale%'

    # ...
```

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 .env File & Environment Variables

There is also a `.env` file which is loaded and its contents become environment variables. This is useful during development, or if setting environment variables is difficult for your deployment.

When you install packages, more environment variables are added to this file. But you can also add your own.

Environment variables can be referenced in any other configuration files by using a special syntax. For example, if you install the `doctrine` package, then you will have an environment variable called `DATABASE_URL` in your `.env` file. This is referenced inside `config/packages/doctrine.yaml`:

Listing 6-5
```yaml
# config/packages/doctrine.yaml
doctrine:
    dbal:
        url: '%env(DATABASE_URL)%'

        # The `resolve:` prefix replaces container params by their values inside the env variable:
        # url: '%env(resolve:DATABASE_URL)%'
```

For more details about environment variables, see Environment Variables.

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

The `.env` file is special, because 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. The `.env` file should contain sensible, non-secret *default* values for all of your environment variables and *should* be commited to your repository.

To override these variables with machine-specific or sensitive values, create a `.env.local` file. This file is **not committed to the shared repository** and is only stored on your machine. In fact, the `.gitignore` file that comes with Symfony prevents it from being committed.

You can also create a few other `.env` files that will be loaded:

- `.env.{environment}`: e.g. `.env.test` will be loaded in the `test` environment and committed to your repository.
- `.env.{environment}.local`: e.g. `.env.prod.local` will be loaded in the `prod` environment but will *not* be committed to your repository.

If you decide to set real environment variables on production, the `.env` files *are* still loaded, but your real environment variables will override those values.

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
- Nov 2018 Changes to .env & How to Update
- 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