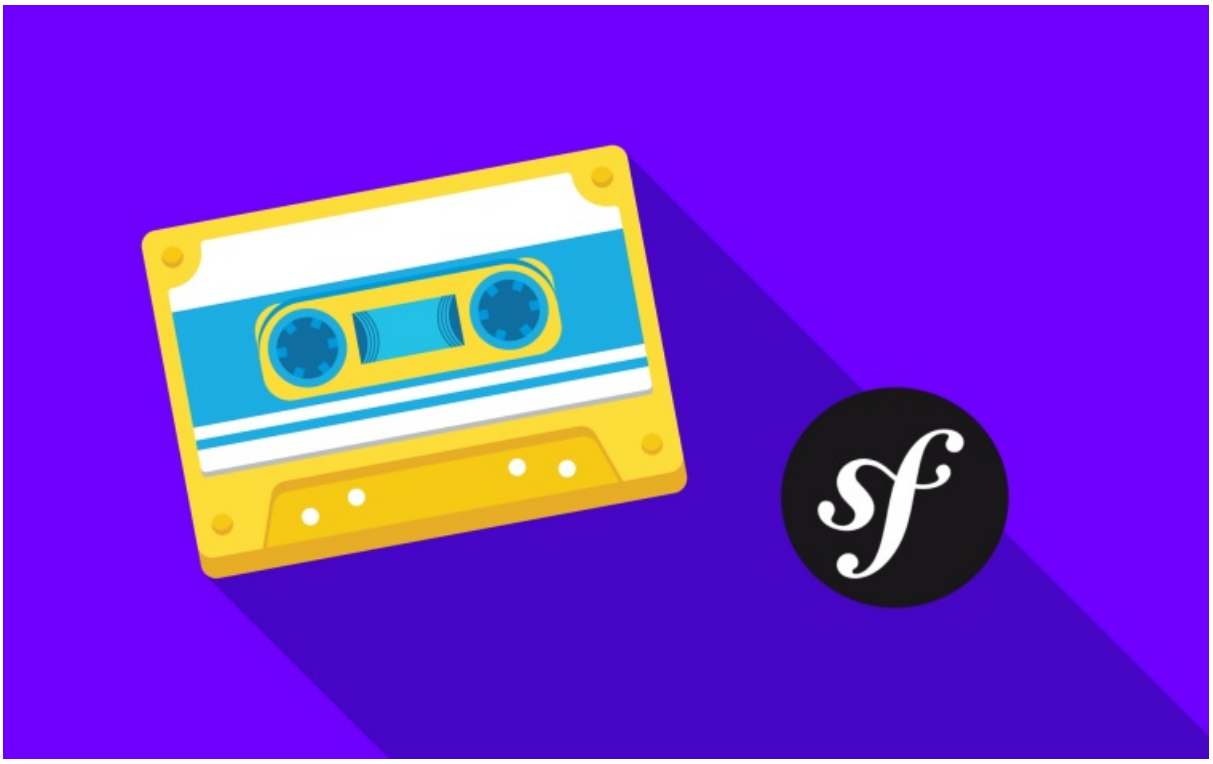
Harmonious Development with Symfony 6



With <3 from SymfonyCasts

Chapter 1: Hello Symfony

Welcome. Hello. Hi, my name is Ryan and I have *the* absolute pleasure to introduce you to the beautiful and fascinating and productive world of Symfony 6. Seriously, I feel like Willie Wonka inviting you into my chocolate factory, except with hopefully less sugar-related injuries. Anyways, if you're new to Symfony, I'm... honestly a bit jealous! You're going to *love* the journey... and hopefully become an even better developer along the way: you're *definitely* going to build some cool stuff.

The secret sauce of Symfony is that it starts *tiny*, which makes it easy to learn. But then, it scales up its features automatically through a unique recipe system. In Symfony 6, those features include new JavaScript tools and a new security system... just to name two of the many new things.

Symfony is also lightning fast with a huge focus on creating a joyful developer experience, but without sacrificing programming best practices. Yea: you get to love coding *and* love your code. I know... that sounded cheesy, but it's true.

So come with me and you'll be in a world of pure elucidation.

That's my first time singing in these tutorials... and maybe my last. Let's get started.

Installing the "symfony" Binary

Head over to https://symfony.com/download. On this page, you'll find some instructions - which will differ based on your operating system - on how to download something called the Symfony binary.

This is... not actually Symfony. It's just a command line tool that will help us *start* new Symfony projects and give us some nice local development tools. It's optional, but I highly recommend it!

Once you've installed this - I already have - open up your favorite terminal app. I'm using iTerm for mac, but it doesn't matter. If you got everything set up correctly, you should be able to run:

```
symfony
```

Or even better:

```
symfony list
```

to see a list of all the "things" that this symfony binary can do. There's a lot of stuff here: things that help with "local" development... and also some optional services for deployment. We'll walk through the stuff *you* need to know along the way.

Let's Start a Symfony App!

Ok, so *we* want to start a brand new shiny Symfony app. To do *that* run:

```
symfony new mixed_vinyl
```

Where "mixed_vinyl" is the directory the new app will be downloaded into. That's our top-secret project to

combine the *best* part of the 90's - no, not dial-up internet, I'm talking about mix tapes - with the auditory delight of records. More on that later.

Behind the scenes, this command uses *composer* - that's PHP's package manager - to create the new project. More on *that* later.

The end-result is that we can move into our new `mixed_vinyl` directory. Open this folder up in your favorite editor. I'm using PhpStorm and I *highly* recommend it.

Meeting our new Project

So what did that `symfony new` command do? It bootstrapped a new Symfony project! Ooh. And we already have a git repository. Run:

```
git status
```

Yep: on branch main, nothing to commit. Try:

```
git log
```

Cool. After downloading the new project, the command committed all of the original files automatically... which was *very* nice of it. Though I do wish that first commit message was a bit more rock n' roll.

What I *really* want to show you is that our new project is super small! Try this command:

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

Yup! Our entire project is... about 17 files. And we'll learn about all of these along the way. But I want you to feel comfortable: there's not a lot of code here.

We're going to add features little-by-little. But if you *did* want to start with a bigger, more fully-featured project, you can do that by running that `symfony new` command with `--webapp`.

If you want a fully-dockerized new Symfony app, check out https://github.com/dunglas/symfony-docker

Checking System Requirements

Before we jump into coding, let's make sure our system is ready. Run another command from the symfony binary:

```
symfony check:req
```

Looks good! If your PHP install is missing any extensions... or there are any other problems... like your computer is actually a teapot, this will let you know.

Starting the Dev Web Server

So: we have a new Symfony app over here... and our system is ready! All we need now is a subwoofer. I mean

web server! You *can* set up a real web server like Nginx or something trendy like Caddy. But for local development, the Symfony binary can help us. Run:

```
symfony serve -d
```

And... we have a web server running! Come back!

The first time you run this, it might ask you to run a different command to set up an SSL certificate, which is nice because then the server supports https.

Moment of truth! Copy the URL, spin over to your browser, hold your breath and woo! Hello Symfony 6 welcome page... complete with fancy color changes whenever we reload.

Next: let's meet - and become friends with - the code inside our app, so we can demystify what each part does. Then we'll code.

Chapter 2: Meet our Tiny App

Let's get to know our new project because my *ultimate* goal is for you to *really* understand how things work. As I mentioned, there isn't a lot here yet... about 15 files. And there are really only three directories that we ever need to think or worry about.

The public/ Directory

The first is `public/` ... and this is simple: it's the document root. In other words, if you need a file to be publicly accessible - like an image file or a CSS file - it needs to live inside `public/` .

Right now, this holds exactly one file: `index.php` , which is called the "front controller".

10 lines | public/index.php

```php
<?php

use App\Kernel;

require_once dirname(__DIR__).'/vendor/autoload_runtime.php';

return function (array $context) {
    return new Kernel($context['APP_ENV'], (bool) $context['APP_DEBUG']);
};
```

Ooo. That's a fancy word that means that - no matter what URL the user goes to - *this* is the script that's always executed first. Its job is to boot up Symfony and run our app. And now that we've looked at it, we'll probably never need to think about or open it ever again.

config/ & src/

And, really, other than putting CSS or image files into `public/` , this is *not* a directory you will deal with on a day-to-day basis. Which means... I kinda lied! There are really only *two* directories that we need to think about: `config/` and `src/` .

The `config/` directory holds... kittens! Oh, I wish. Nah, it holds config files. And `src/` holds 100% of your PHP classes. We will spend 95% of our time inside the `src/` directory.

composer.json & vendor/

Okay... so where is "Symfony"? Our project started with a `composer.json` file. This lists all of the third party libraries that our app needs. The "symfony new" command that we ran secretly used "Composer" - that's PHP's package manager - to install these libraries... which is really just a way of saying that Composer downloaded these libraries into the `vendor/` directory.

> If you're not familiar with Composer package manager - check out our separate course called Wonderful World of Composer.

Symfony itself is actually a collection of a bunch of small libraries that each solve a specific problem. In the `vendor/symfony/` directory, it looks like we already have about 25 of these. *Technically* our app only requires these six packages, but some of *those* packages require *other* packages... and Composer is smart enough to download everything we need.

Anyways, "Symfony", or really, a set of Symfony libraries, lives in the `vendor/` directory and our new app leverages that code to do its job. We're going to talk more about Composer and installing third party packages later. But for the most part, `vendor/` is yet another directory that... we don't need to worry about!

bin/ and var/

So what's left? Well, `bin/` holds exactly one file... and will always hold just this one file. We'll talk about what `bin/console` does a bit later. And the `var/` directory holds cache and log files. Those files are important... but *we* will never need to look at or think about that stuff.

Yup, we're going to live pretty much entirely inside of the `config/` and `src/` directories.

PhpStorm Setup

Ok, one last piece of homework before we start coding. Feel free to use whatever code editor you want: PhpStorm, VS Code, code carrier pigeon, whatever. But I *highly* recommend PhpStorm. It makes developing with Symfony a dream... and they're not even paying me to say that! Though, if they *do* want to start paying me, I accept payment in stroopwafels.

Part of what makes PhpStorm *so* great is a plugin that's designed specifically for Symfony. I'll go to my PhpStorm preferences and, inside, find Plugins, Marketplace then search for Symfony. Here it is. This plugin is amazing.... which you can see because it's been downloaded 5.4 million times! It adds tons of crazy auto-completion features that are specific to Symfony.

If you don't have it already, get it installed. Once it *is* installed, go *back* to Settings and search up here for "Symfony" to find a new Symfony area. The one trick about this plugin is that you need to enable it for each project. So click that check box. Also, it's not too important, but change the web directory to `public/`.

Hit Ok and... we are ready! Let's bring our app to life by creating our very first page next.

Chapter 3: Routes, Controllers & Responses

I gotta say, I miss the 90's. Well, not the beanie babies and... definitely not the way I dressed back then, but... the mix tapes. If you weren't a kid in the 80's or 90's, you may not know how hard it was to share your favorite tunes with your friends. Oh yea, I'm talking about a Michael Jackson, Phil Collins, Paula Abdul mashup. Perfection.

To capitalize off of that nostalgia, but with a hipster twist, we're going to create a brand new app called Mixed Vinyl: a store where users can create mix tapes - complete with Boyz || Men, Mariah Carey and Smashing Pumpkins... except pressed onto a vinyl record. Hmm, I might need to put a record player in my car.

The page we're looking at, which is super cute and changes colors when we refresh... is not a real page. It's just a way for Symfony to say "hi" and link us to the documentation. And by the way, Symfony's documentation is great, so definitely check it out as you're learning.

Routes & Controllers

Ok: *every* web framework in *any* language has the same job: to help us create pages, whether those are HTML pages, JSON API responses or ASCII art. And pretty much every framework does this in the same way: via a route & controller system. The route defines the URL for the page and points to a controller. The controller is a PHP function that builds that page.

So route + controller = page. It's math people.

Creating the Controller

We're going to build these two things... kind of in reverse. So first, let's create the controller function. In Symfony, the controller function is always a *method* inside of a PHP class. I'll show you: in the `src/Controller/` directory, create a new PHP class. Let's call it `VinylController`, but the name could be anything.

8 lines | src/Controller/VinylController.php

```
<?php

namespace App\Controller;

class VinylController
{
}
```

And, congrats! It's our first PHP class! And guess where it lives? In the `src/` directory, where *all* PHP classes will live. And for the most part, it doesn't matter how you organize things inside `src/`: you can usually put things into whatever directory you want and name the *classes* whatever you want. So flex your creativity.

> Controllers actually *do* need to live in `src/Controller/`, unless you change some config. Most PHP classes can live anywhere in `src/`.

But there *are* two important rules. First, notice the namespace that PhpStorm added on top of the class: `App\Controller`. *However* you decide to organize your `src/` directory, the namespace of a class *must* match the directory structure... starting with `App`. You can imagine that the `App\` namespace points to the `src/` directory. Then, if you put a file in a `Controller/` sub-directory, it needs a `Controller` part in its namespace.

If you ever mess this up, like you typo something or forget this, you're gonna have a bad time. PHP will not be able to find the class: you'll get a "class not found" error. Oh, and the *other* rule is that the *name* of a file must match the class name inside of it, plus `.php`. Hence, `VinylController.php`. We'll follow those two rules for *all* files we create in `src/`.

Creating the Controller

Back to our job of creating a controller function. Inside, add a new public method called `homepage()`. And no, the name of this method doesn't matter either: try naming it after your cat: it'll work!

For now, I'm just going to put a `die()` statement with a message.

12 lines | src/Controller/VinylController.php

```
<?php

namespace App\Controller;

class VinylController
{
    public function homepage()
    {
        die('Vinyl: Definitely NOT a fancy-looking frisbee!');
    }
}
```

Creating the Route

Good start! Now that we have a controller function, let's create a route, which defines the *URL* to our new page and *points* to this controller. There are a few ways to create routes in Symfony, but almost everyone uses attributes.

Here's how it works. Right above this method, say `#[]`. This is the PHP 8 attribute syntax, which is a way to add configuration to your code. Start typing `Route`. But before you finish that, notice that PhpStorm is auto-completing it. Hit tab to let *it* finish.

That, *nicely*, completed the word `Route` for me. But *more* importantly, it added a `use` statement up on top. Whenever you use an attribute, you *must* have a corresponding `use` statement for it at the top of the file.

Inside `Route`, pass `/`, which will be the URL to our page.

15 lines | src/Controller/VinylController.php

```
<?php

namespace App\Controller;

use Symfony\Component\Routing\Annotation\Route;

class VinylController
{
    #[Route('/')]
    public function homepage()
    {
        die('Vinyl: Definitely NOT a fancy-looking frisbee!');
    }
}
```

And... done! This route defines the URL and points to this controller... simply because it's right *above* this controller.

Let's try it! Refresh and... congratulations! Symfony looked at the URL, saw that it matched the route - `/` or no slash is the same for the homepage - executed our controller and hit the `die` statement!

Oh, and by the way, I keep saying controller function. That's commonly just called the "controller" or the "action"... just to confuse things.

Returning a Response

Ok, so inside of the controller - or action - we can write *whatever* code we want to build the page, like make database queries, API calls, render a template, whatever. We are going to do all of that eventually.

The *only* thing that Symfony cares about is that your controller returns a `Response` object. Check it out: type `return` and then start typing `Response`. Woh: there are quite a few `Response` classes already in our code... and two are from Symfony! We want the one from HTTP foundation. HTTP foundation is one of those Symfony libraries... and it gives us nice classes for things like the Request, Response and Session. Hit tab to auto-complete and finish that.

Oh, I should have said return new response. That's better. *Now* hit tab. When I let `Response` auto-complete the first time, *very* importantly, PhpStorm added this use statement on top. *Every* time we reference a class or interface, we will need to add a `use` statement to the top of the file we're working in.

By letting PhpStorm auto-complete that for me, it added the `use` statement automatically. I'll do that *every* time I reference a class. Oh, and if you're still a bit new to PHP namespaces and `use` statements, check out our short and free PHP namespaces tutorial.

Anyways, inside of `Response`, we can put whatever we want to return to the user: HTML, JSON or, for now, a simple message, like the title of the Mixed vinyl we're working on: PB and jams.

16 lines | src/Controller/VinylController.php

```php
<?php

namespace App\Controller;

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

class VinylController
{
    #[Route('/')]
    public function homepage()
    {
        return new Response('Title: "PB and Jams"');
    }
}
```

Ok team, let's see what happens! Refresh and... PB and Jams! It may not like much, but we just built our first fully-functional Symfony page! Route + controller = profit!

And you've just learned the most *foundational* part of Symfony... and we're just getting started. Oh, and because our controllers always return a `Response` object, it's optional, but you can add a return type to this function if you want to. But that doesn't change anything: it's just a nice way to code.

16 lines | src/Controller/VinylController.php

```
... lines 1 - 9
    #[Route('/')]
    public function homepage(): Response
... lines 12 - 16
```

Next I'm feeling pretty confident. So let's create *another* page, but with a much fancier route that matches a wildcard pattern.

Chapter 4: Wildcard Routes

The homepage will eventually be the place where a user can design and build their next sweet mix tape. But in addition to *creating* new tapes, users will also be able to browse *other* people's creations.

Creating a Second Page

Let's make a second page for that. How? By adding a second controller: public function, how about `browse`: the name doesn't really matter. And to be responsible, I'll add a `Response` return type.

Above this, we need our route. This will look exactly the same, except set the URL to `/browse`. Inside the method, what do we *always* return from a controller? That's right: a `Response` object! Return a new `Response` ... with a short message to start.

22 lines | src/Controller/VinylController.php

```
... lines 1 - 7
class VinylController
{
... lines 10 - 15
    #[Route('/browse')]
    public function browse(): Response
    {
        return new Response('Breakup vinyl? Angsty 90s rock? Browse the collection!');
    }
}
```

Let's try it! If we refresh the homepage, nothing changes. But if we go to `/browse` ... we're crushing it! A second page in under a minute! Dang!

On this page, we'll eventually list mix tapes from other users. To help find something we like, I want users to *also* be able to browse by *genre*. For example, if I go to `/browse/death-metal`, that would show me all the death metal vinyl mix tapes. Hardcore.

Of course, if we try this URL right now... it doesn't work.

> Not Route found

No matching routes were found for this URL, so it shows us a 404 page. By the way, what you're seeing is Symfony's fancy exception page, because we're currently *developing*. It gives us *tons* of details whenever something goes wrong. When you eventually deploy to production, you can design a *different* error page that your users would see.

{Wildcard} Routes

Anyways, the simplest way to make this URL work is just... to change the URL to `/browse/death-metal`.

22 lines | src/Controller/VinylController.php

```
... lines 1 - 7
class VinylController
{
... lines 10 - 15
    #[Route('/browse/death-metal')]
    public function browse(): Response
    {
        return new Response('Breakup vinyl? Angsty 90s rock? Browse the collection!');
    }
}
```

But... not super flexible, right? We would need one route for *every* genre... which could be hundreds! And also, we just killed the `/browse` URL! *It* now 404's.

What we *really* want is a route that match `/browse/<ANYTHING>`. And we can do that with a *wildcard*. Replace the hard-coded `death-metal` with `{}` and, inside, `slug`. Slug is just a technical word for a "URL-safe name". Really, we could have put anything inside the curly-braces, like `{genre}` or `{coolMusicCategory}`: it makes no difference. But *whatever* we put inside this wildcard, we are then *allowed* to have an argument with that same name: `$slug`.

22 lines | src/Controller/VinylController.php

```
... lines 1 - 7
class VinylController
{
... lines 10 - 15
    #[Route('/browse/{slug}')]
    public function browse(): Response
    {
        return new Response('Breakup vinyl? Angsty 90s rock? Browse the collection!');
    }
}
```

Yup, if we go to `/browse/death-metal`, it will match this route and pass the string `death-metal` to that argument. The matching is done by name: `{slug}` connects to `$slug`.

To see if it's working, let's return a different response: `Genre` then the `$slug`.

24 lines | src/Controller/VinylController.php

```
... lines 1 - 7
class VinylController
{
... lines 10 - 15
    #[Route('/browse/{slug}')]
    public function browse($slug): Response
    {
        return new Response('Genre: '.$slug);

        //return new Response('Breakup vinyl? Angsty 90s rock? Browse the collection!');
    }
... lines 23 - 24
```

Testing time! Head back to `/browse/death-metal` and... yes! Try `/browse/emo` and yea! I'm *that* much closer to my Dashboard Confessional mix tape!

Oh, and it's optional, but you can add a `string` type to the `$slug` argument. That doesn't change anything... it's just a nice way to program: the `$slug` was *already* always going to be a string.

24 lines | src/Controller/VinylController.php

```
... lines 1 - 7
class VinylController
{
... lines 10 - 15
    #[Route('/browse/{slug}')]
    public function browse(string $slug): Response
    {
... lines 19 - 21
    }
... lines 23 - 24
```

A bit later, we'll learn how you could turn a *number* wildcard - like the number 5 - into an integer if you want to.

Using Symfony's String Component

Let's make this page a bit fancier. Instead of printing out the slug exactly, let's convert it to a title. Say

`$title = str_replace()` and replace any dashes with spaces. Then, down here, use title in the response. In a future tutorial, we're going to query the database for these genres, but, for right now, we can at least make it look nicer.

```
26 lines | src/Controller/VinylController.php
... lines 1 - 7
class VinylController
{
... lines 10 - 15
    #[Route('/browse/{slug}')]
    public function browse(string $slug): Response
    {
        $title = str_replace('-', ' ', $slug);

        return new Response('Genre: '.$title);

... line 23
    }
... lines 25 - 26
```

If we try it, Emo doesn't look any different... but death metal *does*. But I want *more* fancy! Add another line with `$title =` then type `u` and auto-complete a function that's literally called... `u`.

We don't use many functions from Symfony, but this is a rare example. This comes from a Symfony library called `symfony/string`. As I mentioned, Symfony is many different libraries - also called components - and we're going to leverage those libraries all the time. This one helps you make string transformations... and it happens to already be installed.

Move the `str_replace()` to the first argument of `u()`. This function returns an object that we can then do string operations on. One method is called `title()`. Say `->title(true)` to convert all words to title case.

```
27 lines | src/Controller/VinylController.php
... lines 1 - 6
use function Symfony\Component\String\u;
... line 8
class VinylController
{
... lines 11 - 15

    #[Route('/browse/{slug}')]
    public function browse(string $slug): Response
    {
        $title = u(str_replace('-', ' ', $slug))->title(true);

        return new Response('Genre: '.$title);
... lines 23 - 24
    }
... lines 26 - 27
```

Now whe n we try it... sweet! It uppercases the letters! The string component isn't particularly important, I just want you to see how we can already leverage parts of Symfony to get our job done.

Making the Wildcard Optional

Ok: one last challenge. Going to `/browse/emo` or `/browse/death-metal` works. But just going to `/browse` ... does *not* work. It's broken! A wild card can match anything, but, by default, a wild card is *required*. We *have* to go to `/browse/<something>`.

Can we make a wildcard optional? Absolutely! And it's delightfully simple: make the corresponding argument optional.

27 lines | src/Controller/VinylController.php

```
... lines 1 - 8
class VinylController
{
... lines 11 - 15

    #[Route('/browse/{slug}')]
    public function browse(string $slug = null): Response
    {
... lines 20 - 24
    }
... lines 26 - 27
```

As soon as we do that, it tells Symfony's routing layer that the `{slug}` does not need to be in the URL. So now when we refresh... it works. Though, that's not a great message for the page.

Let's see. *If* there's a slug, then set the title the way we were. Else, set `$title` to "All genres". Oh, and move the "Genre:" up here... so that down in the `Response` we can just pass `$title`.

31 lines | src/Controller/VinylController.php

```
... lines 1 - 8
class VinylController
{
... lines 11 - 15

    #[Route('/browse/{slug}')]
    public function browse(string $slug = null): Response
    {
        if ($slug) {
            $title = 'Genre: '.u(str_replace('-', ' ', $slug))->title(true);
        } else {
            $title = 'All Genres';
        }

        return new Response($title);
... lines 27 - 28
    }
... lines 30 - 31
```

Try that. On `/browse` ... "All Genres". On `/browse/emo` ... "Genre: Emo".

Next: putting text like this into a controller.... isn't very clean or scalable, especially if we start including HTML. Nope, we need to render a template. To do *that*, we're going to install our first third-party package and witness the massively important Symfony recipe system in action.

Chapter 5: Symfony Flex: Aliases, Packs & Recipes

Symfony is a set of libraries that gives us tons of tools: tools for logging, making database queries, sending emails, rendering templates and making API calls, just to name a few. If you counted them, I did, Symfony consists of about 100 separate libraries. Wow!

Right now, I want to start turning our pages into true HTML pages... instead of just returning text. But we are *not* going to jam a bunch of HTML into our PHP classes. Yuck. Instead, we're going to render a template.

Symfony's Start Small & Install Features Philosophy

But guess what? There is no templating library in our project! What? But I thought you just said that Symfony has a tool for rendering templates!? Lies!

Well... Symfony *does* have a tool for that. But our app currently uses very *few* of the Symfony libraries. The tools we have so far don't amount to much more than a route-controller-response system. If you need to render a template or make a database query, we do *not* have *those* tools installed in our app... yet.

I actually *love* this about Symfony. Instead of starting us with a gigantic project, with everything we need, plus *tons* of stuff that we *don't* need, Symfony starts tiny. *Then*, if you need something, you install it!

But before we install a templating library, at your terminal, run:

```
git status
```

Let's commit everything:

```
git add .
```

I can safely run `git add .` - which adds *everything* in my directory to git - because one of the files that our project *originally* came with was a `.gitignore` file, which already ignores stuff like the `vendor/` directory, `var/` directory, and several other paths. If you're wondering what these weird marker things are, that's related to the recipe system, which we're about to talk about.

Anyways, run `git commit` and add a message:

```
git commit -m "route -> controller -> response -> profit"
```

Perfect! And now, we are clean.

Installing a Templating Library (Twig)

Okay. So how can we install a templating library? And what templating libraries are even *available* for Symfony? And which is recommended? Well, of course, a great way to answer these questions would be check the Symfony documentation.

But we can also just... guess! In *any* PHP project, you can add new third-party libraries to your app by saying "composer require" and then the package name. We don't *know* the package name we need yet, so I'll just

guess:

```
composer require templates
```

Now, if you've used Composer before, you might be *screaming* at your screen right about now! Why? Because in Composer, package names are *always* `something/something`. It is literally *not* possible to have a package just named `templates`.

But watch: when we run this, it works! And up on top, it says using version 1 for `symfony/twig-pack`. Twig is the name of the templating engine for Symfony.

Flex Aliases

To understand this, let's take a tiny step backwards. Our project started with a `composer.json` file containing several Symfony libraries. One of these is called `symfony/flex`. Flex is a composer *plugin*. So it *adds* more features to composer. Actually, it adds *three* superpowers to composer.

> 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!

The first, which we just saw, is called Flex aliases. Head to https://flex.symfony.com to see a *giant* page full of packages. Search for "templates". Here it is. Under `symfony/twig-pack`, it says Aliases: template, templates, twig, and twig-pack.

The idea between behind Flex aliases is dead simple. We type `composer require templates`. And then, internally, Flex *changes* that to `symfony/twig-pack`. Ultimately, *that* is the package that Composer installs.

This means that, most of the time, you can just "composer require" whatever you want, like `composer require logger`, `composer require orm`, `composer require icecream`, whatever. It's just a shortcut system. The important point is that, what *really* got installed was `symfony/twig-pack`.

Flex Packs

And *that* means that, in our `composer.json` file, we *should* now see `symfony/twig-pack` under the `require` key. But if you spin over, it's not there! Gasp! Instead, it added `symfony/twig-bundle`, `twig/extra-bundle`, and `twig/twig`.

We're witnessing the *second* superpower of Symfony Flex: unpacking packs. Copy the original package name and... we can actually find that repository on GitHub by going to https://github.com/symfony/twig-pack.

And... it contains just *one* file: `composer.json`. And *this* requires three *other* packages: the three we just saw added to *our* project.

This is called a Symfony pack. It's... really just a fake package that helps us install *other* packages. It turns out, if you want a rich template engine to be added to your app, we recommend installing these *three* packages. But instead of making you add them manually, you can composer require `symfony/twig-pack` and get them automatically. When you install a "pack", like this, Flex automatically "unpacks" it: it finds the three packages that the pack depends on and adds *those* into your `composer.json` file.

So, packs are a shortcut so that you can run one `composer require` command and get *multiple* libraries added to your project.

Ok, so what is the third and final superpower of Flex? So glad you asked! To find out, at your terminal, run:

```
git status
```

Flex Recipes

Whoa. Normally when you run `composer require`, the only files it should modify - other than downloading packages into `vendor/` - are `composer.json` and `composer.lock`. Flex's third superpower is its recipes system.

Whenever you install a package, that package *may* have a *recipe*. If it does, in addition to downloading the package into the `vendor/` directory, Flex will also *execute* its recipe. Recipes can do things like add new files or even modify a few existing files.

Watch: if we scroll up a little, ah yes: it says "configuring 2 recipes". So apparently there was a recipe for `symfony/twig-bundle` and also a recipe for `twig/extra-bundle`. And these recipes apparently updated the `config/bundles.php` file and added a new directory and file.

The recipe system is *sweet*. All *we* need to do is composer require a new library and its recipe will then add all the configuration files or other setup needed so that we can start *using* that library immediately! No more following 5 manual "installation" steps in a README. When you add a library, it works out-of-the-box.

Next: I want to dive a bit deeper into the recipes. Like, where do they live? What's their favorite color? And what did this recipe added specifically to our app and why? I'm also going to let you in on a little secret: *every* file on our project - all the files in `config/`, the `public/` directory... *all* of this stuff - was added via a recipe. And I'll prove it.

Chapter 6: Flex Recipes

We just installed a new package by running `composer require templates`. *Normally* when you do that, Composer will update the `composer.json` and `composer.lock` files, but nothing else.

But when we run:

```
git status
```

There are *other* changes. This is thanks to Flex's recipe system. Each time we install a new package, Flex checks a central repository to see if that package has a recipe. And if it does, it installs it.

Where do Recipes Live?

Where do these recipes live? In the cloud... or more specifically GitHub. Check it out. Run:

```
composer recipes
```

This is a command *added* to composer by Flex. It lists all of the recipes that have been installed. And if you want more info about one, just run:

```
composer recipes symfony/twig-bundle
```

This is one of the recipes that was just executed. And... cool! It shows us a couple of nice things! The first is a tree of the files it *added* to our project. The second is a URL to the recipe that was installed. I'll click to open that.

Yep! Symfony recipes live in a special repository called `symfony/recipes`. This is a big directory organized by package name. There's a `symfony` directory that holds recipes for all packages starting with `symfony/`. The one we were just looking at... is way down here: `twig-bundle`. And then there are different versions of the recipe based on your version of the package. We're using the latest 5.4 version.

Every recipe has a `manifest.json` file, which controls what it does. The recipe system can only do a specific set of operations, including adding new files to your project and modifying a few *specific* files. For example, this `bundles` section tells flex to add this line to our `config/bundles.php` file.

If we run `git status` again... yup! That file *was* modified. If we diff it:

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

It added *two* lines, probably one for each of the *two* recipes.

Symfony Bundles?

By the way, `config/bundles.php` is not a file that you need to think about much. A bundle, in Symfony land, is basically a plugin. So if you install a new bundle into your app, that gives you new Symfony features. In order to *activate* that bundle, its name needs to live in this file.

So the first thing that the recipe did for twig-bundle, thanks to this line up here, was to activate itself inside `bundles.php` ... so that we didn't need to do it manually. Recipes are like automated installation.

New, Copied Files

The *second* section in the manifest is called `copy-from-recipe`. This is simple: it says to copy the `config/` and `templates/` directories from the recipe into the project. If we look... the recipe contains a `config/packages/twig.yaml` file... and also a `templates/base.html.twig` file.

Back at the terminal, run `git status` again. We see these two files at the bottom: `config/packages/twig.yaml` ... and inside of `templates/`, `base.html.twig`.

I *love* this. Think about it: if you install a templating tool into your app, we're going to need some configuration *somewhere* that tells that templating tool which directory to look inside of to find our templates. Whelp, go check out that `config/packages/twig.yaml` file. We're going talk about these Yaml files more in the next tutorial. But on a high level, this file controls how Twig - the templating engine for Symfony - behaves. And check out the `default_path` key set to `%kernel.project_dir%/templates`. Don't worry about this percent syntax: that's a fancy way to refer to the root of our project.

The point is, this config says:

> Hey Twig! When you look for templates, look for them in the `templates/` directory.

And then the recipe even *created* that directory with a layout file inside. We'll use this in a few minutes.

symfony.lock & Committing Files

The *last* unexplained file that was modified is `symfony.lock`. This is *not* important: it just keeps track of which recipes have been installed... and you *should* commit it.

In fact, we should commit *all* of this stuff. The recipe might give us files, but then they are *our's* to modify. Run:

```
git add .
```

Then:

```
git status
```

Cool. Let's commit!

```
git commit -m "Adding Twig and its beautiful recipe"
```

Updating Recipes

Done! By the way, a few months down the road, there might be *changes* to some of the recipes that you've installed. And if there *are*, when you run

```
composer recipes
```

you'll see a little "update available" next to them. Run `composer recipes:update` to upgrade to the latest version.

Oh, and before I forget, in addition to `symfony/recipes`, there is also a `symfony/recipes-contrib` repository. So recipes can live in *either* of these two places. The recipes in `symfony/recipes` are approved by Symfony's core team, so they're a bit more vetted for quality. Other than that, there's no difference.

Our Project Started as One File

Now, the recipe system is *so* powerful that *every* single file in our project was *added* via a recipe! I can prove it. Go to https://github.com/symfony/skeleton.

When we originally ran that `symfony new` command to start our project, what that *really* did was *clone* this repository... and then ran `composer install` inside of it, which downloads everything into the `vendor/` directory.

Yup! Our project - the one that we see right here - was originally *just* a single file: `composer.json`. But then, when the packages were installed, the *recipes* for those packages added *everything* else we see. Run:

```
composer recipes
```

again. One recipe is for something called `symfony/console`. Check out its details:

```
composer recipes symfony/console
```

And... yes! The recipe for `symfony/console` added the `bin/console` file! The recipe for `symfony/framework-bundle` - one of the other packages that was originally installed - added almost everything else, including the `public/index.php` file. How cool is that?

Okay next: we installed Twig! So let's get back to work and use it to render some templates! You're going to love Twig.

Chapter 7: Twig ❤

Symfony controller classes do *not* need to extend a base class. As long as your controller function returns a `Response` object, Symfony doesn't care *what* your controller looks like. But usually, you *will* extend a class called `AbstractController`.

Why? Because it gives us shortcut methods.

Rendering a Template

And the *first* shortcut is `render()`: the method for rendering a template. So return `$this->render()` and pass it two things. The first is the name of the template. How about `vinyl/homepage.html.twig`.

It's not required, but it's common to have a directory with the same name as your controller class and filename that's the same as your method, but you can do whatever. The second argument is an array of any variables that you want to pass *into* the template. Let's pass in a variable called `title` and set it to our mix tape title: "PB and Jams".

34 lines | src/Controller/VinylController.php

```
... lines 1 - 4
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
... lines 6 - 8

class VinylController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
        return $this->render('vinyl/homepage.html.twig', [
            'title' => 'PB & Jams',
        ]);
    }
... lines 19 - 34
```

Done in here. Oh, but pop quiz! What do you think the `render()` method returns? Yea, it's the thing I *keep* repeating: a controller must always return a `Response` object. `render()` is just a shortcut to render a template, get that string and put it into a `Response` object. `render()` returns a `Response`.

Creating the Template

We know from earlier that when you render a template, Twig looks in the `templates/` directory. So create a new `vinyl/` sub-directory... and inside of that, a file called `homepage.html.twig`. To start, add an `h1` and then print the `title` variable with a special Twig syntax: `{{ title }}`. And... I'll add some hardcoded TODO text.

8 lines | templates/vinyl/homepage.html.twig

```
<h1>{{ title }}</h1>

{# TODO: add an image of the record #}

<div>
    Our schweet track list: TODO
</div>
```

Let's... go see if this works! We were working on our homepage, so go there and... hello Twig!

Twigs 3 Syntax

Twig is one of the *nicest* parts of Symfony, and also one of the easiest. We're going to go through everything you need to know... in basically the next ten minutes.

Twig has exactly *three* different syntaxes. If you need to print something, use `{{` . I call this the "say something" syntax. If I say `{{ saySomething }}` that would print a variable called `saySomething` . Once you're *inside* Twig, it looks a *lot* like JavaScript. For example, if I surround this in quotes, now I'm printing the *string* `saySomething` . Twig has functions... so that would call the function and print the result.

So syntax #1 - the "say something" syntax - is `{{`

The second syntax... doesn't really count. It's `{#` to create a comment... and that's it.

8 lines | templates/vinyl/homepage.html.twig

```
<h1>{{ title }}</h1>

{# TODO: add an image of the record #}

<div>
    Our schweet track list: TODO
</div>
```

The *third* and final syntax I call the "do something" syntax. This is when you're not *printing*, your *doing* something in the language. Examples of "doing something" would be if statements, for loops or setting variables.

The for Loop

Let's try a `for` loop. Go back to the controller. I'm going to paste in a tracks list... and then pass a `tracks` variable into the template set to that array.

44 lines | src/Controller/VinylController.php

```
<?php
... lines 2 - 8

class VinylController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
        $tracks = [
            'Gangsta\'s Paradise - Coolio',
            'Waterfalls - TLC',
            'Creep - Radiohead',
            'Kiss from a Rose - Seal',
            'On Bended Knee - Boyz II Men',
            'Fantasy - Mariah Carey',
        ];

        return $this->render('vinyl/homepage.html.twig', [
            'title' => 'PB & Jams',
            'tracks' => $tracks,
        ]);
    }
... lines 29 - 42
}
```

Now, unlike `title` , tracks is an array... so we can't just print it. But, we can try! Ha! That gives us an array to string conversion. Nope, we need to *loop* over tracks.

Add a header and a `ul` . To loop, we'll use the "do something" syntax, which is `{%` and then the *thing* that you want to do, like `for` , `if` or `set` . I'll show you the full list of do something tags in a minute. A for loop looks like this: `for track in tracks` , where tracks is the variable we're looping over and `track` will be the variable *inside* the loop.

After this, add `{% endfor %}` : most "do something" tags have an end tag. *Inside* the loop, add an `li` and then use the say something syntax to print `track` .

16 lines | templates/vinyl/homepage.html.twig

```
<h1>{{ title }}</h1>

{# TODO: add an image of the record #}

<div>
    Tracks:

    <ul>
        {% for track in tracks %}
            <li>
                {{ track }}
            </li>
        {% endfor %}
    </ul>
</div>
```

Using Sub.keys

When we try it... nice! Oh, but let's get *trickier*. Back in the controller, instead of using a simple array, I'll restructure this to make each track an associative array with `song` and `artist` keys. I'll paste in that same change for the rest.

44 lines | src/Controller/VinylController.php

```
<?php
... lines 2 - 8

class VinylController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
        $tracks = [
            ['song' => 'Gangsta\'s Paradise', 'artist' => 'Coolio'],
            ['song' => 'Waterfalls', 'artist' => 'TLC'],
            ['song' => 'Creep', 'artist' => 'Radiohead'],
            ['song' => 'Kiss from a Rose', 'artist' => 'Seal'],
            ['song' => 'On Bended Knee', 'artist' => 'Boyz II Men'],
            ['song' => 'Fantasy', 'artist' => 'Mariah Carey'],
        ];
... lines 23 - 27
    }
... lines 29 - 42
}
```

What happens if we try it? Ah, we're back to the "array to string" conversion. When we loop, each track *itself* is now an array. How can we read the `song` and `artist` keys?

Remember when I said that Twig looks a lot like JavaScript? Well then, it shouldn't be a surprise that the answer is `track.song` and `track.artist`.

16 lines | templates/vinyl/homepage.html.twig

```
... lines 1 - 7
    <ul>
        {% for track in tracks %}
            <li>
                {{ track.song }} - {{ track.artist }}
            </li>
        {% endfor %}
    </ul>
... lines 15 - 16
```

And... *that* gets our list working.

Now that we have the basics of Twig down, next, let's look at the *full* list of "do something" tags, learn about Twig "filters" *and* tackle the all-important template inheritance system.

Chapter 8: Twig Inheritance

Head to https://twig.symfony.com... and then click to check its documentation. There's lots of good stuff here. But what I want you to do is scroll down to the Twig reference. Yea!

Tags

The first things to look at, on the left, are these things called tags. This list represents *every* possible thing you can use with the do something syntax. Yup, it will always be `{%` and then *one* of these things, like `for` or `if`. And honestly, you're only going to use about 5 of these on an everyday basis. If you want to know the syntax for one of these, just click to see its docs.

Filters

In addition to the 20 tags, Twig also has something called *filters*. These are sweet. Filters are basically functions, but with a more hipster syntax. Twig *does* also have functions, but there are fewer: Twig really prefers filters: they're way cooler!

For example, there's a filter called `upper`. Using a filter is like using the `|` key on the command line. You have some value - then you "pipe it into" the filter you want, like `upper`.

Let's try this! Print `track.artist|upper`.

16 lines | templates/vinyl/homepage.html.twig

```
... lines 1 - 10
11              {{ track.song }} - {{ track.artist|upper }}
... lines 12 - 16
```

And now... it's uppercase! If you want to confuse your coworkers, you can pipe *that* to `lower` ... which sends things *back* to lowercase. There's no *real* reason to do this, but filters *can* be chained like this.

16 lines | templates/vinyl/homepage.html.twig

```
... lines 1 - 10
11              {{ track.song }} - {{ track.artist|upper|lower }}
... lines 12 - 16
```

Anyways, check out the filters list because there's probably something you'll find useful.

And... that's pretty much it! Beyond functions, there's also something called "tests", which are handy in if statements: you can say things like "if number is divisibleby 5".

Template Inheritance

Ok, just *one* more thing to learn about Twig, and it's cool.

View the HTML source of the page. Notice that there is *no* HTML structure: there's no `html`, `head` or `body` tags. Literally the HTML that we have inside of our template, is what we get. Nothing more.

So is there... some sort of layout system in Twig where we can add a base layout around us? Absolutely. And it's incredible. It's called template inheritance. If you have a template and you want that to use some base layout, at the very top of the file, use a "do something" tag called `extends`. Pass this the name of the layout file: `base.html.twig`.

18 lines | templates/vinyl/homepage.html.twig

```
1  {% extends 'base.html.twig' %}
... lines 2 - 18
```

That's referring to this template right here. Before we check that out, if we try this now, yikes! Big error:

> A template that extends another cannot include content outside Twig blocks.

To figure out what that means, open `base.html.twig`. This is your base layout file... and you're *totally* free to customize it however you want. Right now... it's mostly just boring HTML tags... except for a number of these "blocks".

Blocks are basically "holes" into which a child template can place content. Let me explain that in a different way. When we say `extends 'base.html.twig'`, that basically says:

> Yo Twig! When you render this template, I want you to *actually* render `base.html.twig` ... and then put *my* content *inside* of it.

Twig then politely replies:

> Ok cool... I can do that. But *where* in `base.html.twig` do you want me to put all of your content? Do you want me to put it at the bottom of the page? At the top? Some random place in the middle?

The way we tell Twig *where* to put our content within `base.html.twig` is by override a block. Notice that `base.html.twig` already has a block called `body` ... and that's *right* where we want to put our template's HTML.

To put it there, in our template, surround all of the content with `{% block body %}` ... and then `{% endblock %}`.

20 lines | templates/vinyl/homepage.html.twig

```
{% extends 'base.html.twig' %}

{% block body %}
<h1>{{ title }}</h1>

{# TODO: add an image of the record #}

<div>
    Tracks:

    <ul>
        {% for track in tracks %}
            <li>
                {{ track.song }} - {{ track.artist }}
            </li>
        {% endfor %}
    </ul>
</div>
{% endblock %}
```

This is called template inheritance because we are *overriding* that `body` block with this new content. So now, when Twig renders `base.html.twig` ... and it gets to this `block body` part, it's going to print the `block body` HTML from *our* template

Watch: refresh and... the error is gone. And if you view the page source, we have a full HTML page!

Block Names

Oh, and the *names* of these blocks are *not* important. If you want to rename them after your favorite 90's sitcom character, do it. Just remember to *also* update its name in any child templates.

You can also add *more* blocks. Every block you add is just another potential override point.

Default Block Content

Oh, and you may have noticed that blocks *can* have default content. Look at the page right now: the title says "Welcome". That's because the `title` block has default content... and we're not overriding it. Let's change the default title to "Mixed Vinyl".

20 lines | templates/base.html.twig

```
... lines 1 - 4
5         <title>{% block title %}Mixed Vinyl{% endblock %}</title>
... lines 6 - 20
```

So now *that* will be the title of every page on our site... *unless* we override that. In our template, either above block body or below - the order of blocks doesn't matter - add `{% block title %}`, `{% endblock %}` and, in between "Create a new Record".

22 lines | templates/vinyl/homepage.html.twig

```
1  {% extends 'base.html.twig' %}
2
3  {% block title %}Create a new Record{% endblock %}
4
5  {% block body %}
... lines 6 - 20
21 {% endblock %}
```

And now... yes! *This* page has a custom title.

Adding to (Instead of Replacing) the Parent Block

Oh, and you might be wondering:

> What if I don't want to *replace* a block entirely.... but instead, I want to *add* to a block?

That's totally possible. In `base.html.twig`, the `title` block is set to "Mixed Vinyl". If we wanted to *prepend* our custom title to that, we could say "Create a new Record" then use the "say something" tag to print a function called `parent()`.

22 lines | templates/vinyl/homepage.html.twig

```
1  {% extends 'base.html.twig' %}
2
3  {% block title %}Create a new Record | {{ parent() }}{% endblock %}
4
5  {% block body %}
... lines 6 - 20
21 {% endblock %}
```

That does exactly what you'd expect: it finds the parent template's content for this block.. and prints it. Refresh and... that's *so* nice.

Template Inheritance is Class Inheritance

If you're ever confused about how template inheritance works, it's useful, for me at least, to think about it *exactly* like object-oriented inheritance. Each template is like a class and each block is like a method. So the homepage "class" extends the `base.html.twig` "class", but overrides two of its methods. If that only confused you, don't worry about it.

So... that's it for Twig. You're basically a Twig expert, which I'm told is a popular topic at parties.

Next: one of the *killer* features of Symfony is its debugging tools. Let's get these installed and check 'em out.

Chapter 9: Profiler: Your Debugging Best Friend

Time to install our *second* package. And this one is *fun*. Let's commit our changes first: it'll makes it easier to check out any changes that the new package's recipe makes.

Add everything:

```
git add .
```

That looks fine so... commit:

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

Beautiful.

The debug Pack

Now run:

```
composer require debug
```

So yes, this is *another* Flex alias... and apparently it's an alias for `symfony/debug-pack`. And *we* know that a pack is a *collection* of packages. So instead of adding this *one* line to our `composer.json` file, if we check, it looks like it added one new package up under the `require` section - this is a logging library - and... *all* the way at the bottom, it added a new `require-dev` section with three *other* libraries.

The difference between `require` and `require-dev` isn't too important: all of these packages were downloaded into our app, But as a best practice, if you install a library that's only meant for *local* development, you should put it into `require-dev`. The pack did that for us! Thanks pack!

Recipe Changes

Back at the terminal, this also installed three recipes! Ooh. Let's see what those did. I'll clear the screen and run:

```
git status
```

So this is familiar: it modified `config/bundles.php` to activate three new bundles. Again, bundles are Symfony plugins, which add more features to our app.

It also added several configuration files to the `config/packages/` directory. We will talk more about these files in the next tutorial, but, on a high level, they control the *behavior* of those bundles.

The Web Debug Toolbar And Profiler

So what *did* these new bundles give us? To find out, head over to your browser and refresh the homepage. Holy cats, Batman! It's the web debug toolbar. This is debugging *madness*: a toolbar *full* of good info. On the left, you can see the controller that was called along with the HTTP status code. There's also the amount of time the page took, the memory it used and also how many templates were rendered through Twig: this is the cute Twig icon.

On the right side, we have details about the Symfony local web server that's running and PHP info.

But you haven't seen the best part: click any of these icons to jump into the *profiler*. This is the web debug toolbar... gone crazy. It's *full* of data about that request, like the request and response, all of the log messages that happened during that request, information about the routes and the route that was matched, Twig shows you which templates were rendered and how many *times* they were rendered... and there's configuration information down here. Phew!

But my *most* favorite section is Performance. This shows a timeline of everything that happened during the request. This is great for two reasons. The first is pretty obvious: you can use this to find which parts of your page are slow. So, for example, our controller took 20.4 millisecond. And *within* the controller's execution, the homepage template rendered in 3.9 milliseconds and `base.html.twig` rendered in 2.8 milliseconds.

The second reason this is really cool is that it uncovers all the hidden layers of Symfony. Set this threshold down to zero. Previously, this only displayed things that took *more* than one millisecond. Now it's showing *everything*. You don't need to worry about the vast majority of the stuff, but it's *super* cool to see the layers of Symfony: the things that happen before and after your controller is executed. We have a deep dive tutorial for Symfony if you want to learn more about this stuff.

The web debug toolbar and profiler will also grow with our app. In a future tutorial, when we install a library to talk to the database, we will suddenly have a new section that lists all of the database queries that a page made and how long each took.

dump() and dd() Functions

Ok, so the debug pack installed the web debug toolbar. It also installed a logging library that we'll use later. *And* it installed a package that gives us two fantastic debug functions.

Head over to `VinylController`. Pretend that we're doing some developing and we need to see what this `$tracks` variable looks like. It's pretty obvious in this case, but sometimes you'll want to see what's inside a complex object.

To do that, say `dd($tracks)` where "dd" stands for dump and die.

45 lines | src/Controller/VinylController.php

```
... lines 1 - 9
class VinylController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
... lines 15 - 22
        dd($tracks);
... lines 24 - 28
    }
... lines 30 - 43
}
```

So if we refresh... yup! That dumps the variable and kills the page. And this is a *lot* more powerful - and prettier - than using `var_dump()`: we can expand sections and see deep data really easily.

Instead of `dd()`, you can also use `dump()`. to dump and *live*. But this might not show up where you expect it to. Instead of printing in the middle of the page, it shows up down in the web debug toolbar, under the target icon.

45 lines | src/Controller/VinylController.php

```
... lines 1 - 9
class VinylController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
... lines 15 - 22
        dump($tracks);
... lines 24 - 28
    }
... lines 30 - 43
}
```

If that's a bit too small, click to see a bigger version in the profiler.

Dumping in Twig

You can *also* use this `dump()` in Twig. Remove the dump from the controller... and then in the template, right before the `ul`, `dump(tracks)`.

23 lines | templates/vinyl/homepage.html.twig

```
... lines 1 - 9
<div>
    Tracks:

    {{ dump(tracks) }}
... lines 14 - 20
</div>
... lines 22 - 23
```

And this... looks exactly the same. Except that when you dump in Twig, it *does* dump right into the middle of the page

And even *more* useful, in Twig *only*, you can use `dump()` with no arguments.

23 lines | templates/vinyl/homepage.html.twig

```
... lines 1 - 9
<div>
    Tracks:

    {{ dump() }}
... lines 14 - 20
</div>
... lines 22 - 23
```

This will dump *all* variables that we have access to. So here's the `title` variable, `tracks` and, surprise! There's a third variable called `app`. This is a global variable that we have in *all* templates... and it gives us access to things like the session and user data. And... we just discovered it by being curious!

So now that we've got these awesome debugging tools, let's turn to our next job... which is to make this site less ugly. Time to add CSS and a proper layout to bring our site to life!

Chapter 10: Assets, CSS, Images, etc

If you download the course code from the page where you're watching this video, after unzipping, you'll find a `start/` directory that contains the same brand new Symfony 6 app that we created earlier. You don't actually *need* that code, but it *does* contain one extra directory called `tutorial/`, like I have here. This holds some files that we're about to use.

So let's talk about our next goal: to make this site look like a *real* site... instead of looking like something I designed myself. And *that* means we need a true HTML layout that brings in some CSS.

Adding a Layout & CSS Files

We know that our layout file is `base.html.twig` ... and there's *also* a `base.html.twig` file in the new `tutorial/` directory. Copy that... paste it into templates, and override the original.

Before we look at that, *also* copy the three `.png` files and put those into the `public/` directory... so that our users can access them.

Beautiful. Open up the new `base.html.twig` file. There's nothing special here. We bring in some *external* CSS files from some CDNs for Bootstrap and FontAwesome. By the end of this tutorial, we'll refactor this into a *fancier* way of handling CSS... but for right now, this will work great.

But otherwise, everything is still hardcoded. We have some hardcoded navigation, the same block `body` ... and a hardcoded footer. Let's go see what it looks like. Refresh and woo! Well, not perfect, but better!

Adding a Custom CSS File

The `tutorial/` directory also holds an `app.css` file with custom CSS. To make this publicly available so that our user's browser can download it, it needs to live *somewhere* in the `public/` directory. But it doesn't matter where or how you organize things inside.

Let's create a `styles/` directory... and then copy `app.css` ... and paste it there.

Back in `base.html.twig`, head to the top. After all the external CSS files, let's add a link tag for *our* `app.css`. So `<link rel="stylesheet"` and `href=""`. Because the `public/` directory is our document root, to refer to a CSS or image file there, the path should be with respect to *that* directory. So this will be `/styles/app.css`.

87 lines | templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
... lines 4 - 15
        <link rel="stylesheet" href="/styles/app.css">
... lines 17 - 25
    </head>
... lines 27 - 85
</html>
```

Let's check it. Refresh now and... even better!

The asset() Function

I want you to notice something. So far, Symfony is *not* involved at *all* in how we organize or use images or CSS files. Nope. Our setup is dead simple: we put things in the `public/` directory... then refer to them with their paths.

But *does* Symfony have any cool features to help work with CSS and JavaScript? Absolutely. It's called Webpack Encore and Stimulus. And we'll talk about both of those towards the end of the tutorial.

But even in this simple setup - where we just put files in `public/` and point to them - Symfony does have one *minor* feature: the `asset()` function.

It works like this: instead of using `/styles/app.css`, say `{{ asset() }}` and then, inside quotes, move our path there... but *without* the opening "/".

87 lines | templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
... lines 4 - 15
        <link rel="stylesheet" href="{{ asset('styles/app.css') }}">
... lines 17 - 25
    </head>
... lines 27 - 85
</html>
```

So the path is still relative to the `public/` directory... you just don't need to include the first "/".

Before we talk about what this does... let's go see if it works. Refresh and... it doesn't! Error:

> Unknown function: did you forget to run `composer require symfony/asset`.

I keep saying that Symfony starts small... and then you install things *as* you need them. Apparently, this `asset()` function comes from a part of Symfony that we don't have yet! But getting it is easy. Copy this composer require command, spin over to your terminal and run it:

```
composer require symfony/asset
```

This is a pretty simple install: it downloads just this one package... and there are no recipes.

But when we try the page now... it works! Check out the HTML source. Interesting: the `link` tag's `href` is still, literally, `/styles/app.css`. That's *exactly* what we had before! So what the heck is this `asset()` function doing?

The answer is... not much. But it's still a good idea to use. The `asset()` function gives you two features. First, imagine you deploy to a sub-directory of a domain. Like, the homepage lives at https://example.com/mixed-vinyl.

If that were the case, then in order for our CSS to work, the `href` would need to be `/mixed-vinyl/styles/app.css`. In this situation, the `asset()` function would detect the sub-directory automatically and add that prefix for you.

The second - and more important thing that the `asset()` function does - is allow you to easily switch to a CDN later. Because this path is now going through the `asset()` function, we could, via a configuration file, say:

> Hey Symfony! When you output this path, please prefix it with the URL to my CDN.

This means that, when we load the the page, instead of `href="/styles/app.css`, it would be something like `https://mycdn.com/styles/app.css`.

So the `asset()` function might not be doing anything you need *today*, but anytime you reference a static file - whether it's a CSS file, JavaScript file, image, whatever, use this function.

In fact, up here, I'm referencing three images. Let's use `asset`: `{{ asset()` ... and then it auto-completes the path! Thanks Symfony plugin! Repeat this for the second image... and the third.

87 lines | templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
... lines 4 - 6
        <link rel="apple-touch-icon" sizes="180x180" href="{{ asset('apple-touch-icon.png') }}">
        <link rel="icon" type="image/png" sizes="32x32" href="{{ asset('favicon-32x32.png') }}">
        <link rel="icon" type="image/png" sizes="16x16" href="{{ asset('favicon-16x16.png') }}">
... lines 10 - 15
        <link rel="stylesheet" href="{{ asset('styles/app.css') }}">
... lines 17 - 25
    </head>
... lines 27 - 85
</html>
```

We know this won't make any difference today... we can refresh the HTML source to see the same paths... but we're ready for a CDN in the future.

Homepage And Browse Page HTML

So the layout now looks great! But the content for our homepage is... just kind of hanging out... looking weird... like me in middle school. Back in the `tutorial/` directory, copy the homepage template... and overwrite our original file.

Open that up. This still extends `base.html.twig` ... and it still overrides the `body` block. And then, it has a bunch of completely hard coded HTML. Let's go see what it looks like. Refresh and... it looks awesome!

Except that... it's 100% hard coded. Let's fix that. All the way on top, here's the name of our record, print the `title` variable.

And then, below for the songs.. we have a long list of hardcoded HTML. Let's turn this into a loop. Add `{% for track in tracks %}` like we had before. And... at the bottom, `endfor` .

For the song details, use `track.song` ... and `track.artist` . And now we can remove *all* the hardcoded songs.

58 lines | templates/vinyl/homepage.html.twig

```
{% extends 'base.html.twig' %}
... lines 2 - 4
{% block body %}
<div class="container">
    <h1 class="d-inline me-3">{{ title }}</h1> <i class="fas fa-edit"></i>
    <div class="row mt-5">
... lines 9 - 34
        <div class="col-12 col-md-8 ps-5">
            <h2 class="mb-4">10 songs (30 minutes of 60 still available)</h2>
            {% for track in tracks %}
            <div class="song-list">
                <div class="d-flex mb-3">
                    <a href="#">
                        <i class="fas fa-play me-3"></i>
                    </a>
                    <span class="song-details">{{ track.song }} - {{ track.artist }}</span>
                    <a href="#">
                        <i class="fas fa-bars mx-3"></i>
                    </a>
                    <a href="#">
                        <i class="fas fa-times"></i>
                    </a>
                </div>
            </div>
            {% endfor %}
            <button type="button" class="btn btn-success"><i class="fas fa-plus"></i> Add a song</button>
        </div>
    </div>
</div>
{% endblock %}
```

Sweet! Let's try that. Hey! It's coming to life people!

One more page to go! The `/browse` page. You know the drill: copy `browse.html.twig`, and paste into our directory. This looks a lot like the homepage: it extends `base.html.twig` and overrides block `body`.

Over in `VinylController`, we weren't rendering a template before... so let's do that now: `return $this->render('vinyl/browse.html.twig')` and let's pass in the genre. Add a variable for that: `$genre =` and if we have a slug... use our fancy title-case code, else set this to null. Then delete the `$title` stuff... and pass `genre` into Twig.

40 lines | src/Controller/VinylController.php

```
<?php

... lines 3 - 9
class VinylController extends AbstractController
{
... lines 12 - 29
    #[Route('/browse/{slug}')]
    public function browse(string $slug = null): Response
    {
        $genre = $slug ? u(str_replace('-', ' ', $slug))->title(true) : null;

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

Back in the template, use this in the `h1`. In Twig, we can *also* use fancy syntax. So *if* we have a `genre`, print `genre`, else print `All Genres`.

48 lines | templates/vinyl/browse.html.twig

```
{% extends 'base.html.twig' %}

{% block body %}
<div class="container">
    <h1>Browse {{ genre ? genre : 'All Genres' }}</h1>
... lines 6 - 45
</div>
{% endblock %}
```

Testing time. Head over to `/browse`: "Browse all genres"! And then `/browse/death-metal`: Browse Death Metal. Friends, this is starting to feel like a real site!

Except that these links up in the nav... go nowhere! Let's fix that next by learning how to generate URLs. We're also going to meet the mega-powerful `bin/console` command line tool.

Chapter 11: Generate Urls & bin/console

There are two different ways that we can interact with our app. The first is via the web server... and that's what we've been doing! We got to a URL and... behind the scenes, it executes `public/index.php`, which boots up Symfony, calls the routing and runs our controller.

Hello bin/console

What's the *second* way we can interact with our app? We haven't seen it yet: it's via a command line tool called `bin/console`. At your terminal run:

```
php bin/console
```

... to see a *bunch* of commands *within* this script. I *love* this thing. It's full of stuff to help us debug, eventually it will have code-generation commands, commands for setting secrets: all kinds of good stuff that we're going to discover little-by-little.

But I *do* want to point out that... there's nothing special about this `bin/console` script! It's just a file: there's literally a `bin/` directory with a `console` file inside. You'll probably never need to open this file or think about it, but it *is* useful. Oh, and on most systems, you can just run:

```
./bin/console
```

... which looks cooler. Or sometimes you may see me run:

```
symfony console
```

... which is just *another* way to execute this file. We'll talk more about this in a future tutorial.

bin/console debug:router

The *first* command I want to check out inside of `bin/console` is `debug:router`:

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

This is awesome. It shows us *every* route in our app, like *our* two routes for `/` and `/browse/{slug}`. What are these other routes? They come form the web debug toolbar and profiler system... and they're only here while we're developing locally.

Ok, back on our site.... at the top of the page, we have two non-functional links to the homepage and browse page. Let's hook these up. Open `templates/base.html.twig` ... and search for `a` tags. Here we go.

So it would be *really* easy to get this working by just `href="/"`. But instead, whenever we link to a page in Symfony, we're going to ask the routing system to *generate* a URL for us. We'll say:

> Please generate the URL to the homepage's route, or the browse page's route.

Then, if we ever change the URL of a route, all our links will instantly update. Magic.

Naming your Route

Let's start with the homepage. How do we ask Symfony to generate a URL to this route? Well first, we need to give the route a *name*. Surprise! *Every* route has an internal name. You can see it back in `debug:router`. Our route's are named `app_vinyl_homepage` and `app_vinyl_browse`. Huh, those are the *exact* names of my pet turtles when I was kid.

Where did these names come from? By default, Symfony automatically generates a name *for* us, which is fine. The name isn't used at *all* until we generate a URL to it. And as soon as we *do* need to generate a URL to a route, I highly recommend taking control of this name... just to make sure it never accidentally changes.

To do this, find the route and add an argument: `name` set to, how about, `app_homepage`. I like using the `app_` prefix: it makes it easier to search for a route name later.

40 lines | src/Controller/VinylController.php

```
<?php

... lines 3 - 9
class VinylController extends AbstractController
{
    #[Route('/', name: 'app_homepage')]
    public function homepage(): Response
    {
... lines 15 - 27
    }
... lines 29 - 38
}
```

By the way, PHP 8 attributes - like this `Route` attribute - are represented by actual, physical PHP classes. If you hold command or ctrl, you can open it and look inside. This is great: the `__construct()` method shows all of the different *options* you can pass to the attribute.

For example, there's a `name` argument... and then we're using PHP's named argument syntax to pass this into the attribute. Opening up an attribute is a *great* way to learn about its options.

Generating a URL from Twig

Anyways, now that we've given this a name, go back to our terminal and run `debug:router` again:

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

This time... yea! The route is named `app_homepage`! Copy that, then head back to `base.html.twig`. To generate a URL inside of twig, say `{{` - because we're going to print something - and then use a Twig function called `path()`. Pass this the route name.

87 lines | templates/base.html.twig

```
<!DOCTYPE html>
<html>
... lines 3 - 26
    <body>
... lines 28 - 31
            <a class="navbar-brand" href="{{ path('app_homepage') }}">
                <i class="fas fa-record-vinyl"></i>
                Mixed Vinyl
            </a>
... lines 36 - 84
    </body>
</html>
```

Done! Refresh... and the link up here works!

One more link to go. We know step one: give the route a name. So `name:` and, how about, `app_browse`.

40 lines | src/Controller/VinylController.php

```
<?php

... lines 3 - 9
class VinylController extends AbstractController
{
... lines 12 - 29
    #[Route('/browse/{slug}', name: 'app_browse')]
    public function browse(string $slug = null): Response
    {
... lines 33 - 37
    }
}
```

Copy that, and... scroll down a bit. Here it is: "Browse Mixes". Change that to `{{ path('app_browse') }}`.

87 lines | templates/base.html.twig

```
<!DOCTYPE html>
<html>
... lines 3 - 26
    <body>
... lines 28 - 40
                    <li class="nav-item">
                        <a class="nav-link" style="margin-top: 20px;" href="{{ path('app_browse') }}">Browse Mixes</a>
                    </li>
... lines 44 - 84
    </body>
</html>
```

And now... that link works too!

Generating URLs with Wildcards

Oh, but on this page, we have some quick links to go to the browse page for a specific genre. And these do *not* work yet.

This is interesting. We want to generate a URL like before... but this time we need to pass something to the `{slug}` wildcard. Open `browse.html.twig`. Here's how we do that. The first part is the same: `{{ path() }}` and then the name of the route: `app_browse`.

If we stopped here, it would generate `/browse`. To pass values to any wildcards in a route, `path()` has a second argument: an associative array of those value. And, again, just like JavaScript, to create an "associative array", you use `{` and `}`. I'm going to hit enter to break this onto multiple lines... just to keep things readable. Inside add a `slug` key to the array... and since this is the "Pop" genre, set it to `pop`.

Cool! Let's repeat this two more times: `{{ path('app_browse') }}`, pass curly braces for an associative array, with `slug` set to `rock`. And then one more time down here... which I'll do really quickly.

54 lines | templates/vinyl/browse.html.twig

```
... lines 1 - 2
{% block body %}
... lines 4 - 7
    <ul class="genre-list ps-0 mt-2 mb-3">
        <li class="d-inline">
            <a class="btn btn-primary btn-sm" href="{{ path('app_browse', {
                slug: 'pop'
            }) }}">Pop</a>
        </li>
        <li class="d-inline">
            <a class="btn btn-primary btn-sm" href="{{ path('app_browse', {
                slug: 'rock'
            }) }}">Rock</a>
        </li>
        <li class="d-inline">
            <a class="btn btn-primary btn-sm" href="{{ path('app_browse', {
                slug: 'all-accordion'
            }) }}">All Accordion</a>
        </li>
    </ul>
... lines 25 - 52
{% endblock %}
```

Let's see if it works! Refresh. Ah! Variable `rock` doesn't exist. I bet some of you saw me do that. I forgot my quotes, so this looks like a *variable*.

Try it again. There we go. And try the links... yes! They work!

Next: we've created two HTML pages. Now let's see what it looks like to create a JSON API endpoint.

Chapter 12: JSON API Endpoint

In a future tutorial, we're going to create a database to manage songs, genres, and the mixed vinyl records that our users are creating. Right now, we're working entirely with hardcoded data... but our controllers - and - especially templates won't feel *that* much different once we make this all dynamic.

So here's our new goal: I want to create an API endpoint that will return the data for a single song as JSON. We're going to use this in a few minutes to bring this play button to life. At the moment, none of these buttons do anything, but they do look pretty.

Creating the JSON Controller

The two steps to creating an API endpoint are... exactly the same as creating an HTML page: we need a route and a controller. Since this API endpoint will be returning *song* data, instead of adding *another* method inside of `VinylController`, let's create a totally new controller class. How you organize this stuff is entirely up to you.

Create a new PHP class called `SongController`... or `SongApiController` would also be a good name. Inside, this will start like any other controller, by extending `AbstractController`. Remember: that's optional... but it gives us shortcut methods with no downside.

Next, create a `public function` called, how about, `getSong()`. Add the route... and hit tab to auto-complete this so that PhpStorm adds the use statement on top. Set the URL to `/api/songs/{id}`, where `id` will eventually be the database id of the song.

And because we have a wildcard in the route, we are *allowed* to have an `$id` argument. *Finally*, even though we don't *need* to do this, because we know that our controller will return a `Response` object, we can set that as the return type. Make sure to auto-complete the one from Symfony's `HttpFoundation` component.

Inside the method, to start, `dd($id)` ... *just* to see if everything is working.

17 lines | src/Controller/SongController.php

```php
<?php

namespace App\Controller;

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

class SongController extends AbstractController
{
    #[Route('/api/songs/{id}')]
    public function getSong($id): Response
    {
        dd($id);
    }
}
```

Let's do this! Head over to `/api/songs/5` and... got it! *Another* new page.

Back in that controller, I'm going to paste in some song data: eventually, this will come from the database. You can copy this from the code block on this page. Our job is to return this as JSON.

So how *do* we return JSON in Symfony? By returning a new `JsonResponse` and passing it the data.

25 lines | src/Controller/SongController.php

```
<?php

... lines 3 - 9
class SongController extends AbstractController
{
    #[Route('/api/songs/{id}')]
    public function getSong($id): Response
    {
        // TODO query the database
        $song = [
            'id' => $id,
            'name' => 'Waterfalls',
            'url' => 'https://symfonycasts.s3.amazonaws.com/sample.mp3',
        ];

        return new JsonResponse($song);
    }
}
```

I know... too easy! Refresh and... hello JSON! Now you *might* be thinking:

> Ryan! You've been telling us - repeatedly - that a controller must all always return a Symfony `Response` object, which is what `render()` returns. Now you're returning some *other* type of `Response` object?

Ok, *fair*... but this works because `JsonResponse` *is* a Response. Let me explain. Sometimes it's useful to jump into core classes to see how they work. To do that, in PHPStorm - if you're on a Mac hold command, otherwise hold control - and then click the class name that you want to jump into. And... surprise! `JsonResponse` *extends* `Response`. Yea, we're still returning a `Response`. But this sub-class is nice because it automatically JSON encodes our data *and* sets the `Content-Type` header to `application/json`.

The ->json() Shortcut Method

Oh, and back in our controller, we can be even *lazier* by saying `return $this->json($song)` ... where `json()` is *another* shortcut method that comes from `AbstractController`.

25 lines | src/Controller/SongController.php

```
<?php

... lines 3 - 9
class SongController extends AbstractController
{
    #[Route('/api/songs/{id}')]
    public function getSong($id): Response
    {
... lines 15 - 20

        return $this->json($song);
    }
}
```

Doing this makes absolutely *no* difference because this is just a shortcut to return ... a `JsonResponse`!

If you're building a serious API, Symfony has a `serializer` component that's really good at turning objects into JSON... and then JSON back into objects. We talk a lot about it in our API Platform tutorial, which is a powerful library for creating APIs in Symfony.

Next, let's learn how to make our routes smarter, like by making a wildcard only match a *number*, instead of matching anything.

Chapter 13: Smart Routes: GET-only & Validate {Wildcards}

Now that we have a new page, at your terminal, run `debug:router` again.

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

Yep, there's our new endpoint! Notice that the table has a column called "Method" that says "any". This means that you can make a request to this URL using *any* HTTP method - like GET or POST - and it will match that route.

Restricting Routes to GET or POST Only

But the purpose of our new API endpoint is to allow users to make a GET request to fetch song data. Technically, right now, you could also make a POST for request to this... and it would work just fine. We might not care, but often with APIs, you'll want to restrict an endpoint to only work with one specific method like GET, POST or PUT. Can we make this route somehow only match GET requests?

Yep! By adding another option to the `Route`. In this case, it's called `methods`, it even auto-completes! Set this to an array and, put `GET`.

25 lines | src/Controller/SongController.php

```
<?php

... lines 3 - 9
class SongController extends AbstractController
{
    #[Route('/api/songs/{id}', methods: ['GET'])]
    public function getSong($id): Response
    {
... lines 15 - 22
    }
}
```

I'm going to hold Command and click into the `Route` class again... so we can see that... yup! `methods` is one of the arguments.

Back over on `debug:router`:

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

Nice. The route will now only match GET requests. It's... kind of hard to test this, since a browser always makes GET requests if you go directly to a URL... but this is where *another* `bin/console` command comes in handy: `router:match`.

If we run this with no arguments:

```
php bin/console router:match
```

It gives us an error but shows how it's used! Try:

```
php bin/console router:match /api/songs/11
```

And... that matches our new route! But *now* ask what would happen if we made a POST request to that URL with `--method=POST`:

```
php bin/console router:match /api/songs/11 --method=POST
```

No routes match this path with that method! But it *does* say that it *almost* matched our route.

Restricting Route Wildcards by Regex

Let's do *one* more thing to tighten up our new endpoint. I'm going to add an `int` type-hint to the `$id` argument.

src/Controller/SongController.php (25 lines)

```php
<?php

... lines 3 - 9
class SongController extends AbstractController
{
    #[Route('/api/songs/{id}', methods: ['GET'])]
    public function getSong(int $id): Response
    {
... lines 15 - 22
    }
}
```

That... doesn't change anything, except that PHP will now take the string `id` from the URL that Symfony passes into this method and cast it into an `int`, which is... just nice because then we're dealing with a true integer in our code.

You can see the subtle difference in the response. Right now, the `id` field is a string. When we refresh, `id` is now a true number in JSON.

But... if somebody was being tricky... and went to `/api/songs/apple` ... yikes! A PHP error, which, on production, would be a 500 error page! I do *not* like that.

But... what can we do? The error comes comes from when Symfony tries to call our controller and passes in that argument. So it's not like we can put code down *in* the controller to check if `$id` is a number: it's too late!

So what if, instead, we could tell Symfony that this route should only *match* if the `id` wildcard is a *number*. Is that possible? Totally!

By default, when you have a wildcard, it matches anything. But you *can* change it match a custom *regular expression*. Inside of the curly braces, right after the name, add a `<` then `>` and, in between, `\d+`. That's a regular expression meaning "a digit of anything length".

25 lines | src/Controller/SongController.php

```php
<?php

... lines 3 - 9
class SongController extends AbstractController
{
    #[Route('/api/songs/{id<\d+>}', methods: ['GET'])]
    public function getSong(int $id): Response
    {
... lines 15 - 22
    }
}
```

Try it! Refresh and... yes! A 404. No route found: it simply didn't match this route. A 404 is great... but a 500 error... that's something we want to avoid. And if we head back to `/api/songs/5` ... that still works.

Next: if you asked me what *the* most central and important part of Symfony is, I wouldn't hesitate: it's services. Let's find out what a service is and how it's the key to unlocking Symfony's potential.

Chapter 14: Service Objects

I see Symfony as two big parts. The first half is the route, controller, response system. It's dead simple and well... you're already an expert on it! The second half of Symfony is *all* about the many useful objects that are floating around... just waiting for us to use them!

Hello Service Objects

For example, when we render a template, what we are *actually* doing is taking advantage of a Twig object and asking *it* to render a template. There's also a logger object, a cache object, a database connection object, an object that helps make API requests, and many, many more! And when you install a new bundle, that give you even *more* useful objects.

The truth is that *everything* that *Symfony* does is... actually done by one of these useful objects. Heck there's even a router object that's responsible for finding the matching route for the given page!

In the Symfony world, and really the object oriented programming world in general, these "objects that do work" have a special name: *services*. But don't let that word confuse you. When you hear service, just think: that's an object that does work! Like a templating object that renders a template or a database connection object that makes queries.

And since service objects do work, they're basically... tools that help you get your job done! The second half of Symfony is *all* about discovering which services are available and how to use them.

The debug:autowiring Command

Let's try something. In our controller, I want to log a message... maybe some debugging message. Since logging a message is work, it's done by a service. Does our app already have a logger service? And if so, how do we get it?

To find out, move over to your terminal and run another `bin/console` command:

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

Say hello to one of the most *powerful* `bin/console` commands. I *love* this thing! This lists *all* of the services that exist in our app. Okay, it's actually not the *full* list, but this shows the services that you're most likely to need. And even though our app is small, there's a lot of stuff here! There's a filesystem service... and down here a cache service. There's even a twig service!

Is there a service for logging? You can look in this list... or you can re-run this command and search for the word log:

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

Excellent! For now, ignore everything except for the first line. This line tells us that there *is* a logger service and that this object implements an interface called `Psr\Log\LoggerInterface`.

Fetching a Service via Autowiring

Ok, so why does knowing that help us? Because if you want a service, you *ask* for it by using the type-hint shown in this command. It's called autowiring.

Let's try it. Head over to our controller and add a second argument. Actually, the order of these arguments doesn't matter. What matters is that the new argument is *type-hinted* with `LoggerInterface`. I'll hit tab to autocomplete that... so that PhpStorm adds the use statement on top.

In this case, the argument can be *called* anything, like `$logger`. When Symfony sees this type-hint, it looks inside of the `debug:autowiring` list... and because there's a match, it will pass us the logger service.

So we now know *two* different types of arguments that we are allowed to have in controller: you can have an argument whose *name* matches a wildcard in the route *or* an argument whose type-hint matches one of the services in our app.

Using the Logger

Ok, so now that we know Symfony will pass us the logger service object, let's use it! I don't know, yet, what *methods* I can call on it but... if we say `$logger->` ... PhpStorm... tells us! That was easy!

I'm going to log something at an `info()` priority level. Let's say:

> Returning API response for song

And then the `$id`.

28 lines | src/Controller/SongController.php

```
<?php

... lines 3 - 4
use Psr\Log\LoggerInterface;
... lines 6 - 10
class SongController extends AbstractController
{
    #[Route('/api/songs/{id<\d+>}', methods: ['GET'])]
    public function getSong(int $id, LoggerInterface $logger): Response
    {
... lines 16 - 22
        $logger->info('Returning API response for song '.$id);
... lines 24 - 25
    }
}
```

Actually, we can do something even cooler with this logger service. Add `{song}` to the message... and add a second argument, which is an array of extra information you want to attach to the log message. Pass `song` set to `$id`. In a minute, you'll see that the logger will print the *actual* id in place of `{song}`.

30 lines | src/Controller/SongController.php

```
<?php

... lines 3 - 10
class SongController extends AbstractController
{
    #[Route('/api/songs/{id<\d+>}', methods: ['GET'])]
    public function getSong(int $id, LoggerInterface $logger): Response
    {
... lines 16 - 22
        $logger->info('Returning API response for song {song}', [
            'song' => $id,
        ]);
... lines 26 - 27
    }
}
```

Anyways, this controller is for our API endpoint. So let's go over and refresh. Um... ok! So no error, that's good! But did it work? Where does the logger service... actually log to?

Let's find out next, learn a trick to see the profiler even for API requests and then leverage our *second* service directly.

Chapter 15: The Twig Service & Profiler for API Requests

Since this page just loaded without an error, we *think* that we just successfully logged a message via the logger service. But... where do log messages go? How can we check?

The logger service is provided by a library that we installed earlier called monolog. It was part of the debug-pack. And you can control its configuration inside the `config/packages/monolog.yaml` file, including *where* log messages are logged to, like which file. We'll focus more on config in the next tutorial.

The Profiler for API Requests

But one way that you can *always* see the log messages for a request is via the profiler! This is super handy. Go to the homepage, click any link on the web debug toolbar... and then go to the Logs section. We're now seeing *all* the log messages that were made only during that *last* request to the homepage.

Great! Except that... our log message is made on an API endpoint... and API endpoints don't have a web debug toolbar we can click! Are we stuck? Nope! Refresh this page one more time... then manually go to `/_profiler`. This is... kind of a secret door into the profiler system... and this page shows the last *ten* requests made into our system. The second to the top is the API request we just made. Click the little token link to see... yea! We're looking at the profiler for that API request! Over in the Logs section... there it is!

> Returning API response for song 5

... and you can even see the extra info we passed.

Rendering a Twig Template Manually

Ok, services are *so* important that... I want to do one more quick example. Go back to `VinylController`. The `render()` method is *really* just a shortcut to fetch the "twig" service, call some method on that object to render the template... and then put the final HTML string into a `Response` object. It's a *great* shortcut and you *should* use it.

But! As a challenge, could we render a template *without* using that method? Of course! Let's do it.

Step one: find the service that does the work you need to do. So, we need to find the Twig service. Let's do our trick again:

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

And... yes! Apparently the type-hint we need to use is `Twig\Environment`.

Ok! Go back to our method, add an argument, type `Environment`, and hit tab to auto-complete that so PhpStorm adds the `use` statement. Let's call it `$twig`.

Below, instead of using `render`, let's say `$html =` and then `$twig->`. Like with the logger, we don't need to know what methods this class has, because, thanks to the type-hint, PhpStorm can *tell* us all the methods. That `render()` method looks like it's probably what we want. The first argument is the string name of the template to render and the `$context` argument holds the variables. So... it has the same arguments that we were already passing.

To see if this is working, `dd($html)`.

42 lines | src/Controller/VinylController.php

```
<?php

... lines 3 - 10
class VinylController extends AbstractController
{
    #[Route('/', name: 'app_homepage')]
    public function homepage(Environment $twig): Response
    {
... lines 16 - 24
        $html = $twig->render('vinyl/homepage.html.twig', [
            'title' => 'PB & Jams',
            'tracks' => $tracks,
        ]);
        dd($html);
    }
... lines 31 - 40
}
```

Testing time! Head to the homepage... and yes! We just rendered a template manually! Seriously awesome! And we can finish this page by wrapping that in a response: `return new Response($html)`.

43 lines | src/Controller/VinylController.php

```
<?php

... lines 3 - 10
class VinylController extends AbstractController
{
    #[Route('/', name: 'app_homepage')]
    public function homepage(Environment $twig): Response
    {
... lines 16 - 24
        $html = $twig->render('vinyl/homepage.html.twig', [
            'title' => 'PB & Jams',
            'tracks' => $tracks,
        ]);

        return new Response($html);
    }
... lines 32 - 41
}
```

And now... the page works! And we understand that the *true* way to render a template is via the Twig service. Someday, you'll find yourself in a situation where you need to render a template but you are *not* in a controller... and so you do *not* have the `$this->render()` shortcut method. Knowing that there's a Twig service you can fetch will be the *key* to solving that problem. More on that in the next tutorial.

But in a real app, in a controller, there's no reason to do all this extra work. So I'm going to revert this... and go back to using `render()`. And... then we don't need to autowire that argument anymore... and we can even clean up the `use` statement.

Here are the three big, gigantic, important takeaways. First, Symfony is *packed* full of objects that do work... which we call services. Services are tools. Second, *all* work in Symfony is done by a service... even things like routing. And third, *we* can use services to help us get *our* work done by autowiring them.

In the next tutorial in this series, we'll dive deeper into this very important concept.

But before we finish *this* tutorial, I *really* really want to talk about one more big awesome, amazing thing: Webpack Encore, the *key* to writing professional CSS and JavaScript. Over these last few chapters, we're going to bring our site to life and even make it as responsive as a single page application.

Chapter 16: Setting up Webpack Encore

Our CSS setup is fine. We put files into the `public/` directory and then... we point to them from inside our templates. We could add JavaScript files the same way.

But if we want get truly serious about writing CSS and JavaScript, we need to take this to the next level. And even if you consider yourself a mostly backend developer, the tools we're about to talk about will allow you to write CSS and JavaScript that feels easier and is less error-prone than what you're probably used to.

The key to taking our setup to the next level is leveraging a node library called Webpack. Webpack is the industry standard tool for packaging, minifying and parsing your frontend CSS, JavaScript, and other files. But don't worry: Node is just JavaScript. And its role in our app will be pretty limited.

Setting up Webpack *can* be tricky. And so, in the Symfony world, we use a lightweight tool called Webpack Encore. It's still Webpack... it just makes it easier! And we have a free tutorial about it if you want to dive deeper.

Installing Encore

But let's do a crash course right now. First, at your command line, make sure you have Node installed:

```
node -v
```

You'll also need either `npm` - which comes with Node automatically - or `yarn`:

```
yarn --version
```

Npm and yarn are Node package managers: they're the Composer for the Node world... and you can use either. If you decide to use yarn - thats what I'll use - make sure to install version 1.

We're about to install a new package... so let's commit everything:

```
git add .
```

And... looks good:

```
git status
```

So commit everything:

```
git commit -m "Look mom! A real app"
```

To install Encore, run:

```
composer require encore
```

This installs WebpackEncoreBundle. Remember, a bundle is a Symfony plugin. And this package has a recipe: a very important recipe. Run:

```
git status
```

The Encore Recipe

Ooh! For the first time, the recipe modified the `.gitignore` file. Let's go check that out. Open `.gitignore`. The stuff on top is what we originally had... and down here is the new stuff added by WebpackEncoreBundle. It's ignoring the `node_modules/` directory, which is basically the `vendor/` directory for Node. We don't need to commit that because those vendor libraries are described in another new file from the recipe: `package.json`. This is Node's `composer.json` file: it describes the Node packages that our app needs. The most important one is Webpack Encore itself, which *is* a Node library. It also has a few other package that will help us get our job done.

The recipe also added an `assets/` directory... and a configuration file to control Webpack: `webpack.config.js`. The `assets/` directory already holds a small set of files to get us started.

Installing Node Dependencies

Ok, with Composer, if we didn't have this `vendor/` directory, we could run `composer install` which would tell it to read the `composer.json` file and re-download all the packages into `vendor/`. The same thing happens with Node: we have a `package.json` file. To *download* this stuff, run:

```
yarn install
```

Or:

```
npm install
```

Go node go! This will take a few moments as it downloads everything. You'll probably get a few warnings like this, which are safe to ignore.

Great! This did two things. First, it downloaded a *bunch* of files into the `node_modules/` directory: the "vendor" directory for Node. It also created a `yarn.lock` file... or `package-lock.json` if you're using npm. This serves the same purpose of `composer.lock`: it stores the *exact* versions of all the packages so that you get the *same* versions next time you install your dependencies.

For the most part, you don't need to worry about these lock files... except that you *should* commit them. Let's do that. Run:

```
git status
```

Then:

```
git add .
```

Beautiful:

```
git status
```

And commit:

```
git commit -m "Adding Webpack Encore"
```

Hey! Webpack Encore is now installed! But... it's not doing anything yet! Freeloader. Next, let's use it to take our JavaScript up to the next level.

Chapter 17: Packaging JS and CSS with Encore

When we installed Webpack Encore, its recipe gave us this new `assets/` directory. Check out the `app.js` file. Interesting. Notice how it *imports* this `bootstrap` file. That's actually `bootstrap.js`: this file right here. The `.js` extension is optional.

JavaScript Imports

This is one of the most *important* things that Webpack gives us: the ability to *import* one JavaScript file from another. We can import functions, objects... really *anything* from another file. We're going to talk more about this `bootstrap.js` file in a little bit.

This also imports a CSS file!? If you haven't seen this before, it might look... weird: JavaScript importing CSS?

To see how this all works, in `app.js`, add a `console.log()`.

15 lines | assets/app.js

```
... lines 1 - 12

console.log('Hi! My name is app.js!');
```

And `app.css` already has a body background... but add an `!important` so that we can *definitely* see if this is being loaded.

4 lines | assets/styles/app.css

```
body {
    background-color: lightgray !important;
}
```

Ok... so who *reads* these files? Because... they don't live in the `public/` directory... so we can't create `script` or `link` tags that point directly to them.

webpack.config.js

To answer that, open `webpack.config.js`. Webpack Encore is an executable binary: we're going to run it in a minute. When we *do*, it will load this file to get its config.

And while there are a *lot* of features inside of Webpack, the only thing we need to focus on right now is this one: `addEntry()`. This `app` could be anything... like `dinosaur`, it doesn't matter. I'll show you how that's used in a minute. The important thing is that it points to the `assets/app.js` file. Because of this, `app.js` will be the first and *only* file that Webpack will parse.

It's pretty cool: Webpack will reads the `app.js` file and then follow *all* of the `import` statements *recursively* until it finally has a giant collection of *all* the JavaScript and CSS our app needs. Then, it will *write* that into the `public/` directory.

Running Webpack Encore

Let's see it in action. Find your terminal and run:

```
yarn watch
```

This is, as it says, a shortcut for running `encore dev --watch`. If you look at your `package.json` file, it came with a `script` section with some shortcuts.

Anyways, `yarn watch` does two things. First, it created a new `public/build/` directory and, inside, `app.css` and `app.js` files! But don't let the names fool you: `app.js` contains a lot more that *just* what's inside `assets/app.js`: it contains *all* the JavaScript from *all* the imports it finds. `app.css` contains all the *CSS* from all the imports.

The reason these files are called `app.css` and `app.js` is because of the entry name.

So the takeaway is that, thanks to Encore, we suddenly have new files in a `public/build/` directory that contain *all* the JavaScript and CSS our app needs!

The Encore Twig Functions

And if you move over to your homepage and refresh... woh! It instantly worked!? The background changed... and in my inspector... there's the console log! How the heck did that happen?

Open your base layout: `templates/base.html.twig`. The secret is in the `encore_entry_link_tags()` and `encore_entry_script_tags()` functions. I bet you can guess what these do: add the `link` tag to `build/app.css` and the `script` tag to `build/app.js`.

You can see this in your browser. View the source for the page and... yup! The link tag for `/build/app.css` ... and `script` tag for `/build/app.js`. Oh, but it also rendered two *other* `script` tags. That's because Webpack is really smart. For performance purposes, instead of dumping one *gigantic* `app.js` file, sometimes Webpack will *split* it into multiple, smaller files. Fortunately, these Encore Twig functions are smart enough to handle that: it will include *all* the link or script tags needed.

The *most* important thing is that the code that we have in our `assets/app.js` file - including anything it imports - is now functioning and showing up on our page!

Watching for Changes

Oh, and because we ran `yarn watch`, Encore is still running in the background watching for changes. Check it out: go into `app.css` ... and change the background color. Save, move over and refresh.

4 lines | assets/styles/app.css

```
1 body {
2     background-color: maroon !important;
3 }
```

It instantly updated! That's because Encore *noticed* the change and recompiled the built file really quickly.

Next: let's move our *existing* CSS into the new system and learn how we can install and import *third party* libraries - look Bootstrap or FontAwesome - right into our Encore setup.

Chapter 18: Installing 3rd Party Code into our JS/CSS

We now have a nice new JavaScript and CSS system that lives entirely inside of the `assets/` directory. Let's move our public styles into this. Open `public/styles/app.css`, copy all of this, delete the *entire* directory... and then paste into the new `app.css`. Thanks to the `encore_entry_link_tags()` in `base.html.twig`, the new CSS *is* being included... and we don't need the old `link` tag anymore.

Go check it out. Refresh and... it still looks great!

Installing 3rd Party JavaScript/CSS Libraries

Go back to `base.html.twig`. What about these external link tags for bootstrap and FontAwesome? Well, you can *totally* keeps these CDN links. But we can *also* process this stuff through Encore. How? By installing Bootstrap and FontAwesome as *vendor* libraries and *importing* them.

Remove *all* of these link tags... and then refresh. Yikes! It's back to looking like *I* designed this site. Let's... *first* re-add bootstrap. Find your terminal. Since the watch command is running, open a new terminal tab and then run:

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

This does three things. First, it adds `bootstrap` to our `package.json` file. Second it downloads bootstrap into our `node_modules/` directory... you would find it down here. And *third*, it updated the `yarn.lock` file with the exact version of bootstrap that it just downloaded.

If we stopped now... this wouldn't make any difference! We downloaded bootstrap - yay - but we're not using it.

To use it, we need to *import* it. Go into `app.css`. Just like in JavaScript files, we can import from inside *CSS* files by saying `@import` and then the file. We could reference a file in the same directory with `./other-file.css`. *Or*, if you want to import something from the `node_modules/` directory in CSS, there's a trick: a `~` and then the package name: `bootstrap`.

34 lines | assets/styles/app.css

```
1  @import '~bootstrap';
   ... lines 2 - 34
```

That's it! As *soon* as we did that, Encore's watch function rebuilt our `app.css` file... which now *includes* Bootstrap! Watch: refresh the page and... we're back! So cool!

The two *other* things we're missing are `FontAwesome` and a specific Font. To add those, head back to the terminal and run:

```
yarn add @fontsource/roboto-condensed --dev
```

Full disclosure: I did some searching *before* recording so that I knew the names of all the packages we need. You can search for packages at https://npmjs.com.

Let's also add the last one we need:

```
yarn add @fortawesome/fontawesome-free --dev
```

Again, this downloaded the two libraries into our project... but doesn't automatically *use* them yet. Because those libraries both hold CSS files, go back to our `app.css` file and import them: `@import '~'` then `@fortawesome/fontawesome-free`. And `@import '~@fontsource/roboto-condensed'`.

34 lines | assets/styles/app.css

```
@import '~bootstrap';
@import '~@fortawesome/fontawesome-free';
@import '~@fontsource/roboto-condensed';

... lines 5 - 34
```

The first package should fix this icon... and the second should cause the font to change on the whole page. Watch the font when we refresh... it *did* change! But, uh... the icons are still kind of broken.

Importing Specific Files from node_modules/

To be totally honest, I'm not sure why that doesn't work out-of-the box. But the fix is kind of interesting. Hold command on a Mac - or ctrl otherwise - and click this `fontawesome-free` string.

When you use this syntax, it goes into your `node_modules/` directory, into `@fortawesome/fontawesome-free` ... and then if you don't put any filename after this, there's a mechanism where this library tells Webpack *which* CSS file it should import. By default, it imports this `fontawesome.css` file. For some reason... that doesn't work. What we want is this `all.css`.

And we can import *that* by adding the path: `/css/all.css`. We don't need the minified file because Encore handles minifying for us.

34 lines | assets/styles/app.css

```
@import '~bootstrap';
@import '~@fortawesome/fontawesome-free/css/all.css';
@import '~@fontsource/roboto-condensed';

... lines 5 - 34
```

And now... we're back!

The *main* reason I love Webpack Encore and this system is that it allows us to use proper imports. We can even organize *our* JavaScript into small files - putting classes or functions into each - and then import them when we need them. There's no more need for global variables.

Webpack also allows us to use more serious stuff like React or Vue: you can even see, in `webpack.config.js`, the methods to activate those.

But usually, I like using a delightful JavaScript library called Stimulus. And I want to tell you about it next.

Chapter 19: Stimulus: Sensible, Beautiful JavaScript

I want to talk about Stimulus. Stimulus is a small, but delightful JavaScript library that I *love*. And Symfony has first-class support for it. It's also heavily used by the Ruby on Rails community.

SPA vs "Traditional" Apps

So there are kind of two philosophies in web development. The first is where you return HTML from your site like we've been doing on our homepage and browse page. And then you *add* JavaScript behavior to that HTML. The second philosophy is to use a front-end JavaScript framework to build *all* of your HTML and JavaScript. That's a single page application.

The right solution depends on your app, but I strongly like the first approach. And by using Stimulus - as well as another tool we'll talk about in a few minutes called Turbo - we can create highly-interactive apps that look and feel *as* responsive as a single page app.

We have an entire tutorial on Stimulus, but let's get a taste. You can already see how it works in the example on their docs. You create a small JavaScript class called a controller... and then *attach* that controller to one or more elements on the page. And that's it! Stimulus allows you to attach event listeners - like click events - and has other goodies.

Stimulus Controllers in our App

In our app, when we installed Encore, it gave us a `controllers/` directory. This is where our Stimulus controllers will live. And in `app.js`, we import `bootstrap.js`. This is not a file that you'll need to look at much, but it's *super* useful. This starts up Stimulus - yes, it's already installed - and registers everything in the `controllers/` directory as a Stimulus controller. This means that if you want to create a new Stimulus controller, all you need to do is add a file to this `controllers/` directory!

And we get one Stimulus controller out-of-the box called `hello_controller.js`. All Stimulus controllers follow the naming practice of something "underscore" `controller.js` or something *dash* `controller.js`. The part before `_controller` - so `hello` - becomes the controller's *name*.

Attaching a Controller to an Element

Let's attach this to an element. Open up `templates/vinyl/homepage.html.twig`. Let's see... on the main part of the page, I'm going to add a div... and then to attach the controller to this element, add `data-controller="hello"`.

59 lines | templates/vinyl/homepage.html.twig

```
... lines 1 - 35
36              <div data-controller="hello"></div>
... lines 37 - 59
```

Let's try it! Refresh and... yes! It worked! Stimulus saw this element, instantiated the controller... and then our code changed the content of the element. The element that this controller is attached to is available as `this.element`.

Stimulus Dynamically sees New Elements!

So... this is already *really* nice... because we get to work inside of a neat JavaScript object... which is tied to a specific element.

But let me show you the *coolest* part of Stimulus: what makes it such a game changer. Inspect element in your browser tools near the element. I'm going to modify the parent element's HTML. Right above this - though it doesn't matter where - add *another* element with `data-controller="hello"`.

And... boom! We see the message! *This* is the killer feature of Stimulus: you can add these `data-controller` elements onto the page *whenever* you want. For example, if you make an Ajax call... which adds fresh HTML to your page, Stimulus will *notice* that and execute any controllers that the new HTML should be attached to. If

you've ever had problems where you add HTML to your page via Ajax... but that new HTML's JavaScript is broken because it's missing some event listeners, well, Stimulus just solved that.

The stimulus_controller () Function

When you use Stimulus inside of Symfony, we get a few helper functions to make life easier. So instead of writing `data-controller="hello"` by hand, we can say `{{ stimulus_controller('hello') }}`.

59 lines | templates/vinyl/homepage.html.twig

```
... lines 1 - 35
36              <div {{ stimulus_controller('hello') }}></div>
... lines 37 - 59
```

But that's just a shortcut to render that attribute *exactly* like it was before.

Ok, now that we have the basics of Stimulus, let's use it to do something real, like make an Ajax request when we click this play icon. That's next.

Chapter 20: Real-World Stimulus Example

Let's put Stimulus to the test. Here's our goal: when we click the play icon, we're going to make an Ajax request to our API endpoint... the one in `SongController`. This returns the URL to where this song can be played. We'll then use that in JavaScript to... play the song!

Take `hello_controller.js` and rename it to, how about `song-controls_controller.js`. Inside, just to see if this is working, in `connect()`, log a message. The `connect()` method is called whenever Stimulus sees a new matching element on the page.

17 lines | assets/controllers/song-controls_controller.js

```
import { Controller } from '@hotwired/stimulus';

/*
 * This is an example Stimulus controller!
 *
 * Any element with a data-controller="hello" attribute will cause
 * this controller to be executed. The name "hello" comes from the filename:
 * hello_controller.js -> "hello"
 *
 * Delete this file or adapt it for your use!
 */
export default class extends Controller {
    connect() {
        console.log('I just appeared into existence!');
    }
}
```

Now, over in the template, hello isn't going to work anymore, so remove that. What I want to do is surround each song *row* with this controller.... so that's this `song-list` element. After the class, add `{{ stimulus_controller('song-controls') }}`.

58 lines | templates/vinyl/homepage.html.twig

```
{% extends 'base.html.twig' %}

{% block title %}Create a new Record | {{ parent() }}{% endblock %}

{% block body %}
<div class="container">
... lines 7 - 36
            {% for track in tracks %}
            <div class="song-list" {{ stimulus_controller('song-controls') }}>
... lines 39 - 50
            </div>
            {% endfor %}
... lines 53 - 55
</div>
{% endblock %}
```

Let's try that! Refresh, check the console and... yes! It hit our code six times! Once for *each* of these elements. And each element gets its own, separate controller *instance*.

Adding Stimulus Actions

Okay, next, when we click play, we want to run some code. To do that, we can add an *action*. It looks like this: on the `a` tag, add `{{ stimulus_action() }}` - another shortcut function - and pass this the controller name that you're attaching the action to - `song-controls` - and then a method inside of that controller that should be called when someone clicks this element. How about `play`.

58 lines | templates/vinyl/homepage.html.twig

```
{% extends 'base.html.twig' %}

{% block title %}Create a new Record | {{ parent() }}{% endblock %}

{% block body %}
<div class="container">
... lines 7 - 36
            {% for track in tracks %}
            <div class="song-list" {{ stimulus_controller('song-controls') }}>
                <div class="d-flex mb-3">
                    <a href="#" {{ stimulus_action('song-controls', 'play') }}>
                        <i class="fas fa-play me-3"></i>
                    </a>
... lines 43 - 49
                </div>
            </div>
            {% endfor %}
... lines 53 - 55
</div>
{% endblock %}
```

Cool huh? Back in song controller, we don't need the `connect()` method anymore: we don't need to *do* anything each time we notice another `song-list` row. But we *do* need a `play()` method.

And like with normal event listeners, this will receive an `event` object... and then we can say `event.preventDefault()` so that our browser doesn't try to follow the link click. To test, `console.log('Playing!')`.

19 lines | assets/controllers/song-controls_controller.js

```
import { Controller } from '@hotwired/stimulus';
... lines 2 - 11
export default class extends Controller {
    play(event) {
        event.preventDefault();

        console.log('Playing!');
    }
}
```

Let's go see what happens! Refresh and... click. It's working. It's that easy to hook up an event listener in Stimulus. Oh, and if you inspect this element... that `stimulus_action()` function is just a shortcut to add a special `data-action` attribute that Stimulus understands.

Installing and Importing Axios

Ok, how can we make an Ajax call from inside of the `play()` method? Well, we could use the built-in `fetch()` function from JavaScript. But instead, I'm going to install a third-party library called Axios. At your terminal, install it by saying:

```
yarn add axios --dev
```

We now know what this does: it downloads this package into our `node_modules` directory, and adds this line to our `package.json` file.

Oh, and side note: you absolutely *can* use jQuery inside of Stimulus. I won't do it, but it works great - and you can install - and import - jQuery like any other package. We talk about that in our Stimulus tutorial.

Ok, so how do we *use* the `axios` library? By importing it!

At the top of this file, we're already importing the `Controller` base class from `stimulus`. Now `import axios from 'axios'`. As soon as we do that, Webpack Encore will grab the `axios` source code and *include* it in our built JavaScript files.

21 lines | assets/controllers/song-controls_controller.js

```
... lines 1 - 11
import axios from 'axios';
... lines 13 - 21
```

Now, down here, we can say `axios.get()` to make a GET request. But... what should we pass for the URL? It needs to be something like `/api/songs/5` ... but how do we know what the "id" is for *this* row?

Stimulus Values

One of the coolest things about Stimulus is that it allows you to pass values from Twig *into* your Stimulus controller. To do that, declare which values you want to *allow* to passed in via a special static property: `static values = {}`. Inside, let's allow an `infoUrl` value to be passed. I totally just made up that name: I'm thinking we'll pass in the *full* URL to the API endpoint. Set this to the *type* that this will be. So, a `String`.

We'll learn *how* we pass this value from Twig into our controller in a minute. But because we have this, below, we can reference the value by saying `this.infoUrlValue`.

26 lines | assets/controllers/song-controls_controller.js

```
... lines 1 - 11
import axios from 'axios';
... line 13
export default class extends Controller {
    static values = {
        infoUrl: String
    }
... line 18
    play(event) {
... lines 20 - 21
        console.log(this.infoUrlValue);
        //axios.get()
    }
}
```

So how do we pass that in? Back in `homepage.html.twig`, add a second argument to `stimulus_controller()`. This is an array of the *values* you want to pass into the controller. Pass `infoUrl` set to the URL.

Hmm, but we need to generate that URL. Does that route have a name yet? Nope! Add `name: 'api_songs_get_one'`.

30 lines | src/Controller/SongController.php

```
<?php

... lines 3 - 10
class SongController extends AbstractController
{
    #[Route('/api/songs/{id<\d+>}', methods: ['GET'], name: 'api_songs_get_one')]
    public function getSong(int $id, LoggerInterface $logger): Response
    {
... lines 16 - 27
    }
}
```

Perfect. Copy that... and back in the template, set `infoURl` to `path()`, the name of the route... and then an array with any wildcards. Our route has an `id` wildcard.

In a real app, these tracks would probably each have a database id that we could pass. We don't have that yet... so to, kind of, fake this, I'm going to use `loop.index`. This is a magic Twig variable: if you're inside of a

Twig `for` loop, you can access the *index* - like 1, 2, 3, 4 - by using `loop.index`. So we're going to use this as a fake ID. Oh, and don't forget to say `id:` then `loop.index`.

60 lines | templates/vinyl/homepage.html.twig

```
... lines 1 - 4
{% block body %}
<div class="container">
... lines 7 - 36
            {% for track in tracks %}
            <div class="song-list" {{ stimulus_controller('song-controls', {
                infoUrl: path('api_songs_get_one', { id: loop.index })
            }) }}>
... lines 41 - 52
            </div>
            {% endfor %}
... lines 55 - 57
</div>
{% endblock %}
```

Testing time! Refresh. The first thing I want you to see is that, when we pass `infoUrl` as the second argument to `stimulus_controller`, all that really does is output a very special `data` attribute that Stimulus knows how to read. That's how you pass a value into a controller.

Click one of the play links and... got it. Every controller object is passed its correct URL!

Making the Ajax Call

Let's celebrate by making the Ajax call! Do it with `axios.get(this.infoUrlValue)` - yes, I just typo'ed that, `.then()` and a callback using an arrow function that will receive a `response` argument. This will be called when the Ajax call finishes. Log the response to start. Oh, and fix to use `this.infoUrlValue`.

28 lines | assets/controllers/song-controls_controller.js

```
import { Controller } from '@hotwired/stimulus';

... lines 3 - 11
import axios from 'axios';
... line 13
export default class extends Controller {
... lines 15 - 18
    play(event) {
        event.preventDefault();

        axios.get(this.infoUrlValue)
            .then((response) => {
                console.log(response);
            });
    }
}
```

Alrighty, refresh... then click a play link! Yes! It dumped the response... and one of its keys is `data` ... which contains the `url`!

Time for our victory lap! Back in the function, we can *play* that audio by creating a new `Audio` object - this is just a normal JavaScript object - passing it `response.data.url` ... and then calling `play()` on this.

29 lines | assets/controllers/song-controls_controller.js

```
import { Controller } from '@hotwired/stimulus';
... lines 2 - 11
import axios from 'axios';

export default class extends Controller {
... lines 15 - 18
    play(event) {
        event.preventDefault();

        axios.get(this.infoUrlValue)
            .then((response) => {
                const audio = new Audio(response.data.url);
                audio.play();
            });
    }
}
```

And now... when we hit play... finally! Music to my ears.

If you want to learn more about Stimulus - this *was* a bit fast - we have an entire tutorial about it... and it's *great*.

To finish off *this* tutorial, let's install one more JavaScript library. This one will instantly make our app feel like a single page app. That's next.

Chapter 21: Turbo: Supercharge your App

Welcome to the *final* chapter of our intro to Symfony 6 tutorial. If you're watching this, you're crushing it! And it's time to celebrate by installing one more package from Symfony. But before we do, as you know, I like to commit everything first... in case the new package installs an interesting recipe:

```
git add .
git commit -m "Never gonna let you go..."
```

Installing symfony/ux-turbo

Ok, let's install the new package:

```
composer require symfony/ux-turbo
```

See that "ux" in the package name? Symfony UX is a set of libraries that add *JavaScript* functionality to your app... often with some PHP code to help. For example, there's a library for rendering charts... and another for using an image Cropper with the form system.

Symfony UX Recipes

So, as you can see, this *did* install a recipe. OoOOo. Run

```
git status
```

so we can see what that did. Most of this is normal, like `config/bundles.php` means it enabled the new bundle. The two interesting changes are `assets/controllers.json` and `package.json`. Let's check out `package.json` first.

When you install a UX package, what that *usually* means is that you're integrating with a third-party JavaScript library. And so, that package's recipe *adds* that library to your code. In this case, the JavaScript library we're integrating with is called `@hotwired/turbo`. Also, the `symfony/ux-turbo` PHP package *itself* comes with some extra JavaScript. This special line says:

> Hey Node! I want to include a package called `@symfony/ux-turbo` ... but instead of downloading that, you can just find its code in the `vendor/symfony/ux-turbo/Resources/assets` directory.

You can literally look at that path: `vendor/symfony/ux-turbo/Resources/assets` to find a mini JavaScript package. Now, because this updated our `package.json` file, we need to re-install our dependencies to download and get this all set up.

In fact, find your terminal that's running `yarn watch`. We've got an error! It says the file `@symfony/ux-turbo/package.json` cannot be found, try running `yarn install --force`.

Let's do that! Hit control+C to stop this... and then run

```
yarn install --force
```

or `npm install --force`. Then, restart Encore with:

```
yarn watch
```

The *other* file the recipe modified was `assets/controllers.json`. Let's go take a look at that: `assets/controllers.json`. This is another thing that's unique to Symfony UX. Normally, if we want to add a stimulus controller, we put it into the `controllers/` directory. But sometimes, we might install a PHP package and *that* may want to add its *own* Stimulus controller into our app. This syntax basically says:

> Hey Stimulus! Go load this Stimulus controller from that new `@symfony/ux-turbo` package.

Now this *particular* Stimulus controller is a little weird. It's not one that we're going to use directly inside of the `stimulus_controller()` Twig function. This is, kind of a, fake controller. What does it do? *Just* by it being loaded, it's going to activate the Turbo library.

Hello Turbo! By Full-Page Refreshes

So I keep talking about Turbo. What *is* Turbo? Well, by running that composer require command... then reinstalling yarn, the Turbo JavaScript *is* now active and running on our site. What does it *do*? It's simple: it turns every link click and form submit on our site into an Ajax call. And *that* makes our site feel lightning fast.

Check it out. Do one last full refresh. And then watch... if I click Browse, there was no full page refresh! If I click these icons, no refresh! Turbo intercepts those clicks, makes an Ajax call to the URL, and then puts that HTML onto our site. This is *huge* because, for free, our app suddenly looks and feels like a single page app... without us doing anything!

The Web Debug Toolbar & Profiler for Ajax Requests

Now, one other cool thing you'll notice is that even though full page reloads are gone, these Ajax calls *do* show up on the web debug toolbar. And you can click to go see the profiler for that Ajax call really easily. This Ajax part of the web debug toolbar is even more useful with Ajax calls for an API endpoint. If we hit the play icon... that 7 just went up to 8... and here's the profiler for that API request! I'll open that link in a new window. That's a *super* easy way to get to the profiler for *any* Ajax request.

So Turbo is amazing... and it can do more. There *are* some things you need to know about it before shipping it to production, and if you're interested, yup! We have a full tutorial about Turbo. I wanted to mention it in *this* tutorial because Turbo is easiest if you add it to your app early on.

All right, congratulations! The first Symfony 6 tutorial is in the books! Pat yourself on the back... or better, find a friend and give them a crisp high five.

And keep going! Join us for the next tutorial in this series, which will take you from a budding Symfony developer to someone who *really* understands what's going on. How services work, the point of all of these configuration files, Symfony environments, environmental variables, and a lot more. Basically everything you'll need to do *whatever* you want with Symfony.

And if you have any questions or ideas, we are here for you down in the comments section below the video.

Alright friends, see you next time!