Stellar Development with Symfony 4

With <3 from KnpUniversity

Chapter 1: Symfony 4: Let's Launch!

Hey guys! Yes! It's Symfony 4 time! I am *so* excited. Why? Because *nothing* makes me happier than sitting down to work inside a framework where coding is actually *fun*, and where I can build features fast, but *without* sacrificing quality. Well, maybe I'd be *even* happier doing all of that on a beach... with, maybe a cold drink?

Anyways, Symfony 4 completely re-imagined the developer experience: you're going to create better features, faster than ever. *And*, Symfony has a *new*, unique super-power: it starts as a microframework, then automatically scales in size as your project grows. How? Stay tuned...

Oh, and did I mention that Symfony 4 is the fastest version ever? And the fastest PHP framework? Honestly, *all* frameworks are fast enough anyways, but the point is this: you're building on a seriously awesome foundation.

Tip

See http://www.phpbenchmarks.com for the third-party benchmark stats!

Prep: Download & Update Composer

Ok, let's get started already! Open a new terminal and move into whatever directory you want. Make sure that you already have Composer installed globally so that you can just say `composer`. If you have any questions, ask us in the comments!

And *also* make sure you have the latest version:

```
$ composer self-update
```

That's important: Composer had a recent bug fix to help Symfony.

Install Symfony!

To download your new Symfony project, run `composer create-project symfony/skeleton` and put this into a new directory called `the_spacebar`.

```
$ composer create-project symfony/skeleton the_spacebar
```

That's the name of our project! "The Spacebar" will be *the* place for people from *across* the galaxy to communicate, share news and argue about celebrities and BitCoin. It's going to be amazing!

This command clones the `symfony/skeleton` project and then runs `composer install` to download its dependencies.

Further down, there's something *special*: something about "recipes". OooOOO. Recipes are a new and very important concept. We'll talk about them in a few minutes.

Starting the Web Server

And at the bottom, cool! Symfony gives us clear instructions about what to do next. Move into the new directory:

```
$ cd the_spacebar
```

Apparently, we can run our app immediately by executing:

```
```

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

This starts the built-in PHP web server, which is *great* for development. `public/` is the document root of the project - but more on that soon!

Tip

If you want to use Nginx or Apache for local development, you can! See http://bit.ly/symfony-web-servers.

Time to blast off! Move to your browser and go to `http://localhost:8000`. Say hello to your new Symfony app!

Our Tiny Project

Back in the terminal, I'll create a new terminal tab. Symfony *already* inititalized a new git repository for us *and* gave us a perfect `.gitignore` file. Thanks Symfony! That means we can create our *first* commit just by saying:

```
$ git add .
$ git commit
```

Create a calm and well-thought-out commit message.

```
$ # Woohoo! OMG WE ARE USING SYMFONY4
```

Woh! Check this out: the entire project - *including* Composer and `.gitignore` stuff - is only 16 files! Our app is teenie-tiny!

Let's learn more about our project next *and* setup our editor to make Symfony development *amazing*!

Chapter 2: Our Micro-App & PhpStorm Setup

Our mission: to boldly go where no one has gone before... by checking out our app! I already opened the new directory in PhpStorm, so fire up your tricorder and let's explore!

The public/ Directory

There are only three directories you need to think about. First, `public/` is the document root: so it will hold *all* publicly accessible files. And... there's just one right now! `index.php`. This is the "front controller": a fancy word programmers invented that means that this is the file that's executed when you go to *any* URL.

But, *really*, you'll almost never need to worry about it. In fact, now that we've talked about this directory, *stop* thinking about it!

src/ and config/

Yea, I lied! There are *truly* only *two* directories you need to think about: `config/` and `src/`. `config/` holds... um... ya know... config files and `src/` is where you'll put *all* your PHP code. It's just that simple.

Where is Symfony? As usual, when we created the project, Composer read our `composer.json` file and downloaded all the third-party libraries - including parts of Symfony - into the `vendor/` directory.

Installing the Server

Go back to your terminal and find the original tab. Check this out: at the bottom, it says that we can get a *better* web server by running `composer require server`. I like better stuff! So let's try it! Press `Ctrl` + `C` to stop the existing server, and then run:

```
$ composer require server
```

If you're familiar with Composer... that package name should look funny! Really, wrong! *Normally*, every package name is "something" *slash* "something", like `symfony/console`. So... `server` just should *not* work! But it *does*! This is part of a cool new system called Flex. More about that soon!

When this finishes, you can now run:

```
$ ./bin/console server:run
```

This does *basically* the same thing as before... but the command is shorter. And when we refresh, it still works!

By the way, this `bin/console` command is going to be our new robot side-kick. But it's *not* magic: our project has a `bin/` directory with a `console` file inside. Windows users should say `php bin/console`... because it's just a PHP file.

So, what amazing things can this `bin/console` robot do? Find your open terminal tab and just run:

```
$ ./bin/console
```

Yes! This is a list of *all* of the `bin/console` commands. Some of these are debugging *gold*. We'll talk about them along the way!

PhpStorm Setup

Ok, we are *almost* ready to start coding! But we *need* talk about our spaceship, I mean, editor! Look, you can

use *whatever* your want... but... I *highly* recommend PhpStorm! Seriously, it makes developing in Symfony a *dream*! And no, those nice guys & gals at PhpStorm aren't paying me to say this... but they can if they want to!

Ahem, If you *do* use it... which would be *awesome* for you... there are 2 secrets you need to know to trick out your spaceship, ah, *editor*! Clearly I was in hyper-sleep too long.

Go to Preferences, Plugins, then click "Browse Repositories". There are 3 must-have plugins. Search for "Symfony". First: the "Symfony Plugin". It has over 2 million downloads for a reason: it will give you *tons* of ridiculous auto-completion. You should also download "PHP Annotations" and "PHP Toolbox". I already have them installed. If you *don't*, you'll see an "Install" button right at the top of the description. Install those and restart PHPStorm.

Then, come *back* to Preferences, search for "symfony" and find the new "Symfony" section. Click the "Enable Plugin" checkbox: you need to enable the Symfony plugin for *each* project. It says you need to restart... but I think that's lie. It's space! What could go wrong?

So that's PhpStorm trick #1. For the second, search "Composer" and click on the "Composer" section. Click to browse for the "Path to composer.json" and select the one in our project. I'm not sure why this isn't automatic... but whatever! Thanks to this, PhpStorm will make it easier to create classes in `src/`. You'll see this *really* soon.

Okay! Our project is set up and it's already working. Let's start building some pages and discovering more cool things about new app.

Chapter 3: Routes, Controllers, Pages, oh my!

Let's create our *first* page! Actually, this is the *main* job of a framework: to give you a *route* and *controller* system. A route is configuration that defines the URL for a page and a controller is a function that *we* write that *actually* builds the content for that page.

And right now... our app is *really* small! Instead of weighing down your project with *every* possible feature you could ever need - after all, we're *not* in zero-gravity yet - a Symfony app is basically just a small route-controller system. Later, we'll install more features when we need them, like a warp drive! Those always come in handy. Adding more features is *actually* going to be pretty awesome. More on that later.

First Route & Controller

Open your app's main routing file: `config/routes.yaml`:

4 lines | config/routes.yaml

```
#index:
#    path: /
#    controller: App\Controller\DefaultController::index
```

Hey! We already have an example! Uncomment that. Ignore the `index` key for now: that's the internal *name* of the route, but it's not important yet.

This says that when someone goes to the homepage - `/` - Symfony should execute an `index()` method in a `DefaultController` class. Change this to `ArticleController` and the method to `homepage`:

4 lines | config/routes.yaml

```
index:
    path: /
    controller: App\Controller\ArticleController::homepage
```

And... yea! That's a route! Hi route! It defines the URL and tells Symfony what controller function to execute.

The controller class doesn't exist yet, so let's create it! Right-click on the `Controller` directory and go to "New" or press `Cmd` + `N` on a Mac. Choose "PHP Class". And, yes! Remember that Composer setup we did in Preferences? Thanks to that, PhpStorm correctly guesses the namespace! The force is strong with this one... The namespace for every class in `src/` should be `App` plus whatever sub-directory it's in.

Name this `ArticleController`:

14 lines | src/Controller/ArticleController.php

```
<?php

namespace App\Controller;
... lines 4 - 6
class ArticleController
{
... lines 9 - 12
}
```

And inside, add `public function homepage()`:

14 lines | src/Controller/ArticleController.php

```
... lines 1 - 2
namespace App\Controller;
... lines 4 - 6
class ArticleController
{
    public function homepage()
    {
... line 11
    }
}
```

This function is the controller... and it's *our* place to build the page. To be more confusing, it's also called an "action", or "ghob" to its Klingon friends.

Anyways, we can do *whatever* we want here: make database queries, API calls, take soil samples looking for organic materials or render a template. There's just *one* rule: a controller must return a Symfony `Response` object.

So let's say: `return new Response()`: we want the one from `HttpFoundation`. Give it a calm message: `OMG! My first page already! WOOO!`:

14 lines | src/Controller/ArticleController.php

```
... lines 1 - 2
namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;

class ArticleController
{
    public function homepage()
    {
        return new Response('OMG! My first page already! WOOO!');
    }
}
```

Ahem. Oh, and check this out: when I let PhpStorm auto-complete the `Response` class it added this `use` statement to the top of the file automatically:

14 lines | src/Controller/ArticleController.php

```
... lines 1 - 4
use Symfony\Component\HttpFoundation\Response;
... lines 6 - 14
```

You'll see me do that a lot. Good job Storm!

Let's try the page! Find your browser. Oh, this "Welcome" page only shows if you don't have *any* routes configured. Refresh! Yes! This is *our* page. Our first of *many*.

Annotation Routes

That was *pretty* easy, but it can be easier! Instead of creating our routes in YAML, let's use a cool feature called *annotations*. This is an extra feature, so we need to install it. Find your open terminal and run:

```
$ composer require annotations
```

Interesting... this `annotations` package *actually* installed `sensio/framework-extra-bundle`. We're going to talk about how that works *very* soon.

Now, about these annotation routes. Comment-out the YAML route:

4 lines | config/routes.yaml

```
#index:
#    path: /
#    controller: App\Controller\ArticleController::homepage
```

Then, in `ArticleController`, above the controller method, add `/**`, hit enter, clear this out, and say `@Route()`. You can use either class - but make sure PhpStorm adds the `use` statement on top. Then add `"/"`:

18 lines | src/Controller/ArticleController.php

```
... lines 1 - 4
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
... lines 6 - 7
class ArticleController
{
    /**
     * @Route("/")
     */
    public function homepage()
    {
... line 15
    }
}
```

That's it! The route is defined *right* above the controller, which is why I *love* annotation routes: everything is in one place. But don't trust me, find your browser and refresh. It's a traaaap! I mean, it works!

Tip

What *exactly* are annotations? They're PHP comments that are read as configuration.

Fancy Wildcard Routes

So what else can we do with routes? Create another public function called `show()`. I want this page to eventually display a full article. Give it a route: `@Route("/news/why-asteroids-taste-like-bacon")`:

26 lines | src/Controller/ArticleController.php

```
... lines 1 - 4
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
... lines 6 - 7
class ArticleController
{
... lines 10 - 17
    /**
     * @Route("/news/why-asteroids-taste-like-bacon")
     */
    public function show()
    {
... line 23
    }
}
```

Eventually, this is how we want our URLs to look. This is called a "slug", it's a URL version of the title. As usual, return a `new Response('Future page to show one space article!')`:

26 lines | src/Controller/ArticleController.php

```
... lines 1 - 4
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;

class ArticleController
{
... lines 10 - 17
    /**
     * @Route("/news/why-asteroids-taste-like-bacon")
     */
    public function show()
    {
        return new Response('Future page to show one space article!');
    }
}
```

Perfect! Copy that URL and try it in your browser. It works... but this sucks! I don't want to build a route and controller for *every* single article that lives in the database. Nope, we need a route that can match `/news/` *anything*. How? Use `{slug}`:

29 lines | src/Controller/ArticleController.php

```
... lines 1 - 7
class ArticleController
{
... lines 10 - 17
    /**
     * @Route("/news/{slug}")
     */
    public function show($slug)
    {
... lines 23 - 26
    }
}
```

This route *now* matches `/news/` anything: that `{slug}` is a *wildcard*. Oh, and the name `slug` could be anything. But whatever you choose now becomes available as an *argument* to your "ghob", I mean your action.

So let's refactor our success message to say:

> Future page to show the article

And then that slug:

29 lines | src/Controller/ArticleController.php

```
... lines 1 - 7
class ArticleController
{
... lines 10 - 17
    /**
     * @Route("/news/{slug}")
     */
    public function show($slug)
    {
        return new Response(sprintf(
            'Future page to show the article: "%s"',
            $slug
        ));
    }
}
```

Try it! Refresh the same URL. Yes! It matches the route *and* the slug prints! Change it to something else: `/why-asteroids-taste-like-tacos`. So delicious! Go back to bacon... because... ya know... everyone knows that's what asteroids *really* taste like.

And... yes! We're 3 chapters in and you *now* know the first *half* of Symfony: the route & controller system. Sure, you can do fancier things with routes, like match regular expressions, HTTP methods or host names - but that will all be pretty easy for you now.

It's time to move on to something *really* important: it's time to learn about Symfony Flex and the *recipe* system. Yum!

Chapter 4: Symfony Flex & Aliases

It's time to demystify something *incredible*: tractor beams. Well actually, we haven't figured those out yet... so let's demystify something *else*, something that's already been happening behind the scenes. First commit everything, with a nice message:

```
$ git add .
$ git commit -m "making so much good progress"
```

Installing the Security Checker

Let's install a new feature called the Symfony Security Checker. This is a *great* tool.... but... full disclosure: we're *mostly* installing it to show of the *recipe* system. Ooooo. Run:

```
$ git status
```

Ok, there are *no* changes. Now run:

```
$ composer require sec-checker
```

Tip

This package will only be used while developing. So, it would be even better to run `composer require sec-checker --dev`.

Hello Symfony Flex

Once again, `sec-checker` should *not* be a valid package name! So what's going on? Move over and open `composer.json`:

64 lines | composer.json

```
{
... lines 2 - 3
    "require": {
... lines 5 - 8
        "symfony/flex": "^1.0",
... lines 10 - 13
    },
... lines 15 - 62
}
```

Our project began with just a *few* dependencies. One of them was `symfony/flex`: this is *super* important. Flex is a Composer plugin with two superpowers.

Flex Aliases

The first superpower is the *alias* system. Find your browser and go to symfony.sh.

This is the Symfony "recipe" server: we'll talk about what that means next. Search for "security". Ah, here's a package called `sensiolabs/security-checker`. And below, it has aliases: `sec-check`, `sec-checker`, `security-check` and

more.

Thanks to Flex, we can say `composer require sec-checker`, or *any* of these aliases, and it will translate that into the real package name. Yep, it's just a shortcut system. But the result is *really* cool. Need a logger? `composer require logger`. Need to send emails? `composer require mailer`. Need a tractor beam? `composer require`, wait, no, we can't help with that one.

Back in `composer.json`, yep! Composer *actually* added `sensiolabs/security-checker`:

64 lines | composer.json

```
{
... lines 2 - 14
    "require-dev": {
        "sensiolabs/security-checker": "^4.1",
... line 17
    },
... lines 19 - 62
}
```

That's the *first* superpower of Flex.

Flex Recipes

The *second* superpower is even better: recipes. Mmmm. Go back to your terminal and... yes! It *did* install and, check this out: "Symfony operations: 1 recipe". Then, "Configuring `sensiolabs/security-checker`".

What does that mean? Run:

```
$ git status
```

Woh! We *expected* `composer.json` and `composer.lock` to be updated. But there are *also* changes to a `symfony.lock` file and we suddenly have a *brand new* config file!

First, `symfony.lock`: this file is managed by Flex. It keeps track of which recipes have been installed. Basically... commit it to git, but don't worry about it.

The second file is `config/packages/dev/security_checker.yaml`:

9 lines | config/packages/dev/security_checker.yaml

```
services:
    SensioLabs\Security\SecurityChecker:
        public: false

    SensioLabs\Security\Command\SecurityCheckerCommand:
        arguments: ['@SensioLabs\Security\SecurityChecker']
        tags:
            - { name: console.command }
```

This was added by the recipe and, cool! It adds a new `bin/console` command to our app! Don't worry about the code itself: you'll understand and be writing code like this soon enough!

The point is this: thanks to this file, we can now run:

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

Cool! This is the recipe system in action! Whenever you install a package, Flex will execute the *recipe* for that package, if there is one. Recipes can add configuration files, create directories, or even modify files like `.gitignore` so that the library *instantly* works without *any* extra setup. I *love* Flex.

By the way, the *purpose* of the security checker is that it checks to see if there are any known vulnerabilities

for packages used in our project. Right now, we're good!

But the recipe made one *other* change. Run:

```
$ git diff composer.json
```

Of course, `composer require` added the package. But the *recipe* added a new *script*!

64 lines | composer.json

```
{
... lines 2 - 40
    "scripts": {
        "auto-scripts": {
... lines 43 - 44
            "security-checker security:check": "script"
        },
... lines 47 - 52
    },
... lines 54 - 62
}
```

Thanks to that, whenever we run:

```
$ composer install
```

when it finishes, it runs the security checker automatically. So cool!

Oh, and I won't show it right now, but Flex is even smart enough to *uninstall* the recipes when you *remove* a package. That makes testing out new packages fast and easy.

The Recipes Repository

So you might be wondering... where do these recipes live? Great question! They live... in the *cloud*. I mean, they live on GitHub. On symfony.sh, click "Recipe" next to the Security checker. Ah, it takes us to the `symfony/recipes` repository. Here, you can see what files will be added and a few other changes described in `manifest.json`.

All recipes either live in this repository, or another one called `symfony/recipes-contrib`. There's no important difference between the two repositories: but the official recipes are watched more closely for quality.

Next! Let's put the recipe system to work by installing Twig so we can create proper templates.

Chapter 5: The Twig Recipe

Do you remember the *only* rule for a controller? It must return a Symfony Response object! But Symfony doesn't care *how* you do that: you could render a template, make API requests or make database queries and build a JSON response.

Tip

Technically, a controller can return *anything*. Eventually, you'll learn how and why to do this.

Really, *most* of learning Symfony involves learning to install and use a *bunch* of powerful, but optional, tools that make this work easier. If your app needs to return HTML, then one of these great tools is called Twig.

Installing Twig

First, make sure you commit all of your changes so far:

```
$ git status
```

I already did this. Recipes are *so* much more fun when you can see what they do! Now run:

```
$ composer require twig
```

By the way, in future tutorials, our app will become a mixture of a traditional HTML app and an API with a JavaScript front-end. So if you want to know about building an API in Symfony, we'll get there!

This installs TwigBundle, a few other libraries and... configures a recipe! What did that recipe do? Let's find out:

```
$ git status
```

Woh! Lot's of good stuff! The first change is `config/bundles.php`:

9 lines | config/bundles.php

```
... lines 1 - 2
return [
... lines 4 - 6
    Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
];
```

Bundles are the "plugin" system for Symfony. And whenever we install a third-party bundle, Flex adds it here so that it's used automatically. Thanks Flex!

The recipe also *created* some stuff, like a `templates/` directory! Yep, no need to guess where templates go: it's pretty obvious! It even added a base layout file that we'll use soon.

Twig also needs some configuration, so the recipe added it in `config/packages/twig.yaml`:

5 lines | config/packages/twig.yaml

```yaml
twig:
    paths: ['%kernel.project_dir%/templates']
    debug: '%kernel.debug%'
    strict_variables: '%kernel.debug%'
```

But even though this file was added by Flex, it's *yours* to modify: you can make whatever changes you want.

Oh, and I *love* this! *Why* do our templates need to live in a `templates/` directory. Is that hardcoded deep inside Symfony? Nope! It's right here!

5 lines | config/packages/twig.yaml

```yaml
twig:
    paths: ['%kernel.project_dir%/templates']
... lines 3 - 5
```

Don't worry about this percent syntax yet - you'll learn about that in a future episode. But, you can probably guess what's going on: `%kernel.project_dir%` is a variable that points to the root of the project.

Anyways, looking at what a recipe did is a *great* way to learn! But the main lesson of Flex is this: install a library and it takes care of the rest.

Now, let's go *use* Twig!

Chapter 6: Twig ❤

Back to work! Open `ArticleController`. As soon as you want to render a template, you need to extend a base class: `AbstractController`:

29 lines | src/Controller/ArticleController.php

```
... lines 1 - 5
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
... lines 7 - 8
class ArticleController extends AbstractController
{
... lines 11 - 27
}
```

Obviously, your controller does not *need* to extend this. But they usually will... because this class gives you shortcut methods! The one we want is `return $this->render()`. Pass it a template filename: how about `article/show.html.twig` to be consistent with the controller name. The second argument is an array of variables that you want to pass *into* your template:

29 lines | src/Controller/ArticleController.php

```
... lines 1 - 8
class ArticleController extends AbstractController
{
... lines 11 - 21
    public function show($slug)
    {
        return $this->render('article/show.html.twig', [
... line 25
        ]);
    }
}
```

Eventually, we're going to load articles from the database. But... hang on! We're not quite ready yet. So let's fake it 'til we make it! Pass a `title` variable set to a title-ized version of the slug:

29 lines | src/Controller/ArticleController.php

```
... lines 1 - 8
class ArticleController extends AbstractController
{
... lines 11 - 21
    public function show($slug)
    {
        return $this->render('article/show.html.twig', [
            'title' => ucwords(str_replace('-', ' ', $slug)),
        ]);
    }
}
```

Great! Let's go add that template! Inside `templates/`, create an `article` directory then the file: `show.html.twig`.

Add an `h1`, then print that `title` variable: `{{ title }}`:

16 lines | templates/article/show.html.twig

```
<h1>{{ title }}</h1>
... lines 2 - 16
```

Twig Basics

If you're new to Twig, welcome! You're going to *love* it! Twig only has *2* syntaxes. The first is `{{ }}`. I call this the "say something" tag, because it *prints*. And just like PHP, you can print anything: a variable, a string or a complex expression.

The second syntax is `{% %}`. I call this the "do something" tag. It's used whenever you need to, um, *do* something, instead of printing, like an `if` statement or `for` loop. We'll look at the *full* list of do something tags in a minute.

And... yea, that's it! Well, ok, I totally lied. There is a *third* syntax: `{# #}`: comments!

At the bottom of this page, I'll paste some extra hard-coded content to *spice* things up!

16 lines | templates/article/show.html.twig

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

<div>
    <p>
        Bacon ipsum dolor amet filet mignon picanha kielbasa jowl hamburger shankle biltong chicken turkey pastrami cupim
    </p>

    <p>
        Kielbasa pork belly meatball cupim burgdoggen chuck turkey buffalo ground round leberkas cow shank short loin bac
    </p>

    <p>
        Pastrami short loin pork chop, chicken kielbasa swine turducken jerky short ribs beef. Short ribs alcatra shoulder, flan
    </p>
</div>
```

Let's go try it! Find your browser and refresh! Boom! We have content!

But check it out: if you view the page source... it's *just* this content: we don't have any layout or HTML structure yet. But, we will soon!

Looping with for

Go back to your controller. Eventually, users will need to be able to comment on the articles, so they can respectfully debate the article's conclusions based on objective analysis and research. Ya know... no different than *any* other news commenting section. Ahem.

I'll paste in 3 fake comments. Add a second variable called `comments` to pass these into the template:

36 lines | src/Controller/ArticleController.php

```
... lines 1 - 8
class ArticleController extends AbstractController
{
... lines 11 - 21
    public function show($slug)
    {
        $comments = [
            'I ate a normal rock once. It did NOT taste like bacon!',
            'Woohoo! I\'m going on an all-asteroid diet!',
            'I like bacon too! Buy some from my site! bakinsomebacon.com',
        ];

        return $this->render('article/show.html.twig', [
... line 31
            'comments' => $comments,
        ]);
    }
}
```

This time, we can't just *print* that array: we need to loop over it. At the bottom, and an `h2` that says "Comments" and then add a `ul`:

24 lines | templates/article/show.html.twig

```
... lines 1 - 16
<h2>Comments</h2>

<ul>
... lines 20 - 22
</ul>
```

To loop, we need our first *do* something tag! Woo! Use `{% for comment in comments %}`. Most "do" something tags also have a closing tag: `{% endfor %}`:

24 lines | templates/article/show.html.twig

```
... lines 1 - 16
<h2>Comments</h2>

<ul>
    {% for comment in comments %}
... line 21
    {% endfor %}
</ul>
```

Inside the loop, `comment` represents the individual comment. So, just print it: `{{ comment }}`:

24 lines | templates/article/show.html.twig

```
... lines 1 - 16
<h2>Comments</h2>

<ul>
    {% for comment in comments %}
        <li>{{ comment }}</li>
    {% endfor %}
</ul>
```

Try it! Brilliant! I mean, it's *really* ugly... oof. But we'll fix that later.

The Amazing Twig Reference

Go to twig.symfony.com and click on the Documentation link. Scroll down a little until you see a set of columns:

the Twig Reference.

This is *awesome*! See the tags on the left? That is the *entire* list of possible "do something" tags. Yep, it will always be `{%` and then one of these: `for`, `if`, `extends`, `tractorbeam`. And honestly, you're only going to use about 5 of these most of the time.

Twig also has functions... which work like every other language - and a cool thing called "tests". Those are a bit unique, but not too difficult, they allow you to say things like `if foo is defined` or... `if space is empty`.

The most *useful* part of this reference is the filter section. Filters are like functions but with a different, way more hipster syntax. Let's try our the `|length` filter.

Go back to our template. I want to print out the total *number* of comments. Add a set of parentheses and then say `{{ comments|length }}`:

```
24 lines | templates/article/show.html.twig
... lines 1 - 16
<h2>Comments ({{ comments|length }})</h2>
... lines 18 - 24
```

That is a filter: the `comments` value passes from the left to right, just like a Unix pipe. The `length` filter counts whatever was passed to it, and we print the result. You can even use *multiple* filters!

Tip

To unnecessarily confuse your teammates, try using the `upper` and `lower` filters over and over again: `{{ name|upper|lower|upper|lower|upper }}` !

Template Inheritance

Twig has *one* last *killer* feature: it's template inheritance system. Because remember! We don't *yet* have a *real* HTML page: just the content from the template.

To fix this, at the top of the template, add `{% extends 'base.html.twig' %}`:

```
26 lines | templates/article/show.html.twig
{% extends 'base.html.twig' %}
... lines 2 - 26
```

This refers to the `base.html.twig` file that was added by the recipe:

```
13 lines | templates/base.html.twig
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </body>
</html>
```

It's simple now, but this is *our* layout file and we'll customize it over time. By extending it, we should *at least* get this basic HTML structure.

But when we refresh... surprise! An error! And probably one that you'll see at some point!

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

Huh. Look at the base template again: it's basically an HTML layout plus a bunch of blocks... most of which are

empty. When you extend a template, you're telling Twig that you want to put your content *inside* of that template. The blocks, are the "holes" *into* which our child template can put content. For example, there's a block called `body`, and that's *really* where we want to put our content:

13 lines | templates/base.html.twig

```
<!DOCTYPE html>
<html>
... lines 3 - 7
    <body>
        {% block body %}{% endblock %}
... line 10
    </body>
</html>
```

To do that, we need to *override* that block. At the top of the content, add `{% block body %}`, and at the bottom, `{% endblock %}`:

28 lines | templates/article/show.html.twig

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

{% block body %}
<h1>{{ title }}</h1>
... lines 5 - 21
<ul>
    {% for comment in comments %}
        <li>{{ comment }}</li>
    {% endfor %}
</ul>
{% endblock %}
```

Now our content should go *inside* of that block in `base.html.twig`. Try it! Refresh! Yes! Well, it doesn't look any different, but we *do* have a proper HTML body.

More about Blocks

You're *completely* free to customize this template as much as you want: rename the blocks, add more blocks, and, hopefully, make the site look less ugly!

Oh, and *most* of the time, the blocks are empty. But you *can* give the block some *default* content, like with `title`:

13 lines | templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
... line 4
        <title>{% block title %}Welcome!{% endblock %}</title>
... line 6
    </head>
... lines 8 - 11
</html>
```

Yep, the browser tab's title *is* `Welcome`.

Let's override that! At the top... or really, *anywhere*, add `{% block title %}`. Then say `Read`, print the `title` variable, and `{% endblock %}`:

30 lines | templates/article/show.html.twig

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

{% block title %}Read: {{ title }}{% endblock %}
... lines 4 - 30
```

Try that! Yes! The page title changes. And... voilà! That's Twig. You're going to *love* it.

Go Deeper!

Check out another screencast from us to learn more about Twig

Next let's check out one of Symfony's most *killer* features: the profiler.

Chapter 7: Web Debug Toolbar & the Profiler!

Make sure you've committed *all* of your changes - I already did. Because we're about to install something *super* fun! Like, floating around space fun! Run:

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

The profiler - also called the "web debug toolbar" is probably the most *awesome* thing in Symfony. This installs a few packages and... one recipe! Run:

```
$ git status
```

Ok cool! It added a couple of configuration files and even some routes in the `dev` environment only that help the profiler work. So... what the heck *is* the profiler? Go back to your browser, make sure you're on the article show page and refresh! Voilà!

Hello Web Debug Toolbar!

See that slick black bar at the bottom of the page! That's the web debug toolbar! It's now automatically injected at the bottom of any valid HTML page during development. Yep, this JavaScript code makes an AJAX call that loads it.

Oh, and it's *packed* with info, like which route was matched, what controller was executed, execution time, cache details and even information about templates.

And as we install more libraries, we're going to get even *more* icons! But the *really* awesome thing is that you can click any of these icons to go into... the *profiler*.

Hello Profiler: The Toolbar's Powerful Sidekick

OoooOoo. This takes us to a totally different page. The profiler is like the web debug toolbar with a fusion reactor taped onto it. The Twig tab shows exactly which templates were rendered. We can also get detailed info about caching, routing and events, which we'll talk about in a future tutorial. Oh, and my *personal* favorite: Performance! This shows you how long each part of the request took, including the controller. In another tutorial, we'll use this to dig into *exactly* how Symfony works under the hood.

When you're ready to go back to the original page, you can click the link at the top.

Magic with The dump() Function

But wait, there's more! The profiler also installed Symfony's `var-dumper` component. Find `ArticleController` and go to `showAction()`. Normally, to debug, I'll use `var_dump()` to print some data. But, no more! Instead, use `dump()`: dump the `$slug` and also the controller object itself:

37 lines | src/Controller/ArticleController.php

```
... lines 1 - 8
class ArticleController extends AbstractController
{
... lines 11 - 21
    public function show($slug)
    {
... lines 24 - 28
        dump($slug, $this);
... lines 30 - 34
    }
}
```

Ok, refresh! Beautiful, colored output. *And*, you can expand objects to dig deeper into them.

Tip

To expand all the nested nodes just press `Ctrl` and click the arrow.

Using dump() in Twig

The `dump()` function is even *more* useful in Twig. Inside the `body` block, add `{{ dump() }}`:

31 lines | templates/article/show.html.twig

```
... lines 1 - 4
{% block body %}
{{ dump() }}
... lines 7 - 29
{% endblock %}
```

In Twig, you're allowed to use `dump()` with *no* arguments. And that's *especially* useful. Why? Because it dumps an associative array of *all* of the variables you have access to. We already knew we had `title` and `comments` variables. But apparently, we *also* have an `app` variable! Actually, *every* template gets this `app` variable automatically. Good to know!

But! Symfony has even *more* debugging tools! Let's get them and learn about "packs" next!

Chapter 8: Debugging & Packs

Symfony has even *more* debugging tools. The easiest way to get *all* of them is to find your terminal and run:

```
$ composer require debug --dev
```

Find your browser, surf back to symfony.sh and search for "debug". Ah, so the `debug` alias will actually install a package called `symfony/debug-pack`. So... what's a pack?

Click to look at the package details, and then go to its GitHub repository.

Whoa! It's just a single file: `composer.json`! Inside, *it* requires six other libraries!

Sometimes, you're going to want to install *several* packages at once related to one feature. To make that easy, Symfony has a number of "packs", and their *whole* purpose is give you *one* easy package that *actually* installs several *other* libraries.

In this case, `composer require debug` will install Monolog - a logging library, `phpunit-bridge` - for testing, and even the `profiler-pack` that we already installed earlier.

If you go back to the terminal... yep! It downloaded all those libraries and configured a few recipes.

And... check this out! Refresh! Hey! Our Twig `dump()` got prettier! The `debug-pack` integrated everything together even better.

Unpacking the Pack!

Go back to your Twig template and remove that dump. Then, open `composer.json`. We just installed two packs: the `debug-pack` and the `profiler-pack`:

67 lines | composer.json

```
{
... lines 2 - 15
    "require-dev": {
... line 17
        "symfony/debug-pack": "^1.0",
... line 19
        "symfony/profiler-pack": "^1.0"
    },
... lines 22 - 65
}
```

And we *now* know that the `debug-pack` is actually a collection of about 6 libraries.

But, packs have a *disadvantage*... a "dark side". What if you wanted to control the version of just *one* of these libraries? Or what if you wanted *most* of these libraries, but you didn't want, for example, the `phpunit-bridge`. Well... right now, there's no way to do that: all we have is this *one* `debug-pack` line.

Don't worry brave space traveler! Just... unpack the pack! Yep, at your terminal, run:

```
$ composer unpack debug
```

The `unpack` command comes from Symfony flex. And... interesting! All it says is "removing symfony/debug-pack". But if you look at your `composer.json`:

71 lines | composer.json

```
{
... lines 2 - 15
    "require-dev": {
        "easycorp/easy-log-handler": "^1.0.2",
... line 18
        "symfony/debug-bundle": "^3.3|^4.0",
... line 20
        "symfony/monolog-bundle": "^3.0",
        "symfony/phpunit-bridge": "^3.3|^4.0",
        "symfony/profiler-pack": "^1.0",
        "symfony/var-dumper": "^3.3|^4.0"
    },
... lines 26 - 69
}
```

Ah! It *did* remove `symfony/debug-pack`, but it *replaced* it with the 6 libraries from that pack! We can *now* control the versions or even *remove* individual libraries if we don't want them.

That is the power of packs!

Chapter 9: Assets: CSS & JavaScript

Even astronauts - who *generally* spend their time staring into the black absyss - demand a site that is *less* ugly than this! Let's fix that!

If you download the course code from the page that you're watching this video on right now, inside the zip file, you'll find a `start/` directory. And inside *that*, you'll see the same `tutorial/` directory that I have here. And inside *that*... I've created a new `base.html.twig`. Copy that and overwrite our version in `templates/`:

67 lines | templates/base.html.twig

```
<!doctype html>
<html lang="en">

    <head>
        <title>{% block title %}Welcome to the SpaceBar{% endblock %}</title>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

        {% block stylesheets %}
            <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/css/bootstrap.min.css" integ
        {% endblock %}
    </head>

    <body>
        <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
            <a style="margin-left: 75px;" class="navbar-brand space-brand" href="#">The Space Bar</a>
            <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNavDropdown" aria-c
            <span class="navbar-toggler-icon"></span>
          </button>
            <div class="collapse navbar-collapse" id="navbarNavDropdown">
                <ul class="navbar-nav mr-auto">
                    <li class="nav-item">
                      <a style="color: #fff;" class="nav-link" href="#">Local Asteroids</a>
                    </li>
                    <li class="nav-item">
                      <a style="color: #fff;" class="nav-link" href="#">Weather</a>
                    </li>
                  </ul>
                <form class="form-inline my-2 my-lg-0">
                    <input class="form-control mr-sm-2" type="search" placeholder="Search" aria-label="Search">
                    <button class="btn btn-info my-2 my-sm-0" type="submit">Search</button>
                </form>
                <ul class="navbar-nav ml-auto">
                    <li class="nav-item dropdown" style="margin-right: 75px;">
                        <a class="nav-link dropdown-toggle" href="http://example.com" id="navbarDropdownMenuLink" data-togg
                  <img class="nav-profile-img rounded-circle" src="images/astronaut-profile.png">
                </a>
                        <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
                            <a class="dropdown-item" href="#">Profile</a>
                            <a class="dropdown-item" href="#">Create Post</a>
                            <a class="dropdown-item" href="#">Logout</a>
                        </div>
                    </li>
```

```
            </ul>
        </div>
    </nav>

    {% block body %}{% endblock %}

    <footer class="footer">
        <div class="container text-center">
            <span class="text-muted">Made with <i class="fa fa-heart" style="color: red;"></i> by the guys and gals at <
        </div>
    </footer>


    {% block javascripts %}
        <script src="https://code.jquery.com/jquery-3.2.1.min.js" integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQui
        <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.3/umd/popper.min.js" integrity="sha384-vFJXuSJp
        <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/js/bootstrap.min.js" integrity="sha384-alpBp
        <script>
            $('.dropdown-toggle').dropdown();
        </script>
    {% endblock %}
</body>
</html>
```

On a technical level, this is basically the same as before: it has the same blocks: `title` `stylesheets`, `body` and `javascripts` at the bottom. But now, we have a nice HTML layout that's styled with Bootstrap.

If you refresh, it should look better. Woh! No change! Weird! Actually... this is *more* weird than you might think. Find your terminal and remove the `var/cache/dev` directory:

```
$ rm -rf var/cache/dev/*
```

What the heck is this? Internally, Symfony caches things in this directory. And... you normally don't need to think about this at *all*: Symfony is smart enough during development to automatically rebuild this cache whenever necessary. So... why am I *manually* clearing it? Well... because we copied *my* file... and because its "last modified" date is *older* than our original `base.html.twig`, Twig gets confused and thinks that the template was *not* updated. Seriously, this is *not* something to worry about in any other situation.

Referencing CSS Files

And when we refresh... there it is! Ok, it's still *pretty* ugly. That's because we're missing some CSS files!

In the `tutorial/` directory, I've also prepped some `css/`, `fonts/` and `images/`. All of these files need to be accessed by the user's browser, and that means they must live inside `public/`. Open that directory and paste them there.

By the way, Symfony has an *awesome* tool called Webpack Encore that helps process, combine, minify and generally do *amazing* things with your CSS and JS files. We *are* going to talk about Webpack Encore... but in a different tutorial. For now, let's get things setup with normal, static files.

The two CSS files we want to include are `font-awesome.css` and `styles.css`. And we don't need to do *anything* complex or special! In `base.html.twig`, find the `stylesheets` block and add a `link` tag.

But wait, why exactly are we adding the `link` tag *inside* the `stylesheets` block? Is that important? Well, technically... it doesn't matter: a `link` tag can live anywhere in `head`. But later, we might want to add additional CSS files on specific *pages*. By putting the `link` tags inside this block, we'll have more flexibility to do that. Don't worry: we're going to see an example of this with a JavaScript file soon.

So... what path should we use? Since `public/` is the document root, it should just be `/css/font-awesome.css`:

69 lines | templates/base.html.twig

```
<!doctype html>
<html lang="en">

    <head>
... lines 5 - 8
        {% block stylesheets %}
... line 10
            <link rel="stylesheet" href="/css/font-awesome.css">
... line 12
        {% endblock %}
    </head>
... lines 15 - 67
</html>
```

Do the same thing for the other file: `/css/styles.css`:

69 lines | templates/base.html.twig

```
<!doctype html>
<html lang="en">

    <head>
... lines 5 - 8
        {% block stylesheets %}
... line 10
            <link rel="stylesheet" href="/css/font-awesome.css">
            <link rel="stylesheet" href="/css/styles.css">
        {% endblock %}
    </head>
... lines 15 - 67
</html>
```

It's that simple! Refresh! Still not perfect, but *much* better!

The Not-So-Mystical asset Function

And *now* I'm going to *slightly* complicate things. Go back into PhpStorm's Preferences, search for "Symfony" and find the "Symfony" plugin. Change the "web" directory to `public` - it was called `web` in Symfony 3.

This is *not* required, but it will give us more auto-completion when working with assets. Delete the "font-awesome" path, re-type it, and hit tab to auto-complete:

69 lines | templates/base.html.twig

```
<!doctype html>
<html lang="en">

    <head>
... lines 5 - 8
        {% block stylesheets %}
... line 10
            <link rel="stylesheet" href="{{ asset('css/font-awesome.css') }}">
... line 12
        {% endblock %}
    </head>
... lines 15 - 67
</html>
```

Woh! It wrapped the path in a Twig `asset()` function! Do the same thing below for `styles.css`:

69 lines | templates/base.html.twig

```
<!doctype html>
<html lang="en">

    <head>
... lines 5 - 8
        {% block stylesheets %}
... line 10
            <link rel="stylesheet" href="{{ asset('css/font-awesome.css') }}">
            <link rel="stylesheet" href="{{ asset('css/styles.css') }}">
        {% endblock %}
    </head>
... lines 15 - 67
</html>
```

Here's the deal: whenever you link to a static asset - CSS, JS or images - you *should* wrap the path in this `asset()` function. But... it's not *really* that important. In fact, right now, it doesn't *do* anything: it will print the same path as before. But! In the future, the `asset()` function will give us more flexibility to *version* our assets or store them on a CDN.

In other words: don't worry about it too much, but *do* remember to use it!

Installing the asset Component

Actually, the `asset()` function *does* do something immediately - it breaks our site! Refresh! Ah!

The `asset()` function comes from a part of Symfony that we don't have installed yet. Fix that by running:

```
$ composer require asset
```

This installs the `symfony/asset` component. And as soon as Composer is done... we can refresh, and it works! To prove that the `asset()` function isn't doing anything magic, you can look at the `link` tag in the HTML source: it's the same boring `/css/styles.css`.

There is one other spot where we need to use `asset()`. In the layout, search for `img`. Ah, an `img` tag! Remove the `src` and re-type `astronaut-profile`:

69 lines | templates/base.html.twig

```
<!doctype html>
<html lang="en">
... lines 3 - 15
    <body>
        <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
... lines 18 - 21
            <div class="collapse navbar-collapse" id="navbarNavDropdown">
... lines 23 - 34
                <ul class="navbar-nav ml-auto">
                    <li class="nav-item dropdown" style="margin-right: 75px;">
                        <a class="nav-link dropdown-toggle" href="http://example.com" id="navbarDropdownMenuLink" data-togg
                 <img class="nav-profile-img rounded-circle" src="{{ asset('images/astronaut-profile.png') }}">
                </a>
... lines 40 - 44
                    </li>
                </ul>
            </div>
        </nav>
... lines 49 - 66
    </body>
</html>
```

Perfect! Refresh and enjoy our new avatar on the user menu. There's a lot of hardcoded data, but we'll make this dynamic over time.

Styling the Article Page

The layout is looking great! But the *inside* of the page... yea... that's still pretty terrible. Back in the `tutorial/` directory, there is *also* an `article.html.twig` file. Don't copy this entire file - just copy its contents. Close it and open `show.html.twig`. Paste the new code at the top of the `body` block:

122 lines | templates/article/show.html.twig

```
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
                <div class="row">
                    <div class="col-sm-12">
                        <img class="show-article-img" src="{{ asset('images/asteroid.jpeg') }}">
                        <div class="show-article-title-container d-inline-block pl-3 align-middle">
                            <span class="show-article-title ">Why do Asteroids Taste Like Bacon?</span>
                            <br>
                            <span class="align-left article-details"><img class="article-author-img rounded-circle" src="{{ asset('i
                            <span class="pl-2 article-details"> 3 hours ago</span>
                            <span class="pl-2 article-details"> 5 <a href="#" class="fa fa-heart-o like-article"></a> </span>
                        </div>
                    </div>
                </div>
                <div class="row">
                    <div class="col-sm-12">
                        <div class="article-text">
                            <p>Spicy jalapeno bacon ipsum dolor amet veniam shank in dolore. Ham hock nisi landjaeger cow,
                                lorem proident beef ribs aute enim veniam ut cillum pork chuck picanha. Dolore reprehenderit
                                labore minim pork belly spare ribs cupim short loin in. Elit exercitation eiusmod dolore cow
```

```
                    turkey shank eu pork belly meatball non cupim.</p>

                <p>Laboris beef ribs fatback fugiat eiusmod jowl kielbasa alcatra dolore velit ea ball tip. Pariatur
                    laboris sunt venison, et laborum dolore minim non meatball. Shankle eu flank aliqua shoulder,
                    capicola biltong frankfurter boudin cupim officia. Exercitation fugiat consectetur ham. Adipisicing
                    picanha shank et filet mignon pork belly ut ullamco. Irure velit turducken ground round doner incididur
                    occaecat lorem meatball prosciutto quis strip steak.</p>

                <p>Meatball adipisicing ribeye bacon strip steak eu. Consectetur ham hock pork hamburger enim strip s
                    mollit quis officia meatloaf tri-tip swine. Cow ut reprehenderit, buffalo incididunt in filet mignon
                    strip steak pork belly aliquip capicola officia. Labore deserunt esse chicken lorem shoulder tail consect
                    cow est ribeye adipisicing. Pig hamburger pork belly enim. Do porchetta minim capicola irure pancetta
                    fugiat.</p>

                <p>Sausage tenderloin officia jerky nostrud. Laborum elit pastrami non, pig kevin buffalo minim ex quis.
                    pork chop officia anim. Irure tempor leberkas kevin adipisicing cupidatat qui buffalo ham aliqua pork b
                    exercitation eiusmod. Exercitation incididunt rump laborum, t-bone short ribs buffalo ut shankle pork c
                    bresaola shoulder burgdoggen fugiat. Adipisicing nostrud chicken consequat beef ribs, quis filet migno
                    Prosciutto capicola mollit shankle aliquip do dolore hamburger brisket turducken eu.</p>

                <p>Do mollit deserunt prosciutto laborum. Duis sint tongue quis nisi. Capicola qui beef ribs dolore pariat
                    Minim strip steak fugiat nisi est, meatloaf pig aute. Swine rump turducken nulla sausage. Reprehender
                    belly tongue alcatra, shoulder excepteur in beef bresaola duis ham bacon eiusmod. Doner drumstick s
                    adipisicing cow cillum tenderloin.</p>
            </div>
        </div>
    </div>
    <div class="row">
        <div class="col-sm-12">
            <p class="share-icons mb-5"><span class="pr-1">Share:</span> <i class="pr-1 fa fa-facebook-square">
        </div>
    </div>
    <div class="row">
        <div class="col-sm-12">
            <h3><i class="pr-3 fa fa-comment"></i>10 Comments</h3>
            <hr>

            <div class="row mb-5">
                <div class="col-sm-12">
                    <img class="comment-img rounded-circle" src="{{ asset('images/astronaut-profile.png') }}">
                    <div class="comment-container d-inline-block pl-3 align-top">
                        <span class="commenter-name">Amy Oort</span>
                        <div class="form-group">
                            <textarea class="form-control comment-form" id="articleText" rows="1"></textarea>
                        </div>
                        <button type="submit" class="btn btn-info">Comment</button>
                    </div>
                </div>
            </div>

            <div class="row">
                <div class="col-sm-12">
                    <img class="comment-img rounded-circle" src="{{ asset('images/alien-profile.png') }}">
                    <div class="comment-container d-inline-block pl-3 align-top">
                        <span class="commenter-name">Mike Ferengi</span>
```

```
                        <br>
                        <span class="comment"> Now would this be apple wood smoked bacon? Or traditional bacon - IMH
                        <p><a href="#">Reply</a></p>
                    </div>
                </div>
            </div>

        </div>
    </div>
</div>
</div>

<h1>{{ title }}</h1>
... lines 99 - 122
```

Check it out in your browser. Yep! It looks cool... but *all* of this info is hardcoded. I mean, that article name is just static text.

Let's take the dynamic code that we have at the bottom and work it into the new HTML. For the title, use `{{ title }}`:

101 lines | templates/article/show.html.twig

```
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
                <div class="row">
                    <div class="col-sm-12">
... line 13
                        <div class="show-article-title-container d-inline-block pl-3 align-middle">
                            <span class="show-article-title ">{{ title }}</span>
... lines 16 - 19
                        </div>
                    </div>
                </div>
... lines 23 - 94
            </div>
        </div>
    </div>
</div>

{% endblock %}
```

Below, it prints the number of comments. Replace that with `{{ comments|length }}`:

101 lines | templates/article/show.html.twig

```
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
... lines 11 - 60
                <div class="row">
                    <div class="col-sm-12">
                        <h3><i class="pr-3 fa fa-comment"></i>{{ comments|length }} Comments</h3>
... lines 64 - 92
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>

{% endblock %}
```

Oh, and at the bottom, there is a comment box and one *actual* comment. Let's find this and... add a loop! For `comment in comments` on top, and `endfor` at the bottom. For the actual comment, use `{{ comment }}`:

101 lines | templates/article/show.html.twig

```
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
... lines 11 - 60
                <div class="row">
                    <div class="col-sm-12">
... lines 63 - 78
                        {% for comment in comments %}
                        <div class="row">
                            <div class="col-sm-12">
... line 82
                                <div class="comment-container d-inline-block pl-3 align-top">
... lines 84 - 85
                                    <span class="comment"> {{ comment }}</span>
... line 87
                                </div>
                            </div>
                        </div>
                        {% endfor %}

                    </div>
                </div>
            </div>
        </div>
    </div>
</div>

{% endblock %}
```

Delete the old code from the bottom... oh, but don't delete the `endblock`:

101 lines | templates/article/show.html.twig

```
... lines 1 - 4
{% block body %}

<div class="container">
... lines 8 - 97
</div>

{% endblock %}
```

Let's try it - refresh! It looks awesome! A bunch of things are still hardcoded, but this is *much* better.

It's time to make our homepage less ugly and learn about the *second* job of routing: route *generation* for linking.

Chapter 10: Generating URLs

Most of these links don't got anywhere yet. Whatever! No problem! We're going to fill them in as we continue. Besides, most of our users will be in hypersleep for at least a few more decades.

But we *can* hook up *some* of these - like the "Space Bar" logo text - that should go to the homepage.

Open `templates/base.html.twig` and search for "The Space Bar":

```
69 lines | templates/base.html.twig
<!doctype html>
<html lang="en">
... lines 3 - 15
    <body>
        <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
            <a style="margin-left: 75px;" class="navbar-brand space-brand" href="#">The Space Bar</a>
... lines 19 - 47
        </nav>
... lines 49 - 66
    </body>
</html>
```

Ok - let's point this link to the homepage. And yep, we *could* just say `href="/"`.

But... there's a *better* way. Instead, we're going to *generate* a URL *to* the route. Yep, we're going to ask Symfony to give us the URL to the route that's above our homepage action:

```
36 lines | src/Controller/ArticleController.php
... lines 1 - 8
class ArticleController extends AbstractController
{
    /**
     * @Route("/")
     */
    public function homepage()
    {
... line 16
    }
... lines 18 - 34
}
```

Why? Because if we ever decided to *change* this route's URL - like to `/news` - if we *generate* the URL instead of hardcoding it, *all* the links will automatically update. Magic!

The Famous debug:router

So how can we do this? First, find your terminal and run:

```
$ ./bin/console debug:router
```

This is an awesome little tool that shows you a list of *all* of the routes in your app. You can see *our* two routes *and* a bunch of routes that help the profiler and web debug toolbar.

There's *one* thing about routes that we haven't really talked about yet: each route has an internal name. This is never shown to the user, it only exists so that we can *refer* to that route in our code. For annotation routes, by default, that name is created for us.

Generating URLs with path()

This means, to generate a URL to the homepage, copy the route name, go back to `base.html.twig`, add `{{ path() }}` and paste the route name:

```
69 lines | templates/base.html.twig
<!doctype html>
<html lang="en">
... lines 3 - 15
    <body>
        <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
            <a style="margin-left: 75px;" class="navbar-brand space-brand" href="{{ path('app_article_homepage') }}">The
... lines 19 - 47
        </nav>
... lines 49 - 66
    </body>
</html>
```

That's it!

Refresh! Click it! Yes! We're back on the homepage.

But... actually I *don't* like to rely on auto-created route names because they could *change* if we renamed certain parts of our code. Instead, as soon as I want to generate a URL to a route, I add a name option: `name="app_homepage"`:

```
36 lines | src/Controller/ArticleController.php
... lines 1 - 8
class ArticleController extends AbstractController
{
    /**
     * @Route("/", name="app_homepage")
     */
    public function homepage()
    {
... line 16
    }
... lines 18 - 34
}
```

Run `debug:router` again:

```
$ ./bin/console debug:router
```

The *only* thing that changed is the *name* of the route. Now go back to `base.html.twig` and use the new route name here:

```
69 lines | templates/base.html.twig
```

```
<!doctype html>
<html lang="en">
... lines 3 - 15
    <body>
        <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
            <a style="margin-left: 75px;" class="navbar-brand space-brand" href="{{ path('app_homepage') }}">The Space
... lines 19 - 47
        </nav>
... lines 49 - 66
    </body>
</html>
```

It still works *exactly* like before, but we're in complete control of the route name.

Making the Homepage Pretty

We now have a link to our homepage... but I don't know why you'd want to go here: it's *super* ugly! So let's render a template. In `ArticleController`, instead of returning a `Response`, return `$this->render()` with `article/homepage.html.twig`:

36 lines | src/Controller/ArticleController.php

```
... lines 1 - 8
class ArticleController extends AbstractController
{
    /**
     * @Route("/", name="app_homepage")
     */
    public function homepage()
    {
        return $this->render('article/homepage.html.twig');
    }
... lines 18 - 34
}
```

For now, don't pass *any* variables to the template.

This template does *not* exist yet. But if you look again in the `tutorial/` directory from the code download, I've created a homepage template for us. Sweet! Copy that and paste it into `templates/article`:

81 lines | templates/article/homepage.html.twig

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

{% block body %}
    <div class="container">
        <div class="row">

            <!-- Article List -->

            <div class="col-sm-12 col-md-8">

                <!-- H1 Article -->
                <a class="main-article-link" href="#">
                    <div class="main-article mb-5 pb-3">
                        <img src="{{ asset('images/meteor-shower.jpg') }}" alt="meteor shower">
                        <h1 class="text-center mt-2">Ursid Meteor Shower: <br>Healthier than a regular shower?</h1>
                    </div>
                </a>

                <!-- Supporting Articles -->
```

```
20
21          <div class="article-container my-1">
22              <a href="#">
23                  <img class="article-img" src="{{ asset('images/asteroid.jpeg') }}">
24                  <div class="article-title d-inline-block pl-3 align-middle">
25                      <span>Why do Asteroids Taste Like Bacon?</span>
26                      <br>
27                      <span class="align-left article-details"><img class="article-author-img rounded-circle" src="{{ asset('in
28                      <span class="pl-5 article-details float-right"> 3 hours ago</span>
29                  </div>
30              </a>
31          </div>
32
33          <div class="article-container my-1">
34              <a href="#">
35                  <img class="article-img"  src="{{ asset('images/mercury.jpeg') }}">
36                  <div class="article-title d-inline-block pl-3 align-middle">
37                      <span>Life on Planet Mercury: <br> Tan, Relaxing and Fabulous</span>
38                      <br>
39                      <span class="align-left article-details"><img class="article-author-img rounded-circle" src="{{ asset('in
40                      <span class="pl-5 article-details float-right"> 6 days ago</span>
41                  </div>
42              </a>
43          </div>
44
45          <div class="article-container my-1">
46              <a href="#">
47                  <img class="article-img" src="{{ asset('images/lightspeed.png') }}">
48                  <div class="article-title d-inline-block pl-3 align-middle">
49                      <span>Light Speed Travel: <br> Fountain of Youth or Fallacy</span>
50                      <br>
51                      <span class="align-left article-details"><img class="article-author-img rounded-circle" src="{{ asset('in
52                      <span class="pl-5 article-details float-right"> 2 weeks ago</span>
53                  </div>
54              </a>
55          </div>
56
57      </div>
58
59      <!-- Right bar ad space -->
60
61
62      <div class="col-sm-12 col-md-4 text-center">
63          <div class="ad-space mx-auto mt-1 pb-2 pt-2">
64              <img class="advertisement-img" src="{{ asset('images/space-ice.png') }}">
65              <p><span class="advertisement-text">New:</span> Space Ice Cream!</p>
66              <button class="btn btn-info">Buy Now!</button>
67          </div>
68
69          <div class="quote-space pb-2 pt-2 px-5">
70              <h3 class="text-center pb-3">Trending Quotes</h3>
71              <p><i class="fa fa-comment"></i> "Our two greatest problems are gravity and paperwork. We can lick grav
72
73              <p class="pt-4"><i class="fa fa-comment"></i> "Let's face it, space is a risky business. I always considered
74
75              <p class="pt-4"><i class="fa fa-comment"></i> "If offered a seat on a rocket ship, don't ask what seat. Just
```

```
76              </div>
77          </div>
78      </div>
79  </div>
80  {% endblock %}
```

It's nothing special: just a bunch of hardcoded information and fascinating space articles. It *does* make for a pretty cool-looking homepage. And yea, we'll make this all dynamic once we have a database.

Generating a URL with a {wildcard}

One of the hardcoded articles is the one we've been playing with: Why Asteroids Taste like Bacon! The link doesn't go anywhere yet, so let's fix that by generating a URL to our article show page!

Step 1: now that we want to link to this route, give it a name: `article_show`:

36 lines | src/Controller/ArticleController.php

```
... lines 1 - 8
class ArticleController extends AbstractController
{
... lines 11 - 18
    /**
     * @Route("/news/{slug}", name="article_show")
     */
    public function show($slug)
    {
... lines 24 - 33
    }
}
```

Step 2: inside `homepage.html.twig`, find the article... and... for the `href`, use `{{ path('article_show') }}`:

81 lines | templates/article/homepage.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">

            <!-- Article List -->

            <div class="col-sm-12 col-md-8">
... lines 10 - 18
                <!-- Supporting Articles -->

                <div class="article-container my-1">
                    <a href="{{ path('article_show') }}">
... lines 23 - 29
                    </a>
                </div>
... lines 32 - 56
            </div>
... lines 58 - 77
        </div>
    </div>
{% endblock %}
```

That should work... right? Refresh! No! It's a huge, horrible, error!

> Some mandatory parameters are missing - `{slug}` - to generate a URL for `article_show`.

That *totally* makes sense! This route has a wildcard... so we can't just generate a URL to it. Nope, we need to *also* tell Symfony what *value* it should use for the `{slug}` part.

How? Add a second argument to `path()`: `{}`. That's the syntax for an associative array when you're inside Twig - it's similar to JavaScript. Give this a `slug` key set to `why-asteroids-taste-like-bacon`:

templates/article/homepage.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">

            <!-- Article List -->

            <div class="col-sm-12 col-md-8">
... lines 10 - 18
                <!-- Supporting Articles -->

                <div class="article-container my-1">
                    <a href="{{ path('article_show', {slug: 'why-asteroids-taste-like-bacon'}) }}">
... lines 23 - 29
                    </a>
                </div>
... lines 32 - 56
            </div>
... lines 58 - 77
        </div>
    </div>
{% endblock %}
```

Try it - refresh! Error gone! And check this out: the link goes to our show page.

Next, let's add some JavaScript and an API endpoint to bring this little heart icon to life!

Chapter 11: JavaScript & Page-Specific Assets

The topic of API's is... ah ... a *huge* topic and *hugely* important these days. We're going to dive deep into API's in a future tutorial. But... I think we at *least* need to get to the basics right now.

So here's the goal: see this heart icon? I want the user to be able to click it to "like" the article. We're going to write some JavaScript that sends an AJAX request to an API endpoint. That endpoint will *return* the *new* number of likes, and we'll update the page. Well, the number of "likes" is just a fake number for now, but we can still get this entire system setup and working.

Creating the new JavaScript File

Oh, and by the way, if you look at the bottom of `base.html.twig`, our page *does* have jQuery, so we can use that:

69 lines | templates/base.html.twig

```
<!doctype html>
<html lang="en">
... lines 3 - 15
    <body>
... lines 17 - 58
        {% block javascripts %}
            <script src="https://code.jquery.com/jquery-3.2.1.min.js" integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQui
... lines 61 - 65
        {% endblock %}
    </body>
</html>
```

In the `public/` directory, create a new `js/` directory and a file inside called, how about, `article_show.js`. The idea is that we'll include this *only* on the article show page.

Start with a jQuery `$(document).ready()` block:

11 lines | public/js/article_show.js

```
$(document).ready(function() {
... lines 2 - 9
});
```

Now, open `show.html.twig` and, scroll down a little. Ah! Here is the hardcoded number and heart link:

101 lines | templates/article/show.html.twig

```
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
                <div class="row">
                    <div class="col-sm-12">
... line 13
                        <div class="show-article-title-container d-inline-block pl-3 align-middle">
... lines 15 - 18
                            <span class="pl-2 article-details"> 5 <a href="#" class="fa fa-heart-o like-article"></a> </span>
                        </div>
                    </div>
                </div>
... lines 23 - 94
            </div>
        </div>
    </div>
</div>

{% endblock %}
```

Yep, we'll start the AJAX request when this link is clicked and update the "5" with the new number.

To set this up, let's make few changes. On the link, add a new class `js-like-article`. And to target the 5, add a span around it with `js-like-article-count`:

107 lines | templates/article/show.html.twig

```
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
                <div class="row">
                    <div class="col-sm-12">
... line 13
                        <div class="show-article-title-container d-inline-block pl-3 align-middle">
... lines 15 - 18
                            <span class="pl-2 article-details">
                                <span class="js-like-article-count">5</span>
                                <a href="#" class="fa fa-heart-o like-article js-like-article"></a>
                            </span>
                        </div>
                    </div>
                </div>
... lines 26 - 97
            </div>
        </div>
    </div>
</div>

{% endblock %}
... lines 104 - 107
```

We can use those to find the elements in JavaScript.

Copy the link's class. Let's write some very straightforward... but still awesome... JavaScript: find that element and, on click, call this function. Start with the classic `e.preventDefault()` so that the browser doesn't follow the link:

```
11 lines | public/js/article_show.js
$(document).ready(function() {
    $('.js-like-article').on('click', function(e) {
        e.preventDefault();
... lines 4 - 8
    });
});
```

Next, set a `$link` variable to `$(e.currentTarget)`:

```
11 lines | public/js/article_show.js
$(document).ready(function() {
    $('.js-like-article').on('click', function(e) {
        e.preventDefault();

        var $link = $(e.currentTarget);
... lines 6 - 8
    });
});
```

This is the link that was just clicked. I want to toggle that heart icon between being empty and full: do that with `$link.toggleClass('fa-heart-o').toggleClass('fa-heart')`:

```
11 lines | public/js/article_show.js
$(document).ready(function() {
    $('.js-like-article').on('click', function(e) {
        e.preventDefault();

        var $link = $(e.currentTarget);
        $link.toggleClass('fa-heart-o').toggleClass('fa-heart');
... lines 7 - 8
    });
});
```

To update the count value, go copy the other class: `js-like-article-count`. Find it and set its HTML, for now, to `TEST`:

```
11 lines | public/js/article_show.js
$(document).ready(function() {
    $('.js-like-article').on('click', function(e) {
        e.preventDefault();

        var $link = $(e.currentTarget);
        $link.toggleClass('fa-heart-o').toggleClass('fa-heart');

        $('.js-like-article-count').html('TEST');
    });
});
```

Including Page-Specific JavaScript

Simple enough! All we need to do *now* is include this JS file on our page. Of course, in `base.html.twig`, we *could* add the script tag right at the bottom with the others:

69 lines | templates/base.html.twig

```
<!doctype html>
<html lang="en">
... lines 3 - 15
    <body>
... lines 17 - 58
        {% block javascripts %}
            <script src="https://code.jquery.com/jquery-3.2.1.min.js" integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQui
            <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.3/umd/popper.min.js" integrity="sha384-vFJXuSJp
            <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/js/bootstrap.min.js" integrity="sha384-alpBp
            <script>
                $('.dropdown-toggle').dropdown();
            </script>
        {% endblock %}
    </body>
</html>
```

But... we don't really want to include this JavaScript file on *every* page, we only need it on the article *show* page.

But how can we do that? If we add it to the `body` block, then on the final page, it will appear too early - *before* even jQuery is included!

To add our new file at the bottom, we can *override* the `javascripts` block. Anywhere in `show.html.twig`, add `{% block javascripts %}` and `{% endblock %}`:

107 lines | templates/article/show.html.twig

```
... lines 1 - 104
{% block javascripts %}
... line 106
{% endblock %}
```

Add the script tag with `src=""`, start typing `article_show`, and auto-complete!

107 lines | templates/article/show.html.twig

```
... lines 1 - 104
{% block javascripts %}
    <script src="{{ asset('js/article_show.js') }}"></script>
{% endblock %}
```

There *is* still a problem with this... and you might already see it. Refresh the page. Click and... it doesn't work!

Check out the console. Woh!

> $ is not defined

That's not good! Check out the HTML source and scroll down towards the bottom. Yep, there is literally only *one* script tag on the page. That makes sense! When you override a block, you *completely* override that block! All the script tags from `base.html.twig` are gone!

Whoops! What we *really* want to do is *append* to the block, not *replace* it. How can we do that? Say `{{ parent() }}`:

109 lines | templates/article/show.html.twig

```
... lines 1 - 104
{% block javascripts %}
    {{ parent() }}

    <script src="{{ asset('js/article_show.js') }}"></script>
{% endblock %}
```

This will print the *parent* template's block content first, and *then* we add our stuff. *This* is why we put CSS in a `stylesheets` block and JavaScript in a `javascripts` block.

Try it now! Refresh! And... it works! Next, let's create our API endpoint and hook this all together.

Chapter 12: JSON API Endpoint

When we click the heart icon, we need to send an AJAX request to the server that will, eventually, update something in a database to show that the we liked this article. That API endpoint also needs to return the new number of hearts to show on the page... ya know... in case 10 other people liked it since we opened the page.

In `ArticleController`, make a new `public function toggleArticleHeart()`:

47 lines | src/Controller/ArticleController.php

```
... lines 1 - 9
class ArticleController extends AbstractController
{
... lines 12 - 39
    public function toggleArticleHeart($slug)
    {
... lines 42 - 44
    }
}
```

Then add the route above: `@Route("/news/{slug}")` - to match the show URL - then `/heart`. Give it a name immediately: `article_toggle_heart`:

47 lines | src/Controller/ArticleController.php

```
... lines 1 - 9
class ArticleController extends AbstractController
{
... lines 12 - 36
    /**
     * @Route("/news/{slug}/heart", name="article_toggle_heart")
     */
    public function toggleArticleHeart($slug)
    {
... lines 42 - 44
    }
}
```

I included the `{slug}` wildcard in the route so that we know *which* article is being liked. We could also use an `{id}` wildcard once we have a database.

Add the corresponding `$slug` argument. But since we *don't* have a database yet, I'll add a TODO: "actually heart/unheart the article!":

47 lines | src/Controller/ArticleController.php

```
... lines 1 - 9
class ArticleController extends AbstractController
{
... lines 12 - 36
    /**
     * @Route("/news/{slug}/heart", name="article_toggle_heart")
     */
    public function toggleArticleHeart($slug)
    {
        // TODO - actually heart/unheart the article!
... lines 43 - 44
    }
}
```

Returning JSON

We want this API endpoint to return JSON... and remember: the *only* rule for a Symfony controller is that it must return a Symfony Response object. So we could literally say `return new Response(json_encode(['hearts' => 5]))`.

But that's too much work! Instead say `return new JsonResponse(['hearts' => rand(5, 100)]`:

47 lines | src/Controller/ArticleController.php

```
... lines 1 - 6
use Symfony\Component\HttpFoundation\JsonResponse;
... lines 8 - 9
class ArticleController extends AbstractController
{
... lines 12 - 36
    /**
     * @Route("/news/{slug}/heart", name="article_toggle_heart")
     */
    public function toggleArticleHeart($slug)
    {
        // TODO - actually heart/unheart the article!

        return new JsonResponse(['hearts' => rand(5, 100)]);
    }
}
```

Tip

Or use the controller shortcut!

```
return $this->json(['hearts' => rand(5, 100)]);
```

There's nothing special here: `JsonResponse` is a *sub-class* of `Response`. It calls `json_encode()` *for* you, and also sets the `Content-Type` header to `application/json`, which helps your JavaScript understand things.

Let's try this in the browser first. Go back and add `/heart` to the URL. Yes! Our *first* API endpoint!

Tip

My JSON looks pretty thanks to the JSONView extension for Chrome!

Making the Route POST-Only

Eventually, this endpoint will *modify* something on the server - it will "like" the article. So as a best-practice, we should *not* be able to make a GET request to it. Let's make this route *only* match when a POST request is made. How? Add another option to the route: `methods={"POST"}`:

47 lines | src/Controller/ArticleController.php

```
... lines 1 - 9
class ArticleController extends AbstractController
{
... lines 12 - 36
    /**
     * @Route("/news/{slug}/heart", name="article_toggle_heart", methods={"POST"})
     */
    public function toggleArticleHeart($slug)
    {
... lines 42 - 44
    }
}
```

As *soon* as we do that, we can no longer make a GET request in the browser: it does *not* match the route anymore! Run:

```
$ ./bin/console debug:router
```

And you'll see that the new route only responds to POST requests. Pretty cool. By the way, Symfony has a *lot* more tools for creating API endpoints - this is *just* the beginning. In future tutorials, we'll go further!

Hooking up the JavaScript & API

Our API endpoint is ready! Copy the route name and go back to `article_show.js`. But wait... if we want to make an AJAX request to the new route... how can we generate the URL? This is a pure JS file... so we can't use the Twig `path()` function!

Actually, there *is* a really cool bundle called FOSJsRoutingBundle that *does* allow you to generate routes in JavaScript. But, I'm going to show you another, simple way.

Back in the template, find the heart section. Let's just... fill in the `href` on the link! Add `path()`, paste the route name, and pass the `slug` wildcard set to a `slug` variable:

109 lines | templates/article/show.html.twig

```
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
                <div class="row">
                    <div class="col-sm-12">
... line 13
                        <div class="show-article-title-container d-inline-block pl-3 align-middle">
... lines 15 - 18
                            <span class="pl-2 article-details">
... line 20
                                <a href="{{ path('article_toggle_heart', {slug: slug}) }}" class="fa fa-heart-o like-article js-like-articl
                            </span>
                        </div>
                    </div>
                </div>
... lines 26 - 97
            </div>
        </div>
    </div>
</div>

{% endblock %}
... lines 104 - 109
```

Actually... there is *not* a `slug` variable in this template yet. If you look at `ArticleController`, we're only passing two variables. Add a third: `slug` set to `$slug`:

48 lines | src/Controller/ArticleController.php

```
... lines 1 - 9
class ArticleController extends AbstractController
{
... lines 12 - 22
    public function show($slug)
    {
... lines 25 - 30
        return $this->render('article/show.html.twig', [
... line 32
            'slug' => $slug,
... line 34
        ]);
    }
... lines 37 - 46
}
```

That *should* at least set the URL on the link. Go back to the show page in your browser and refresh. Yep! The heart link *is* hooked up.

Why did we do this? Because now we can get that URL *really* easily in JavaScript. Add `$.ajax({})` and pass `method: 'POST'` and `url` set to `$link.attr('href')`:

16 lines | public/js/article_show.js

```
$(document).ready(function() {
    $('.js-like-article').on('click', function(e) {
... lines 3 - 5
        $link.toggleClass('fa-heart-o').toggleClass('fa-heart');

        $.ajax({
            method: 'POST',
            url: $link.attr('href')
... lines 11 - 12
        })
    });
});
```

That's it! At the end, add `.done()` with a callback that has a `data` argument:

16 lines | public/js/article_show.js

```
$(document).ready(function() {
    $('.js-like-article').on('click', function(e) {
... lines 3 - 7
        $.ajax({
            method: 'POST',
            url: $link.attr('href')
        }).done(function(data) {
... line 12
        })
    });
});
```

The `data` will be whatever our API endpoint sends back. That means that we can move the article count HTML line into this, and set it to `data.heart`:

16 lines | public/js/article_show.js

```
$(document).ready(function() {
    $('.js-like-article').on('click', function(e) {
... lines 3 - 7
        $.ajax({
... lines 9 - 10
        }).done(function(data) {
            $('.js-like-article-count').html(data.hearts);
        })
    });
});
```

Oh, and if you're not familiar with the `.done()` function or Promises, I'd highly recommend checking out our JavaScript Track. It's not beginner stuff: it's meant to take your JS up to the next level.

Anyways... let's try it already! Refresh! And... click! It works!

And... I have a surprise! See this little arrow icon in the web debug toolbar? This showed up as soon as we made the first AJAX request. Actually, every time we make an AJAX request, it's added to the top of this list! That's awesome because - remember the profiler? - you can click to view the profiler for *any* AJAX request. Yep, you now have all the performance and debugging tools at your fingertips... even for AJAX calls.

Oh, and if there *were* an error, you would see it in all its beautiful, styled glory on the Exception tab. Being able to load the profiler for an AJAX call is kind of an easter egg: not everyone knows about it. But you *should*.

I think it's time to talk about the most important part of Symfony: Fabien. I mean, services.

Chapter 13: Services

It's time to talk about the most fundamental part of Symfony: services!

Honestly, Symfony is nothing more than a bunch of useful objects that work together. For example, there's a router object that matches routes and generates URLs. There's a Twig object that renders templates. And there's a Logger object that Symfony is already using internally to store things in a `var/log/dev.log` file.

Actually, *everything* in Symfony - I mean *everything* - is done by one of these useful objects. And these useful objects have a special name: *services*.

What's a Service?

But don't get too excited about that word - service. It's a special word for a *really* simple idea: a service is any object that does *work*, like generating URLs, sending emails or saving things to a database.

Symfony comes with a *huge* number of services, and I want you to think of services as your *tools*.

Like, if I gave you the logger service, or object, then you could use it to log messages. If I gave you a mailer service, you could send some emails! Tools!

The *entire* second half of Symfony is all about learning where to find these services and how to use them. Every time you learn about a new service, you get a new tool, and become just a *little* bit more dangerous!

Using the Logger Service

Let's check out the logging system. Find your terminal and run:

```
$ tail -f var/log/dev.log
```

I'll clear the screen. Now, refresh the page, and move back. Awesome! This *proves* that Symfony has some sort of logging system. And since *everything* is done by a service, there must be a logger object. So here's the question: how can *we* get the logger service so that *we* can log our *own* messages?

Here's the answer: inside the controller, on the method, add an additional argument. Give it a `LoggerInterface` type hint - hit tab to auto-complete that and call it whatever you want, how about `$logger`:

51 lines | src/Controller/ArticleController.php

```
... lines 1 - 4
use Psr\Log\LoggerInterface;
... lines 6 - 10
class ArticleController extends AbstractController
{
... lines 13 - 41
    public function toggleArticleHeart($slug, LoggerInterface $logger)
    {
... lines 44 - 48
    }
}
```

Remember: when you autocomplete, PhpStorm adds the `use` statement to the top for you.

Now, we can use one of its methods: `$logger->info('Article is being hearted')`:

51 lines | src/Controller/ArticleController.php

```
... lines 1 - 10
class ArticleController extends AbstractController
{
... lines 13 - 41
    public function toggleArticleHeart($slug, LoggerInterface $logger)
    {
        // TODO - actually heart/unheart the article!

        $logger->info('Article is being hearted!');
... lines 47 - 48
    }
}
```

Before we talk about this, let's try it! Find your browser and click the heart. That hit the AJAX endpoint. Go back to the terminal. Yes! There it is at the bottom. Hit `Ctrl` + `C` to exit `tail`.

Service Autowiring

Ok cool! But... how the heck did that work? Here's the deal: before Symfony executes our controller, it looks at each argument. For simple arguments like `$slug`, it passes us the wildcard value from the router:

51 lines | src/Controller/ArticleController.php

```
... lines 1 - 10
class ArticleController extends AbstractController
{
... lines 13 - 38
    /**
     * @Route("/news/{slug}/heart", name="article_toggle_heart", methods={"POST"})
     */
    public function toggleArticleHeart($slug, LoggerInterface $logger)
    {
... lines 44 - 48
    }
}
```

But for `$logger`, it looks at the *type-hint* and *realizes* that we *want* Symfony to pass us the logger object. Oh, and the order of the arguments does *not* matter.

This is a *very* powerful idea called autowiring: if you need a service object, you just need to know the correct *type-hint* to use! So... how the heck did I know to use `LoggerInterface`? Well, of course, if you look at the official Symfony docs about the logger, it'll tell you. But, there's a *cooler* way.

Go to your terminal and run:

```
$ ./bin/console debug:autowiring
```

Boom! This is a full list of *all* of the type-hints that you can use to get a service. Notice that most of them say that they are an *alias* to something. Don't worry about that too much: like routes, each service has an internal name you can use to reference it. We'll learn more about that later. Oh, and whenever you install a *new* package, you'll get more and more services in this list. More tools!

Using Twig Directly

And check this out! If you want to get the Twig service, you can use either of these two type-hints.

And remember how I said that *everything* in Symfony is done by a service? Well, when we call `$this->render()` in a controller, that's just a shortcut to fetch the Twig service and call a method on it:

51 lines | src/Controller/ArticleController.php

```
... lines 1 - 10
class ArticleController extends AbstractController
{
... lines 13 - 23
    public function show($slug)
    {
... lines 26 - 31
        return $this->render('article/show.html.twig', [
... lines 33 - 35
        ]);
    }
... lines 38 - 49
}
```

In fact, let's pretend that the `$this->render()` shortcut does *not* exist. How could we render a template? No problem: we just need the Twig service. Add a second argument with an `Environment` type-hint, because that's the class name we saw in `debug:autowiring`. Call the arg `$twigEnvironment`:

54 lines | src/Controller/ArticleController.php

```
... lines 1 - 9
use Twig\Environment;

class ArticleController extends AbstractController
{
... lines 14 - 24
    public function show($slug, Environment $twigEnvironment)
    {
... lines 27 - 39
    }
... lines 41 - 52
}
```

Next, change the `return` statement to be `$html = $twigEnvironment->render()`:

54 lines | src/Controller/ArticleController.php

```
... lines 1 - 9
use Twig\Environment;

class ArticleController extends AbstractController
{
... lines 14 - 24
    public function show($slug, Environment $twigEnvironment)
    {
... lines 27 - 32
        $html = $twigEnvironment->render('article/show.html.twig', [
            'title' => ucwords(str_replace('-', ' ', $slug)),
            'slug' => $slug,
            'comments' => $comments,
        ]);
... lines 38 - 39
    }
... lines 41 - 52
}
```

The method we want to call on the Twig object is coincidentally the same as the controller shortcut.

Then at the bottom, return `new Response()` and pass `$html`:

54 lines | src/Controller/ArticleController.php

```
... lines 1 - 8
use Symfony\Component\HttpFoundation\Response;
use Twig\Environment;

class ArticleController extends AbstractController
{
... lines 14 - 24
    public function show($slug, Environment $twigEnvironment)
    {
... lines 27 - 32
        $html = $twigEnvironment->render('article/show.html.twig', [
            'title' => ucwords(str_replace('-', ' ', $slug)),
            'slug' => $slug,
            'comments' => $comments,
        ]);

        return new Response($html);
    }
... lines 41 - 52
}
```

Ok, this is *way* more work than before... and I would *not* do this in a real project. But, I wanted to prove a point: when you use the `$this->render()` shortcut method on the controller, all it *really* does is call `render()` on the Twig service and then wrap it inside a `Response` object for you.

Try it! Go back and refresh the page. It works exactly like before! Of course we *will* use shortcut methods, because they make our life *way* more awesome. I'll change my code back to look like it did before. But the point is this: *everything* is done by a service. If you learn to master services, you can do *anything* from *anywhere* in Symfony.

There's a lot more to say about the topic of services, and *so* many other parts of Symfony: configuration, Doctrine & the database, forms, Security and APIs, to just name a few. The Space Bar is far from being the galactic information source that we know it will be!

But, congrats! You just spent an hour getting an *awesome* foundation in Symfony. You will *not* regret your hard work: you're on your way to building *great* things and, as always, becoming a better and better developer.

Alright guys, seeya next time!