FREE eBook

LEARNING symfony

Free unaffiliated eBook created from
Stack Overflow contributors.

#symfony

Table of Contents

About ... 1

Chapter 1: Getting started with symfony ... 2

Remarks ... 2

Open-Source ... 2

Official documentation ... 2

Versions ... 2

Symfony 3 ... 2

Symfony 2 ... 2

Examples ... 3

Creating a new Symfony project using the Symfony Installer ... 3

Downloading and installing the Symfony Installer on Linux / MacOS ... 3

Creating a new project with the latest Symfony version ... 3

Creating a new project using a specific Symfony version ... 4

Creating a new Symfony project using Composer ... 4

Installing a specific Symfony version ... 4

Running the Symfony application using PHP's built-in web server ... 5

Chapter 2: Controllers ... 6

Introduction ... 6

Syntax ... 6

Remarks ... 6

Examples ... 6

A simple controller class ... 6

Rendering a Twig template ... 7

Returning a 404 (Not Found) page ... 7

Using data from the Request object ... 8

Chapter 3: Routing ... 9

Introduction ... 9

Parameters ... 9

Examples ... 9

Simple routes ... 9

Routes with placeholders 10
Default values for placeholders 10
Chapter 4: Service Container 12
Introduction 12
Examples 12
Retrieve a service from the container 12
Chapter 5: The Request 13
Introduction 13
Syntax 13
Examples 13
Getting a query string parameter 13
Creating a Request object from global variables 14
Accessing a POST variable 14
Accessing the contents of a cookie 14
Credits 15

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: symfony

It is an unofficial and free symfony ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official symfony.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with symfony

Remarks

Symfony is a set of reusable PHP components, which can be used seperately or as part of the Symfony Framework.

As most frameworks, Symfony solves recurring technical problems for you (such as authentication, routing, etc.) so you can focus your time on the actual business problems you're trying to solve.

In contrary to other frameworks, however, the Symfony components are decoupled from each other, allowing you to select the ones you need. Instead of having to adapt your application to your framework, you can adapt the framework to your needs.

This is what makes Symfony very popular and allows other projects and frameworks (including Laravel, Drupal, Magento and Composer) to utilize the components without having to use the full framework.

Open-Source

Symfony is an open-source project. See how you can contribute.

Official documentation

The official Symfony documentation can be found on the Symfony website.

Versions

Symfony 3

Version	End of Life	Release Date
3.3	07/2018	2017-05-29
3.2	01/2018	2016-11-30
3.1	07/2017	2016-05-30
3.0	01/2017	2015-11-30

Symfony 2

Version	End of Life	Release Date
2.8	11/2019	2015-11-30
2.7	05/2019	2015-05-30
2.6	01/2016	2014-11-28
2.5	07/2015	2014-05-31
2.4	01/2015	2013-12-03
2.3	05/2017	2013-06-03
2.2	05/2014	2013-03-01
2.1	11/2013	2012-09-06
2.0	09/2013	2011-07-28

Examples

Creating a new Symfony project using the Symfony Installer

The Symfony Installer is a command line tool that helps you to create new Symfony applications. It requires PHP 5.4 or higher.

Downloading and installing the Symfony Installer on Linux / MacOS

Open a terminal and execute the following commands:

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

This creates a global `symfony` executable that can be called from anywhere. You have to do this only once: now you can create as many Symfony projects with it as you want.

Creating a new project with the latest Symfony version

Once the installer is installed, you can use it to create a new Symfony project. Run the following command:

```
symfony new my_project_name
```

This command will create a new directory (called `my_project_name`) containing the most recent

version of the Symfony Standard Edition. It will also install all of its dependencies (including the actual Symfony components) using Composer.

Creating a new project using a specific Symfony version

If you want to select a specific Symfony version instead of the latest one, you can use the optional second argument of the `new` command.

To select a minor version:

```
symfony new my_project_name 3.2
```

To select a patch version:

```
symfony new my_project_name 3.2.9
```

To select a beta version or release candidate:

```
symfony new my_project 2.7.0-BETA1
symfony new my_project 2.7.0-RC1
```

To select the most recent Long Term Support (LTS) version:

```
symfony new my_project_name lts
```

Creating a new Symfony project using Composer

If for some reason using the Symfony Installer is not an option, you can also create a new project using Composer. First of all, make sure you have installed Composer.

Next, you can use the `create-project` command to create a new project:

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

Similar to the Symfony Installer, this will install the latest version of the Symfony Standard Edition in a directory called `my_project_name` and will then install its dependencies (including the Symfony components).

Installing a specific Symfony version

As with the Symfony Installer, you can select a specific version of Symfony by supplying an optional third argument:

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

Note however that not all version aliases (such as `lts` for example) are available here.

Running the Symfony application using PHP's built-in web server

After creating a new Symfony application, you can use the `server:run` command to start a simple PHP web server, so you can access your new application from your web browser:

```
cd my_project_name/
php bin/console server:run
```

You can now visit http://localhost:8000/ to see the Symfony welcome page.

Important: while using the built-in web server is great for development, you should **not** use it in production. Use a full-featured web server such as Apache or Nginx instead.

Read Getting started with symfony online: https://riptutorial.com/symfony/topic/9448/getting-started-with-symfony

Chapter 2: Controllers

Introduction

A controller in Symfony is a PHP callable (a function, a method on an object, or a closure) that receives an HTTP request and returns an HTTP response. An HTTP response can contain anything: an HTML page, a JSON string, a file download, etc.

In order to tell Symfony which controller should handle a certain request, you need to configure a route.

Syntax

- $this->generateUrl('route_name', ['placeholder' => 'value']); // generates a URL for the route `route_name` with a placeholder
- $this->render('template.html.twig'); // renders a Twig template and returns a Response object
- $this->render('template.html.twig', ['parameter' => $value]); // renders a Twig template with a parameter
- throw $this->createNotFoundException('Message'); // throws a NotFoundHttpException which will cause Symfony to return a 404 response

Remarks

Controllers should be small and focus on handling HTTP requests: the actual business logic of your application should be delegated to different parts of your application, for instance your domain model.

Examples

A simple controller class

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

use Symfony\Component\HttpFoundation\Response;

class HelloWorldController
{
    public function helloWorldAction()
    {
        return new Response(
            '<html><body>Hello World!</body></html>'
        );
    }
}
```

Rendering a Twig template

Most of the time, you will want to render HTML responses from a template instead of hard-coding the HTML in your controller. Also, your templates will not be static but will contain placeholders for application data. By default Symfony comes with Twig, a powerful templating language.

In order to use Twig in your controller, extend Symfony's base `Controller` class:

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

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class HelloWorldController extends Controller
{
    public function helloWorldAction()
    {
        $text = 'Hello World!';

        return $this->render('hello-world.html.twig', ['text' => $text]);
    }
}
```

Create the Twig template (located in `app/Resources/views/hello-world.html.twig`):

```
<html><body>{{ text }}</body></html>
```

Twig will automatically replace the `{{ text }}` placeholder with the value of the `text` parameter, passed by the controller. This will render the following HTML output:

```
<html><body>Hello World!</body></html>
```

Returning a 404 (Not Found) page

Sometimes you want to return a 404 (Not Found) response, because the requested resource does not exist. Symfony allows you to do so by throwing a `NotFoundHttpException`.

The Symfony base Controller exposes a `createNotFoundException` method which creates the exception for you:

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

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

    // continue with the normal flow if no exception is thrown
    return $this->render(...);
}
```

Using data from the Request object

If you need to access the `Request` object (for instance to read the query parameters, to read an HTTP header or to process an uploaded file), you can receive the request as a method argument by adding a type-hinted argument:

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

public function indexAction(Request $request)
{
    $queryParam = $request->query->get('param');

    // ...
}
```

Symfony will recognize the type hint and add the request argument to the controller call.

Read Controllers online: https://riptutorial.com/symfony/topic/10085/controllers

Chapter 3: Routing

Introduction

Routing is the process of mapping a URL to a controller. Symfony has a powerfull Routing component which allows you to define routes.

The Routing component supports a number of configuration formats: annotations, YAML, XML and raw PHP.

Parameters

Parameter	Details
name	The name of the route. Example: `book_show`
path	The path (may contain wildcards). Example: `/book/{isbn}`
defaults	Default values of parameters

Examples

Simple routes

Using YAML:

```
# app/config/routing.yml
blog_list:
    path:     /blog
    defaults: { _controller: AppBundle:Blog:list }
```

Using Annotations:

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

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

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

```
}
```

A request for the `/blog` URL will be handled by the `listAction()` method of the `BlogController` inside `AppBundle`.

Routes with placeholders

Using YAML:

```
# app/config/routing.yml
blog_show:
    path:     /blog/{slug}
    defaults: { _controller: AppBundle:Blog:show }
```

Using Annotations:

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

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

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

Any request with a URL matching `/blog/*` will be handled by the `showAction()` method of the `BlogController` within `AppBundle`. The controller action will receive the value of the placeholder as a method argument.

For example, a request for `/blog/my-post` will trigger a call to `showAction()` with an argument `$slug` containing the value `my-post`. Using that argument, the controller action can change the response depending on the value of the placeholder, for instance by retrieving the blog post with the slug `my-post` from the database.

Default values for placeholders

If you want to have a placeholder that may be omitted, you can give it a default value:

Using YAML:

```
# app/config/routing.yml
blog_list:
    path:      /blog/{page}
    defaults:  { _controller: AppBundle:Blog:list, page: 1 }
```

```
    requirements:
        page: '\d+'
```

Using Annotations:

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

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

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

In this example, both the `/blog` and `/blog/1` URLs will match the `blog_list` route and will be handled by the `listAction()` method. In the case of `/blog`, `listAction()` will still receive the `$page` argument, with the default value `1`.

Read Routing online: https://riptutorial.com/symfony/topic/10084/routing

Chapter 4: Service Container

Introduction

A Symfony application is typically composed of a lot of objects that perform different tasks, such as repositories, controllers, mailers, etc. In Symfony, these objects are called **services**, and are defined in `app/config/services.yml` or in one of the installed bundles.

The **Service Container** knows how to instantiate these services, and keeps a reference of them so they don't have to be instantiated twice. If a service has dependencies it will instantiate those too.

Examples

Retrieve a service from the container

```
$logger = $container->get('logger');
```

This will fetch the service with the service ID "logger" from the container, an object that implements `Psr\Log\LoggerInterface`.

Read Service Container online: https://riptutorial.com/symfony/topic/10183/service-container

Chapter 5: The Request

Introduction

Symfony's Request class is an object-oriented representation of the HTTP request. It contains information such as the URL, query string, uploaded files, cookies and other headers coming from the browser.

Syntax

- $request->getPathInfo(); // returns the path (local part of the URL) that is being requested (but without the query string). I.e. when visiting https://example.com/foo/bar?key=value, this will contain /foo/bar
- $request->query->get('id'); // returns a query string parameter ($_GET)
- $request->query->get('id', 1); // returns a query string parameter with a default value
- $request->request->get('name'); // returns a request body variable ($_POST)
- $request->files->get('attachment'); // returns an instance of UploadedFile identified by "attachment"
- $request->cookies->get('PHPSESSID'); // returns the value of a cookie ($_COOKIE)
- $request->headers->get('content_type'); // returns an HTTP request header
- $request->getMethod(); // returns the HTTP request method (GET, POST, PUT, DELETE, etc.)
- $request->getLanguages(); // returns an array of languages the client accepts

Examples

Getting a query string parameter

Lets say we want to build a paginated list of products, where the number of the page is passed as a query string parameter. For instance, to fetch the 3rd page, you'd go to:

```
http://example.com/products?page=3
```

The raw HTTP request would look something like this:

```
GET /products?page=3 HTTP/1.1
Host: example.com
Accept: text/html
User-Agent: Mozilla/5.0 (Macintosh)
```

In order to get the page number from the request object, you can access the `query` property:

```
$page = $request->query->get('page'); // 3
```

In the case of a `page` parameter, you will probably want to pass a default value in case the query string parameter is not set:

```
$page = $request->query->get('page', 1);
```

This means that when someone visits http://example.com/products (note the absence of the query string), the `$page` variable will contain the default value `1`.

Creating a Request object from global variables

PHP exposes a number of so-called *global variables* which contain information about the HTTP request, such as `$_POST`, `$_GET`, `$_FILES`, `$_SESSION`, etc. The `Request` class contains a static `createFromGlobals()` method in order to instantiate a request object based on these variables:

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

$request = Request::createFromGlobals();
```

When using the Symfony framework, you should not instantiate the request object yourself. Instead, you should use the object that is instantiated when the framework is bootstrapped in `app.php` / `app_dev.php`. For instance by type hinting the request object in your controller.

Accessing a POST variable

To get the contents of a form that is submitted with `method="post"`, use the `post` property:

```
$name = $request->request->get('name');
```

Accessing the contents of a cookie

```
$id = $request->cookies->get('PHPSESSID');
```

This will return the value of the 'PHPSESSID' cookie sent by the browser.

Read The Request online: https://riptutorial.com/symfony/topic/10097/the-request

Credits

S. No	Chapters	Contributors
1	Getting started with symfony	Braiam, Code Lᴝver, Community, Nic Wortel, Paul Crovella
2	Controllers	Nic Wortel
3	Routing	Nic Wortel
4	Service Container	Nic Wortel
5	The Request	Nic Wortel