Symfony

The Quick Tour

Version: 3.0

generated on September 1, 2016

What could be better to make up your own mind than to try out Symfony yourself? Aside from a little time, it will cost you nothing. Step by step you will explore the Symfony universe. Be careful, Symfony can become addictive from the very first encounter!

The Quick Tour (3.0)

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
The Big Picture

Start using Symfony in 10 minutes! This chapter will walk you through the most important concepts behind Symfony and explain how you can get started quickly by showing you a simple project in action.

If you've used a web framework before, you should feel right at home with Symfony. If not, welcome to a whole new way of developing web applications.

Installing Symfony

Before continuing reading this chapter, make sure to have installed both PHP and Symfony as explained in the *Installing & Setting up the Symfony Framework* article.

Understanding the Fundamentals

One of the main goals of a framework is to keep your code organized and to allow your application to evolve easily over time by avoiding the mixing of database calls, HTML tags and other PHP code in the same script. To achieve this goal with Symfony, you'll first need to learn a few fundamental concepts.

When developing a Symfony application, your responsibility as a developer is to write the code that maps the user's *request* (e.g. `http://localhost:8000/`) to the *resource* associated with it (the `Homepage` HTML page).

The code to execute is defined as methods of PHP classes. The methods are called **actions** and the classes **controllers**, but in practice most developers use **controllers** to refer to both of them. The mapping between user's requests and that code is defined via the **routing** configuration. And the contents displayed in the browser are usually rendered using **templates**.

When you go to `http://localhost:8000/app/example`, Symfony will execute the controller in `src/AppBundle/Controller/DefaultController.php` and render the `app/Resources/views/default/index.html.twig` template.

In the following sections you'll learn in detail the inner workings of Symfony controllers, routes and templates.

Actions and Controllers

Open the `src/AppBundle/Controller/DefaultController.php` file and you'll see the following code (for now, don't look at the `@Route` configuration because that will be explained in the next section):

Listing 1-1
```php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    /**
     * @Route("/", name="homepage")
     */
    public function indexAction()
    {
        return $this->render('default/index.html.twig');
    }
}
```

In Symfony applications, **controllers** are usually PHP classes whose names are suffixed with the `Controller` word. In this example, the controller is called `Default` and the PHP class is called `DefaultController`.

The methods defined in a controller are called **actions**, they are usually associated with one URL of the application and their names are suffixed with `Action`. In this example, the `Default` controller has only one action called `index` and defined in the `indexAction()` method.

Actions are usually very short - around 10-15 lines of code - because they just call other parts of the application to get or generate the needed information and then they render a template to show the results to the user.

In this example, the `index` action is practically empty because it doesn't need to call any other method. The action just renders a template with the *Homepage.* content.

Routing

Symfony routes each request to the action that handles it by matching the requested URL against the paths configured by the application. Open again the `src/AppBundle/Controller/DefaultController.php` file and take a look at the three lines of code above the `indexAction()` method:

Listing 1-2
```php
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    /**
     * @Route("/", name="homepage")
     */
    public function indexAction()
    {
        return $this->render('default/index.html.twig');
    }
}
```

These three lines define the routing configuration via the `@Route()` annotation. A **PHP annotation** is a convenient way to configure a method without having to write regular PHP code. Beware that annotation blocks start with `/**`, whereas regular PHP comments start with `/*`.

The first value of `@Route()` defines the URL that will trigger the execution of the action. As you don't have to add the host of your application to the URL (e.g. `http://example.com`), these URLs are always relative and they are usually called *paths*. In this case, the `/` path refers to the application homepage. The second value of `@Route()` (e.g. `name="homepage"`) is optional and sets the name of this route. For now this name is not needed, but later it'll be useful for linking pages.

Considering all this, the `@Route("/", name="homepage")` annotation creates a new route called `homepage` which makes Symfony execute the `index` action of the `Default` controller when the user browses the `/` path of the application.

In addition to PHP annotations, routes can be configured in YAML, XML or PHP files, as explained in the *Routing* guide. This flexibility is one of the main features of Symfony, a framework that never imposes a particular configuration format on you.

Templates

The only content of the `index` action is this PHP instruction:

Listing 1-3
```
return $this->render('default/index.html.twig');
```

The `$this->render()` method is a convenient shortcut to render a template. Symfony provides some useful shortcuts to any controller extending from the base Symfony *`Controller`*[1] class.

By default, application templates are stored in the `app/Resources/views/` directory. Therefore, the `default/index.html.twig` template corresponds to the `app/Resources/views/default/index.html.twig`. Open that file and you'll see the following code:

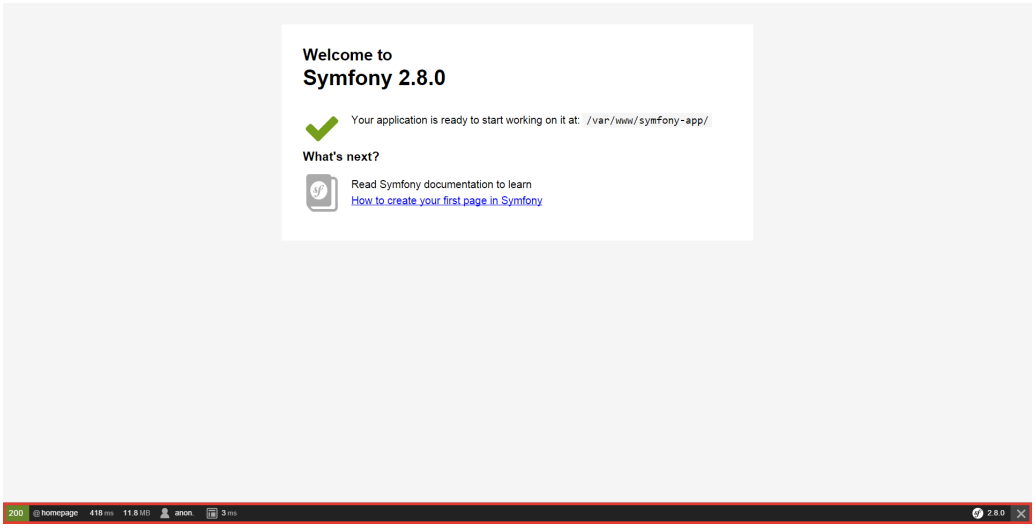
Listing 1-4
```
{# app/Resources/views/default/index.html.twig #}
{% extends 'base.html.twig' %}

{% block body %}
    <h1>Welcome to Symfony</h1>

    {# ... #}
{% endblock %}
```

This template is created with *Twig*[2], a template engine created for modern PHP applications. The *second part of this tutorial* explains how templates work in Symfony.

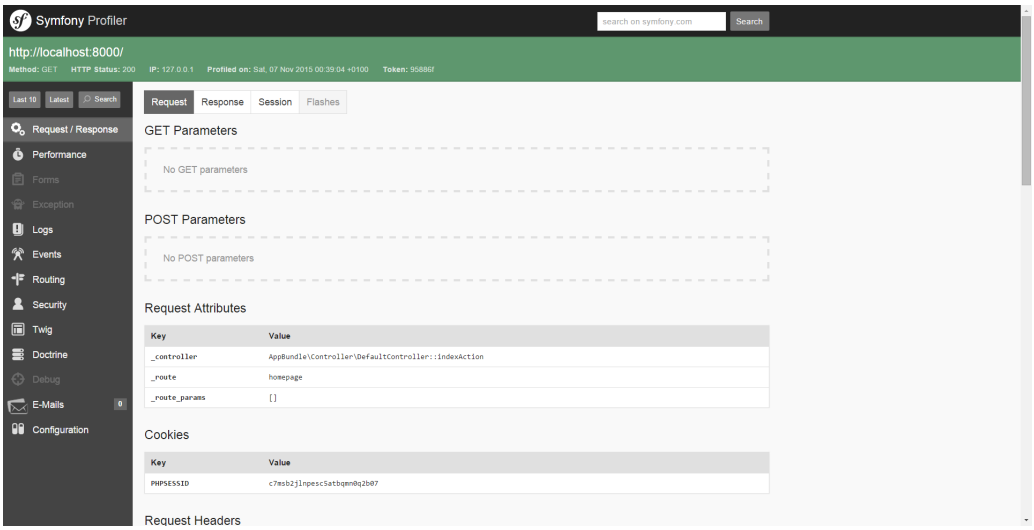
Working with Environments

Now that you have a better understanding of how Symfony works, take a closer look at the bottom of any Symfony rendered page. You should notice a small bar with the Symfony logo. This is the "web debug toolbar" and it is a Symfony developer's best friend!

1. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html
2. http://twig.sensiolabs.org/



But what you see initially is only the tip of the iceberg; click on any of the bar sections to open the profiler and get much more detailed information about the request, the query parameters, security details and database queries:



This tool provides so much internal information about your application that you may be worried about your visitors accessing sensible information. Symfony is aware of this issue and for that reason, it won't display this bar when your application is running in the production server.

How does Symfony know whether your application is running locally or on a production server? Keep reading to discover the concept of **execution environments**.

What is an Environment?

An environment represents a group of configurations that's used to run your application. Symfony defines two environments by default: `dev` (suited for when developing the application locally) and `prod` (optimized for when executing the application on production).

When you visit the `http://localhost:8000` URL in your browser, you're executing your Symfony application in the `dev` environment. To visit your application in the `prod` environment, visit the `http://localhost:8000/app.php` URL instead. If you prefer to always show the `dev` environment in the URL, you can visit `http://localhost:8000/app_dev.php` URL.

The main difference between environments is that `dev` is optimized to provide lots of information to the developer, which means worse application performance. Meanwhile, `prod` is optimized to get the best performance, which means that debug information is disabled, as well as the web debug toolbar.

The other difference between environments is the configuration options used to execute the application. When you access the `dev` environment, Symfony loads the `app/config/config_dev.yml` configuration file. When you access the `prod` environment, Symfony loads `app/config/config_prod.yml` file.

Typically, the environments share a large amount of configuration options. For that reason, you put your common configuration in `config.yml` and override the specific configuration file for each environment where necessary:

Listing 1-5
```yaml
# app/config/config_dev.yml
imports:
    - { resource: config.yml }

web_profiler:
    toolbar: true
    intercept_redirects: false
```

In this example, the `config_dev.yml` configuration file imports the common `config.yml` file and then overrides any existing web debug toolbar configuration with its own options.

For more details on environments, see the "Environments" section of the Configuration guide.

Final Thoughts

Congratulations! You've had your first taste of Symfony code. That wasn't so hard, was it? There's a lot more to explore, but you should already see how Symfony makes it really easy to implement web sites better and faster. If you are eager to learn more about Symfony, dive into the next section: "*The View*".

Chapter 2
The View

After reading the first part of this tutorial, you have decided that Symfony was worth another 10 minutes. In this second part, you will learn more about *Twig*[1], the fast, flexible and secure template engine for PHP applications. Twig makes your templates more readable and concise; it also makes them more friendly for web designers.

Getting Familiar with Twig

The official *Twig documentation*[2] is the best resource to learn everything about this template engine. This section just gives you a quick overview of its main concepts.

A Twig template is a text file that can generate any type of content (HTML, CSS, JavaScript, XML, CSV, LaTeX, etc.). Twig elements are separated from the rest of the template contents using any of these delimiters:

`{{ ... }}`
: Prints the content of a variable or the result of evaluating an expression;

`{% ... %}`
: Controls the logic of the template; it is used for example to execute `for` loops and `if` statements.

`{# ... #}`
: Allows including comments inside templates. Contrary to HTML comments, they aren't included in the rendered template.

Below is a minimal template that illustrates a few basics, using two variables `page_title` and `navigation`, which would be passed into the template:

Listing 2-1

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>{{ page_title }}</title>
5      </head>
```

1. http://twig.sensiolabs.org/
2. http://twig.sensiolabs.org/documentation

```
    <body>
        <h1>{{ page_title }}</h1>

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

To render a template in Symfony, use the `render()` method from within a controller. If the template needs variables to generate its contents, pass them as an array using the second optional argument:

Listing 2-2
```
$this->render('default/index.html.twig', array(
    'variable_name' => 'variable_value',
));
```

Variables passed to a template can be strings, arrays or even objects. Twig abstracts the difference between them and lets you access "attributes" of a variable with the dot (`.`) notation. The following code listing shows how to display the content of a variable passed by the controller depending on its type:

Listing 2-3
```
{# 1. Simple variables #}
{# $this->render('template.html.twig', array(
        'name' => 'Fabien')
    ) #}
{{ name }}

{# 2. Arrays #}
{# $this->render('template.html.twig', array(
        'user' => array('name' => 'Fabien'))
    ) #}
{{ user.name }}

{# alternative syntax for arrays #}
{{ user['name'] }}

{# 3. Objects #}
{# $this->render('template.html.twig', array(
        'user' => new User('Fabien'))
    ) #}
{{ user.name }}
{{ user.getName }}

{# alternative syntax for objects #}
{{ user.name() }}
{{ user.getName() }}
```

Decorating Templates

More often than not, templates in a project share common elements, like the well-known header and footer. Twig solves this problem elegantly with a concept called "template inheritance". This feature allows you to build a base template that contains all the common elements of your site and defines "blocks" of contents that child templates can override.

The `index.html.twig` template uses the `extends` tag to indicate that it inherits from the `base.html.twig` template:

Listing 2-4
```
{# app/Resources/views/default/index.html.twig #}
{% extends 'base.html.twig' %}

{% block body %}
```

```
    <h1>Welcome to Symfony!</h1>
{% endblock %}
```

Open the `app/Resources/views/base.html.twig` file that corresponds to the `base.html.twig` template and you'll find the following Twig code:

Listing 2-5
```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
        <link rel="icon" type="image/x-icon" href="{{ asset('favicon.ico') }}" />
    </head>
    <body>
        {% block body %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </body>
</html>
```

The `{% block %}` tags tell the template engine that a child template may override those portions of the template. In this example, the `index.html.twig` template overrides the `body` block, but not the `title` block, which will display the default content defined in the `base.html.twig` template.

Using Tags, Filters and Functions

One of the best features of Twig is its extensibility via tags, filters and functions. Take a look at the following sample template that uses filters extensively to modify the information before displaying it to the user:

Listing 2-6
```
<h1>{{ article.title|capitalize }}</h1>

<p>{{ article.content|striptags|slice(0, 255) }} ...</p>

<p>Tags: {{ article.tags|sort|join(", ") }}</p>

<p>Activate your account before {{ 'next Monday'|date('M j, Y') }}</p>
```

Don't forget to check out the official *Twig documentation*[3] to learn everything about filters, functions and tags.

Including other Templates

The best way to share a snippet of code between several templates is to create a new template fragment that can then be included from other templates.

Imagine that we want to display ads on some pages of our application. First, create a `banner.html.twig` template:

Listing 2-7
```
{# app/Resources/views/ads/banner.html.twig #}
<div id="ad-banner">
    ...
</div>
```

3. http://twig.sensiolabs.org/documentation

To display this ad on any page, include the `banner.html.twig` template using the `include()` function:

Listing 2-8
```twig
{# app/Resources/views/default/index.html.twig #}
{% extends 'base.html.twig' %}

{% block body %}
    <h1>Welcome to Symfony!</h1>

    {{ include('ads/banner.html.twig') }}
{% endblock %}
```

Embedding other Controllers

And what if you want to embed the result of another controller in a template? That's very useful when working with Ajax, or when the embedded template needs some variable not available in the main template.

Suppose you've created a `topArticlesAction` controller method to display the most popular articles of your website. If you want to "render" the result of that method (usually some HTML content) inside the `index` template, use the `render()` function:

Listing 2-9
```twig
{# app/Resources/views/index.html.twig #}
{{ render(controller('AppBundle:Default:topArticles')) }}
```

Here, the `render()` and `controller()` functions use the special `AppBundle:Default:topArticles` syntax to refer to the `topArticlesAction` action of the `Default` controller (the `AppBundle` part will be explained later):

Listing 2-10
```php
// src/AppBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    public function topArticlesAction()
    {
        // look for the most popular articles in the database
        $articles = ...;

        return $this->render('default/top_articles.html.twig', array(
            'articles' => $articles,
        ));
    }

    // ...
}
```

Creating Links between Pages

Creating links between pages is a must for web applications. Instead of hardcoding URLs in templates, the `path()` function knows how to generate URLs based on the routing configuration. That way, all your URLs can be easily updated by just changing the configuration:

Listing 2-11
```twig
<a href="{{ path('homepage') }}">Return to homepage</a>
```

The `path()` function takes the route name as the first argument and you can optionally pass an array of route parameters as the second argument.

The `url()` function is very similar to the `path()` function, but generates *absolute* URLs, which is very handy when rendering emails and RSS files: `<a href="{{ url('homepage') }}">Visit our website</a>`.

Including Assets: Images, JavaScripts and Stylesheets

What would the Internet be without images, JavaScripts and stylesheets? Symfony provides the `asset()` function to deal with them easily:

Listing 2-12
```
1 <link href="{{ asset('css/blog.css') }}" rel="stylesheet" type="text/css" />
2
3 <img src="{{ asset('images/logo.png') }}" />
```

The `asset()` function looks for the web assets inside the `web/` directory. If you store them in another directory, read *this article* to learn how to manage web assets.

Using the `asset()` function, your application is more portable. The reason is that you can move the application root directory anywhere under your web root directory without changing anything in your template's code.

Final Thoughts

Twig is simple yet powerful. Thanks to layouts, blocks, templates and action inclusions, it is very easy to organize your templates in a logical and extensible way.

You have only been working with Symfony for about 20 minutes, but you can already do pretty amazing stuff with it. That's the power of Symfony. Learning the basics is easy and you will soon learn that this simplicity is hidden under a very flexible architecture.

But I'm getting ahead of myself. First, you need to learn more about the controller and that's exactly the topic of the *next part of this tutorial*. Ready for another 10 minutes with Symfony?

Chapter 3
The Controller

Still here after the first two parts? You are already becoming a Symfony fan! Without further ado, discover what controllers can do for you.

Returning Raw Responses

Symfony defines itself as a Request-Response framework. When the user makes a request to your application, Symfony creates a `Request` object to encapsulate all the information related to that request. Similarly, the result of executing any action of any controller is the creation of a `Response` object which Symfony uses to generate the HTML content returned to the user.

So far, all the actions shown in this tutorial used the `$this->render()` controller shortcut method to return a rendered template as result. In case you need it, you can also create a raw `Response` object to return any text content:

Listing 3-1

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

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    /**
     * @Route("/", name="homepage")
     */
    public function indexAction()
    {
        return new Response('Welcome to Symfony!');
    }
}
```

Route Parameters

Most of the time, the URLs of applications include variable parts on them. If you are creating for example a blog application, the URL to display the articles should include their title or some other unique identifier to let the application know the exact article to display.

In Symfony applications, the variable parts of the routes are enclosed in curly braces (e.g. `/blog/read/{article_title}/`). Each variable part is assigned a unique name that can be used later in the controller to retrieve each value.

Let's create a new action with route variables to show this feature in action. Open the `src/AppBundle/Controller/DefaultController.php` file and add a new method called `helloAction()` with the following content:

Listing 3-2

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

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    // ...

    /**
     * @Route("/hello/{name}", name="hello")
     */
    public function helloAction($name)
    {
        return $this->render('default/hello.html.twig', array(
            'name' => $name
        ));
    }
}
```

Open your browser and access the `http://localhost:8000/hello/fabien` URL to see the result of executing this new action. Instead of the action result, you'll see an error page. As you probably guessed, the cause of this error is that we're trying to render a template (`default/hello.html.twig`) that doesn't exist yet.

Create the new `app/Resources/views/default/hello.html.twig` template with the following content:

Listing 3-3

```
{# app/Resources/views/default/hello.html.twig #}
{% extends 'base.html.twig' %}

{% block body %}
    <h1>Hi {{ name }}! Welcome to Symfony!</h1>
{% endblock %}
```

Browse again the `http://localhost:8000/hello/fabien` URL and you'll see this new template rendered with the information passed by the controller. If you change the last part of the URL (e.g. `http://localhost:8000/hello/thomas`) and reload your browser, the page will display a different message. And if you remove the last part of the URL (e.g. `http://localhost:8000/hello`), Symfony will display an error because the route expects a name and you haven't provided it.

Using Formats

Nowadays, a web application should be able to deliver more than just HTML pages. From XML for RSS feeds or Web Services, to JSON for Ajax requests, there are plenty of different formats to choose from.

Supporting those formats in Symfony is straightforward thanks to a special variable called `_format` which stores the format requested by the user.

Tweak the `hello` route by adding a new `_format` variable with `html` as its default value:

Listing 3-4
```
// src/AppBundle/Controller/DefaultController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

// ...

/**
 * @Route("/hello/{name}.{_format}", defaults={"_format"="html"}, name="hello")
 */
public function helloAction($name, $_format)
{
    return $this->render('default/hello.'.$_format.'.twig', array(
        'name' => $name
    ));
}
```

Obviously, when you support several request formats, you have to provide a template for each of the supported formats. In this case, you should create a new `hello.xml.twig` template:

Listing 3-5
```
<!-- app/Resources/views/default/hello.xml.twig -->
<hello>
    <name>{{ name }}</name>
</hello>
```

Now, when you browse to `http://localhost:8000/hello/fabien`, you'll see the regular HTML page because `html` is the default format. When visiting `http://localhost:8000/hello/fabien.html` you'll get again the HTML page, this time because you explicitly asked for the `html` format. Lastly, if you visit `http://localhost:8000/hello/fabien.xml` you'll see the new XML template rendered in your browser.

That's all there is to it. For standard formats, Symfony will also automatically choose the best `Content-Type` header for the response. To restrict the formats supported by a given action, use the `requirements` option of the `@Route()` annotation:

Listing 3-6
```
// src/AppBundle/Controller/DefaultController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

// ...

/**
 * @Route("/hello/{name}.{_format}",
 *     defaults = {"_format"="html"},
 *     requirements = { "_format" = "html|xml|json" },
 *     name = "hello"
 * )
 */
public function helloAction($name, $_format)
{
    return $this->render('default/hello.'.$_format.'.twig', array(
        'name' => $name
    ));
}
```

The `hello` action will now match URLs like `/hello/fabien.xml` or `/hello/fabien.json`, but it will show a 404 error if you try to get URLs like `/hello/fabien.js`, because the value of the `_format` variable doesn't meet its requirements.

Redirecting

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

Listing 3-7
```php
// src/AppBundle/Controller/DefaultController.php
class DefaultController extends Controller
{
    /**
     * @Route("/", name="homepage")
     */
    public function indexAction()
    {
        return $this->redirectToRoute('hello', array('name' => 'Fabien'));
    }
}
```

The `redirectToRoute()` method takes as arguments the route name and an optional array of parameters and redirects the user to the URL generated with those arguments.

Displaying Error Pages

Errors will inevitably happen during the execution of every web application. In the case of `404` errors, Symfony includes a handy shortcut that you can use in your controllers:

Listing 3-8
```php
// src/AppBundle/Controller/DefaultController.php
// ...

class DefaultController extends Controller
{
    /**
     * @Route("/", name="homepage")
     */
    public function indexAction()
    {
        // ...
        throw $this->createNotFoundException();
    }
}
```

For `500` errors, just throw a regular PHP exception inside the controller and Symfony will transform it into a proper `500` error page:

Listing 3-9
```php
// src/AppBundle/Controller/DefaultController.php
// ...

class DefaultController extends Controller
{
    /**
     * @Route("/", name="homepage")
     */
    public function indexAction()
    {
        // ...
        throw new \Exception('Something went horribly wrong!');
    }
}
```

Getting Information from the Request

Sometimes your controllers need to access the information related to the user request, such as their preferred language, IP address or the URL query parameters. To get access to this information, add a new argument of type `Request` to the action. The name of this new argument doesn't matter, but it must be preceded by the `Request` type in order to work (don't forget to add the new `use` statement that imports this `Request` class):

Listing 3-10
```php
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    /**
     * @Route("/", name="homepage")
     */
    public function indexAction(Request $request)
    {
        // is it an Ajax request?
        $isAjax = $request->isXmlHttpRequest();

        // what's the preferred language of the user?
        $language = $request->getPreferredLanguage(array('en', 'fr'));

        // get the value of a $_GET parameter
        $pageName = $request->query->get('page');

        // get the value of a $_POST parameter
        $pageName = $request->request->get('page');
    }
}
```

In a template, you can also access the `Request` object via the special `app.request` variable automatically provided by Symfony:

Listing 3-11
```twig
{{ app.request.query.get('page') }}

{{ app.request.request.get('page') }}
```

Persisting Data in the Session

Even if the HTTP protocol is stateless, Symfony provides a nice session object that represents the client (be it a real person using a browser, a bot, or a web service). Between two requests, Symfony stores the attributes in a cookie by using native PHP sessions.

Storing and retrieving information from the session can be easily achieved from any controller:

Listing 3-12
```php
use Symfony\Component\HttpFoundation\Request;

public function indexAction(Request $request)
{
    $session = $request->getSession();

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

    // get the value of a session attribute
    $foo = $session->get('foo');
```

```
12
13      // use a default value if the attribute doesn't exist
14      $foo = $session->get('foo', 'default_value');
15  }
```

You can also store "flash messages" that will auto-delete after the next request. They are useful when you need to set a success message before redirecting the user to another page (which will then show the message):

Listing 3-13
```
1  public function indexAction(Request $request)
2  {
3      // ...
4
5      // store a message for the very next request
6      $this->addFlash('notice', 'Congratulations, your action succeeded!');
7  }
```

And you can display the flash message in the template like this:

Listing 3-14
```
1  {% for flashMessage in app.session.flashBag.get('notice') %}
2      <div class="flash-notice">
3          {{ flashMessage }}
4      </div>
5  {% endfor %}
```

Final Thoughts

That's all there is to it and I'm not even sure you'll have spent the full 10 minutes. You were briefly introduced to bundles in the first part and all the features you've learned about so far are part of the core FrameworkBundle. But thanks to bundles, everything in Symfony can be extended or replaced. That's the topic of the *next part of this tutorial*.

Chapter 4
The Architecture

You are my hero! Who would have thought that you would still be here after the first three parts? Your efforts will be well rewarded soon. The first three parts didn't look too deeply at the architecture of the framework. Because it makes Symfony stand apart from the framework crowd, let's dive into the architecture now.

Understanding the Directory Structure

The directory structure of a Symfony application is rather flexible, but the recommended structure is as follows:

`app/`
: The application configuration, templates and translations.

`bin/`
: Executable files (e.g. `bin/console`).

`src/`
: The project's PHP code.

`tests/`
: Automatic tests (e.g. Unit tests).

`var/`
: Generated files (cache, logs, etc.).

`vendor/`
: The third-party dependencies.

`web/`
: The web root directory.

The `web/` Directory

The web root directory is the home of all public and static files like images, stylesheets and JavaScript files. It is also where each front controller (the file that handles all requests to your application) lives, such as the production controller shown here:

Listing 4-1
```php
// web/app.php
require_once __DIR__.'/../var/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
```

The controller first bootstraps the application using a kernel class (`AppKernel` in this case). Then, it creates the `Request` object using the PHP's global variables and passes it to the kernel. The last step is to send the response contents returned by the kernel back to the user.

The `app/` Directory

The `AppKernel` class is the main entry point of the application configuration and as such, it is stored in the `app/` directory.

This class must implement two methods:

`registerBundles()`
: Must return an array of all bundles needed to run the application, as explained in the next section.

`registerContainerConfiguration()`
: Loads the application configuration (more on this later).

Autoloading is handled automatically via *Composer*[1], which means that you can use any PHP class without doing anything at all! All dependencies are stored under the `vendor/` directory, but this is just a convention. You can store them wherever you want, globally on your server or locally in your projects.

Understanding the Bundle System

This section introduces one of the greatest and most powerful features of Symfony: The bundle system.

A bundle is kind of like a plugin in other software. So why is it called a *bundle* and not a *plugin*? This is because *everything* is a bundle in Symfony, from the core framework features to the code you write for your application.

All the code you write for your application is organized in bundles. In Symfony speak, a bundle is a structured set of files (PHP files, stylesheets, JavaScripts, images, ...) that implements a single feature (a blog, a forum, ...) and which can be easily shared with other developers.

Bundles are first-class citizens in Symfony. This gives you the flexibility to use pre-built features packaged in third-party bundles or to distribute your own bundles. It makes it easy to pick and choose which features to enable in your application and optimize them the way you want. And at the end of the day, your application code is just as *important* as the core framework itself.

Symfony already includes an AppBundle that you may use to start developing your application. Then, if you need to split the application into reusable components, you can create your own bundles.

1. https://getcomposer.org

Registering a Bundle

An application is made up of bundles as defined in the `registerBundles()` method of the `AppKernel` class. Each bundle is a directory that contains a single Bundle class that describes it:

Listing 4-2
```php
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\SecurityBundle\SecurityBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        new Symfony\Bundle\MonologBundle\MonologBundle(),
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
        new Symfony\Bundle\DoctrineBundle\DoctrineBundle(),
        new Symfony\Bundle\AsseticBundle\AsseticBundle(),
        new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        new AppBundle\AppBundle(),
    );

    if (in_array($this->getEnvironment(), array('dev', 'test'))) {
        $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
        $bundles[] = new Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
        $bundles[] = new Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();
    }

    return $bundles;
}
```

In addition to the AppBundle that was already talked about, notice that the kernel also enables other bundles that are part of Symfony, such as FrameworkBundle, DoctrineBundle, SwiftmailerBundle and AsseticBundle.

Configuring a Bundle

Each bundle can be customized via configuration files written in YAML, XML, or PHP. Have a look at this sample of the default Symfony configuration:

Listing 4-3
```yaml
# app/config/config.yml
imports:
    - { resource: parameters.yml }
    - { resource: security.yml }
    - { resource: services.yml }

framework:
    #esi:             ~
    #translator:      { fallbacks: ['%locale%'] }
    secret:          '%secret%'
    router:
        resource: '%kernel.root_dir%/config/routing.yml'
        strict_requirements: '%kernel.debug%'
    form:            true
    csrf_protection: true
    validation:      { enable_annotations: true }
    templating:      { engines: ['twig'] }
    default_locale:  '%locale%'
    trusted_proxies: ~
    session:         ~

# Twig Configuration
twig:
    debug:            '%kernel.debug%'
    strict_variables: '%kernel.debug%'

# Swift Mailer Configuration
swiftmailer:
    transport: '%mailer_transport%'
```

```
    host:      '%mailer_host%'
    username:  '%mailer_user%'
    password:  '%mailer_password%'
    spool:     { type: memory }

# ...
```

Each first level entry like `framework`, `twig` and `swiftmailer` defines the configuration for a specific bundle. For example, `framework` configures the FrameworkBundle while `swiftmailer` configures the SwiftmailerBundle.

Each environment can override the default configuration by providing a specific configuration file. For example, the `dev` environment loads the `config_dev.yml` file, which loads the main configuration (i.e. `config.yml`) and then modifies it to add some debugging tools:

Listing 4-4

```yaml
# app/config/config_dev.yml
imports:
    - { resource: config.yml }

framework:
    router:   { resource: '%kernel.root_dir%/config/routing_dev.yml' }
    profiler: { only_exceptions: false }

web_profiler:
    toolbar: true
    intercept_redirects: false

# ...
```

Extending a Bundle

In addition to being a nice way to organize and configure your code, a bundle can extend another bundle. Bundle inheritance allows you to override any existing bundle in order to customize its controllers, templates, or any of its files.

Logical File Names

When you want to reference a file from a bundle, use this notation: `@BUNDLE_NAME/path/to/file`; Symfony will resolve `@BUNDLE_NAME` to the real path to the bundle. For instance, the logical path `@AppBundle/Controller/DefaultController.php` would be converted to `src/AppBundle/Controller/DefaultController.php`, because Symfony knows the location of the AppBundle.

Logical Controller Names

For controllers, you need to reference actions using the format `BUNDLE_NAME:CONTROLLER_NAME:ACTION_NAME`. For instance, `AppBundle:Default:index` maps to the `indexAction` method from the `AppBundle\Controller\DefaultController` class.

Extending Bundles

If you follow these conventions, then you can use *bundle inheritance* to override files, controllers or templates. For example, you can create a bundle - NewBundle - and specify that it overrides AppBundle. When Symfony loads the `AppBundle:Default:index` controller, it will first look for the `DefaultController` class in NewBundle and, if it doesn't exist, then look inside AppBundle. This means that one bundle can override almost any part of another bundle!

Do you understand now why Symfony is so flexible? Share your bundles between applications, store them locally or globally, your choice.

Using Vendors

Odds are that your application will depend on third-party libraries. Those should be stored in the `vendor/` directory. You should never touch anything in this directory, because it is exclusively managed by Composer. This directory already contains the Symfony libraries, the SwiftMailer library, the Doctrine ORM, the Twig templating system and some other third party libraries and bundles.

Understanding the Cache and Logs

Symfony applications can contain several configuration files defined in several formats (YAML, XML, PHP, etc.). Instead of parsing and combining all those files for each request, Symfony uses its own cache system. In fact, the application configuration is only parsed for the very first request and then compiled down to plain PHP code stored in the `var/cache/` directory.

In the development environment, Symfony is smart enough to update the cache when you change a file. But in the production environment, to speed things up, it is your responsibility to clear the cache when you update your code or change its configuration. Execute this command to clear the cache in the `prod` environment:

Listing 4-5

```
1 $ php bin/console cache:clear --env=prod
```

When developing a web application, things can go wrong in many ways. The log files in the `var/logs/` directory tell you everything about the requests and help you fix the problem quickly.

Using the Command Line Interface

Each application comes with a command line interface tool (`bin/console`) that helps you maintain your application. It provides commands that boost your productivity by automating tedious and repetitive tasks.

Run it without any arguments to learn more about its capabilities:

Listing 4-6

```
1 $ php bin/console
```

The `--help` option helps you discover the usage of a command:

Listing 4-7

```
1 $ php bin/console debug:router --help
```

Final Thoughts

Call me crazy, but after reading this part, you should be comfortable with moving things around and making Symfony work for you. Everything in Symfony is designed to get out of your way. So, feel free to rename and move directories around as you see fit.

And that's all for the quick tour. From testing to sending emails, you still need to learn a lot to become a Symfony master. Ready to dig into these topics now? Look no further - go to the official *Symfony Documentation* and pick any topic you want.