

Charming Development in Symfony 5



Chapter 1: Creating a new Symfony 5 Project

Hey friends! And *welcome* to the world of Symfony 5... which just *happens* to be my favorite world! Ok, maybe Disney World is my *favorite* world... but programming in Symfony 5 is a *close* second.

Symfony 5 is lean and mean: it's lightning fast, starts tiny, but grows with you as your app gets bigger. And that's *not* marketing jargon! Your Symfony app will *literally* grow as you need more features. But more on that later.

Symfony 5 is *also* the product of *years* of work on developer experience. Basically, the people behind Symfony want you to *love* using it but *without* sacrificing quality. Yep, you get to write code that you're proud of, love the process, *and* build things quickly.

Symfony is also the *fastest* major PHP framework, which is no surprise: - its creator *also* created the PHP profiling system Blackfire. So... performance is always a focus.

Go Deeper!

Watch our [Blackfire.io: Revealing Performance Secrets with Profiling](#) course to learn about Blackfire.

Downloading the Symfony Installer

So... let's do this! Start off by going to <http://symfony.com> and clicking "Download". What we're *about* to download is *not* actually Symfony. It's an executable tool that will help make local development with Symfony... well.. awesome.

Because I'm on a Mac, I'll copy this command and then go open a terminal - I already have one waiting. It doesn't matter *where* on your filesystem you run this. Paste!

```
curl -sS https://get.symfony.com/cli/installer | bash
```

This downloads a *single* executable file and, for me, puts it into my home directory. To make it so that I can run this executable from *anywhere* on my system, I'll follow the command's advice and move the file somewhere else:

```
mv /Users/weaverryan/.symfony/bin/symfony /usr/local/bin/symfony
```

Ok, try it!

```
symfony
```

It's alive! Say hello to the Symfony CLI: a command-line tool that will help us with various things along our path to programming glory.

Starting a new Symfony App

Its *first* job will be to help us create a new Symfony 5 project. Run:

```
symfony new cauldron_overflow
```

Where `cauldron_overflow` will be the *directory* that the new app will live in. This *also* happens to be the name of the site we're building... but more on that later.

Behind the scenes, this command isn't doing anything special: it clones a Git repository called `symfony/skeleton` and then uses Composer to install that project's dependencies. We'll talk more about that repository *and* Composer a bit later.

When it's done, move into the new directory:

```
cd cauldron_overflow
```

And then *open* this directory in your favorite editor. I already have it open in *my* favorite: PhpStorm, which I did by going to File -> Open Directory and selecting the new project folder. Anyways, say hello to your brand new, shiny, full-of-potential new Symfony 5 project.

Our App is Small!

Before we start hacking away at things, let's create a new git repository and commit. But wait... run:

```
git status
```

“On branch master, nothing to commit.”

Surprise! The `symfony new` command *already* initialized a git repository *for* us and made the first commit. You can see it by running:

```
git log
```

“Add initial set of files”

Nice! Though, I personally would have liked a slightly more epic *first* commit message... but that's fine.

I'll hit "q" to exit this mode.

I mentioned earlier that Symfony starts *small*. To prove it, we can see a list of *all* the files that were committed by running:

```
git show --name-only
```

Yea... that's it! Our project - which is *fully* set up and ready to leverage Symfony - is less than 15 files... if you don't count things like `.gitignore`. Lean and mean.

Checking Requirements

Let's hook up a web server to our app and see it in action! First, make sure your computer has everything Symfony needs by running:

```
symfony check:req
```

For check requirements. We're good - but if you have any issues and need help fixing them, let us know in the comments.

Starting the PHP Web Server

To actually get the project running, look back in PhpStorm. We're going to talk more about each directory soon. But the *first* thing you need to know is that the `public/` directory is the "document root". This means that you need to point your web server - like Apache or Nginx - at this directory. Symfony has docs on how to do that.

But! To keep life simple, instead of setting up a *real* server on our local machine, we can use PHP's built-in web server. At the root of your project, run:

```
php -S 127.0.0.1:8000 -t public/
```

As soon as we do that, we can spin back over to our browser and go to <http://localhost:8000> to find... Welcome to Symfony 5! Ooh, fancy!

Next: as *easy* as it was to run that PHP web server, I'm going to show you an even *better* option for local development. Then we'll get to know the *significance* of the directories in our new app *and* make sure that we have a few plugins installed in PhpStorm... which... make working with Symfony an absolute pleasure.

Chapter 2: Meet our Tiny App + PhpStorm Setup

One of my *main* goals in these tutorials will be to help you *really* understand how Symfony - how your *application* - works.

To start with that, let's take a quick look at the directory structure.

The public/ Directory.

There are only three directories you need to think about. First, `public/` is the document root: so it will hold all files that need to be accessible by a browser. And... there's only one right now: `index.php`:

public/index.php

```
↕ // ... lines 1 - 2
3 use App\Kernel;
4 use Symfony\Component\ErrorHandler\Debug;
5 use Symfony\Component\HttpFoundation\Request;
6
7 require dirname(__DIR__).'/config/bootstrap.php';
8
9 if ($_SERVER['APP_DEBUG']) {
10     umask(0000);
11
12     Debug::enable();
13 }
14
15 if ($trustedProxies = $_SERVER['TRUSTED_PROXIES'] ??
    $_ENV['TRUSTED_PROXIES'] ?? false) {
16     Request::setTrustedProxies(explode(',', $trustedProxies),
    Request::HEADER_X_FORWARDED_ALL ^ Request::HEADER_X_FORWARDED_HOST);
17 }
18
19 if ($trustedHosts = $_SERVER['TRUSTED_HOSTS'] ?? $_ENV['TRUSTED_HOSTS'] ??
    false) {
20     Request::setTrustedHosts([$trustedHosts]);
21 }
22
23 $kernel = new Kernel($_SERVER['APP_ENV'], (bool) $_SERVER['APP_DEBUG']);
24 $request = Request::createFromGlobals();
25 $response = $kernel->handle($request);
26 $response->send();
27 $kernel->terminate($request, $response);
```

This is called the "front controller": a fancy word that programmers invented to mean that this is the file that's executed by your web server.

But, really, other than putting CSS or image files into `public/`, you'll almost never need to think about it.

src/ and config/

So... I kinda lied. There are *truly* only *two* directories that you need to think about: `config/` and `src/`. `config/` holds... um... puppies? No, `config/` holds config files and `src/` is where all your PHP code will go. It's just that simple.

Where is Symfony? Our project *started* with a `composer.json`:

```
composer.json
1 {
2     "type": "project",
3     "license": "proprietary",
4     "require": {
5         "php": "^7.2.5",
6         "ext-ctype": "*",
7         "ext-iconv": "*",
8         "symfony/console": "5.0.*",
9         "symfony/dotenv": "5.0.*",
10        "symfony/flex": "^1.3.1",
11        "symfony/framework-bundle": "5.0.*",
12        "symfony/yaml": "5.0.*"
13    },
14    "require-dev": {
15    },
16    // ... lines 16 - 64
65 }
```

file, which lists all the third-party libraries that our app *requires*. Behind the scenes, that `symfony new` command used composer to install these... which is a *fancy* way of saying that Composer downloaded a bunch of libraries into the `vendor/` directory... including Symfony itself.

We'll talk more about the other files and directories along the way... but they're not important yet.

Using the Symfony Local Web Server

A few minutes ago, we used PHP itself to start a local web server. Cool. But hit Ctrl+C to quit that. Why? Because that handy symfony binary tool we installed comes with a more *powerful* local server. Run:

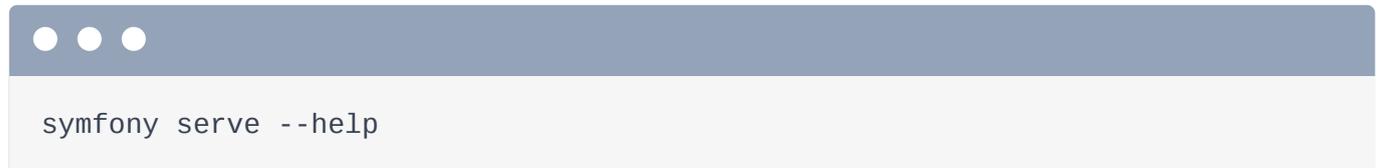
```
symfony serve
```

That's it. The first time you run this, it may ask you about installing a certificate. That's optional. If you *do* install it - I did - it will start the web server with https. Yep, you get https locally with

zero config.

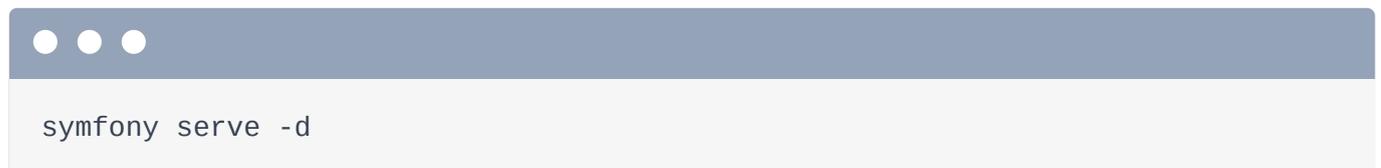
Once it's running, move over to your browser and refresh. It works! And the little lock icon proves that we're now using https.

To stop the web server, just hit Control + C. You can see all of this command's options by running:



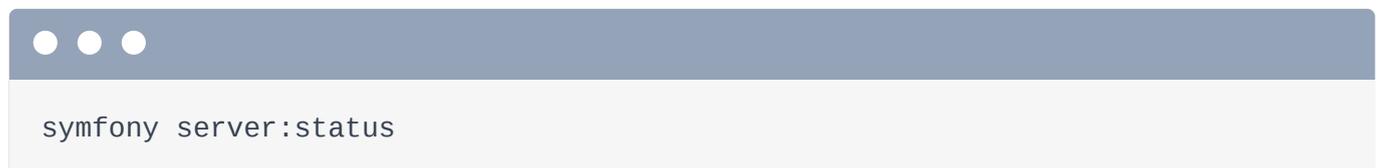
```
symfony serve --help
```

Like ways to control the port number. When I use this command, I usually run:



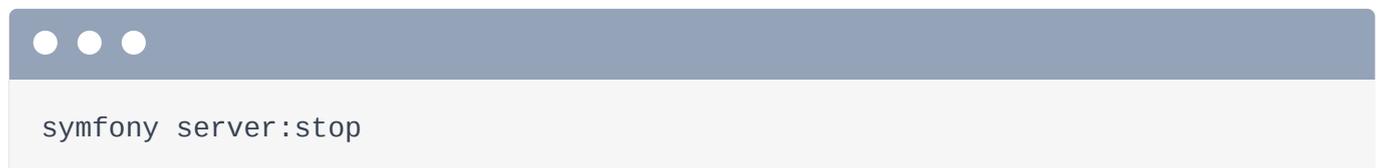
```
symfony serve -d
```

The `-d` means to run as a daemon. It does the exact same thing except that *now* it runs in the background... which means I can still use this terminal. Running:



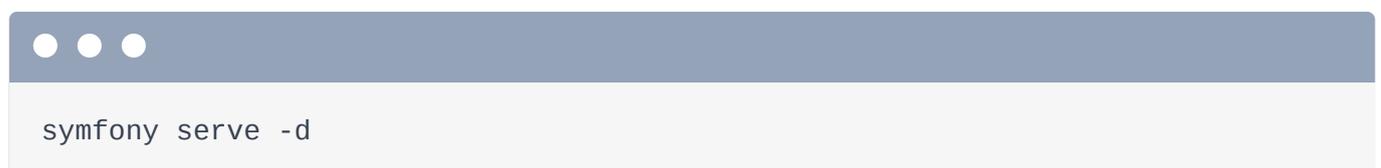
```
symfony server:status
```

Shows me that the server is running and:



```
symfony server:stop
```

Will stop it. Let's start it again:



```
symfony serve -d
```

Installing PhpStorm Plugins

Ok: we're about to start doing a *lot* of coding... so I want to make sure your editor is ready to go. And, yea, you can use *whatever* editor your want. But I *highly* recommend PhpStorm! Seriously, it makes developing in Symfony a *dream*! And no, the nice people at PhpStorm aren't paying me to say this... though... they *do* actually sponsor several open source PHP devs... which is kinda better.

To *really* make PhpStorm awesome, you need to do two things. First, open the Preferences, select "Plugins" and click "Marketplace". Search for "Symfony".

This plugin is *amazing*... proven by the nearly 4 million downloads. This will give us all *kinds* of extra auto-completion & intelligence for Symfony. If you don't have it already, install it. You should *also* install the "PHP Annotations" and "PHP toolbox" plugins. If you search for "php toolbox"... you can see all three of them. Install them and restart PhpStorm.

Once you've restarted, go *back* to Preferences and Search for Symfony. In addition to installing the plugin, you *also* need to enable it on a project-by-project basis. Check Enable and then apply. It says you need to restart PhpStorm... but I don't think that's true.

The *second* thing you need to do in PhpStorm is to search for Composer and find the "Languages and Frameworks", "PHP", "Composer" section. Make sure the "Synchronize IDE settings with composer.json" box is checked... which automatically configures several useful things.

Hit "Ok" and... we are ready! Let's create our very first page and see what Symfony is all about, next.

Chapter 3: Route, Controllers & Responses!

The page we're looking at right now... which is super fun... and even changes colors... is *just* here to say "Hello!". Symfony is rendering this because, in reality, our app doesn't have *any* real pages yet. Let's change that.

Route + Controller = Page

Every web framework... in *any* language... has the same main job: to give you a route & controller system: a 2-step system to build pages. A route defines the URL of the page and the controller is where we write PHP code to *build* that page, like the HTML or JSON.

Open up `config/routes.yaml`:

```
config/routes.yaml
```

```
1 #index:
2 #   path: /
3 #   controller: App\Controller\DefaultController::index
```

Hey! We already have an example! Uncomment that. If you're not familiar with YAML, it's super friendly: it's a key-value config format that's separated by colons. Indentation is also important.

This creates a single route whose URL is `/`. The controller points to a *function* that will *build* this page... really, it points to a method on a class. Overall, this route says:

"when the user goes to the homepage, please execute the `index` method on the `DefaultController` class."

Oh, and you can ignore that `index` key at the top of the YAML: that's an internal name for the route... and it's not important yet.

Our App

The project we're building is called "Cauldron Overflow". We *originally* wanted to create a site where developers could ask questions and other developers answered them but... someone beat us to it... by... like 10 years. So like all impressive startups, we're pivoting! We've noticed a lot of wizards accidentally blowing themselves up... or conjuring fire-breathing dragons when they meant to create a small fire for roasting marshmallows. And so... Cauldron Overflow is here to become *the* place for witches and wizards to ask and answer questions about magical misadventures.

Creating a Controller

On the homepage, we will eventually list some of the most recent questions. So let's change the controller class to `QuestionController` and the method to `homepage`.

```
config/routes.yaml
1 index:
2     path: /
3     controller: App\Controller\QuestionController::homepage
```

Ok, route done: it defines the URL and points to the controller that will build the page. Now... we need to create that controller! Inside the `src/` directory, there's already a `Controller/` directory... but it's empty. I'll right click on this and select "New PHP class". Call it `QuestionController`.

Namespaces & the src/ Directory

Ooh, check this out. It pre-filled the *namespace*! That's awesome! This is thanks to the Composer PhpStorm configuration we did in the last chapter.

Here's the deal: every class we create in the `src/` directory will need a namespace. And... for reasons that aren't super important, the namespace must be `App\` followed whatever directory the file lives in. Because we're creating this file in the `Controller/` directory, its namespace must be `App\Controller`. PhpStorm will pre-fill this every time.

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 2
3 namespace App\Controller;
↕ // ... lines 4 - 6
7 class QuestionController
8 {
↕ // ... lines 9 - 12
13 }
```

Perfect! Now, because in `routes.yaml` we decided to call the method `homepage`, create that here: `public function homepage()`.

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 2
3 namespace App\Controller;
↕ // ... lines 4 - 6
7 class QuestionController
8 {
9     public function homepage()
10    {
↕ // ... line 11
12    }
13 }
```

Controllers Return a Response

And.. congratulations! You are inside of a controller function, which is also sometimes called an "action"... to confuse things. Our job here is simple: to build the page. We can write *any* code we need to do that - like to make database queries, cache things, perform API calls, mine cryptocurrencies... whatever. The *only* rule is that a controller function *must* return a `Symfony Response` object.

Say `return new Response()`. PhpStorm tries to auto-complete this... but there are *multiple* `Response` classes in our app. The one we want is from `Symfony\Component\HttpFoundation`. `HttpFoundation` is one of the most important parts - or "components" - in Symfony. Hit tab to auto-complete it.

But stop! Did you see that? Because we let PhpStorm auto-complete that class for us, it wrote `Response`, but it *also* added the `use` statement for this class at the top of the file! That is one of the *best* features of PhpStorm and I'm going to use it a *lot*. You will *constantly* see me type a

class and allow PhpStorm to auto-complete it so that it adds the `use` statement to the top of the file for me.

Inside `new Response()`, add some text:

"What a bewitching controller we have conjured!"

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 2
3 namespace App\Controller;
4
5 use Symfony\Component\HttpFoundation\Response;
6
7 class QuestionController
8 {
9     public function homepage()
10    {
11        return new Response('What a bewitching controller we have
12        conjured!');
13    }
14 }
```

And... done! We just created our first page! Let's try it! When we go to the homepage, it should execute our controller function... which returns the message.

Find your browser. We're already on the homepage... so just refresh. Say hello to our *very* first page. I know, it's not much to look at yet, but we've already covered the most *foundational* part of Symfony: the route and controller system.

Next, let's make our route fancier by using something called annotations. We'll also create a second page with a route that matches a *wildcard* path.

Chapter 4: Annotation & Wildcard Routes

Creating a route in YAML that points to a controller function is pretty simple. But there's an even *easier* way to create routes... and I *love* it. It's called: annotations.

First, comment-out the YAML route. Basically, remove it entirely. To prove it's not working, refresh the homepage. Yep! It's back to the welcome page.

```
config/routes.yaml
```

```
1 #index:
2 #   path: /
3 #   controller: App\Controller\QuestionController::homepage
```

Installing Annotations Support

Annotations are a special config format... and support for annotations is *not* something that comes standard in our tiny Symfony app. And... that's fine! In fact, that's the *whole* philosophy of Symfony: start small and add features when you need them.

To add annotations support, we'll use Composer to require a new package. If you don't have Composer installed already, go to <https://getcomposer.org>.

Once you *do*, run:

```
composer require "annotations:<6.2.10"
```

If you're familiar with composer, that package name might look strange. And in reality, it installed a totally *different* package: `sensio/framework-extra-bundle`. Near the bottom of the command, it mentions something about two recipes. We'll talk about what's going on soon: it's part of what makes Symfony special.

Adding a Route Annotation

Anyways, now that annotations support is installed, we can re-add our route via annotations. What does that mean? Above your controller function, say `/**` and hit enter to create a PHPDoc section. Then say `@Route` and auto-complete the one from the Routing component. Just like before, PhpStorm added the `use` statement at the top of the class automatically.

Inside the parentheses, say `"/"`.

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 5
6 use Symfony\Component\Routing\Annotation\Route;
7
8 class QuestionController
9 {
10     /**
11      * @Route("/")
12      */
13     public function homepage()
14     {
↕ // ... line 15
16     }
17 }
```

That's it! When the user goes to the homepage, it will execute the function right below this. I *love* annotations because they're simple to read and keep the route and controller right next to each other. And yes... annotations are *literally* configuration inside PHP comments. If you don't like them, you can always use YAML or XML instead: Symfony is super flexible. From a performance standpoint, all the formats are the same.

Now when we refresh the homepage... we're back!

A Second Route and Controller

This page will eventually list some recently-asked questions. When you click on a specific question, it will need its *own* page. Let's create a second route and controller for that. How? By creating a second method. How about: `public function show()`.

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 7
8 class QuestionController
9 {
↕ // ... lines 10 - 20
21     public function show()
22     {
↕ // ... line 23
24     }
25 }
```

Above this, add `@Route()` and set the URL to, how about, `/questions/how-to-tie-my-shoes-with-magic`. That would be awesome!

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 7
8 class QuestionController
9 {
↕ // ... lines 10 - 17
18     /**
19      * @Route("/questions/how-to-tie-my-shoes-with-magic")
20      */
21     public function show()
22     {
↕ // ... line 23
24     }
25 }
```

Inside, just like last time, return a new `Response`: the one from `HttpFoundation`.

"Future page to show a question"

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 7
8 class QuestionController
9 {
↕ // ... lines 10 - 17
18     /**
19      * @Route("/questions/how-to-tie-my-shoes-with-magic")
20      */
21     public function show()
22     {
23         return new Response('Future page to show a question!');
24     }
25 }
```

Let's try it! Copy the URL, move over to your browser, paste and... it works! We just created a *second* page... in less than a minute.

The Front Controller: Working Behind-the-Scenes

By the way, no matter what URL we go to - like this one or the homepage - the PHP file that our web server is executing is `index.php`. It's as *if* we are going to `/index.php/questions/how-to-tie-my-shoes-with-magic`. The only reason you don't *need* to have `index.php` in the URL is because our local web server is configured to execute `index.php` automatically. On production, your Nginx or Apache config will do the same. Check the Symfony docs to learn how.

A Wildcard Route

Eventually, we're going to have a database *full* of questions. And so, no, we are *not* going to manually create one route per question. Instead, we can make this route smarter. Replace the `how-to-tie-my-shoes-with-magic` part with `{slug}`.

When you have something between curly braces in a route, it becomes a *wildcard*. This route now matches `/questions/ANYTHING`. The name `{slug}` is not important: we could have used anything... like `{slugulusErecto}`! That makes no difference.

But *whatever* we call this wildcard - like `{slug}` - we are now *allowed* to have an argument to our controller with the same *name*: `$slug`... which will be set to whatever that part of the URL is.

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 7
8  class QuestionController
9  {
↕ // ... lines 10 - 17
18     /**
19      * @Route("/questions/{slug}")
20      */
21     public function show($slug)
22     {
↕ // ... lines 23 - 26
27     }
28 }
```

Let's use that to make our page fancier! Let's use `sprintf()`, say "the" question and add a `%s` placeholder. Pass `$slug` for that placeholder.

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 7
8 class QuestionController
9 {
↕ // ... lines 10 - 17
18 /**
19  * @Route("/questions/{slug}")
20  */
21 public function show($slug)
22 {
23     return new Response(sprintf(
24         'Future page to show the question "%s"!',
25         $slug
26     ));
27 }
28 }
```

Sweet! Move over, refresh and... love it! Change the URL to `/questions/accidentally-turned-cat-into-furry-shoes` and... that works too.

In the future, we'll use the `$slug` to query the database for the question. But since we're not there yet, I'll use `str_replace()` ... and `ucwords()` to make this *just* a little more elegant. It's still early, but the page is *starting* come alive!

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 7
8 class QuestionController
9 {
↕ // ... lines 10 - 20
21 public function show($slug)
22 {
23     return new Response(sprintf(
↕ // ... line 24
25         ucwords(str_replace('-', ' ', $slug))
26     ));
27 }
28 }
```

Next, our new app is hiding a secret! A little command-line executable that's *filled* with goodies.

Chapter 5: The Lovely bin/console Tool

Let's commit our progress so far. I'll clear the screen and run:

```
git status
```

Interesting: there are a few *new* files here that I didn't create. Don't worry: we're going to talk about that *exactly* in the next chapter. Add everything with:

```
git add .
```

Normally... this command can be dangerous - we might accidentally add some files that we *don't* want to commit! Fortunately, our project came with a pre-filled `.gitignore` file which ignores the important stuff, like `vendor/` and some other paths we'll talk about later. For example, `var/` holds cache and log files. The point is, Symfony has our back.

Commit with:

```
git commit -m "we are ROCKING this Symfony thing"
```

Hello bin/console Command

You can interact with your Symfony app in *two* different ways. The first is by loading a page in your browser. The *second* is with a handy command-line script called `bin/console`. At your terminal, run:

```
php bin/console
```

Woh! This command lists a *bunch* of different things you can do with it, including a *lot* of debugging tools. Now, just to demystify this a little, there is *literally* a `bin/` directory in our app with a file called `console` inside. So this `bin/console` thing is not some global command that got installed on our system: we are literally executing a physical PHP file.

The `bin/console` command can do *many* things - and we'll discover my favorite features along the way. For example, want to see a list of every route in your app? Run:



```
php bin/console debug:router
```

Yep! There are our *two* routes... plus another one that Symfony adds automatically during development.

The `bin/console` tool *already* contains *many* useful commands like this. But the list of commands it supports is *not* static. New commands can be added by *us... or* by new packages that we install into our project. That's my "not-so-subtle" foreshadowing.

Next: let's talk about Symfony Flex, Composer aliases and the recipes system. Basically, the tools that makes Symfony truly unique.

Chapter 6: Flex, Recipes & Aliases

We're going to install a *totally* new package into our app called the "security checker". The security checker is a tool that looks at your application's dependencies and tell you if any of them have known security vulnerabilities. But, full disclosure, as *cool* as that is... the *real* reason I want to install this library is because it's a *great* way to look at Symfony's all-important "recipe" system.

At your terminal, run:

```
composer require sec-checker --no-scripts
```

Tip

You can still download the security checker to see how its recipe works, but the API it uses has been discontinued in favor of other solutions. If you want to know more, see <https://github.com/sensiolabs/security-checker>

In a real app, you *should* probably pass `--dev` to add this to your *dev* dependencies... but it won't matter for us.

Flex Aliases

There *is*, however, something weird here. Specifically... `sec-checker` is *not* a valid package name! In the Composer world, *every* package *must* be `something/something-else`: it can't just be `sec-checker`. So what the heck is going on?

Back in PhpStorm, open up `composer.json`. When we started the project, we had just a *few* dependencies in this file. One of them is `symfony/flex`.

```
composer.json
```

```
1 {
2 // ... lines 2 - 3
4     "require": {
5         "php": "^7.2.5",
6         "ext-ctype": "*",
7         "ext-iconv": "*",
8         "sensio/framework-extra-bundle": "^5.5",
9         "sensiolabs/security-checker": "^6.0",
10        "symfony/console": "5.0.*",
11        "symfony/dotenv": "5.0.*",
12        "symfony/flex": "^1.3.1",
13        "symfony/framework-bundle": "5.0.*",
14        "symfony/yaml": "5.0.*"
15    },
16 // ... lines 16 - 67
68 }
```

This is a composer *plugin* that adds *two* special features to Composer itself. The first is called "aliases".

At your browser, go to <http://flex.symfony.com> to find and big page full of packages.

💡 Tip

The flex.symfony.com server was shut down in favor of a new system. But you can still see a list of all of the available recipes at <https://bit.ly/flex-recipes>!

Search for `security`. Better, search for `sec-checker`. Boom! This says that there is a package called `sensiolabs/security-checker` and it has aliases of `sec-check`, `sec-checker`, `security-checker` and some more.

The alias system is simple: because Symfony Flex is in our app, we can say `composer require security-checker`, and it will *really* download `sensiolabs/security-checker`.

You can see this in our terminal: we said `sec-checker`, but ultimately it downloaded `sensiolabs/security-checker`. That's also what Composer added to our `composer.json` file. So... aliases are just a nice shortcut feature... but it's kinda cool! You can almost *guess* an alias when you want to install something. Want a logger? Run `composer require logger` to get the recommended logger. Need to mail something? `composer require mailer`. Need to eat a cake? `composer require cake`!

Flex Recipes

The *second* feature that Flex adds to Composer is the *really* important one. It's the recipe system.

Back at the terminal, after installing the package, it said:

```
"Symfony operations: 1 recipe configuring sensiolabs/security-checker."
```

Interesting. Run:

```
git status
```

Whoa! We expected `composer.json` and `composer.lock` to be modified... that's how composer works. But something *also* modified a `symfony.lock` file... and added a totally *new* `security_checker.yaml` file!

Ok, first, `symfony.lock` is a file that's managed by Flex. You don't need to worry about it, but you *should* commit it. It keeps a big list of which recipes have been installed.

So, who created the other file? Open it up: `config/packages/security_checker.yaml`.

```
config/packages/security_checker.yaml
```

```
1 services:
2     _defaults:
3         autowire: true
4         autoconfigure: true
5
6     SensioLabs\Security\SecurityChecker: null
7
8     SensioLabs\Security\Command\SecurityCheckerCommand: null
```

Each package you install *may* have a Flex "recipe". The idea is *beautifully* simple. Instead of telling people to install a package and *then* create this file, and update this other file in order to get things working, Flex executes a *recipe* which... just does that stuff for you! This file was added by the `sensiolabs/security-checker` recipe!

You don't need to worry about the specifics of what's *inside* this file right now. The point is, *thanks* to this file, we have a new `bin/console` command. Run:

```
php bin/console
```

See that `security:check` command? That wasn't there a second ago. It's there *now* thanks to the new YAML file. Try it:

```
php bin/console security:check
```

No packages have known vulnerabilities! Awesome!

How Recipes Work

Here is the *big* picture: thanks to the recipe system, whenever you install a package, Flex will check to see if that package has a recipe and, if it does, will install it. A recipe can do many things, like add files, create directories or even *modify a few* files, like adding new lines to your `.gitignore` file.

The recipe system is a *game-changer*. I *love* it because anytime I need a new package, all I need to do is install it. I don't need to add configuration files or modify anything because the recipe automates all that boring work.

Recipes can Modify Files

In fact, this recipe did something *else* we didn't notice. At the terminal, run:

```
git diff composer.json
```

We expected that Composer would add this new line to the `require` section. But there is *also* a new line under the `scripts` section. That was done by the recipe.

composer.json

```
1 {
2 // ... lines 2 - 3
4     "require": {
5 // ... lines 5 - 8
9         "sensiolabs/security-checker": "^6.0",
10 // ... lines 10 - 14
15     },
16 // ... lines 16 - 45
46     "scripts": {
47         "auto-scripts": {
48 // ... lines 48 - 49
50             "security-checker security:check": "script"
51         },
52 // ... lines 52 - 57
58     },
59 // ... lines 59 - 67
68 }
```

Thanks to this, whenever you run `composer install` after it finishes, it automatically runs the security checker.

Tip

Running `composer install` will fail with 403 API error. It's ok, we will remove security checker in the next chapter so it won't be an issue. If you want to know more, see <https://github.com/sensiolabs/security-checker>

The point is: to use the security checker, the *only* thing we needed to do was... install it. Its recipe took care of the rest of the setup.

Now... if you're wondering:

"Hey! Where the heck does this recipe live? Can I see it?"

That's a *great* question! Let's find out where these recipes live and what they look like next.

Chapter 7: How Recipes Work

Where do these Flex recipes live? They live... in the *cloud*. More specifically, if you look back at <https://flex.symfony.com>, you can click to view the Recipe for any of the packages.

Tip

The flex.symfony.com server was shut down in favor of a new system. But you can still see a list of all of the available recipes at <https://bit.ly/flex-recipes>!

This goes to... interesting: a GitHub repository called `symfony/recipes`.

Go to the homepage of this repository. This is *the* central repository for recipes, organized by the name of the package... and then each package can have different recipes for different versions. Our recipe lives in `sensiolabs/security-checker/4.0`.

Looking at the Source of a Recipe

Every recipe has *at least* this `manifest.json` file, which describes all of the "things" it should do. This `copy-from-recipe` says that the contents of the `config/` directory in the recipe should be copied into our project. *This* is why a `config/packages/security_checker.yaml` file was added to our app.

Back in the manifest, the `composer-scripts` section tells Flex to add this line to our `composer.json` file... and `aliases` define... well... the aliases that should *map* to this package.

There *are* a few other things that a recipe can do, but this is the basic idea.

So... *all* Symfony recipes live in this *one* repository. Hmm, actually, that's not true: all Symfony recipes live in this repository *or* in another one called `recipes-contrib`. There's no difference between these, except that quality control is higher on recipes merged into the *main* repository.

Using Composer to View Recipes

Another way you can see details about the recipes is via Composer itself. Run:

```
composer recipes
```

These are the 7 recipes that have been installed into our app. And if we run:

```
composer recipes sensiolabs/security-checker
```

We can see more details, like the URL to the recipe and files it copied into our app.

Anyways, the recipe system is going to be our *best* friend: allowing our app to start tiny, but grow *automatically* when we install new packages.

Removing a Package & Recipe

Oh, and if you decide that you want to *remove* a package, its recipe will be *uninstalled*. Check it out:

```
composer remove sec-checker
```

That - of course - will remove the package... but it *also* "unconfigured" the recipe. When we run:

```
git status
```

It's clean! It reverted the change in `composer.json` and removed the config file.

composer.json

```
1 {
2 // ... lines 2 - 3
3
4     "require": {
5         "php": "^7.2.5",
6         "ext-ctype": "*",
7         "ext-iconv": "*",
8         "sensio/framework-extra-bundle": "^5.5",
9         "symfony/console": "5.0.*",
10        "symfony/dotenv": "5.0.*",
11        "symfony/flex": "^1.3.1",
12        "symfony/framework-bundle": "5.0.*",
13        "symfony/yaml": "5.0.*"
14    },
15 // ... lines 15 - 44
16
17    "scripts": {
18        "auto-scripts": {
19            "cache:clear": "symfony-cmd",
20            "assets:install %PUBLIC_DIR%": "symfony-cmd"
21        },
22 // ... lines 50 - 55
23    },
24 // ... lines 57 - 65
25 }
26 }
```

Next: let's install Twig - Symfony's templating engine - so we can create HTML templates. The Twig recipe is going to make this so easy.

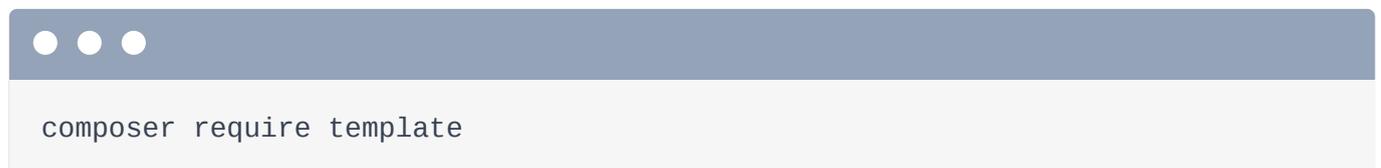
Chapter 8: The Twig Recipe

Unless you're building a pure API - and we *will* talk about returning JSON later in this tutorial - you're going to need to write some HTML. And... putting text or HTML in a controller like this is... ugly.

No worries! Symfony has *great* integration with an *incredible* template library called Twig. There's just one problem: our Symfony app is so small that Twig isn't even installed yet! Ah, but that's not *really* a problem... thanks to the recipe system.

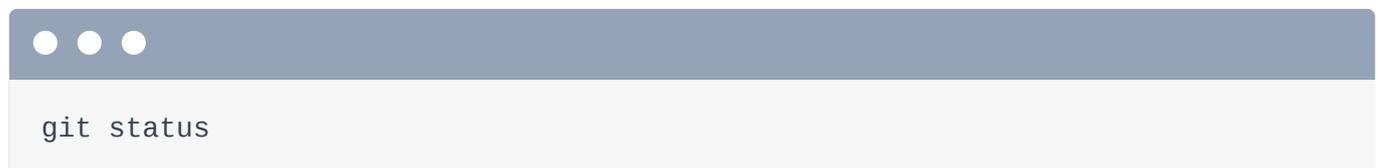
Installing Twig

Head back to <https://flex.symfony.com> and search for "template". There it is! Apparently Symfony's recommended "template" library is something called `twig-pack`. Let's install it!



```
composer require template
```

This installs a few packages... and yea! 2 recipes! Let's see what they did:



```
git status
```

Checking out the Recipe Changes

Whoa, awesome. Okay: we expected changes to `composer.json`, `composer.lock` and `symfony.lock`. Everything *else* was done by those recipes.

What are Bundles?

Let's look at `bundles.php` first:

```
git diff config/bundles.php
```

Interesting: it added two lines. Go open that: `config/bundles.php`.

```
config/bundles.php
```

```
↕ // ... lines 1 - 2
```

```
3 return [
4     Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' =>
true],
5     Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle::class
=> ['all' => true],
6     Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
7     Twig\Extra\TwigExtraBundle\TwigExtraBundle::class => ['all' => true],
8 ];
```

A "bundle" is a Symfony *plugin*. Pretty commonly, when you want to add a new feature to your app, you'll install a bundle. And when you install a bundle, you need to *enable* it in your application. A long time ago, doing this was manual. But thanks to Symfony Flex, whenever you install a Symfony bundle, it automatically updates this to enable it for you. So... now that we've talked about this file, you'll probably *never* need to think about it again.

The templates/ Directory and Config

The recipe *also* added a `templates/` directory. So if you were wondering where your templates are supposed to live... the recipe kinda answered that question! It also added a `base.html.twig` layout file that we'll talk about soon.

```
templates/base.html.twig
```

```
1 <!DOCTYPE html>
2 <html>
3     <head>
4         <meta charset="UTF-8">
5         <title>{% block title %}Welcome!{% endblock %}</title>
6         {% block stylesheets %}{% endblock %}
7     </head>
8     <body>
9         {% block body %}{% endblock %}
10        {% block javascripts %}{% endblock %}
11    </body>
12 </html>
```

So... apparently our templates are supposed to live in `templates/`. But why? I mean, is that path hardcoded deep in some core Twig file? Nope! It lives right in *our* code, thanks to a `twig.yaml` file that was created by the recipe. Let's check that out:
`config/packages/twig.yaml`.

```
config/packages/twig.yaml
1 twig:
2   default_path: '%kernel.project_dir%/templates'
```

We're going to talk more about these YAML files in another tutorial. But without understanding a lot about this file, it... already makes sense! This `default_path` config points to the `templates/` directory. Want your templates to live somewhere else? Just change this and... done! You're in control.

By the way, don't worry about this weird `%kernel.project_dir%` syntax. We'll learn about that later. But basically, it's a fancy way to point to the root of our project.

The recipe also created one other `twig.yaml` file which is less important:
`config/packages/test/twig.yaml`. This makes a *tiny* change to Twig inside your automated tests. The details don't really matter. The point is: we installed Twig and its recipe handled everything else. We are 100% ready to use it in our app. Let's do that next.

Chapter 9: Twig

Let's make our `show()` controller render some *real* HTML by using a template. As *soon* as you want to render a template, you need to make your controller extend `AbstractController`. Don't forget to let PhpStorm auto-complete this so it adds the `use` statement.

Now, obviously, a controller doesn't *need* to extend this base class - Symfony doesn't really care about that. *But*, you usually *will* extend `AbstractController` for one simple reason: it gives us shortcut methods!

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 4
5 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
↕ // ... lines 6 - 8
9 class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 27
28 }
```

Rendering a Template

The first useful shortcut method is `render`. We can say: `return this->render()` and pass two arguments. The first is the filename of the template: we can put anything here, but usually - because we value our sanity - we name this after our controller:

```
question/show.html.twig.
```

The second argument is an array of any variables that we want to pass into the template. Eventually, we're going to query the database for a specific question and pass that data into the template. Right now, let's fake it. I'll copy my `ucwords()` line and delete the old code. Let's pass a variable into the template called - how about, `question` - set to this string.

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 8
9 class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 21
22     public function show($slug)
23     {
24         return $this->render('question/show.html.twig', [
25             'question' => ucwords(str_replace('-', ' ', $slug))
26         ]);
27     }
28 }
```

Pop quiz time! What do you think that `render()` method returns? A string? Something else? The answer is: a `Response` object... with HTML inside. Because remember: the *one* rule of a controller is that it must *always* return a `Response`.

💡 Tip

A controller *can* actually return something *other* than a `Response`, but don't worry about that right now... or maybe ever.

Creating the Template

Anyways, let's go create that template! Inside `templates/`, create a `question` sub-directory, then a new file called `show.html.twig`. Let's start simple: an `<h1>` and then `{{ question }}` to render the question *variable*. And... I'll put some extra markup below this.

```
templates/question/show.html.twig
```

```
1 <h1>{{ question }}</h1>
2
3 <div>
4     Eventually, we'll print the full question here!
5 </div>
6
```

The 3 Syntaxes of Twig!

We *just* wrote our first Twig code! Twig is *super* friendly: it's a plain HTML file until you write one of its *two* syntaxes.

The first is the "say something" syntax. Anytime you want to print something, use `{{`, the thing you want to print, then `}}`. Inside the curly braces, you're writing Twig code... which is a lot like JavaScript. This prints the `question` variable. If we put quotes around it, it would print the *string* `question`. And yea, you can do more complex stuff - like the ternary operator. Again, it's very much like JavaScript.

The *second* syntax I call the "do something" syntax. It's `{%` followed by whatever you need to do, like `if` or `for` to do a loop. We'll talk more about this in a second.

And... that's it! You're either printing something with `{{` or *doing* something, like an `if` statement, with `{%`.

Ok, *small lie*, there *is* a third syntax... but it's just comments: `{#`, a comment... then `#}`.

```
templates/question/show.html.twig
```

```
1 <h1>{{ question }}</h1>
2
3 {# oh, I'm just a comment hiding here #}
4
5 <div>
6     Eventually, we'll print the full question here!
7 </div>
8
```

Let's see if this works! Move over refresh and... got it! If you view the HTML source, notice that there is *no* HTML layout yet. It's literally the markup from our template and nothing else. We'll add a layout in a few minutes.

Looping with the `{%` for Tag

Ok: we have a fake question. I think it deserves some fake answers! Back in the controller, up in the `show()` action, I'm going to paste in three fake answers.

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 8
9 class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 21
22     public function show($slug)
23     {
24         $answers = [
25             'Make sure your cat is sitting purrrfectly still ?',
26             'Honestly, I like furry shoes better than MY cat',
27             'Maybe... try saying the spell backwards?',
28         ];
↕ // ... lines 29 - 33
34     }
35 }
```

Again, once we talked about databases, we will query the database for these. But this will work beautifully to start. Pass these into the template as a *second* variable called `answers`.

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 8
9 class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 21
22     public function show($slug)
23     {
24         $answers = [
25             'Make sure your cat is sitting purrrfectly still ?',
26             'Honestly, I like furry shoes better than MY cat',
27             'Maybe... try saying the spell backwards?',
28         ];
29
30         return $this->render('question/show.html.twig', [
31             'question' => ucwords(str_replace('-', ' ', $slug)),
32             'answers' => $answers,
33         ]);
34     }
35 }
```

Back in the template, how can we print those? We can't just say `{{ answers }}`... because it's an array. What we *really* want to do is loop over that array and print each individual answer. To do that, we get to use our first "do something" tag! It looks like this:

```
{% for answer in answers %}. And most "do something" tags also have an end tag:
{% endfor %}.
```

Let's surround this with a `ul` and, inside the loop, say `` and `{{ answer }}`.

```
templates/question/show.html.twig
↕ // ... lines 1 - 8
9 <h2>Answers</h2>
10
11 <ul>
12     {% for answer in answers %}
13         <li>{{ answer }}</li>
14     {% endfor %}
15 </ul>
↕ // ... lines 16 - 17
```

I love that! Ok browser, reload! It works! I mean, it's so, so, ugly... but we'll fix that soon.

The Twig Reference: Tags, Filters, Functions

Head to <https://twig.symfony.com>. Twig is its own library with its *own* documentation. There's a lot of good stuff here... but what I *really* love is down here: the Twig Reference.

See these "Tags" on the left? These are *all* of the "do something" tags that exist. Yep, it will *always* be `{%` and then *one* of these words - like `for`, `if` or `{% set`. If you try `{% pizza`, I'll think it's funny, but Twig will yell at you.

Twig also has functions... like every language... and a cool feature called "tests", which is a bit unique. These allow you to say things like: `if foo is defined` or `if number is even`.

But the *biggest* and *coolest* section is for "filters". Filters are basically functions... but more hipster. Check out the `length` filter. Filters work like "pipes" on the command line: we "pipe" the `users` variable into the `length` filter, which counts it. The value goes from left to right. Filters are really functions... with a friendlier syntax.

Let's use this filter to print out the *number* of answers. I'll add some parenthesis, then `{{ answers|length }}`. When we try that... super nice!

```
templates/question/show.html.twig
↕ // ... lines 1 - 7
8
9 <h2>Answers {{ answers|length }}</h2>
10
↕ // ... lines 11 - 17
```

Twig Template Inheritance: extends

At this point, you're *well* on your way to being a Twig pro. There's just *one* last big feature we need to talk about, and it's a good one: template inheritance.

Most of our pages will share an HTML layout. Right now, we don't have *any* HTML structure. To give it some, at the *top* of the template, add `{% extends 'base.html.twig' %}`.

```
templates/question/show.html.twig
1  {% extends 'base.html.twig' %}
2
3  <h1>{{ question }}</h1>
↕  // ... lines 4 - 19
```

This tells Twig that we want to use this `base.html.twig` template as our layout. This file is *super* basic right now, but it's *ours* to customize - and we will soon.

But if you refresh the page... hide! Huge error!

“A template that extends another one cannot include content outside Twig blocks.”

When you add `extends` to a template, you're saying that you want the content from this template to go *inside* of `base.html.twig`. But... where? Should Twig put it all the way on top? On the bottom? Somewhere in the middle? Twig doesn't know!

I'm sure you already noticed these `block` things, like `stylesheets`, `title` and `body`. Blocks are "holes" that a child template can put content *into*. We can't *just* extend `base.html.twig`: we need to tell it which *block* the content should go into. The `body` block is a perfect spot.

How do we do this? By *overriding* the block. Above the content add `{% block body %}`, and after, `{% endblock %}`.

```
templates/question/show.html.twig
1  {% extends 'base.html.twig' %}
2
3  {% block body %}
↕  // ... lines 4 - 18
19 {% endblock %}
↕  // ... lines 20 - 21
```

Try it now. It works! It doesn't look like much yet... because our base layout is so simple, but if you check out the page source, we *do* have the basic HTML structure.

Adding, Removing, Changing Blocks?

By the way, these blocks in `base.html.twig` aren't special: you can rename them, move them around, add more or remove some. The more blocks you add, the more flexibility your "child" templates have to put content into different spots.

Most of the existing blocks are empty... but a block *can* define *default* content... like the `title` block. See this `Welcome`? No surprise, that's the current `title` of the page.

Because this is surrounded by a block, we can *override* that in any template. Check it out: anywhere in `show.html.twig`, add `{% block title %}`, Question, print the question, then `{% endblock %}`.

```
templates/question/show.html.twig
```

```
1  {% extends 'base.html.twig' %}
2
3  {% block title %}Question: {{ question }}{% endblock %}
4
5  {% block body %}
↕ // ... lines 6 - 20
21 {% endblock %}
↕ // ... lines 22 - 23
```

This time when we reload... we have a *new* title!

Ok, with Twig behind us, let's look at one of the *killer* features of Symfony... and your new best friend for debugging: the Symfony profiler.

Chapter 10: Profiler: Your Debugging Best Friend

We're making some *pretty* serious progress - you should be proud! Let's check out what files we've modified:

```
git status
```

Add everything:

```
git add .
```

And commit:

```
git commit -m "Added some Twiggy goodness"
```

Installing the Profiler

Because *now* I want to install one of my *absolute* favorite tools in Symfony. Run:

```
composer require profiler --dev
```

I'm using `--dev` because the profiler is a tool that we'll only need while we're *developing*: it won't be used on production. This means Composer adds it to the `require-dev` section of `composer.json`. This isn't that important, but this is the right way to do it.

Tip

In newer projects, instead of `symfony/profiler-pack`, you may see 3 packages here, including `symfony/web-profiler-bundle`. That's ok! We'll explain what's going on in a few minutes.

```
composer.json
```

```
1 {
  // ... lines 2 - 15
16     "require-dev": {
17         "symfony/profiler-pack": "^1.0"
18     },
  // ... lines 19 - 67
68 }
```

And... at this point, it should be *no* surprise that this configured a recipe! Run:

```
git status
```

Hello Web Debug Toolbar

Oh, wow! It added *three* config files. Thanks to these, the feature will work *instantly*. Try it: back at your browser, refresh the page. Hello web debug toolbar! The fancy little black bar on the bottom. This will now show up on *every* HTML page while we're developing. It tells us the status code, which controller and route were used, speed, memory, Twig calls and even *more* icons will show up as we start using more parts of Symfony.

And Hello Profiler

The *best* part is that you can click any of these icons to jump into... the *profiler*. This is basically the expanded version of the toolbar and it is *packed* with information about that page load, including request info, response info and even a super-cool performance tab. This is not *only* a nice way to debug the performance of your app, it's *also* a great way to... just understand what's going on inside Symfony.

There are other sections for Twig, configuration, caching and eventually there will be a tab to debug database queries. By the way, this isn't just for HTML pages: you can *also* access the profiler for AJAX calls that you make to your app. I'll show you how later.

The dump() and dd() Functions

When we installed the profiler, we also got one other handy tool called `dump()`. I'll click back a few times to get to the page. Open up the controller:

```
src/Controller/QuestionController.php.
```

Imagine we want to debug a variable. Normally I'd use `var_dump()`. Instead, use `dump()` and let's dump the `$slug` and... how about `$this` object itself.

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 8
9  class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 21
22     public function show($slug)
23     {
↕ // ... lines 24 - 29
30         dump($slug, $this);
↕ // ... lines 31 - 35
36     }
37 }
```

When we refresh, woh! It works *exactly* like `var_dump()` except... way more beautiful and useful. The controller apparently has a `container` property... and we can dig deeper and deeper.

If you're *really* lazy... like most of us are... you can also use `dd()` which stands for `dump()` and `die()`.

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 8
9 class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 21
22     public function show($slug)
23     {
↕ // ... lines 24 - 29
30         dd($slug, $this);
↕ // ... lines 31 - 35
36     }
37 }
```

Now when we reload... it dumps, but *also* kills the page. We've now perfected dump-and-die-driven development. I think we should be proud?

Installing the debug Package

Change that back to `dump()`... and let's *just* `dump()` `$this`.

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 8
9 class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 21
22     public function show($slug)
23     {
↕ // ... lines 24 - 29
30         dump($this);
↕ // ... lines 31 - 35
36     }
37 }
```

There's *one* other library that we can install for debugging tools. This one is less important - but still nice to have. At your terminal, run:

```
composer require debug
```

This time I'm *not* using `--dev` because this will install something that I *do* want on production. It installs DebugBundle - that's not something we need on production - but *also* Monolog, which

is a logging library. And we probably *do* want to log things on production.

Composer Packs?

Before we talk about what this gave us, check out the name of the package it installed:

`debug-pack`. This is not the first time that we've installed a library with "pack" in its name.

A "pack" is a special concept in Symfony: it's sort of a "fake" package whose only job is to help install several packages at once. Check it out: copy the package name, find your browser, and go to <https://github.com/symfony/debug-pack>. Woh! It's nothing more than a `composer.json` file! This gives us an easy way to install just *this* package... but actually get *all* of these libraries.

💡 Tip

In my project, installing a "pack" would add just *one* line to `composer.json`:
`symfony/debug-pack`. But starting in `symfony/flex` 1.9, when you install a pack, instead of adding `symfony/debug-pack` to `composer.json`, it will add these 5 packages instead. You still get the same code, but this makes it easier to manage the package versions.

So thanks to this, we have two new things in our app. The first is a logger! If we refresh the page... and click into the profiler, we have a "Logs" section that shows us *all* the logs for that request. These are *also* being saved to a `var/log/dev.log` file.

The second new thing in our app is... well... if you were watching closely, the `dump()` is gone from the page! The DebugBundle integrates the `dump()` function even *more* into Symfony. Now if you use `dump()`, instead of printing in the middle of the page, it puts it down here on the web debug toolbar. You can click it to see a bigger version. It's not that important... just another example of how Symfony gets smarter as you install more stuff.

The server:dump Command

Oh, while we're talking about it, the DebugBundle gave us one handle new console command. At your terminal, run:

```
php bin/console server:dump
```

This starts a little server in the background. Now whenever `dump()` is called in our code, it still shows up on the toolbar... but it *also* gets dumped out in the terminal! That's a great way to see dumped data for AJAX requests. I'll hit Control-C to stop that.

Unpacking Packs

Oh, and about these "packs", if you open your `composer.json` file, the one problem with packs is that we only have `debug-pack` version `1.0` here: we can't control the versions of the packages inside. You just get whatever versions the pack allows.

```
composer.json
```

```
1 {
2 // ... lines 2 - 3
4     "require": {
5 // ... lines 5 - 9
10         "symfony/debug-pack": "^1.0",
11 // ... lines 11 - 15
16     },
17 // ... lines 17 - 68
69 }
```

If you need more control, no problem... just unpack the pack:

```
composer unpack symfony/debug-pack
```

That does exactly what you expect: it removes `debug-pack` from `composer.json` and *adds* its underlying packages, like `debug-bundle` and `monolog`. Oh, and because the `profiler-pack` is a dependency of the `debug-pack`, it's in both places. I'll remove the extra one from `require`.

composer.json

```
1 {
2 // ... lines 2 - 3
3
4     "require": {
5         "php": "^7.2.5",
6         "ext-ctype": "*",
7         "ext-iconv": "*",
8         "easycorp/easy-log-handler": "^1.0.7",
9         "sensio/framework-extra-bundle": "^5.5",
10        "symfony/console": "5.0.*",
11        "symfony/debug-bundle": "5.0.*",
12        "symfony/dotenv": "5.0.*",
13        "symfony/flex": "^1.3.1",
14        "symfony/framework-bundle": "5.0.*",
15        "symfony/monolog-bundle": "^3.0",
16        "symfony/profiler-pack": "*",
17        "symfony/twig-pack": "^1.0",
18        "symfony/var-dumper": "5.0.*",
19        "symfony/yaml": "5.0.*"
20    },
21 // ... lines 21 - 72
22
23 }
```

Next, let's make our site prettier by bringing CSS into our app.

Chapter 11: Assets: CSS, Images, etc

We're doing really well, but yikes! Our site is *ugly*. Time to fix that.

If you download the course code from this page, after you unzip it, you'll find a `start/` directory with a `tutorial/` directory inside: the same `tutorial/` directory you see here. We're going to copy a few files from it over the next few minutes.

Copying the Base Layout & Main CSS File

The first is `base.html.twig`. I'll open it up, copy its contents, close it, and then open *our* `templates/base.html.twig`. Paste the new stuff here.

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8">
5     <title>{% block title %}Welcome!{% endblock %}</title>
6     {% block stylesheets %}
7       <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.c
integrity="sha384-
Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
crossorigin="anonymous">
8       <link rel="stylesheet" href="https://fonts.googleapis.com/css?
family=Spartan&display=swap">
9       <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/5.12.1/css/all.min.css" integrity="sha256-
mmgLkCYLUQbXn0B1SRqzHar6dCnv9oZFPec1g1cwlkk=" crossorigin="anonymous" />
10      <link rel="stylesheet" href="/css/app.css">
11    {% endblock %}
12  </head>
13  <body>
14    <nav class="navbar navbar-light bg-light" style="height: 100px;">
15      <a class="navbar-brand" href="#">
16        <i style="color: #444; font-size: 2rem;" class="pb-1 fad fa-
cauldron"></i>
17        <p class="pl-2 d-inline font-weight-bold" style="color:
#444;">Cauldron Overflow</p>
18      </a>
19      <button class="btn btn-dark">Sign up</button>
20    </nav>
21
22    {% block body %}{% endblock %}
23    <footer class="mt-5 p-3 text-center">
24      Made with <i style="color: red;" class="fa fa-heart"></i> by
the guys and gals at <a style="color: #444; text-decoration: underline;"
href="https://symfonycasts.com">SymfonyCasts</a>
25    </footer>
26    {% block javascripts %}{% endblock %}
27  </body>
28 </html>
```

This was *not* a huge change: this added some CSS files - including Bootstrap - and some basic HTML markup. But we have the same blocks as before: `{% block body %}` in the middle, `{% block javascripts %}`, `{% block title %}`, etc.

Notice that the link tags are *inside* a block called `stylesheets`. But that's not important yet. I'll explain why it's done that way a bit later.

```
templates/base.html.twig
1 <!DOCTYPE html>
2 <html>
3   <head>
4   // ... lines 4 - 5
6     {% block stylesheets %}
7       <link rel="stylesheet"
8         href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.c
9         integrity="sha384-
10        Vkoo8x4CGs03+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
11        crossorigin="anonymous">
12       <link rel="stylesheet" href="https://fonts.googleapis.com/css?
13         family=Spartan&display=swap">
14       <link rel="stylesheet"
15         href="https://cdnjs.cloudflare.com/ajax/libs/font-
16         awesome/5.12.1/css/all.min.css" integrity="sha256-
17         mmgLkCYLUQbXn0B1SRqzHar6dCnv9oZFPEC1g1cwlkk=" crossorigin="anonymous" />
18       <link rel="stylesheet" href="/css/app.css">
19     {% endblock %}
20   </head>
21   // ... lines 13 - 27
28 </html>
```

One of the link tags is pointing to `/css/app.css`. That's *another* file that lives in this `tutorial/` directory. In fact, select the `images/` directory *and* `app.css` and copy both. Now, select the `public/` folder and paste. Add another `css/` directory and move `app.css` inside.

Remember: the `public/` directory is our document root. So if you need a file to be accessible by a user's browser, it needs to live here. The path `/css/app.css` will load this `public/css/app.css` file.

Let's see what this looks like! Spin over to your browser and refresh. *Much* better. The middle still looks terrible... but that's because we haven't added any markup to the template for this page.

Does Symfony Care about your Assets

So let me ask a question... and answer it: what features does Symfony offer when it comes to CSS and JavaScript? The answer is... none... or a lot!

Symfony has two different levels of integration with CSS and JavaScript. Right now, we're using the basic level. Really, right now, Symfony isn't doing *anything* for us: we created a CSS file, then added a very traditional link tag to it in HTML. Symfony is doing *nothing*: it's all up to you.

The *other, bigger* level of integration is to use something called Webpack Encore: a *fantastic* library that handles minification, Sass support, React or Vue.js support and many other things. I'll give you a crash course into Webpack Encore at the end of this tutorial.

But right now, we're going to keep it simple: you create CSS or JavaScript files, put them in the `public/` directory, and then create `link` or `script` tags that point to them.

The Not-So-Important `asset()` Function

Well, actually, even with this, "basic" integration, there is *one* small Symfony feature you should use.

Before I show you, go into your PhpStorm preference... and search again for "Symfony" to find the Symfony plugin. See this web directory option? Change that to `public/` - this was called `web/` in older versions of Symfony. This will give us better auto-completion soon. Hit "Ok".

Here's the deal: whenever you reference a static file on your site - like a CSS file, JavaScript file or image, instead of just putting `/css/app.css`, you should use a Twig function called `asset()`. So, `{{ asset() }}` and then the *same* path as before, but without the opening `/`: `css/app.css`.

```
templates/base.html.twig
↕ // ... line 1
2 <html>
3   <head>
↕ // ... lines 4 - 5
6     {% block stylesheets %}
↕ // ... lines 7 - 9
10       <link rel="stylesheet" href="{{ asset('css/app.css') }}">
11     {% endblock %}
12   </head>
↕ // ... lines 13 - 27
28 </html>
```

What does this super-cool-looking `asset()` function do? Almost... nothing. In fact, this will output the *exact* same path as before: `/css/app.css`.

So why are we bothering to use a function that does nothing? Well, it *does* do *two* things... which you may or may not care about. First, if you decide to deploy your app to a *subdirectory* of a domain - like `ILikeMagic.com/cauldron_overflow`, the `asset()` function will automatically prefix all the paths with `/cauldron_overflow`. *Super* great... if you care.

The *second* thing it does is more useful: if you decide to deploy your assets to a CDN, by adding one line to one config file, suddenly, Symfony will prefix *every* path with the URL to your CDN.

So... it's really not *that* important, but if you use `asset()` everywhere, you'll be happy later when you need it.

But... if we move over and refresh... surprise! It explodes!

"Did you forget to run `composer require symfony/asset`? Unknown function `asset`."

How cool is that? Remember, Symfony starts small: you install things *when* you need them. In this case, we're trying to use a feature that's not installed... so Symfony gives us the *exact* command we need to run. Copy it, move over and go:



```
composer require symfony/asset
```

When this finishes... move back over and... it works. If you look at the HTML source and search for `app.css`... yep! It's printing the same path as before.

Making the "show" page Pretty

Let's make the middle of our page look a bit nicer. Back in the `tutorial/` directory, open `show.html.twig`, copy its contents, close it, then open up our version: `templates/question/show.html.twig`. Paste the new code.


```

42         
43     </div>
44     <div class="mr-3 pt-2">
45         {{ answer }}
46         <p>-- Mallory</p>
47     </div>
48     <div class="vote-arrows flex-fill pt-2" style="min-
width: 90px;">
49         <a class="vote-up" href="#"><i class="far fa-
arrow-alt-circle-up"></i></a>
50         <a class="vote-down" href="#"><i class="far fa-
arrow-alt-circle-down"></i></a>
51         <span>+ 6</span>
52     </div>
53 </div>
54 </li>
55 {% endfor %}
56 </ul>
57 </div>
58 {% endblock %}

```

Once again, there's nothing important happening here: we're still overriding the same `title` and `body` blocks. We're still using the same `question` variable and we're still looping over the `answers` down here. There's just a lot of extra markup... which... ya know... makes things pretty.

When we refresh... see! Pretty! Back in the template, notice that this page has a few `img` tags... but these are *not* using the `asset()` function. Let's fix that. I'll use a shortcut! I can just type "tisha", hit tab and... boom! It takes care of the rest. Search for `img`... and replace this one too with "tisha". Wondering who tisha is? Oh, just one of the several cats we keep on staff here at SymphonyCasts. This one manages Vladimir.

```
templates/question/show.html.twig
```

```
↕ // ... lines 1 - 4
5  {% block body %}
6  <div class="container">
7      <div class="row">
8          <div class="col-12">
↕ // ... line 9
10         <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11             <div class="q-container-show p-4">
12                 <div class="row">
13                     <div class="col-2 text-center">
14                         
15                     </div>
↕ // ... lines 16 - 23
24                 </div>
25             </div>
26         </div>
27     </div>
28 </div>
↕ // ... lines 29 - 36
37     <ul class="list-unstyled">
38         {% for answer in answers %}
39             <li class="mb-4">
40                 <div class="d-flex justify-content-center">
41                     <div class="mr-2 pt-2">
42                         
43                     </div>
↕ // ... lines 44 - 52
53                 </div>
54             </li>
55         {% endfor %}
56     </ul>
57 </div>
58 {% endblock %}
```

By the way, in a real app, instead of these images being static files in our project, that might be files that users *upload*. Don't worry: we have an entire tutorial on [handling file uploads](#).

Make sure this works and... it does.

Styling the Homepage

The *last* page that we haven't styled is the homepage... which right now... prints some text. Open its controller: `src/Controller/QuestionController.php`. Yep! It's just `return new Response()` and text. We can do better. Replace this with `return $this->render()`. Let's call the template `question/homepage.html.twig`. And... right now... I don't think we need to pass any variables into the template... so I'll leave the second argument off.

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 8
9 class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 13
14     public function homepage()
15     {
16         return $this->render('question/homepage.html.twig');
17     }
↕ // ... lines 18 - 34
35 }
```

Inside `templates/question/`, create the new file: `homepage.html.twig`.

Most templates start the exact same way. Yay consistency! On top, `{% extends 'base.html.twig' %}`, `{% block body %}` and `{% endblock %}`. In between, add some markup so we can see if this is working.

```
templates/question/homepage.html.twig
1 {% extends 'base.html.twig' %}
2
3 {% block body %}
4     <h1>Voilà</h1>
5 {% endblock %}
```

Ok... refresh the page and... excellent! Except for the "this looks totally awful" part.

Let's steal some code from the `tutorial/` directory *one* last time. Open `homepage.html.twig`. This is *just* a bunch of hardcoded markup to make things look nicer. Copy it, close that file... and then paste it over our `homepage.html.twig` code.

```
1  {% extends 'base.html.twig' %}
2
3  {% block body %}
4  <div class="jumbotron-img jumbotron jumbotron-fluid">
5      <div class="container">
6          <h1 class="display-4">Your Questions Answered</h1>
7          <p class="lead">And even answers for those questions you didn't
think to ask!</p>
8      </div>
9  </div>
10 <div class="container">
11     <div class="row mb-3">
12         <div class="col">
13             <button class="btn btn-question">Ask a Question</button>
14         </div>
15     </div>
16     <div class="row">
17         <div class="col-12">
18             <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
19                 <div class="q-container p-4">
20                     <div class="row">
21                         <div class="col-2 text-center">
22                             
23                             <div class="d-block mt-3 vote-arrows">
24                                 <a class="vote-up" href="#"><i class="far
fa-arrow-alt-circle-up"></i></a>
25                                 <a class="vote-down" href="#"><i
class="far fa-arrow-alt-circle-down"></i></a>
26                             </div>
27                         </div>
28                         <div class="col">
29                             <a class="q-title" href="#"><h2>Reversing a
Spell</h2></a>
30                             <div class="q-display p-3">
31                                 <i class="fa fa-quote-left mr-3"></i>
32                                 <p class="d-inline">I've been turned into
a cat, any thoughts on how to turn back? While I'm adorable, I don't
really care for cat food.</p>
33                                 <p class="pt-4"><strong>--Tisha</strong>
</p>
34                             </div>
35                         </div>
36                     </div>
37                 </div>
38                 <a class="answer-link" href="#" style="color: #fff;">
39                     <p class="q-display-response text-center p-3">
```

```

40         <i class="fa fa-magic magic-wand"></i> 6 answers
41     </p>
42 </a>
43 </div>
44 </div>
45
46 <div class="col-12 mt-3">
47     <div class="q-container p-4">
48         <div class="row">
49             <div class="col-2 text-center">
50                 
51                 <div class="d-block mt-3 vote-arrows">
52                     <a class="vote-up" href="#"><i class="far fa-
arrow-alt-circle-up"></i></a>
53                     <a class="vote-down" href="#"><i class="far
fa-arrow-alt-circle-down"></i></a>
54                 </div>
55             </div>
56             <div class="col">
57                 <a class="q-title" href="#"><h2>Pausing a
Spell</h2></a>
58                 <div class="q-display p-3">
59                     <i class="fa fa-quote-left mr-3"></i>
60                     <p class="d-inline">I mastered the floating
card, but now how do I get it back to the ground?</p>
61                     <p class="pt-4"><strong>-- Jerry</strong></p>
62                 </div>
63             </div>
64         </div>
65     </div>
66     <a class="answer-link" href="#" style="color: #fff;">
67         <p class="q-display-response text-center p-3">
68             <i class="fa fa-magic magic-wand"></i> 15 answers
69         </p>
70     </a>
71 </div>
72 </div>
73 </div>
74 {% endblock %}
75

```

And now... it looks *much* better.

So that's the *basic* CSS and JavaScript integration inside of Symfony: you manage it yourself. Sure, you *should* use this `asset()` function, but it's not doing anything too impressive.

If you want more, you're in luck! In the *last* chapter, we'll take our assets up to the next level. You're going to love it.

Next: our site now has some links on it! And they all go nowhere! Let's learn how to generate URLs to routes.

Chapter 12: Generate URLs

Go back to the "show" page for a question. The logo on top is a link... that doesn't go anywhere yet. This *should* take us back to the homepage.

Because this is part of the layout, the link lives in `base.html.twig`. Here it is: `navbar-brand` with `href="#"`.

```
templates/base.html.twig
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 12
13 <body>
14 <nav class="navbar navbar-light bg-light" style="height: 100px;">
15 <a class="navbar-brand" href="#">
↕ // ... lines 16 - 17
18 </a>
↕ // ... line 19
20 </nav>
↕ // ... lines 21 - 26
27 </body>
28 </html>
```

To make this link back to the homepage, we can just change this to `/`, right? You *could* do this, but in Symfony, a better way is to ask Symfony to *generate* the URL to this route. That way, if we decide to change this URL later, all our links will update automatically.

Each Route Has a Name!

To see how to do that, find your terminal and run:

```
php bin/console debug:router
```

This lists every route in the system... and hey! Since the last time we ran this, there are a *bunch* of new routes. These power the web debug toolbar and the profiler and are added automatically

by the WebProfilerBundle when we're in `dev` mode.

Anyways, what I *really* want to look at is the "Name" column. *Every* route has an internal name, *including* the two routes that we made. Apparently their names are `app_question_homepage` and `app_question_show`. But... uh... where did those come from? I don't remember typing either of these!

So... every route *must* be given an internal name. But when you use annotation routes... it lets you cheat: it chooses a name *for* you based on the controller class and method... which is awesome!

But... as soon as you need to generate the URL to a route, I recommend giving it an *explicit* name, instead of relying on this auto-generated name, which could change suddenly if you rename your method. To give the route a name, add `name=""` and... how about: `app_homepage`.

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 8
9 class QuestionController extends AbstractController
10 {
11     /**
12      * @Route("/", name="app_homepage")
13      */
14     public function homepage()
15     {
↕ // ... line 16
17     }
↕ // ... lines 18 - 34
35 }
```

I like to keep my route names short, but `app_` makes it long enough that I could search my project for this string if I ever needed to.

Now if we run `debug:router` again:

```
php bin/console debug:router
```

Nice! We are in control of the route's name. Copy the `app_homepage` name and then go back to `base.html.twig`. The goal is simple, we want to say:

“Hey Symfony! Can you please give me the URL to the `app_homepage` route?”

To do that in Twig, use `{{ path() }}` and pass it the route name.

```
templates/base.html.twig
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 12
13 <body>
14     <nav class="navbar navbar-light bg-light" style="height: 100px;">
15         <a class="navbar-brand" href="{{ path('app_homepage') }}">
↕ // ... lines 16 - 17
18     </a>
↕ // ... line 19
20 </nav>
↕ // ... lines 21 - 26
27 </body>
28 </html>
```

That's it! When we move over and refresh... *now* this links to the homepage.

Linking to a Route with {Wildcards}

On the homepage, we have two hard-coded questions... and each has two links that currently go nowhere. Let's fix these!

Step one: now that we want to generate a URL to this route, find the route and add `name="app_question_show"`.

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 8
9 class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 18
19     /**
20      * @Route("/questions/{slug}", name="app_question_show")
21      */
22     public function show($slug)
23     {
↕ // ... lines 24 - 33
34     }
35 }
```

Copy this and open the template: `templates/question/homepage.html.twig`. Let's see... right below the voting stuff, here's the first link to a "Reversing a spell" question. Remove the pound sign, add `{{ path() }}` and paste in `app_question_show`.

But... we can't stop here. If we try the page now, a glorious error!

"Some mandatory parameters are missing - "slug""

That makes sense! We can't just say "generate the URL to `app_question_show`" because that route has a wildcard! Symfony needs to know what value it should use for `{slug}`. How do we tell it? Add a *second* argument to `path()` with `{}`. The `{}` is a Twig associative array... again, just like JavaScript. Pass `slug` set to... let's see... this is a hardcoded question right now, so hardcode `reversing-a-spell`.

```
templates/question/homepage.html.twig
```

```
↕ // ... lines 1 - 2
3  {% block body %}
↕ // ... lines 4 - 9
10 <div class="container">
↕ // ... lines 11 - 15
16     <div class="row">
17         <div class="col-12">
18             <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
19                 <div class="q-container p-4">
20                     <div class="row">
↕ // ... lines 21 - 27
28                         <div class="col">
29                             <a class="q-title" href="{{
path('app_question_show', { slug: 'reversing-a-spell' }) }}"><h2>Reversing
a Spell</h2></a>
↕ // ... lines 30 - 34
35                             </div>
36                         </div>
37                     </div>
38                     <a class="answer-link" href="{{ path('app_question_show',
{ slug: 'reversing-a-spell' }) }}" style="color: #fff;">
↕ // ... lines 39 - 41
42                         </a>
43                     </div>
44                 </div>
↕ // ... lines 45 - 71
72             </div>
73 </div>
74 {% endblock %}
↕ // ... lines 75 - 76
```

Copy that *entire* thing, because there's one other link down here for the same question. For the second question... paste again, but change it to `pausing-a-spell` to match the name. I'll copy that... find the last spot... and paste.

```
templates/question/homepage.html.twig
```

```
↕ // ... lines 1 - 2
3  {% block body %}
↕ // ... lines 4 - 9
10 <div class="container">
↕ // ... lines 11 - 15
16     <div class="row">
↕ // ... lines 17 - 45
46         <div class="col-12 mt-3">
47             <div class="q-container p-4">
48                 <div class="row">
↕ // ... lines 49 - 55
56                     <div class="col">
57                         <a class="q-title" href="{{
path('app_question_show', { slug: 'pausing-a-spell' }) }}"><h2>Pausing a
Spell</h2></a>
↕ // ... lines 58 - 62
63                             </div>
64                         </div>
65                     </div>
66                         <a class="answer-link" href="{{ path('app_question_show', {
slug: 'pausing-a-spell' }) }}" style="color: #fff;">
↕ // ... lines 67 - 69
70                             </a>
71                         </div>
72                     </div>
73 </div>
74 {% endblock %}
↕ // ... lines 75 - 76
```

Later, when we introduce a database, we'll make this fancier and avoid repeating ourselves so many times. But! If we move over, refresh... and click a link, it works! Both pages go to the same route, but with a different slug value.

Next, let's take our site to the next level by creating a JSON API endpoint that we will consume with JavaScript.

Chapter 13: JSON API Endpoint

One of the features on our site... which doesn't work yet... is that you can up and down vote answers to a question. Eventually, when you click up or down, this will make an AJAX request to an API endpoint that we will make. That endpoint will save the vote to the database and *respond* with JSON that contains the *new* vote count so that our JavaScript can update this vote number.

We don't have a database in our app yet, but we're ready to build every other part of this feature.

Creating a JSON Endpoint

Let's start by creating a JSON API endpoint that will be hit via AJAX when a user up or down votes an answer.

We *could* create this in `QuestionController` as a new method. But since this endpoint *really* deals with a "comment", let's create a *new* controller class. Call it `CommentController`.

Like before, we're going to say `extends AbstractController` and hit tab so that PhpStorm autocompletes this and adds the `use` statement on top. Extending this class gives us shortcut methods... and I love shortcuts!

```
src/Controller/CommentController.php
```

```
↕ // ... lines 1 - 2
3 namespace App\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6
7 class CommentController extends AbstractController
8 {
9 }
```

Inside, create a `public function`. This could be called anything... how about `commentVote()`. Add the route above: `/**`, then `@Route`. Auto-complete the one from the Routing component so that PhpStorm adds its `use` statement.

For the URL, how about `/comments/{id}` - this will eventually be the id of the specific comment in the database - `/vote/{direction}`, where `{direction}` will either be the word `up` or the word `down`.

And because we have these two wildcards, we can add two arguments: `$id` and `$direction`. I'll start with a comment: the `$id` will be *super* important later when we have a database... but we won't use it at all right now.

```
src/Controller/CommentController.php
↕ // ... lines 1 - 6
7 use Symfony\Component\Routing\Annotation\Route;
↕ // ... line 8
9 class CommentController extends AbstractController
10 {
11     /**
12      * @Route("/comments/{id}/vote/{direction}")
13      */
14     public function commentVote($id, $direction)
15     {
↕ // ... lines 16 - 25
26     }
27 }
```

Without a database, we'll kinda fake the logic. If `$direction === 'up'`, then we would normally save this up-vote to the database and query for the new vote count. Instead, say `$currentVoteCount = rand(7, 100)`.

```
src/Controller/CommentController.php
```

```
↕ // ... lines 1 - 8
9 class CommentController extends AbstractController
10 {
↕ // ... lines 11 - 13
14     public function commentVote($id, $direction)
15     {
16         // todo - use id to query the database
17
18         // use real logic here to save this to the database
19         if ($direction === 'up') {
20             $currentVoteCount = rand(7, 100);
21         } else {
↕ // ... line 22
23         }
↕ // ... lines 24 - 25
26     }
27 }
```

The vote counts in the template are hardcoded to 6. So this will make the new vote count appear to be *some* random number higher than that. In the else, do the opposite: a random number between 0 and 5.

```
src/Controller/CommentController.php
```

```
↕ // ... lines 1 - 8
9 class CommentController extends AbstractController
10 {
↕ // ... lines 11 - 13
14     public function commentVote($id, $direction)
15     {
16         // todo - use id to query the database
17
18         // use real logic here to save this to the database
19         if ($direction === 'up') {
20             $currentVoteCount = rand(7, 100);
21         } else {
22             $currentVoteCount = rand(0, 5);
23         }
↕ // ... lines 24 - 25
26     }
27 }
```

Yes, this will all be *much* cooler when we have a database, but it will work great for our purposes.

Returning JSON?

The question now is: after "saving" the vote to the database, what should this controller return? Well it should probably return JSON... and I *know* that I want to include the new vote count in its data so our JavaScript can use that to update the vote number text.

So... how do we return JSON? Remember: our *only* job in a controller is to return a Symfony `Response` object. JSON is nothing more than a response whose body is a JSON string instead of HTML. So we *could* say: `return new Response()` with `json_encode()` of some data.

But! Instead, `return new JsonResponse()` - auto-complete this so that PhpStorm adds the `use` statement. Pass this an array with the data we want. How about a `votes` key set to `$currentVoteCount`.

```
src/Controller/CommentController.php
↕ // ... lines 1 - 5
6 use Symfony\Component\HttpFoundation\JsonResponse;
↕ // ... lines 7 - 8
9 class CommentController extends AbstractController
10 {
↕ // ... lines 11 - 13
14     public function commentVote($id, $direction)
15     {
16         // todo - use id to query the database
17
18         // use real logic here to save this to the database
19         if ($direction === 'up') {
20             $currentVoteCount = rand(7, 100);
21         } else {
22             $currentVoteCount = rand(0, 5);
23         }
24
25         return new JsonResponse(['votes' => $currentVoteCount]);
26     }
27 }
```

Now... you may be thinking:

"Ryan! You keep saying that we must return a Response object... and you just returned something different. This is madness!"

Fair point. But! If you hold Command or Ctrl and click the `JsonResponse` class, you'll learn that `JsonResponse` extends `Response`. This class is nothing more than a shortcut for creating JSON responses: it JSON encodes the data we pass to it *and* makes sure that the `Content-Type` header is set to `application/json`, which helps AJAX libraries understand that we're returning JSON data.

So... ah! Let's test out our shiny-new API endpoint! Copy the URL, open a new browser tab, paste and fill in the wildcards: how about 10 for `{id}` and vote "up". Hit enter. Hello JSON endpoint!

The big takeaway is this: JSON responses are nothing special.

The json() Shortcut Method

The `JsonResponse` class makes life easier... but we can be even *lazier*! Instead of `new JsonResponse`, just say `return $this->json()`.

```
src/Controller/CommentController.php
↕ // ... lines 1 - 8
9 class CommentController extends AbstractController
10 {
↕ // ... lines 11 - 13
14     public function commentVote($id, $direction)
15     {
↕ // ... lines 16 - 24
25         return $this->json(['votes' => $currentVoteCount]);
26     }
27 }
```

That changes nothing: it's a shortcut method to create the *same* `JsonResponse` object. Easy peasy.

The Symfony Serializer

By the way, one of the "components" in Symfony is called the "Serializer", and it's *really* good at converting *objects* into JSON or XML. We don't have it installed yet, but if we *did*, the `$this->json()` would start using it to serialize whatever we pass. That wouldn't make any difference in our case with an array, but it means that you could start passing *objects* to

`$this->json()`. If you want to learn more - or want to build a super-rich API - check out our tutorial about [API Platform](#): an amazing Symfony bundle for building APIs.

Next, let's write some JavaScript that will make an AJAX call to our new endpoint. We'll also learn how to add global Javascript as *well* as page-specific JavaScript.

Chapter 14: JavaScript, AJAX & the Profiler

Here's our next goal: write some JavaScript so that that when we click the up or down vote icons, it will make an AJAX request to our JSON endpoint. This "fakes" saving the vote to the database and returns the new vote count, which we will use to update the vote number on the page.

Adding.js- Classes to the Template

The template for this page is: `templates/question/show.html.twig`. For each answer, we have these `vote-up` and `vote-down` links. I'm going to add a few classes to this section to help our JavaScript. On the `vote-arrows` element, add a `js-vote-arrows` class: we'll use that in JavaScript to find this element. Then, on the `vote-up` link, add a data attribute called `data-direction="up"`. Do the same for the down link: `data-direction="down"`. This will help us know which link was clicked. Finally, surround the vote number - the 6 - with a span that has another class: `js-vote-total`. We'll use that to find the element so we can update that number.

```
templates/question/show.html.twig
```

```
↕ // ... lines 1 - 4
5  {% block body %}
6  <div class="container">
↕ // ... lines 7 - 36
37     <ul class="list-unstyled">
38         {% for answer in answers %}
39             <li class="mb-4">
40                 <div class="d-flex justify-content-center">
↕ // ... lines 41 - 47
48                     <div class="vote-arrows flex-fill pt-2 js-vote-arrows"
49 style="min-width: 90px;">
49                         <a class="vote-up" href="#" data-direction="up"><i
49 class="far fa-arrow-alt-circle-up"></i></a>
50                         <a class="vote-down" href="#" data-
50 direction="down"><i class="far fa-arrow-alt-circle-down"></i></a>
51                         <span>+ <span class="js-vote-total">6</span>
51 </span>
52                     </div>
53                 </div>
54             </li>
55         {% endfor %}
56     </ul>
57 </div>
58 {% endblock %}
```

Adding JavaScript inside the `javascripts` Block.

To keep things simple, the JavaScript code we are going to write will use jQuery. In fact, *if* your site uses jQuery, you *probably* will want to include jQuery on *every* page... which means that we want to add a `script` tag to `base.html.twig`. At the bottom, notice that we have a block called `javascripts`. Inside this block, I'm going to paste a `<script>` tag to bring in jQuery from a CDN. You can copy this from the code block on this page, or go to jQuery to get it.

Tip

In new Symfony projects, the `javascripts` block is at the top of this file - inside the `<head>` tag. You can keep the `javascripts` block up in `<head>` or move it down here. If you keep it up inside `head`, be sure to add a `defer` attribute to every `script` tag: this will cause your JavaScript to be executed *after* the page loads.

```
templates/base.html.twig
```

```
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 12
13 <body>
↕ // ... lines 14 - 25
26     {% block javascripts %}
27         <script
28             src="https://code.jquery.com/jquery-3.4.1.min.js"
29             integrity="sha256-
CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSF1Bw8HfCJo="
30             crossorigin="anonymous"></script>
31     {% endblock %}
32 </body>
33 </html>
```

If you're wondering *why* we put this inside of the `javascripts` block... other than it "seems" like a logical place, I'll show you why in a minute. Because *technically*, if we put this *after* the `javascripts` block or before, it would make no difference right now. But putting it inside will be useful soon.

For our custom JavaScript, inside the `public/` directory, create a new directory called `js/`. And then a new file: `question_show.js`.

Here's the idea: usually you will have some custom JavaScript that you want to include on every page. We don't have any right now, but if we *did*, I would create an `app.js` file and add a `script` tag for it in `base.html.twig`. Then, on certain pages, you might *also* need to include some page-specific JavaScript, like to power a comment-voting feature that only lives on one page.

That's what I'm doing and that's why I created a file called `question_show.js`: it's custom JavaScript for that page.

Inside `question_show.js`, I'm going to paste about 15 lines of code.

```
public/js/question_show.js
```

```
1  /**
2   * Simple (ugly) code to handle the comment vote up/down
3   */
4  var $container = $('.js-vote-arrows');
5  $container.find('a').on('click', function(e) {
6      e.preventDefault();
7      var $link = $(e.currentTarget);
8
9      $.ajax({
10         url: '/comments/10/vote/'+$link.data('direction'),
11         method: 'POST'
12     }).then(function(response) {
13         $container.find('.js-vote-total').text(response.votes);
14     });
15 });
```

This finds the `.js-vote-arrows` element - which we added here - finds any `a` tags inside, and registers a `click` listener on them. On click, we make an AJAX request to `/comments/10` - the 10 is hardcoded for now - `/vote/` and then we read the `data-direction` attribute off of the anchor element to know if this is an `up` vote or `down` vote. On success, jQuery passes us the JSON data from our endpoint. Let's rename that variable to `data` to be more accurate.

```
public/js/question_show.js
```

```
↕ // ... lines 1 - 4
5  $container.find('a').on('click', function(e) {
↕ // ... lines 6 - 8
9      $.ajax({
↕ // ... lines 10 - 11
12     }).then(function(data) {
13         $container.find('.js-vote-total').text(data.votes);
14     });
15 });
```

Then we use the `votes` field from the data - because in our controller we're returning a `votes` key - to update the vote total.

Overriding the javascripts Block

So... how do we include this file? If we wanted to include this on every page, it would be pretty easy: add another script tag below jQuery in `base.html.twig`. But we want to include this

only on the show page. This is where having the jQuery script tag inside of a `javascripts` block is handy. Because, in a "child" template, we can *override* that block.

Check it out: in `show.html.twig`, it doesn't matter where - but let's go to the bottom, say `{% block javascripts %} {% endblock %}`. Inside, add a `<script>` tag with `src=""`. Oh, we need to remember to use the `asset()` function. But... PhpStorm is suggesting `js/question_show.js`. Select that. Nice! It added the `asset()` function for us.

```
templates/question/show.html.twig
↕ // ... lines 1 - 59
60 {% block javascripts %}
↕ // ... lines 61 - 62
63     <script src="{{ asset('js/question_show.js') }}"></script>
64 {% endblock %}
```

If we stopped now, this would literally *override* the `javascripts` block of `base.html.twig`. So, jQuery would not be included on the page. Instead of *overriding* the block, what we *really* want to do is *add to it*! In the final HTML, we want our new `script` tag to go right *below* jQuery.

How can we do this? Above our script tag, say `{{ parent() }}`.

```
templates/question/show.html.twig
↕ // ... lines 1 - 59
60 {% block javascripts %}
61     {{ parent() }}
62
63     <script src="{{ asset('js/question_show.js') }}"></script>
64 {% endblock %}
```

I love that! The `parent()` function gets the content of the *parent* block, and prints it.

Let's try this! Refresh and... click up. It updates! And if we hit down, we see a really low number.

AJAX Requests on the Profiler

Oh, and see this number "6" down on the web debug toolbar? This is really cool. Refresh the page. Notice that the icon is *not* down here. But as soon as our page makes an AJAX requests, it shows up! Yep, the web debug toolbar *detects* AJAX requests and lists them here. The *best*

part is that you can use this to jump into the *profiler* for any of these requests! I'll right click and open this "down" vote link in a new tab.

This is the *full* profiler for that request in all its glory. If you use `dump()` somewhere in your code, the dumped variable for that AJAX requests will be here. And later, a database section will be here. This is a *killer* feature.

Next, let's tighten up our API endpoint: we shouldn't be able to make a GET request to it - like loading it in our browser. And... do we have anything that validates that the `{direction}` wildcard in the URL is either `up` or `down` but nothing else? Not yet.

Chapter 15: Smart Routes: POST-only & Validate {Wildcards}

Inside our JavaScript, we're making a POST request to the endpoint. And that makes sense. The topic of "which HTTP method" - like GET, POST, PUT, etc - you're *supposed* to use for an API endpoint... can get complicated. But because our endpoint will eventually *change* something in the database, as a best-practice, we don't want to allow people to make a **GET** request to it. Right now, we can make a GET request by just putting the URL in our browser. Hey! I just voted!

To tighten this up, in **CommentController**, we can make our route smarter: we can tell it to *only* match if the method is POST. To do that add **methods="POST"**.

```
src/Controller/CommentController.php
↕ // ... lines 1 - 8
9 class CommentController extends AbstractController
10 {
11     /**
12      * @Route("/comments/{id}/vote/{direction}", methods="POST")
13      */
14     public function commentVote($id, $direction)
15     {
↕ // ... lines 16 - 25
26     }
27 }
```

As soon as we do that, when we refresh... 404 not found! The route no longer matches.

 **Tip**

Actually, it's a 405 response code! HTTP Method Not Allowed.

The router:match Command

Another cool way to see this is at your terminal. Run: **php bin/console router:match**. Then go copy the URL... and paste it.

```
php bin/console router:match /comments/10/vote/up
```

This fun command tells us which *route* matches a given URL. In this case, *no* routes match, but it tells us that it *almost* matched the `app_comment_commentvote` route.

To see if a `POST` request would match this route, pass `--method=POST`:

```
php bin/console router:match /comments/10/vote/up --method=POST
```

And... boom! It shows us the route that matched and ALL its details, including the controller.

Restricting what a {Wildcard} Matches

But there's something else that's not quite right with our route. We're *expecting* that the `{direction}` part will either be `up` or `down`. But... technically, somebody could put `banana` in the URL. In fact, let's try that: change the direction to `banana`:

```
php bin/console router:match /comments/10/vote/banana --method=POST
```

Yes! We vote "banana" for this comment! This isn't the end of the world... if a bad user tried to hack our system and did this, it would just be a down vote. But we can make this better.

As you know, *normally* a wildcard matches *anything*. However, if you want, you can control that with a regular expression. Inside the `{}`, but after the name, add `<>`. Inside, say `up|down`.

```
src/Controller/CommentController.php
```

```
↕ // ... lines 1 - 8
9 class CommentController extends AbstractController
10 {
11     /**
12      * @Route("/comments/{id}/vote/{direction<up|down>}", methods="POST")
13      */
14     public function commentVote($id, $direction)
15     {
16     // ... lines 16 - 25
17     }
18 }
19 }
```

Now try the `router:match` command:

```
php bin/console router:match /comments/10/vote/banana --method=POST
```

Yes! It does *not* match because `banana` is not up or down. If we change this to `up`, it works:

```
php bin/console router:match /comments/10/vote/up --method=POST
```

Making id Only Match an Integer?

By the way, you might be tempted to *also* make the `{id}` wildcard smarter. Assuming we're using auto-increment database ids, we know that `id` should be an integer. To make this route only match if the `id` part is a number, you can add `<\d+>`, which means: match a "digit" of any length.

```
src/Controller/CommentController.php
```

```
↕ // ... lines 1 - 8
9 class CommentController extends AbstractController
10 {
11     /**
12      * @Route("/comments/{id<\d+>}/vote/{direction<up|down>}",
13      methods="POST")
14      */
15     public function commentVote($id, $direction)
16     {
17         // ... lines 16 - 25
18     }
19 }
20 }
21 }
```

But... I'm actually *not* going to put that here. Why? Eventually, we're going to use `$id` to query the database. If somebody puts `banana` here, who cares? The query won't find any comment with an id of `banana` and we will add some code to return a 404 page. Even if somebody tries an SQL injection attack, as you'll learn later in our database tutorial, it will *still* be ok, because the database layer protects against this.

```
src/Controller/CommentController.php
```

```
↕ // ... lines 1 - 8
9 class CommentController extends AbstractController
10 {
11     /**
12      * @Route("/comments/{id}/vote/{direction<up|down>}", methods="POST")
13      */
14     public function commentVote($id, $direction)
15     {
16         // ... lines 16 - 25
17     }
18 }
19 }
```

Let's make sure everything still works. I'll close one browser tab and refresh the show page. Yea! Voting still looks good.

Next, let's get a sneak peek into the most *fundamental* part of Symfony: services.

Chapter 16: Service Objects

Symfony is really two parts... and we've already learned *one* of them.

The first part is the route and controller system. And I hope you're feeling pretty comfortable: create a route, it executes a controller function, we return a response.

The *second* half of Symfony is all about the many "useful objects" that are floating around inside Symfony. For example, when we render a template, what we're *actually* doing is taking advantage of a twig object and asking it to render. The `render()` method is just a shortcut to use that object. There is also a logger object, a cache object and many more, like a database connection object and an object that helps make HTTP requests to other APIs.

Basically... *every single thing* that Symfony does - or that *we* do - is *actually* done by one of these useful objects. Heck, even the *router* is an object that figures out which route matches the current request.

In the Symfony world - well, really, in the object-oriented programming world - these "objects that do work" are given a special name: services. But don't let that confuse you: when you hear "service", just think:

"Hey! That's an object that does some work - like a logger object or a database object that makes queries."

Listing All Services

Inside `CommentController`, let's log something. To do that work, we need the "logger" service. How can we get it?

Find your terminal and run:

```
php bin/console debug:autowiring
```

Say hello to one of the *most* important `bin/console` commands. This gives us a list of *all* the service objects in our app. Well, ok, this isn't *all* of them: but it *is* a full list of all the services that you are *likely* to need.

Even in our small app, there's a lot of stuff in here: there's something called `Psr\Log\LoggerInterface`, there's stuff for caching and plenty more. As we install more bundles, this list will grow. More services, means more tools.

To find which service allows us to "log" things, run:

```
php bin/console debug:autowiring log
```

This returns a *bunch* of things... but ignore all of these down here for now and focus on the top line. This tells us that there is a logger service object and its class implements some `Psr\Log\LoggerInterface`. Why is that important? Because if you want the logger service, you ask for it by using this type-hint. It's called "autowiring".

Autowiring the Logger Service

Here's how you get a service from inside a controller. Add a third argument to your method - though the argument order doesn't matter. Say `LoggerInterface` - auto-complete the one from `Psr\Log\LoggerInterface` - and `$logger`.

```
src/Controller/CommentController.php
```

```
↕ // ... lines 1 - 4
5 use Psr\Log\LoggerInterface;
↕ // ... lines 6 - 9
10 class CommentController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function commentVote($id, $direction, LoggerInterface $logger)
16     {
↕ // ... lines 17 - 28
29     }
30 }
```

This added a `use` statement above the class for `Psr\Log\LoggerInterface`, which *matches* the type-hint that `debug:autowiring` told us to use. Thanks to this type-hint, when

Symfony renders our controller, it will know that we want the logger service to be passed to this argument.

So... yea: there are now *two* types of arguments that you can add to your controller method. First, you can have an argument whose *name* matches a wildcard in your route. And second, you can have an argument whose *type-hint* matches one of the class or interface names listed in `debug:autowiring`. `CacheInterface` is another type-hint we could use to get a *caching* service.

Using the Logger Service

So... let's use this object! What methods can we call on it? I have no idea! But because we properly type-hinted the argument, we can say `$logger->` and PhpStorm tells us *exactly* what methods it has. Let's use `$logger->info()` to say "Voting up!". Copy that and say "Voting down!" on the else.

```
src/Controller/CommentController.php
↕ // ... lines 1 - 9
10 class CommentController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function commentVote($id, $direction, LoggerInterface $logger)
16     {
↕ // ... lines 17 - 19
20         if ($direction === 'up') {
21             $logger->info('Voting up!');
↕ // ... line 22
23         } else {
24             $logger->info('Voting down!');
↕ // ... line 25
26         }
↕ // ... lines 27 - 28
29     }
30 }
```

Testing time! Refresh the page and... let's click up, down, up. It... at least doesn't look *broken*.

Hover over the AJAX part of the web debug toolbar and open the profiler for one of these requests. The profiler has a "Logs" section, which is the *easiest* way to see the log entries for a single request. There it is! "Voting up!". You could also find this in the `var/log/dev.log` file.

The point is: Symfony has many, *many* useful objects, I mean "services". And little-by-little, we're going to start using more of them... each time by adding a *type-hint* to tell Symfony which service we need.

Autowiring & Using the Twig Service

Let's look at one other example. The *first* service that we used in our code was the *Twig* service. We used it... kind of "indirectly" by saying `$this->render()`. In reality, that method is a shortcut to use the *Twig service* behind the scenes. And that should *not* surprise you. Like I said, *everything* that's done in Symfony is *actually* done by a service.

As a challenge, let's pretend that the `render()` function doesn't exist. Gasp! In the `homepage()` controller, comment-out the `render()` line.

So... how can we use the Twig service directly to render a template? I don't know! We could *definitely* find some documentation about this... but let's see if we can figure it out by ourselves with the help of the `debug:autowiring` command:

```
php bin/console debug:autowiring twig
```

And, voilà! There is apparently a class called `Twig\Environment` that we can use as a "type-hint" to get a Twig service. In our controller, add `Environment` and hit tab to add the `use` statement on top. I'll call the argument `$twigEnvironment`.

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 7
8 use Twig\Environment;
↕ // ... line 9
10 class QuestionController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function homepage(Environment $twigEnvironment)
16     {
↕ // ... lines 17 - 21
22         //return $this->render('question/homepage.html.twig');
23     }
↕ // ... lines 24 - 40
41 }
```

Inside, add `$html = $twigEnvironment->`. Once again, without reading *any* documentation, thanks to the fact that we're coding responsibly and using type-hints, PhpStorm shows us *all* the methods on this class. Hey! This `render()` method looks like it might be what we need! Pass the same template name as before.

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 9
10 class QuestionController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function homepage(Environment $twigEnvironment)
16     {
17         // fun example of using the Twig service directly!
18         $html = $twigEnvironment->render('question/homepage.html.twig');
↕ // ... lines 19 - 21
22         //return $this->render('question/homepage.html.twig');
23     }
↕ // ... lines 24 - 40
41 }
```

When you use twig directly, instead of returning a Response object, it returns a *string* with the HTML. No problem: finish with `return new Response()` - the one from `HttpFoundation` - and pass `$html`.

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 5
6 use Symfony\Component\HttpFoundation\Response;
↕ // ... lines 7 - 9
10 class QuestionController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function homepage(Environment $twigEnvironment)
16     {
17         // fun example of using the Twig service directly!
18         $html = $twigEnvironment->render('question/homepage.html.twig');
19
20         return new Response($html);
21
22         //return $this->render('question/homepage.html.twig');
23     }
↕ // ... lines 24 - 40
41 }
```

This is now doing the *exact* same thing as `$this->render()`. To prove it, click the homepage link. It still works.

Now in reality, other than being a "great exercise" to understand services, there's no reason to do this the *long* way. I just want you to understand that services are *really* the "things" doing the work behind the scenes. And if you want to do something - like log or render a template - what you *really* need is to find out which *service* does that work. Trust me, *this* is the key to unlocking your full potential in Symfony.

Let's put the old, shorter code back, and comment out the longer example.

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 9
10 class QuestionController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function homepage(Environment $twigEnvironment)
16     {
17         /*
18         // fun example of using the Twig service directly!
19         $html = $twigEnvironment->render('question/homepage.html.twig');
20
21         return new Response($html);
22         */
23
24         return $this->render('question/homepage.html.twig');
25     }
↕ // ... lines 26 - 42
43 }
```

Ok, you've *almost* made it through the first Symfony tutorial. You rock! As a reward, we're going to finish with something fun: an introduction into a system called Webpack Encore that will allow you to do *crazy* things with your CSS and JavaScript.

Chapter 17: Hello Webpack Encore

Our CSS and JavaScript setup is fine: we have a `public/` directory with `app.css` and `question_show.js`. Inside our templates - like `base.html.twig` - we include the files with traditional link or script tags. Sure, we use this `{{ asset() }}` function, but it doesn't do anything important. Symfony isn't touching our frontend assets at all.

That's fine. But if you want to get serious about frontend development - like using a frontend framework like React or Vue - you need to take this up to the next level.

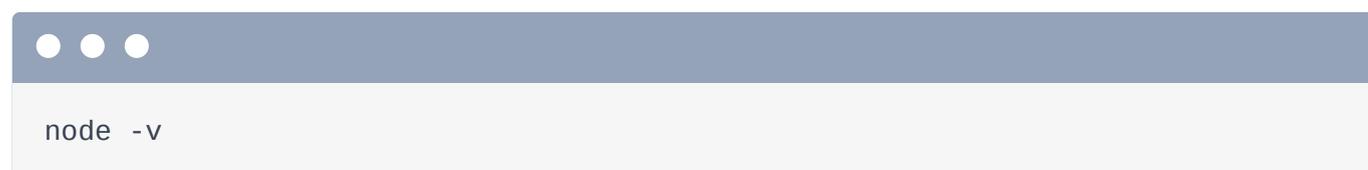
To do that, we're going to use a Node library called Webpack: which is the industry-standard tool for managing your frontend assets. It combines and minifies your CSS and JavaScript files... though that's just the tip of the iceberg of what it can do.

But... to get Webpack to work *really* well requires a lot of complex config. So, in the Symfony world, we use a *wonderful* library called Webpack Encore. It's a lightweight layer on *top* of Webpack that... makes it easier! And we have an entire [free tutorial](#) about it here on SymfonyCasts.

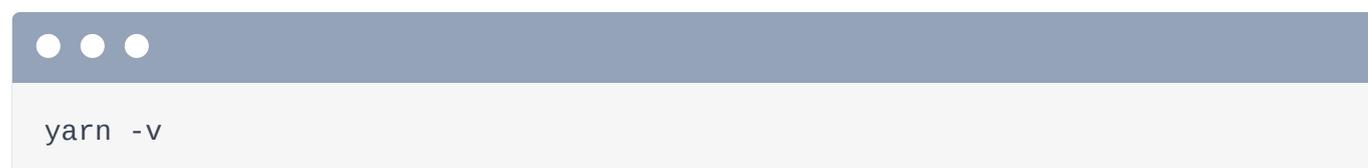
But let's go through a crash course right now.

Installing Webpack Encore

First, make sure you have node installed:

A terminal window with a dark blue header and a light gray body. The text 'node -v' is displayed in the terminal.

And also yarn:

A terminal window with a dark blue header and a light gray body. The text 'yarn -v' is displayed in the terminal.

Tip

If you don't have Node or Yarn installed - see official manuals about how to install it for *your* OS. For Node, see <https://nodejs.org/en/download/> and for Yarn: <https://classic.yarnpkg.com/en/docs/install> . We recommend using Yarn version 1.x to follow this tutorial.

Yarn is one of the package managers for Node... basically Composer for Node.

Before we install Encore, make sure you've committed all your changes - I already have. Then run:

```
composer require "encore:^1.8"
```

Wait... a minute ago, I said that Encore is a *Node* library. So why are we installing it via Composer? Great question! This command does *not* actually install Encore. Nope, it installs a very small bundle called `webpack-encore-bundle`, which helps our Symfony app *integrate* with Webpack Encore. The *real* beauty is that this bundle has a *very* useful recipe. Check it out, run:

```
git status
```

Woh! Its recipe did a *lot* for us! One cool thing is that it modified our `.gitignore` file. Go open it in your editor.

```
.gitignore
```

```
↕ // ... lines 1 - 11
12 ###> symfony/webpack-encore-bundle ###
13 /node_modules/
14 /public/build/
15 npm-debug.log
16 yarn-error.log
17 ###
```

Cool! We're now ignoring `node_modules/` - which is Node's version of the `vendor/` directory - and a few other paths.

The recipe also added some YAML files, which help set things up - but you don't really need to look at them.

The *most* important thing the recipe did was give us these 2 files: `package.json` - which is the `composer.json` of Node - and `webpack.config.js`, which is the Webpack Encore configuration file.

Check out the `package.json` file. This tells Node which libraries it should download and it already has the basic stuff we need. Most importantly: `@symfony/webpack-encore`.

`package.json`

```
1 {
2   "devDependencies": {
3     "@symfony/webpack-encore": "^0.28.2",
4     "core-js": "^3.0.0",
5     "regenerator-runtime": "^0.13.2",
6     "webpack-notifier": "^1.6.0"
7   },
8   "license": "UNLICENSED",
9   "private": true,
10  "scripts": {
11    "dev-server": "encore dev-server",
12    "dev": "encore dev",
13    "watch": "encore dev --watch",
14    "build": "encore production --progress"
15  }
16 }
```

Installing Node Dependencies with yarn

To tell Node to install these dependencies, run:

```
yarn install
```

This command reads `package.json` and downloads a *ton* of files and directories into a new `node_modules/` directory. It might take a few minutes to download everything and build a couple of packages.

When it's done, you'll see two new things. First, you have a fancy new `node_modules/` directory with *tons* of stuff in it. And this is already being ignored from git. Second, it created a `yarn.lock` file, which has the same function as `composer.lock`. So... you should commit the `yarn.lock` file, but not worry about it otherwise.

Ok, Encore is installed! Next, let's refactor our frontend setup to use it.

Chapter 18: Webpack Encore: JavaScript Greatness

Tip

The recipe now adds these 2 files in a slightly different location:

- `assets/app.js`
- `assets/styles/app.css`

But the purpose of each file is exactly the same.

Okay: here's how this whole thing works. The recipe added a new `assets/` directory with a couple of example CSS and JS files. The `app.js` file basically just `console.log()`'s something:

`assets/js/app.js`

```
1 /*
2  * Welcome to your app's main JavaScript file!
3  *
4  * We recommend including the built version of this JavaScript file
5  * (and its CSS file) in your base layout (base.html.twig).
6  */
7
8 // any CSS you import will output into a single css file (app.css in this
9 // case)
10 import '../css/app.css';
11
12 // Need jQuery? Install it with "yarn add jquery", then uncomment to
13 // import it.
14 // import $ from 'jquery';
15
16 console.log('Hello Webpack Encore! Edit me in assets/js/app.js');
```

The `app.css` changes the background color to light gray:

```
assets/css/app.css
```

```
1 body {  
2     background-color: lightgray;  
3 }
```

Webpack Encore is entirely configured by one file: `webpack.config.js`:

```
1 var Encore = require('@symfony/webpack-encore');
2
3 // Manually configure the runtime environment if not already configured
  // yet by the "encore" command.
4 // It's useful when you use tools that rely on webpack.config.js file.
5 if (!Encore.isRuntimeEnvironmentConfigured()) {
6     Encore.configureRuntimeEnvironment(process.env.NODE_ENV || 'dev');
7 }
8
9 Encore
10     // directory where compiled assets will be stored
11     .setOutputPath('public/build/')
12     // public path used by the web server to access the output path
13     .setPublicPath('/build')
14     // only needed for CDN's or sub-directory deploy
15     //.setManifestKeyPrefix('build/')
16
17     /*
18      * ENTRY CONFIG
19      *
20      * Add 1 entry for each "page" of your app
21      * (including one that's included on every page - e.g. "app")
22      *
23      * Each entry will result in one JavaScript file (e.g. app.js)
24      * and one CSS file (e.g. app.css) if your JavaScript imports CSS.
25      */
26     .addEntry('app', './assets/js/app.js')
27     //.addEntry('page1', './assets/js/page1.js')
28     //.addEntry('page2', './assets/js/page2.js')
29
30     // When enabled, Webpack "splits" your files into smaller pieces for
  // greater optimization.
31     .splitEntryChunks()
32
33     // will require an extra script tag for runtime.js
34     // but, you probably want this, unless you're building a single-page
  app
35     .enableSingleRuntimeChunk()
36
37     /*
38      * FEATURE CONFIG
39      *
40      * Enable & configure other features below. For a full
41      * list of features, see:
42      * https://symfony.com/doc/current/frontend.html#adding-more-features
43      */
```

```

44 .cleanupOutputBeforeBuild()
45 .enableBuildNotifications()
46 .enableSourceMaps(!Encore.isProduction())
47 // enables hashed filenames (e.g. app.abc123.css)
48 .enableVersioning(Encore.isProduction())
49
50 // enables @babel/preset-env polyfills
51 .configureBabelPresetEnv((config) => {
52     config.useBuiltIns = 'usage';
53     config.corejs = 3;
54 })
55
56 // enables Sass/SCSS support
57 // .enableSassLoader()
58
59 // uncomment if you use TypeScript
60 // .enableTypeScriptLoader()
61
62 // uncomment to get integrity="..." attributes on your script & link
tags
63 // requires WebpackEncoreBundle 1.4 or higher
64 // .enableIntegrityHashes(Encore.isProduction())
65
66 // uncomment if you're having problems with a jQuery plugin
67 // .autoProvidejQuery()
68
69 // uncomment if you use API Platform Admin (composer req api-admin)
70 // .enableReactPreset()
71 // .addEntry('admin', './assets/js/admin.js')
72 ;
73
74 module.exports = Encore.getWebpackConfig();

```

We won't talk much about this file - we'll save that for the Encore tutorial - but it's already configured to *point* at the `app.js` and `app.css` files: it knows that it needs to process them.

Running Encore

To execute Encore, find your terminal and run:



```
yarn watch
```

This is a shortcut for running `yarn run encore dev --watch`. What does this do? It reads those 2 files in `assets/`, does some processing on them, and outputs a *built* version of each inside a new `public/build/` directory. Here is the "built" `app.css` file... and the built `app.js` file. If we ran Encore in production mode - which is just a different command - it would *minimize* the contents of each file.

Including the Built CSS and JS Files

There's a lot more cool stuff going on, but this is the basic idea: we code in the `assets/` directory, but point to the *built* files in our templates.

For example, in `base.html.twig`, instead of pointing at the old `app.css` file, we want to point at the one in the `build/` directory. That's simple enough, but the WebpackEncoreBundle has a shortcut to make it even easier: `{{ encore_entry_link_tags() }}` and pass this `app`, because that's the name of the source files - called an "entry" in Webpack land.

```
templates/base.html.twig
↕ // ... line 1
2 <html>
3   <head>
↕ // ... lines 4 - 5
6     {% block stylesheets %}
7       <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.c
integrity="sha384-
Vko08x4CGs03+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
crossorigin="anonymous">
8       <link rel="stylesheet" href="https://fonts.googleapis.com/css?
family=Spartan&display=swap">
9       <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/5.12.1/css/all.min.css" integrity="sha256-
mmgLkCYLUQbXn0B1SRqzHar6dCnv9oZFPEC1g1cwlkk=" crossorigin="anonymous" />
10      {{ encore_entry_link_tags('app') }}
11      {% endblock %}
12    </head>
↕ // ... lines 13 - 33
34 </html>
```

At the bottom, render the script tag with `{{ encore_entry_script_tags('app') }}`.

```
templates/base.html.twig
```

```
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 12
13 <body>
↕ // ... lines 14 - 25
26     {% block javascripts %}
27         <script
28             src="https://code.jquery.com/jquery-3.4.1.min.js"
29             integrity="sha256-
CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSF1Bw8HfCJo="
30             crossorigin="anonymous"></script>
31         {{ encore_entry_script_tags('app') }}
32     {% endblock %}
33 </body>
34 </html>
```

Let's try it! Move over and refresh. Did it work? It did! The background color is gray... and if I bring up the console, there's the log:

```
“Hello Webpack Encore!”
```

If you look at the HTML source, there's nothing special going on: we have a normal link tag pointing to `/build/app.css`.

Moving our Code into Encore

Now that this is working, let's move *our* CSS into the new system. Open `public/css/app.css`, copy all of this, then right click and delete the file. Now open the *new* `app.css` inside `assets/` and paste.

```
1 body {
2     font-family: spartan;
3     color: #444;
4 }
5
6 .jumbotron-img {
7     background: rgb(237,116,88);
8     background: linear-gradient(302deg, rgba(237,116,88,1) 16%,
9     rgba(51,61,81,1) 97%);
10    color: #fff;
11 }
12
13 .q-container {
14     border-top-right-radius: .25rem;
15     border-top-left-radius: .25rem;
16     background-color: #efefee;
17 }
18
19 .q-container-show {
20     border-top-right-radius: .25rem;
21     border-top-left-radius: .25rem;
22     background-color: #ED7458 ;
23 }
24
25 .q-container img, .q-container-show img {
26     border: 2px solid #fff;
27     border-radius: 50%;
28 }
29
30 .q-display {
31     background: #fff;
32     border-radius: .25rem;
33 }
34
35 .q-title-show {
36     text-transform: uppercase;
37     font-size: 1.3rem;
38     color: #fff;
39 }
40
41 .q-title {
42     text-transform: uppercase;
43     color: #444;
44 }
45
46 .q-title:hover {
47     color: #2B2B2B;
48 }
```

```
46
47 .q-title h2 {
48     font-size: 1.3rem;
49 }
50
51 .q-display-response {
52     background: #333D51;
53     color: #fff;
54 }
55
56 .answer-link:hover .magic-wand {
57     transform: rotate(20deg);
58 }
59
60 .vote-arrows {
61     font-size: 1.5rem;
62 }
63
64 .vote-arrows span {
65     font-size: 1rem;
66 }
67
68 .vote-arrows a {
69     color: #444;
70 }
71
72 .vote-up:hover {
73     color: #3D9970;
74 }
75 .vote-down:hover {
76     color: #FF4136;
77 }
78
79 .btn-question {
80     color: #FFFFFF;
81     background-color: #ED7458;
82     border-color: #D45B3F;
83 }
84
85 .btn-question:hover,
86 .btn-question:focus,
87 .btn-question:active,
88 .btn-question.active,
89 .open .dropdown-toggle.btn-question {
90     color: #FFFFFF;
91     background-color: #D45B3F;
92     border-color: #D45B3F;
```

```

93 }
94
95 .btn-question:active,
96 .btn-question.active,
97 .open .dropdown-toggle.btn-question {
98     background-image: none;
99 }
100
101 .btn-question.disabled,
102 .btn-question[disabled],
103 fieldset[disabled] .btn-question,
104 .btn-question.disabled:hover,
105 .btn-question[disabled]:hover,
106 fieldset[disabled] .btn-question:hover,
107 .btn-question.disabled:focus,
108 .btn-question[disabled]:focus,
109 fieldset[disabled] .btn-question:focus,
110 .btn-question.disabled:active,
111 .btn-question[disabled]:active,
112 fieldset[disabled] .btn-question:active,
113 .btn-question.disabled.active,
114 .btn-question[disabled].active,
115 fieldset[disabled] .btn-question.active {
116     background-color: #ED7458;
117     border-color: #D45B3F;
118 }
119
120 .btn-question .badge {
121     color: #ED7458;
122     background-color: #FFFFFF;
123 }
124
125 footer {
126     background-color: #efefee;
127 }

```

As soon as I do that, when I refresh... it works! Our CSS is back! The reason is that - if you check your terminal - `yarn watch` is *watching* our files for changes. As soon as we modified the `app.css` file, it re-read that file and dumped a new version into the `public/build` directory. That's why we keep this running in the background.

Let's do the same thing for our custom JavaScript. Open `question_show.js` and, instead of having a page-specific JavaScript file - where we only include this on our "show" page - to keep things simple, I'm going to put this into the new `app.js`, which is loaded on every page.

```
assets/js/app.js
```

```
↕ // ... lines 1 - 13
14 /**
15  * Simple (ugly) code to handle the comment vote up/down
16  */
17 var $container = $('.js-vote-arrows');
18 $container.find('a').on('click', function(e) {
19     e.preventDefault();
20     var $link = $(e.currentTarget);
21
22     $.ajax({
23         url: '/comments/10/vote/'+$link.data('direction'),
24         method: 'POST'
25     }).then(function(data) {
26         $container.find('.js-vote-total').text(data.votes);
27     });
28 });
```

Then go delete the `public/js/` directory entirely... and `public/css/`. Also open up `templates/question/show.html.twig` and, at the bottom, remove the old script tag.

```
templates/question/show.html.twig
```

```
1 {% extends 'base.html.twig' %}
2
3 {% block title %}Question: {{ question }}{% endblock %}
4
5 {% block body %}
↕ // ... lines 6 - 57
58 {% endblock %}
```

With any luck, Encore *already* rebuilt my `app.js`. So if we click to view a question - I'll refresh just to be safe - and... click the vote icons. Yes! This still works.

Installing & Importing Outside Libraries (jQuery).

Now that we're using Encore, there are some *really* cool things we can do. Here's one: instead of linking to a CDN or downloading jQuery directly into our project and committing it, we can *require* jQuery and install it into our `node_modules/` directory... which is *exactly* how we're used to doing things in PHP: we install third-party libraries into `vendor/` instead of downloading them manually.

To do that, open a new terminal tab and run:

```
yarn add jquery --dev
```

This is equivalent to the `composer require` command: it adds `jquery` to the `package.json` file and downloads it into `node_modules/`. The `--dev` part is not important.

Next, inside `base.html.twig`, remove jQuery entirely from the layout.

```
templates/base.html.twig
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 12
13 <body>
↕ // ... lines 14 - 25
26     {% block javascripts %}
27         {{ encore_entry_script_tags('app') }}
28     {% endblock %}
29 </body>
30 </html>
```

If you go back to your browser and refresh the page now... it's totally broken:

“\$ is not defined”

...coming from `app.js`. That makes sense: we *did* just download jQuery into our `node_modules/` directory - you can find a directory here called `jquery` - but we're not *using* it yet.

How do we use it? Inside `app.js`, uncomment this import line: `import $ from 'jquery'`.

```
assets/js/app.js
↕ // ... lines 1 - 9
10
11 // Need jQuery? Install it with "yarn add jquery", then uncomment to
  import it.
12 import $ from 'jquery';
13
↕ // ... lines 14 - 29
```

This "loads" the `jquery` package we installed and *assigns* it to a `$` variable. All these `$` variables below are referencing the value we imported.

Here's the *really* cool part: without making *any* other changes, when we refresh, it works! Webpack *noticed* that we're importing `jquery` and automatically packaged it *inside* of the built `app.js` file. We import the stuff we need, and Webpack takes care of... packaging it all together.

Tip

Actually, Webpack splits the final files into multiple for efficiency. jQuery actually lives inside a different file in `public/build/`, though that doesn't matter!

Importing the Bootstrap CSS

We can do the same thing for the Bootstrap CSS. On the top of `base.html.twig`, delete the `link` tag for Bootstrap:

```
templates/base.html.twig
↕ // ... line 1
2 <html>
3   <head>
↕ // ... lines 4 - 5
6     {% block stylesheets %}
7       <link rel="stylesheet" href="https://fonts.googleapis.com/css?
family=Spartan&display=swap">
8       <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/5.12.1/css/all.min.css" integrity="sha256-
mmgLkCYLUQbXn0B1SRqzHar6dCnv9oZFPEC1g1cwlkk=" crossorigin="anonymous" />
9       {{ encore_entry_link_tags('app') }}
10      {% endblock %}
11    </head>
↕ // ... lines 12 - 28
29 </html>
```

No surprise, when we refresh, our site looks terrible.

To fix it, find your terminal and run:

```
yarn add bootstrap --dev
```

This downloads the `bootstrap` package into `node_modules/`. This package contains *both* JavaScript and CSS. We want to bring in the CSS.

To do that, open `app.css` and, at the top, use the good-old-fashioned `@import` syntax. Inside double quotes, say `~bootstrap`:

```
assets/css/app.css
1 @import "~bootstrap";
2
⬆ // ... lines 3 - 129
```

In CSS, this `~` is a special way to say that you want to load the CSS from a `bootstrap` package inside `node_modules/`.

Move over, refresh and... we are back! Webpack saw this import, grabbed the CSS from the `bootstrap` package, and included it in the final `app.css` file. How cool is that?

What Else can Encore Do?

This is just the start of what Webpack Encore can do. It can also minimize your files for production, supports Sass or LESS compiling, comes with React and Vue.js support, has automatic filename versioning and more. To go further, check out our free [Webpack Encore tutorial](#).

And... that's it for this tutorial! Congratulations for sticking with me to the end! You already understand the most important parts of Symfony. In the next tutorial, we're going to unlock even *more* of your Symfony potential by uncovering the secrets of services. You'll be unstoppable.

As always, if you have questions, problems or have a really funny story - especially if it involves your cat - we would *love* to hear from you in the comments.

Alright friends - seeya next time!

With <3 from SymphonyCasts