Symfony

The Quick Tour

Version: 4.2

generated on January 4, 2019

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 (4.2)

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! Really! That's all you need to understand the most important concepts and start building a real project!

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. Symfony *embraces* best practices, keeps backwards compatibility (Yes! Upgrading is always safe & easy!) and offers long-term support.

Downloading Symfony

First, make sure you've installed *Composer*[1] and have PHP 7.1.3 or higher.

Ready? In a terminal, run:

Listing 1-1
```
$ composer create-project symfony/skeleton quick_tour
```

This creates a new `quick_tour/` directory with a small, but powerful new Symfony application:

Listing 1-2
```
quick_tour/
├── .env
├── bin/console
├── composer.json
├── composer.lock
├── config/
├── public/index.php
├── src/
├── symfony.lock
├── var/
└── vendor/
```

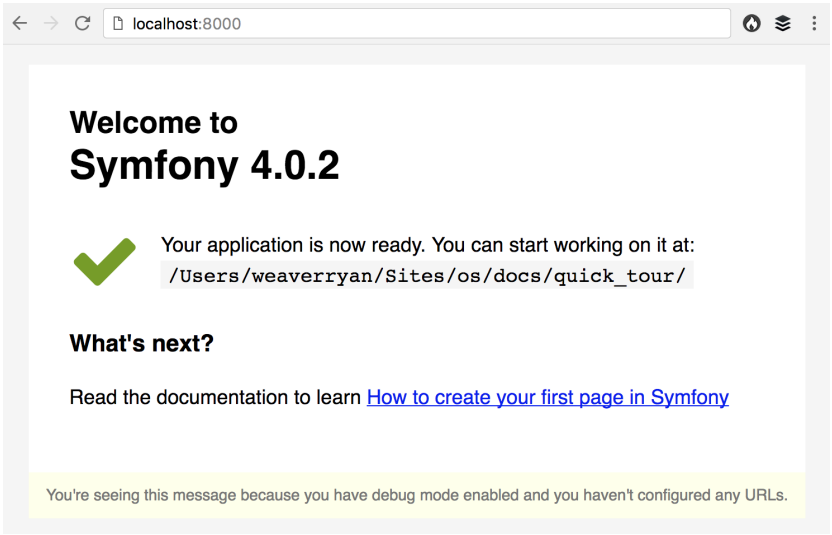
Can we already load the project in a browser? Of course! You can setup *Nginx or Apache* and configure their document root to be the `public/` directory. But, for development, Symfony has its own server. Install and run it with:

Listing 1-3

1. https://getcomposer.org/

```
$ composer require server --dev
$ php bin/console server:start
```

Try your new app by going to `http://localhost:8000` in a browser!



Fundamentals: Route, Controller, Response

Our project only has about 15 files, but it's ready to become a sleek API, a robust web app, or a microservice. Symfony starts small, but scales with you.

But before we go too far, let's dig into the fundamentals by building our first page.

Start in `config/routes.yaml`: this is where *we* can define the URL to our new page. Uncomment the example that already lives in the file:

Listing 1-4
```yaml
# config/routes.yaml
index:
    path: /
    controller: 'App\Controller\DefaultController::index'
```

This is called a *route*: it defines the URL to your page (`/`) and the "controller": the *function* that will be called whenever anyone goes to this URL. That function doesn't exist yet, so let's create it!

In `src/Controller`, create a new `DefaultController` class and an `index` method inside:

Listing 1-5
```php
// src/Controller/DefaultController.php
namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;

class DefaultController
{
    public function index()
    {
        return new Response('Hello!');
    }
}
```

That's it! Try going to the homepage: `http://localhost:8000/`. Symfony sees that the URL matches our route and then executes the new `index()` method.

A controller is just a normal function with *one* rule: it must return a Symfony `Response` object. But that response can contain anything: simple text, JSON or a full HTML page.

But the routing system is *much* more powerful. So let's make the route more interesting:

Listing 1-6
```
# config/routes.yaml
index:
-     path: /
+     path: /hello/{name}
      controller: 'App\Controller\DefaultController::index'
```

The URL to this page has changed: it is *now* `/hello/*`: the `{name}` acts like a wildcard that matches anything. And it gets better! Update the controller too:

Listing 1-7
```
// src/Controller/DefaultController.php
namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;

class DefaultController
{
-     public function index()
+     public function index($name)
      {
-         return new Response('Hello!');
+         return new Response("Hello $name!");
      }
}
```

Try the page out by going to `http://localhost:8000/hello/Symfony`. You should see: Hello Symfony! The value of the `{name}` in the URL is available as a `$name` argument in your controller.

But this can be even simpler! So let's install annotations support:

Listing 1-8
```
$ composer require annotations
```

Now, comment-out the YAML route by adding the `#` character:

Listing 1-9
```
# config/routes.yaml
# index:
#     path: /hello/{name}
#     controller: 'App\Controller\DefaultController::index'
```

Instead, add the route *right above* the controller method:

Listing 1-10
```
// src/Controller/DefaultController.php
namespace App\Controller;

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

class DefaultController
{
+     /**
+      * @Route("/hello/{name}")
+      */
      public function index($name) {
          // ...
      }
}
```

This works just like before! But by using annotations, the route and controller live right next to each other. Need another page? Add another route and method in `DefaultController`:

Listing 1-11

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

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

class DefaultController
{
    // ...

    /**
     * @Route("/simplicity")
     */
    public function simple()
    {
        return new Response('Simple! Easy! Great!');
    }
}
```

Routing can do *even* more, but we'll save that for another time! Right now, our app needs more features! Like a template engine, logging, debugging tools and more.

Keep reading with *Flex: Compose your Application*.

Chapter 2
Flex: Compose your Application

After reading the first part of this tutorial, you have decided that Symfony was worth another 10 minutes. Great choice! In this second part, you'll learn about Symfony Flex: the amazing tool that makes adding new features as simple as running one command. It's also the reason why Symfony is ideal for a small micro-service or a huge application. Curious? Perfect!

Symfony: Start Micro!

Unless you're building a pure API (more on that soon!), you'll probably want to render HTML. To do that, you'll use *Twig*[1]. Twig is a flexible, fast, and secure template engine for PHP. It makes your templates more readable and concise; it also makes them more friendly for web designers.

Is Twig already installed in our application? Actually, not yet! And that's great! When you start a new Symfony project, it's *small*: only the most critical dependencies are included in your `composer.json` file:

Listing 2-1
```
"require": {
    "...",
    "symfony/console": "^4.1",
    "symfony/flex": "^1.0",
    "symfony/framework-bundle": "^4.1",
    "symfony/yaml": "^4.1"
}
```

This makes Symfony different than any other PHP framework! Instead of starting with a *bulky* app with *every* possible feature you might ever need, a Symfony app is small, simple and *fast*. And you're in total control of what you add.

Flex Recipes and Aliases

So how can we install and configure Twig? By running one single command:

Listing 2-2

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

```
$ composer require twig
```

Two *very* interesting things happen behind the scenes thanks to Symfony Flex: a Composer plugin that is already installed in our project.

First, `twig` is not the name of a Composer package: it's a Flex *alias* that points to `symfony/twig-bundle`. Flex resolves that alias for Composer.

And second, Flex installs a *recipe* for `symfony/twig-bundle`. What's a recipe? It's a way for a library to automatically configure itself by adding and modifying files. Thanks to recipes, adding features is seamless and automated: install a package and you're done!

You can find a full list of recipes and aliases by going to *https://flex.symfony.com*.

What did this recipe do? In addition to automatically enabling the feature in `config/bundles.php`, it added 3 things:

`config/packages/twig.yaml`
: A configuration file that sets up Twig with sensible defaults.

`config/routes/dev/twig.yaml`
: A route that helps you debug your error pages.

`templates/`
: This is the directory where template files will live. The recipe also added a `base.html.twig` layout file.

Twig: Rendering a Template

Thanks to Flex, after one command, you can start using Twig immediately:

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

use Symfony\Component\Routing\Annotation\Route;
- use Symfony\Component\HttpFoundation\Response;
+ use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

-class DefaultController
+class DefaultController extends AbstractController
 {
     /**
      * @Route("/hello/{name}")
      */
     public function index($name)
     {
-        return new Response("Hello $name!");
+        return $this->render('default/index.html.twig', [
+            'name' => $name,
+        ]);
     }
```

By extending `AbstractController`, you now have access to a number of shortcut methods and tools, like `render()`. Create the new template:

Listing 2-4
```
{# templates/default/index.html.twig #}
<h1>Hello {{ name }}</h1>
```

That's it! The `{{ name }}` syntax will print the `name` variable that's passed in from the controller. If you're new to Twig, welcome! You'll learn more about its syntax and power later.

But, right now, the page *only* contains the `h1` tag. To give it an HTML layout, extend `base.html.twig`:

Listing 2-5
```twig
{# templates/default/index.html.twig #}
{% extends 'base.html.twig' %}

{% block body %}
    <h1>Hello {{ name }}</h1>
{% endblock %}
```

This is called template inheritance: our page now inherits the HTML structure from `base.html.twig`.

Profiler: Debugging Paradise

One of the *coolest* features of Symfony isn't even installed yet! Let's fix that:

Listing 2-6
```
$ composer require profiler
```

Yes! This is another alias! And Flex *also* installs another recipe, which automates the configuration of Symfony's Profiler. What's the result? Refresh!

See that black bar on the bottom? That's the web debug toolbar, and it's your new best friend. By hovering over each icon, you can get information about what controller was executed, performance information, cache hits & misses and a lot more. Click any icon to go into the *profiler* where you have even *more* detailed debugging and performance data!

Oh, and as you install more libraries, you'll get more tools (like a web debug toolbar icon that shows database queries).

You can now directly use the profiler because it configured *itself* thanks to the recipe. What else can we install this easily?

Rich API Support

Are you building an API? You can already return JSON easily from any controller:

Listing 2-7
```php
// src/Controller/DefaultController.php
namespace App\Controller;

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

class DefaultController extends AbstractController
{
    // ...

    /**
     * @Route("/api/hello/{name}")
     */
    public function apiExample($name)
    {
        return $this->json([
            'name' => $name,
            'symfony' => 'rocks',
        ]);
    }
}
```

But for a *truly* rich API, try installing *Api Platform*[2]:

Listing 2-8
```
$ composer require api
```

2. https://api-platform.com/

This is an alias to `api-platform/api-pack`, which has dependencies on several other packages, like Symfony's Validator and Security components, as well as the Doctrine ORM. In fact, Flex installed *5* recipes!

But like usual, we can immediately start using the new library. Want to create a rich API for a `product` table? Create a `Product` entity and give it the `@ApiResource()` annotation:

Listing 2-9
```php
// src/Entity/Product.php
namespace App\Entity;

use ApiPlatform\Core\Annotation\ApiResource;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity()
 * @ApiResource()
 */
class Product
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string")
     */
    private $name;

    /**
     * @ORM\Column(type="int")
     */
    private $price;

    // ...
}
```

Done! You now have endpoints to list, add, update and delete products! Don't believe me? List your routes by running:

Listing 2-10
```
$ php bin/console debug:router

------------------------------ -------- -----------------------------------
 Name                           Method   Path
------------------------------ -------- -----------------------------------
 api_products_get_collection    GET      /api/products.{_format}
 api_products_post_collection   POST     /api/products.{_format}
 api_products_get_item          GET      /api/products/{id}.{_format}
 api_products_put_item          PUT      /api/products/{id}.{_format}
 api_products_delete_item       DELETE   /api/products/{id}.{_format}
 ...
------------------------------ -------- -----------------------------------
```

Easily Remove Recipes

Not convinced yet? No problem: remove the library:

Listing 2-11
```
$ composer remove api
```

Flex will *uninstall* the recipes: removing files and un-doing changes to put your app back in its original state. Experiment without worry.

More Features, Architecture and Speed

I hope you're as excited about Flex as I am! But we still have *one* more chapter, and it's the most important yet. I want to show you how Symfony empowers you to quickly build features *without* sacrificing code quality or performance. It's all about the service container, and it's Symfony's super power. Read on: about *The Architecture*.

Chapter 3
The Architecture

You are my hero! Who would have thought that you would still be here after the first two parts? Your efforts will be well-rewarded soon. The first two 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.

Add Logging

A new Symfony app is micro: it's basically just a routing & controller system. But thanks to Flex, installing more features is simple.

Want a logging system? No problem:

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

This installs and configures (via a recipe) the powerful *Monolog*[1] library. To use the logger in a controller, add a new argument type-hinted with `LoggerInterface`:

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

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

class DefaultController extends AbstractController
{
    /**
     * @Route("/hello/{name}")
     */
    public function index($name, LoggerInterface $logger)
    {
        $logger->info("Saying hello to $name!");

        // ...
```

1. https://github.com/Seldaek/monolog

```
18         }
19     }
```

That's it! The new log message will be written to `var/log/dev.log`. The log file path or even a different method of logging can be configured by updating one of the config files added by the recipe.

Services & Autowiring

But wait! Something *very* cool just happened. Symfony read the `LoggerInterface` type-hint and automatically figured out that it should pass us the Logger object! This is called *autowiring*.

Every bit of work that's done in a Symfony app is done by an *object*: the Logger object logs things and the Twig object renders templates. These objects are called *services* and they are *tools* that help you build rich features.

To make life awesome, you can ask Symfony to pass you a service by using a type-hint. What other possible classes or interfaces could you use? Find out by running:

Listing 3-3
```
1  $ php bin/console debug:autowiring
```

Class/Interface Type	Alias Service ID
Psr\Cache\CacheItemPoolInterface	alias for "cache.app.recorder"
Psr\Log\LoggerInterface	alias for "monolog.logger"
Symfony\Component\EventDispatcher\EventDispatcherInterface	alias for "debug.event_dispatcher"
Symfony\Component\HttpFoundation\RequestStack	alias for "request_stack"
Symfony\Component\HttpFoundation\Session\SessionInterface	alias for "session"
Symfony\Component\Routing\RouterInterface	alias for "router.default"

This is just a short summary of the full list! And as you add more packages, this list of tools will grow!

Creating Services

To keep your code organized, you can even create your own services! Suppose you want to generate a random greeting (e.g. "Hello", "Yo", etc). Instead of putting this code directly in your controller, create a new class:

Listing 3-4
```
1  // src/GreetingGenerator.php
2  namespace App;
3
4  class GreetingGenerator
5  {
6      public function getRandomGreeting()
7      {
8          $greetings = ['Hey', 'Yo', 'Aloha'];
9          $greeting = $greetings[array_rand($greetings)];
10
11         return $greeting;
12     }
13 }
```

Great! You can use this immediately in your controller:

Listing 3-5

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

use App\GreetingGenerator;
use Psr\Log\LoggerInterface;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class DefaultController extends AbstractController
{
    /**
     * @Route("/hello/{name}")
     */
    public function index($name, LoggerInterface $logger, GreetingGenerator $generator)
    {
        $greeting = $generator->getRandomGreeting();

        $logger->info("Saying $greeting to $name!");

        // ...
    }
}
```

That's it! Symfony will instantiate the `GreetingGenerator` automatically and pass it as an argument. But, could we *also* move the logger logic to `GreetingGenerator`? Yes! You can use autowiring inside a service to access *other* services. The only difference is that it's done in the constructor:

Listing 3-6

```
// src/GreetingGenerator.php
+ use Psr\Log\LoggerInterface;

class GreetingGenerator
{
+     private $logger;
+
+     public function __construct(LoggerInterface $logger)
+     {
+         $this->logger = $logger;
+     }

    public function getRandomGreeting()
    {
        // ...

 +        $this->logger->info('Using the greeting: '.$greeting);

         return $greeting;
    }
}
```

Yes! This works too: no configuration, no time wasted. Keep coding!

Twig Extension & Autoconfiguration

Thanks to Symfony's service handling, you can *extend* Symfony in many ways, like by creating an event subscriber or a security voter for complex authorization rules. Let's add a new filter to Twig called `greet`. How? Create a class that extends `AbstractExtension`:

Listing 3-7

```php
// src/Twig/GreetExtension.php
namespace App\Twig;

use App\GreetingGenerator;
use Twig\Extension\AbstractExtension;
use Twig\TwigFilter;

```

```php
class GreetExtension extends AbstractExtension
{
    private $greetingGenerator;

    public function __construct(GreetingGenerator $greetingGenerator)
    {
        $this->greetingGenerator = $greetingGenerator;
    }

    public function getFilters()
    {
        return [
            new TwigFilter('greet', [$this, 'greetUser']),
        ];
    }

    public function greetUser($name)
    {
        $greeting =  $this->greetingGenerator->getRandomGreeting();

        return "$greeting $name!";
    }
}
```

After creating just *one* file, you can use this immediately:

Listing 3-8

```twig
{# templates/default/index.html.twig #}
{# Will print something like "Hey Symfony!" #}
<h1>{{ name|greet }}</h1>
```

How does this work? Symfony notices that your class extends `AbstractExtension` and so *automatically* registers it as a Twig extension. This is called autoconfiguration, and it works for *many* many things. Create a class and then extend a base class (or implement an interface). Symfony takes care of the rest.

Blazing Speed: The Cached Container

After seeing how much Symfony handles automatically, you might be wondering: "Doesn't this hurt performance?" Actually, no! Symfony is blazing fast.

How is that possible? The service system is managed by a very important object called the "container". Most frameworks have a container, but Symfony's is unique because it's *cached*. When you loaded your first page, all of the service information was compiled and saved. This means that the autowiring and autoconfiguration features add *no* overhead! It also means that you get *great* errors: Symfony inspects and validates *everything* when the container is built.

Now you might be wondering what happens when you update a file and the cache needs to rebuild? I like your thinking! It's smart enough to rebuild on the next page load. But that's really the topic of the next section.

Development Versus Production: Environments

One of a framework's main jobs is to make debugging easy! And our app is *full* of great tools for this: the web debug toolbar displays at the bottom of the page, errors are big, beautiful & explicit, and any configuration cache is automatically rebuilt whenever needed.

But what about when you deploy to production? We will need to hide those tools and optimize for speed!

This is solved by Symfony's *environment* system and there are three: `dev`, `prod` and `test`. Based on the environment, Symfony loads different files in the `config/` directory:

Listing 3-9
```
config/
├─ services.yaml
├─ ...
└─ packages/
    ├─ framework.yaml
    ├─ ...
    ├─ **dev/**
        ├─ monolog.yaml
        └─ ...
    ├─ **prod/**
        └─ monolog.yaml
    └─ **test/**
        ├─ framework.yaml
        └─ ...
└─ routes/
    ├─ annotations.yaml
    └─ **dev/**
        ├─ twig.yaml
        └─ web_profiler.yaml
```

This is a *powerful* idea: by changing one piece of configuration (the environment), your app is transformed from a debugging-friendly experience to one that's optimized for speed.

Oh, how do you change the environment? Change the `APP_ENV` environment variable from `dev` to `prod`:

Listing 3-10
```
# .env
- APP_ENV=dev
+ APP_ENV=prod
```

But I want to talk more about environment variables next. Change the value back to `dev`: debugging tools are great when you're working locally.

Environment Variables

Every app contains configuration that's different on each server - like database connection information or passwords. How should these be stored? In files? Or some other way?

Symfony follows the industry best practice by storing server-based configuration as *environment* variables. This means that Symfony works *perfectly* with Platform as a Service (PaaS) deployment systems as well as Docker.

But setting environment variables while developing can be a pain. That's why your app automatically loads a `.env` file, if the `APP_ENV` environment variable isn't set in the environment. The keys in this file then become environment variables and are read by your app:

Listing 3-11
```
# .env
###> symfony/framework-bundle ###
APP_ENV=dev
APP_SECRET=cc86c7ca937636d5ddf1b754beb22a10
###< symfony/framework-bundle ###
```

At first, the file doesn't contain much. But as your app grows, you'll add more configuration as you need it. But, actually, it gets much more interesting! Suppose your app needs a database ORM. Let's install the Doctrine ORM:

Listing 3-12
```
$ composer require doctrine
```

Thanks to a new recipe installed by Flex, look at the `.env` file again:

Listing 3-13

```
###> symfony/framework-bundle ###
APP_ENV=dev
APP_SECRET=cc86c7ca937636d5ddf1b754beb22a10
###< symfony/framework-bundle ###

+ ###> doctrine/doctrine-bundle ###
+ # ...
+ DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name
+ ###< doctrine/doctrine-bundle ###
```

The new `DATABASE_URL` environment variable was added *automatically* and is already referenced by the new `doctrine.yaml` configuration file. By combining environment variables and Flex, you're using industry best practices without any extra effort.

Keep Going!

Call me crazy, but after reading this part, you should be comfortable with the most *important* parts of Symfony. Everything in Symfony is designed to get out of your way so you can keep coding and adding features, all with the speed and quality you demand.

That's all for the quick tour. From authentication, to forms, to caching, there is so much more to discover. Ready to dig into these topics now? Look no further - go to the official *Symfony Documentation* and pick any guide you want.