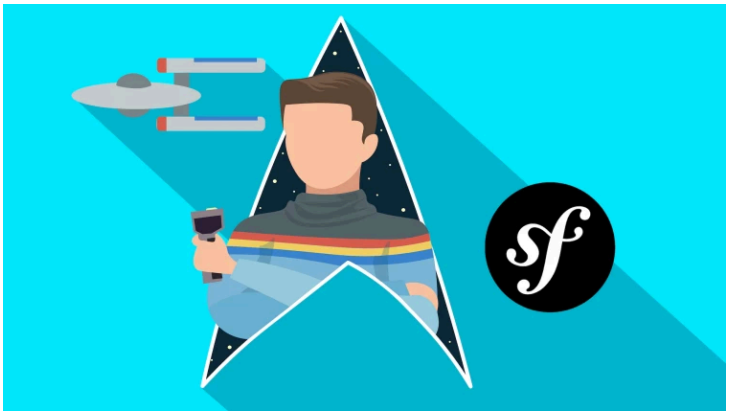
Cosmic Coding with Symfony 7



Chapter 1: Setting up our Symfony App

Welcome to the first Symfony 7 tutorial! My name is Ryan - I live here in the fantasy world of Symfonycasts and... I am *beyond* excited to be *your* guide through this series all about Symfony, web development... bad jokes... space animations, and most importantly, building *real* things we can be proud of. For me, it feels like *I'm* the lucky person that gets to give you a personal tour of the Enterprise... or whatever nerdy thing gets *you* most excited.

And that's because, I *love* this stuff. Bootstrapping databases, building beautiful user interfaces, writing high-quality code... it gets me out of bed in the morning. And Symfony is *the* best tool to do all of this... and become a better developer along the way.

And that's really my goal: I want *you* to enjoy all of this as much as I do ... and to feel empowered to build all the amazing things you have floating around in your mind.

What Makes Symfony Special

Now, one of my *favorite* things about teaching Symfony is that our project is going to start *tiny*. That makes it easy to learn. But then, it'll scale up automatically as we need more tools via a unique *recipe* system. Symfony is actually a collection of over *200* small PHP libraries. So that's a *ton* of tools... but *we* get to choose what we need.

Because, you might be building a pure API... or a full web app, which is what we'll focus on in *this* tutorial. Though, if you *are* building an API, follow the first few tutorials in this series, then pop over to our API Platform tutorials. API Platform is a mind-blowingly fun & powerful system for making APIs, built right on top of Symfony.

Symfony is also *blazingly* fast, has long-term support versions and works a lot on creating a delightful developer experience *while* keeping to programming best-practices. This means we get to write high-quality code and *still* get our work done quickly.

Ok, enough of me gushing about Symfony. Ready to get to work? Then beam aboard.

Installing the Symfony Binary

And head over to https://symfony.com/download. This page has instructions on how to download a standalone binary called `symfony`. Now this is *not* Symfony itself... it's just a little tool that'll help us do things, like start new Symfony projects, run a local web server or even deploy our app to production.

Once you've downloaded and installed it, open a terminal and move into *any* directory. Check that the `symfony` binary is ready to go by running:

```
symfony --help
```

It's got a *bunch* of commands, but we'll just need a few. Before we start a project, also run

```
symfony check:req
```

which stands for check requirements. This makes sure that we have everything on our system needed to run Symfony, like PHP at the correct version and some PHP extensions.

Once this is happy, we can start a new project! Do it with `symfony new` and then a directory name. I'll call mine `starshop`. More on that later.

```
symfony new starshop
```

This will give us a *tiny* project with only the *base* things installed. Then, we'll add more stuff little-by-little along the way. It's gonna be great! But later, when you feel comfortable with Symfony, if you want to get started more quickly, you can run the same command, but with `--webapp` to get a project with *much* more stuff pre-installed.

Anyway, move into the directory - `cd starshop` - then I'll type `ls` to check things out. Cool! We'll get to know these files in the next chapter, but this is *our* project... and it's already working!

Starting the symfony Web Server

To *see* it working in a browser, we need to start a web server. You can use *any* web server you want - Apache, Nginx, Caddy, whatever. But for local development, I highly recommend using the `symfony` binary we just installed. Run:

```
symfony serve
```

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

And... bam! We have a new web server for our project running at https://127.0.0.1:8000. Copy that, spin over to your most favorite browser, paste and... welcome to Symfony 7! That's what I was going to say!

Next, let's sit down, order some Earl Grey tea, and become friends with every file in our new app... which isn't very many.

Chapter 2: Getting to Know our Tiny Project

Sprint back to your command center (aka terminal). This first tab is running the web server. If you need to stop it, press Ctrl-C... then restart it with:

```
symfony serve
```

> **Tip**
>
> You can use `symfony serve -d` to run the command in the "background" so that you can continue using this terminal tab.

We'll leave that alone and let it do its thing.

Our Project's 15 Files

Open a second terminal tab in the same directory. When we ran the `symfony new` command, it downloaded a tiny project *and* initialized a Git repository with an initial commit. That was super nice! To see our files, I'm going to open this directory in my favorite editor: PhpStorm. More on this editor in a few minutes.

Right now, I want you to notice just how *small* our project is! To see the full list of committed files, back at your terminal, run:

```
git ls-files
```

Yea, that's it. Only about 15 files committed to git!

Where's Symfony?

So then... where the heck is Symfony? One of our 15 files is especially important: `composer.json`.

composer.json

```
{
// ... lines 2 - 5
    "require": {
        "php": ">=8.2",
        "ext-ctype": "*",
        "ext-iconv": "*",
        "symfony/console": "7.0.*",
        "symfony/dotenv": "7.0.*",
        "symfony/flex": "^2",
        "symfony/framework-bundle": "7.0.*",
        "symfony/runtime": "7.0.*",
        "symfony/yaml": "7.0.*"
    },
// ... lines 17 - 70
}
```

Composer is the package manager for PHP. Its job is simple: read the package names under this `require` key and download them. When we ran the `symfony new` command, it downloaded these 15 files and *also* ran `composer install`. That downloaded all of these packages into the `vendor/` directory.

So where is Symfony? It's in `vendor/symfony/`... and we're already using about 20 of its packages!

Running Composer

The `vendor/` directory is *not* committed to git. It's ignored thanks to another file we started with: `.gitignore`.

.gitignore

```
###> symfony/framework-bundle ###
/.env.local
/.env.local.php
/.env.*.local
/config/secrets/prod/prod.decrypt.private.php
/public/bundles/
/var/
/vendor/
###< symfony/framework-bundle ###
```

This means that if a teammate clones our project, they will *not* have this directory. And that's okay! We can always repopulate it by running `composer install`.

Watch: I'll right-click and delete the entire `vendor/` directory. Gasp!

If we try our app now, it's busted. Bad feels! To fix it & save the day, at your terminal, run:

```
composer install
```

And... presto! The directory is back.... and over here, the site works again.

The 2 Directories you Care About

Looking back at our files, there are only two directories that we even need to think about. The first is `config/`: this holds... configuration! We'll learn about what these files do along the way.

The second is `src/`. This is where *all* your PHP code will live.

And that's really it! 99% of the time you're either configuring something or writing PHP code. That happens in `config/` & `src/`.

What about the other 4 directories? `bin/` holds a single `console` executable file that we'll try out soon. But we're never going to look at or modify that file. The `public/` directory is known as your document root. Anything you put here - like an image - will be publicly accessible. More about that stuff later. It also holds `index.php`.

public/index.php

```php
<?php

use App\Kernel;

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

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

This is known as your "front controller": it's the main PHP file that your web server executes at the start of every request. And while it *is* super important... you'll never edit or even think about this file.

Up next is `var/`. This is *also* ignored from git: it's where Symfony stores log files and cache files that it needs internally. So very important... but not something we need to think about. And we already talked about `vendor/`. That's everything!

Prepping PhpStorm

Now before we get coding, I mentioned that I use PhpStorm. You're free to use whatever editor you want. However, PhpStorm is *incredible*. And one big reason is the unmatched Symfony *plugin*. If you go to PhpStorm -> Settings and search for "Symfony", down here under Plugins and then Marketplace, you can find it. Download & install the plugin if you don't already have it. *After* installation, restart PhpStorm. Then there's one more step. Go back into settings and search for Symfony again. This time you'll have a Symfony section. Be sure to enable the plugin for each Symfony project you work on... otherwise you won't see all the same magic I have.

Ok! Let's start coding and build our first page in Symfony next.

Chapter 3: Routes, Controllers & Responses

Ok, here's the scoop. Wesley Crusher - everyone's favorite ensign from Star Trek - has retired from Starfleet and is working with *us* to start a new business: Wesley's Star Shop. Someone's gotta break the Ferengi monopoly on the galaxy's starship repair business, and he's hired *us* to build the site to do that. We're about to give the Ferengi a run for their latinum!

Creating the Controller

And it starts with building our first page. The idea behind every page is always the same. Step one, give it a cool URL. That's called the route. Step two, write a PHP function that *generates* the page. That's known as the controller. And that page could be HTML, JSON, ASCII art, anything.

In Symfony, the controller is always a method inside a PHP class. So, we need to create our first PHP code! Where does PHP code live in our app? That's right, the `src/` directory.

Inside this `src/Controller/` directory, create a new file. I would normally select new "PHP class", but for this first time, create an empty file. We'll do each part by hand. Call it `MainController.php`, but you can name this whatever you want.

Inside, add the open PHP tag, and then say `class MainController`. Above this, add a namespace of `App\Controller`.

src/Controller/MainController.php

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

class MainController
{
}
```

Namespaces & Directories

Okay, a few things about this. First, the fact that I put this class inside a directory called `Controller` is optional. That's just a convention. You could rename this to whatever the Klingon word for `Controller` is and everything would the same... and probably be more interesting!

However, there *are* a few rules about PHP classes in general. The first is that every class *must* have a namespace *and* that namespace needs to match your directory structure. It's always going to be `App\` then whatever directory you're inside. Without going into too much detail, that's a rule you'll find in *every* PHP project.

The second rule is that your class name must match your filename `.php`. If you mess either of these up, you'll get an error from PHP that it can't find your class. The Ferengi never make this mistake.

Creating the Controller Method & Route

Anyway, our goal is to create a *controller*, which is a *method* in a class that builds the page. Add a new public function and call it `homepage`. But, again, the name doesn't matter. And... yea! It's not done yet, but *this* is our controller!

src/Controller/MainController.php

```
// ... lines 1 - 4
class MainController
{
    public function homepage()
    {

    }
}
```

But remember, a page is the combination of a controller and a *route*, which defines the page's URL. Where do we put the route? Right *above* the controller method using a feature of PHP called an *attribute*. Write `#[]` then start typing `Route` with a capital `R`. Check out that auto-completion!

Either option will work, but use the one from `Attribute` - which is newer - then hit tab. When I did that, something super important happened: my editor added a `use` statement at the top of the class. Anytime you use a PHP attribute, you *must* have a corresponding `use` statement for it in the same file.

These attributes work almost like PHP functions: you can pass a bunch of arguments. The first one is the path. Set this to `/`.

src/Controller/MainController.php

```
// ... lines 1 - 4
use Symfony\Component\Routing\Attribute\Route;

class MainController
{
    #[Route('/')]
    public function homepage()
    {

    }
}
```

Thanks to this, when someone goes to the homepage - `/` - Symfony will call this controller method to build the page!

Controllers & Responses

What... should our method return? Just the HTML we want, right? Or the JSON if we're building an API?

Almost. The web works on a well-known system. First, a user *requests* a page. They say:

> *"Hey, I want to see `/products`... or I want to see `/users.json`."*

What we return back to them, yes, contains HTML or JSON. But it's more than that. We also communicate back a status code - which says whether the response was okay or had an error - as well as these things called headers, which communicate a bit more info, like the format of what we're returning.

This whole beautiful package is called the *response*. So yeah, most of the time, we'll just be thinking about returning HTML or JSON. But what we're *truly* sending is this bigger, nerdier thing called a *response*.

And so our entire job as web developers - no matter *what* language we're programming in - is to understand the request from the user, then create and return the response.

And this brings us back to something I *love* about Symfony. What does our controller return? A new `Response` object from Symfony! And again, PhpStorm wants to auto-complete this, suggesting a few different `Response` classes. We want the one from the Symfony `HttpFoundation` component. That's the Symfony library that contains everything related to requests & responses.

Hit tab. Once again, when we did that, PhpStorm added a `use` statement at the top of the file. I'm going to use this trick *constantly*. Anytime you reference a class name, you *must* have a corresponding `use` statement, else PHP will give you an error that it can't find the `Response` class.

Inside this, the first argument is the content that we want to return. Start with a hardcoded string.

src/Controller/MainController.php

```
// ... lines 1 - 4
use Symfony\Component\HttpFoundation\Response;
// ... lines 6 - 7
class MainController
{
    #[Route('/')]
    public function homepage()
    {
        return new Response('<strong>Starshop</strong>: your monopoly-busting option for Starship parts!');
    }
}
```

Route, check! Controller that returns a Response, check! Let's try this. Back at the browser, this page was just a demo that shows before we have a *real* homepage. Now that we do, when we refresh... there it is!

I know it's not much yet, but we just learned the first *fundamental* part of Symfony: that every page is a route & controller... and that every controller returns a response.

Oh, and it's optional, but because our controller always returns a `Response`, we can add a `Response` return type. That doesn't change how our code works, but it makes it more descriptive to read. And if we ever did something silly and returned something *other* than a response, PHP would give us a clear reminder.

src/Controller/MainController.php

```
// ... lines 1 - 7
class MainController
{
    #[Route('/')]
    public function homepage(): Response
    {
        return new Response('<strong>Starshop</strong>: your monopoly-busting option for Starship parts!');
    }
}
```

Next up: to supercharge our development, let's install our first third-party package and learn about Symfony's amazing recipe system.

Chapter 4: Magical Flex Recipes

I have a secret. When our project was created, it wasn't 15 files. It was... *one* file. If you peeked inside the code for the `symfony new` command, you'd discover that it's a shortcut for just two things. First, it clones a repository called `symfony/skeleton`... which is just *one* file if you ignore the license. And second, it runs `composer install`.

That's it! But hold on, if that's the case, where in the world did all these other files come from? Like, the stuff in `bin/`, `config/` and `src/`? The answer starts with a special package inside our `composer.json` file called `symfony/flex`. Flex is a Composer *plugin* that adds two superpowers to Composer: aliases and recipes.

composer.json

```
{
// ... lines 2 - 5
    "require": {
// ... lines 7 - 11
        "symfony/flex": "^2",
// ... lines 13 - 15
    },
// ... lines 17 - 70
}
```

Flex Aliases

Aliases are simple. To add a *new* package to your app - which we'll do in a minute - you run `composer require` then the name of the package like `symfony/http-client`. Flex gives the most important packages in the Symfony ecosystem a *shorter* name, called an alias. For example, `symfony/http-client` has an alias called `http-client`. Yup, we could run `composer require http-client` and Flex would translate that to the final package name. It's just a shortcut when adding packages.

If you want to see all the available aliases, go to a repository called symfony/recipes... then click the link to `RECIPES.md`. On the right, there they are!

The Recipes System

The second superpower that Symfony Flex adds to Composer is *recipes*. These are fascinating. When you add a new package, it *may* have a recipe, which is basically a set of files that will be added to your project. And it turns out that *every* file that we started with - in `bin/`, `config/`, `public/` - these *all* came from the recipes of the packages that were originally installed.

For example, `symfony/framework-bundle` is the "core" package of the Symfony Framework. You can check out its recipe by going to the `symfony/recipes` repository and navigating to `symfony`, `framework-bundle`, then the latest version. Boom! Check out `config/packages/`: most of the stuff we started with came from this recipe!

Another way to see the recipes is at your command line. Run:

```
composer recipes
```

Apparently the recipes of *four* different packages were installed. And we could get info about any of these by adding its name to the end of the command.

Anyway, recipes are amazing because we can install a package and instantly get any files we need. Instead of fussing around with configuration, we get right to work.

Installing PHP CS Fixer

Let's try this out: let's add a new package called PHP-CS-Fixer that will give us an executable file to fix the *styling* of our code. For example, in `src/Controller/MainController.php`, if you follow PHP coding standards, the curly brace should live on the next line after a function. If we did something like this, our file now violates those standards. That wouldn't hurt anything, but you know, we want to keep our code looking clean. And PHP-CS-Fixer can help us do that.

To install it, run:

```
composer require cs-fixer-shim
```

And yes, this is an *alias*. On top, the true package is `php-cs-fixer/shim`.

Did this package come with a recipe? It did! The `Configuring php-cs-fixer/shim` tells us that. But, we can *also* see it by running:

```
git status
```

The fact that `composer.json` and `composer.lock` are modified is 100% normal Composer behavior. You can see that `composer.json` has the new library under the `require` key.

composer.json

```
{
// ... lines 2 - 5
    "require": {
// ... lines 7 - 9
        "php-cs-fixer/shim": "^3.46",
// ... lines 11 - 16
    },
// ... lines 18 - 69
}
```

But every *other* modified or new file *is* thanks to the package's recipe.

Investigating the Recipe

Let's investigate these! Open up `.gitignore`. Cool! At the bottom, it added two new entries for two common files that you want to ignore when you use PHP CS fixer.

.gitignore

```
// ... lines 1 - 11
###> php-cs-fixer/shim ###
/.php-cs-fixer.php
/.php-cs-fixer.cache
###< php-cs-fixer/shim ###
```

The recipe also added a new `.php-cs-fixer.dist.php` file. This is CS Fixer's configuration file. And check it out!

.php-cs-fixer.dist.php

```
<?php

$finder = (new PhpCsFixer\Finder())
    ->in(__DIR__)
    ->exclude('var')
;

return (new PhpCsFixer\Config())
    ->setRules([
        '@Symfony' => true,
    ])
    ->setFinder($finder)
;
```

It's pre-built to work for our Symfony app. It tells it to fix all files in the current directory, but ignore the `var/` directory because that's where Symfony stores its cache files. It also tells it to use a ruleset called Symfony. That means that we want our code style to match Symfony's style. The point is: instead of *us* wasting time hunting down this default config... we just get it!

The last modified file is `symfony.lock`. This keeps track of which recipes we have installed and at what version. And yes, we *are* going to commit all these files to our repository.

Using PHP-CS-Fixer

Now that we've installed the package, let's use it. Do that by running:

```
./vendor/bin/php-cs-fixer
```

That'll show all the available commands. The one we want is called fix. Try it:

```
./vendor/bin/php-cs-fixer fix
```

And... yes! It found the violation in `MainController.php`! When we go to that file... yea! It moved my curly brace from the end of the line back down to the next line. That's awesome.

Next up, let's meet and install one of my favorite libraries in all of PHP: the Twig templating engine.

Chapter 5: Twig & Templates

I want to return HTML for this page. We *could* put that HTML right inside the controller... but that's going to get ugly fast. Fortunately, there's a better way: by using a templating library called Twig.

Installing Twig

At your terminal, make sure you've committed your changes, because I want to see what this new package's recipe adds to our project. I've already done that. Install it with:

```
composer require twig
```

Composer "Packs"

You probably recognize that `twig` is an alias... this time to a package called `symfony/twig-pack`. And the word "pack" is important in Symfony. A pack is... kind of a fake package that helps install *multiple* packages at once.

Watch: open up `composer.json`. Instead of *one* new package in here called `symfony/twig-pack`, we have *three* new packages... and `twig-pack` isn't even one of them!

composer.json

```
{
// ... lines 2 - 5
    "require": {
// ... lines 7 - 15
        "symfony/twig-bundle": "7.0.*",
// ... line 17
        "twig/extra-bundle": "^2.12|^3.0",
        "twig/twig": "^2.12|^3.0"
    },
// ... lines 21 - 72
}
```

The three packages give us everything we need for a full, robust Twig setup. So when you see the word "pack", it's not a huge deal: just a shortcut to install multiple packages at once.

Symfony Bundles

Ok, let's see what the recipe did! Run:

```
git status
```

We see the usual `composer.json`, `composer.lock` and `symfony.lock`. But for the first time, we also see a modification to `config/bundles.php`. A bundle is a PHP package that integrates with Symfony... it's basically a Symfony plugin. Whenever you install a bundle, you need to activate it in this `bundles.php` file. But honestly, the recipe system will always do that *for* us... so it's a good thing to notice, but we'll never edit this file by hand.

config/bundles.php

```
<?php

return [
    Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
    Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
    Twig\Extra\TwigExtraBundle\TwigExtraBundle::class => ['all' => true],
];
```

The Twig Recipe

The second thing the recipe did was create a `config/packages/twig.yaml` file. The purpose of each file in `config/packages/` is to configure a *bundle*.

config/packages/twig.yaml

```yaml
twig:
    default_path: '%kernel.project_dir%/templates'

when@test:
    twig:
        strict_variables: true
```

For example, `twig.yaml` controls the behavior of TwigBundle. This line here tells Twig:

> *"Hey! All my template files will end in `.twig`."*

There's a lot more that we *could* configure, but we don't need to. And we'll dive deeper into these config files in the next tutorial.

The final thing the recipe did was add a `templates/` directory, which.... you guessed it! Is where our template files will live! It even started us with a `base.html.twig` file that we'll talk about in a few minutes.

Rendering a Template

So let's render our first template! To do that, make your controller extend a base class called `AbstractController`. Be sure to hit tab so that it adds the `use` statement on top. Extending this base class is optional, but it gives us a bunch of shortcut methods.

src/Controller/MainController.php

```php
// ... lines 1 - 4
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
// ... lines 6 - 8
class MainController extends AbstractController
{
// ... lines 11 - 15
}
```

For example, copy the string and then, to render a template type `return $this->render()` and pass a filename to a template. Use: `main/homepage.html.twig`.

src/Controller/MainController.php

```
// ... lines 1 - 8
class MainController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
        return $this->render('main/homepage.html.twig');
    }
}
```

Your template filename can be whatever you want, but the standard is to have a directory that matches your controller name and a filename that matches your method name.

Let's go create that! In `templates/`, add a new directory called `main`. And inside that, a file called `homepage.html.twig`. I'll paste... then add an `h1` and put it around everything.

templates/main/homepage.html.twig

```
<h1>
    Starshop: your monopoly-busting option for Starship parts!
</h1>
```

Let's do this! Refresh. Got it!

And by the way, what is our controller returning? It's *still* a `Response` object! I *know* because we have a `Response` return type... and our code isn't exploding. `render()` is just a shortcut to render this template, grab that string of HTML and put it into a `Response` object. So even though we're rendering a template, it still goes back to the idea that a controller returns a response.

Passing Data to a Template

What about passing data to the template? Maybe we query the database and pass in the total number of starships. We don't have a database in our app yet, so let's fake it by saying `$starshipCount` equals... I don't know... 457. That seems like a believable fake number.

src/Controller/MainController.php

```
// ... lines 1 - 8
class MainController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
        $starshipCount = 457;
// ... lines 15 - 18
    }
}
```

To pass variables to the template, add a second argument to `render()`: an array. Pass `numberOfStarships` set to `$starshipCount`. The *key* will become the name of the variable inside the Twig template.

src/Controller/MainController.php

```
// ... lines 1 - 8
class MainController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
        $starshipCount = 457;

        return $this->render('main/homepage.html.twig', [
            'numberOfStarships' => $starshipCount,
        ]);
    }
}
```

Rendering Variables

In the template, I'll add a div, and some text. To print the number, write `{{`, the variable name, close `}}`.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
<div>
    Browse through {{ numberOfStarships }} starships!
</div>
```

Ok! Move over and try it. Got it! And we just saw our first Twig code!

Twig is its own language, but it's super friendly. It has just three different syntaxes. The first is `{{` and I call this the "say something" syntax. If you're printing something, you'll use `{{`. Inside the curlies, we're writing Twig, which is *very* similar to JavaScript.

Twig Tags & the "do something" Syntax

For example, we could print the string `'numberOfStarships'`... or the variable `numberOfStarships`... or even `numberOfStarships` times 10.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
<div>
    Browse through {{ numberOfStarships * 10 }} starships!
</div>
```

The second syntax of the three starts with `{%`. I call this the "do something" syntax. This doesn't print anything. Instead, it's used for language constructs like `if` statements, for loops or setting a variable.

To do an if statement say `if numberOfStarships > 400`, then close this with `{% endif %}`. Inside, I'll add a comment.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
<div>
    Browse through {{ numberOfStarships * 10 }} starships!

    {% if numberOfStarships > 400 %}
        <p>
            That's a shiploads of ships!
        </p>
    {% endif %}
</div>
```

Try it out! That works too!

Twig is its own library, but it's maintained by Symfony... so its docs live at https://twig.symfony.com. Click the "Docs" link then scroll down. See the "tags"? It turns out that there are a *finite* number of things you can use with the *do* something syntax: it's these tags. Like, you can't say `{% applesauce`... it just won't work. You can only use `{%` then one of these tags. The list is pretty short... and I probably only use 5 of these on a daily basis.

The third and final syntax of Twig isn't even a syntax at all: it's for comments. `{#` to write a comment.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
<div>
    Browse through {{ numberOfStarships * 10 }} starships!

    {% if numberOfStarships > 400 %}
        <p>
            {# Do you think "shiploads" will pass the legal team? #}
            That's a shiploads of ships!
        </p>
    {% endif %}
</div>
```

Rendering an Associative Array

So we're passing a simple number to Twig and printing it. But Twig can handle whatever complex data you throw at it. For example, in the controller, create a new `$myShip` variable, set to an associative array. Then pass that into the template as a new variable: `myShip`.

src/Controller/MainController.php

```
// ... lines 1 - 8
class MainController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
        $starshipCount = 457;
        $myShip = [
            'name' => 'USS LeafyCruiser (NCC-0001)',
            'class' => 'Garden',
            'captain' => 'Jean-Luc Pickles',
            'status' => 'under construction',
        ];

        return $this->render('main/homepage.html.twig', [
            'numberOfStarships' => $starshipCount,
            'myShip' => $myShip,
        ]);
    }
}
```

In the template, add another `div`... some text and a table to print the data. In the `<td>`, we can't just print `myShip`... because printing an associative array doesn't make sense in PHP... and so it doesn't make sense in Twig. You get the famous error about array to string conversion.

What we want is to print the `name` key on that array. The way we do that looks exactly like JavaScript: `myShip.name`.

That's it! And... it works. I'll paste in the rest of our template, which prints the other keys from the array. Looking good.

templates/main/homepage.html.twig

```
// ... lines 1 - 15
<div>
    <h2>My Ship</h2>

    <table>
        <tr>
            <th>Name</th>
            <td>{{ myShip.name }}</td>
        </tr>
        <tr>
            <th>Class</th>
            <td>{{ myShip.class }}</td>
        </tr>
        <tr>
            <th>Captain</th>
            <td>{{ myShip.captain }}</td>
        </tr>
        <tr>
            <th>Status</th>
            <td>{{ myShip.status }}</td>
        </tr>
    </table>
</div>
```

Twig Functions & Filters

Twig does have a few other tricks up its sleeve, but nothing complex. It has functions... which work like functions in any language. It also has something called tests, which *are* a bit unique to Twig, but simple enough to understand. My favorite concept is probably filters, which are basically functions with a cooler, more hipster syntax.

For example, there's a filter called `upper` to send a string to uppercase. To use a filter, find the string that you want to turn into uppercase then add a `|` and `upper`.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 15
16 <div>
17     <h2>My Ship</h2>
18
19     <table>
↕ // ... lines 20 - 27
28         <tr>
29             <th>Captain</th>
30             <td>{{ myShip.captain|upper }}</td>
31         </tr>
↕ // ... lines 32 - 35
36     </table>
37 </div>
```

The value on the left gets passed through the filter, a lot like using a pipe at the command line. It works beautifully.... and you can go crazy with filters: piping to `upper`, then `lower` then to `title` case *just* to confuse your teammates.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 15
16 <div>
17     <h2>My Ship</h2>
18
19     <table>
↕ // ... lines 20 - 27
28         <tr>
29             <th>Captain</th>
30             <td>{{ myShip.captain|upper|lower|title }}</td>
31         </tr>
↕ // ... lines 32 - 35
36     </table>
37 </div>
```

Okay, we pretty much just learned all of Twig in one session except for one thing: template inheritance. That's next.

Chapter 6: Twig Template Inheritance

What about adding a layout to our page - like a header and a footer? Take a peek at the HTML for the page: it's *just* the HTML from the template. There's nothing special in Twig where a base layout with a header and a footer is automatically wrapped around our content. Whatever you have in your template is what you get on the page.

However, the Twig recipe *did* add a base layout file called `base.html.twig`.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        <link rel="icon" href="data:image/svg+xml,<svg
xmlns=%22http://www.w3.org/2000/svg%22 viewBox=%220 0 128 128%22><text
y=%221.2em%22 font-size=%2296%22>⚫</text></svg>">
        {% block stylesheets %}
        {% endblock %}

        {% block javascripts %}
        {% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

It's really simple now, but *this* is where we'll add our top nav, footer and any other things that should live on every page. The question is: how can we make *our* template use this?

Extending the Base Layout

With a *cool* feature called template inheritance. In `homepage.html.twig`, at the top, type `{% extends` then the name of the base template: `base.html.twig`. And notice: this is the *do* something tag. We're not *printing* this template, we're telling Twig that we want to *extend* it.

templates/main/homepage.html.twig

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

<h1>
    Starshop: your monopoly-busting option for Starship parts!
</h1>
// ... lines 6 - 40
```

If we do nothing else and refresh, we get an error:

> *"a template that extends another one cannot include content outside Twig blocks."*

Hmm. When you extend a template, it tells Twig that you want to render your template *inside* that base layout. But... Twig has no idea *where* our content should go. Should it take our homepage HTML and put it down here? Or up here? Or right there? It doesn't know! So it throws that error.

The way we tell it is via these *blocks*. Blocks are holes into which a child template can put content. And you may have noticed one block called `body`... which is *exactly* where we want our content to go. To put it there, surround all the content with a `{% block body %}`... and at the bottom, `{% endblock %}`.

templates/main/homepage.html.twig

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

{% block body %}
<h1>
    Starshop: your monopoly-busting option for Starship parts!
</h1>

<div>
// ... lines 9 - 16
</div>

<div>
// ... lines 20 - 39
</div>
{% endblock %}
```

And now... it's alive! It doesn't look much different, but we *are* inside the base layout.

This is called template inheritance because it works exactly like PHP class inheritance. Imagine you have a `Homepage` class that extends a `Base` class. That `Base` class has a `body()` method,

and we override that `body()` method in the `Homepage` class. It's the same concept in Twig.

Overriding the Page Title

And these block names - like `javascripts`, `stylesheets` and `body` - aren't special names... and they're not registered anywhere. Feel free to create new blocks however and whenever you want. For example, suppose we want to change the `title` of the page from a child template. In this case, the recipe already gave us a block called `title` to do that. And *this* block has default content... which is why we already see `Welcome` on the browser tab. Let's override this in *our* template.

templates/base.html.twig

```
// ... line 1
<html>
    <head>
// ... line 4
        <title>{% block title %}Welcome!{% endblock %}</title>
// ... lines 6 - 11
    </head>
// ... lines 13 - 15
</html>
```

Anywhere outside the `body` block, say `{% block title %}`, type something, then `{% endblock %}`.

templates/main/homepage.html.twig

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

{% block title %}Starshop: Beam up some parts!{% endblock %}

{% block body %}
// ... lines 6 - 42
{% endblock %}
```

Replacing vs Appending the Parent Block

And now, got it! New title! And notice that when we override a block, we override it *completely*. We don't see the word `Welcome` anymore. Occasionally, you *may* want to *add* to the parent block instead of replacing it. You can do that by saying `{{ parent() }}`.

This is really neat! The `parent()` function grabs the content from the `title` block of the parent template. Then we use `{{` to print it. This time we see welcome and *then* our title.

But since we don't really want that, I'll remove it.

Status check: we're returning HTML and we have a base layout. Yeah, our site is still horribly ugly, but we'll fix that in a bit.

Next up, let's run one command and instantly gain access to some of the most powerful debugging tools on the web.

Chapter 7: Debugging with the Amazing Profiler

Symfony boasts some of the most *epic* debugging tools of all the Internet. But because Symfony apps start so small, we don't even have them installed yet. Time to fix that. Head over to your terminal and, like before, commit all of your changes so we can check out what the recipes will do. I already did that.

Installing the Debugging Tools

Then run:

```
composer require debug
```

Yup! That's another Flex alias. *And*... it installs a *pack*. This installs four different packages that add a variety of debugging goodness to our project. Spin over and open `composer.json`.

composer.json

```
1  {
↕  // ... lines 2 - 5
6      "require": {
↕  // ... lines 7 - 14
15         "symfony/monolog-bundle": "^3.0",
↕  // ... lines 16 - 20
21     },
↕  // ... lines 22 - 78
79 }
```

Ok, the pack added one new line under the `require` key for `monolog-bundle`. Monolog is a logging library.

Then all the way at the bottom, it added three packages to a `require-dev` section.

composer.json

```
{
// ... lines 2 - 73
    "require-dev": {
        "symfony/debug-bundle": "7.0.*",
        "symfony/stopwatch": "7.0.*",
        "symfony/web-profiler-bundle": "7.0.*"
    }
}
```

These are known as *dev* dependencies... which means they won't be downloaded when you deploy to production. But otherwise, they work the same as packages under the `require` key. All three of these help power something called the *profiler*. We'll see that in *just* a minute.

Before we do, go back to your terminal and run

```
git status
```

so we can see what the recipes did. Ok: it updated the normal files, enabled a few new bundles and gave us three new configuration files *for* those bundles.

What's the end result of all this new stuff? Well, first, we now have a logging library. So, like magic, logs will start popping into a `var/log/` directory.

Hello Web Debug Toolbar & Profiler

But the *mind-blowing* moment happens when we refresh the page. Woh! A beautiful new black bar at the bottom called the web debug toolbar.

This is *bursting* with info. Over here, we can see the route and *controller* for this page. That it makes it easy to go to *any* page on your site - maybe one you didn't even build - and quickly find the code behind it. We can also see how long this page took the load, how much memory it used, and even the twig template that was rendered and how long *that* took.

But the *real* magic of the web debug toolbar happens when you click any of these links: you hop into the *profiler*. This has ten *times* more info: details about the request and response, logs that occurred while loading that page, routing details, and even stats about which Twig templates were rendered. Apparently *six* templates were rendering: our main one, the base layout and a

few others that power the web debug toolbar, which, by the way, won't be rendered or shown when we deploy to production. But we'll talk about that in the next tutorial.

Then there's probably my favorite section: Performance. This slices our entire page load time into different pieces. I love this. As you learn more about Symfony, you'll get more familiar with what these different pieces *are*. This section is useful for knowing which part of your code might be slowing down the page... but it's also a fantastic way to dive deeper into Symfony and understand all its moving pieces.

We're going to use the profiler throughout this series, but let's turn to another debugging tool: one that's been installed in our app this whole time!

Hello bin/console!

Head over to the command line and run:

```
php bin/console
```

Or, on most machines, you can just say `./bin/console`. This is Symfony's console, and it's *packed* with commands that can do all sorts of stuff! We'll learn about them along the way. You can also add your *own* commands, which we'll do at the end of the tutorial.

Notice that a bunch of these start with `debug` - like `debug:router`. Try that:

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

Cool! This shows us *every* route in our app: the homepage route at the bottom and a bunch of routes added by Symfony in the `dev` environment that power the web debug toolbar and profiler.

Another command is `debug:twig`:

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

This tells us every Twig function, filter or other thing that exists in our app. This is *like* the Twig docs... except it also includes *extra* functions and filters that are *added* to Twig by bundles that we have installed. Pretty cool.

These `debug` commands are *super* useful, and we'll keep trying more of them along the way.

Next, let's create our first API endpoint and learn about Symfony's powerful serializer component.

Chapter 8: Creating JSON API Endpoints

If you want to build an API, you can *absolutely* do that with Symfony. In fact, it's a *fantastic* option in part, because of API Platform. That's a framework for creating APIs built on top of Symfony that both makes building your API fast *and* creates an API that's more robust than you could imagine.

But, it's also simple enough to return JSON from a controller. Let's see if we can return some ship data as JSON.

Creating the new Route & Controller

This will be our *second* page. Well, it's really an "endpoint", but this will be our second route & controller combo. In `MainController`, we could add another method here. But for organization, let's create a totally new controller class. I'll go to New -> PHP Class and call it `StarshipApiController`.

Because I went to New -> PHP Class, it created the class and the namespace for me. Super nice! Also, going forward, each time I create a controller, I'll immediately extend `AbstractController`... because those shortcuts are nice and there's no downside.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 2
3  namespace App\Controller;
4
5  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
↕ // ... lines 6 - 8
9  class StarshipApiController extends AbstractController
10 {
↕ // ... lines 11 - 36
37 }
```

Add a `public function getCollection()` because this will return info about a *collection* of starships. And, like always, you can add the `Response` return type or skip it. Above this, add the route with `#[Route()]`. Select the one from `Attribute` and hit tab.

So I just used auto-completion to add the `use` statements for `AbstractController`, `Route`, *and* `Response`. Make sure you have all of those. For the URL, how about `/api/starships`.

Inside, I'll paste a `$starships` variable that's set to an array of three associative arrays of starship data.

Returning JSON

You can probably imagine how this will look as JSON. How do we *turn* it into JSON? Well, it can be this simple: `return new Response` with `json_encode($starships)`.

But we can do better! Instead, return `$this->json($starships)`.

src/Controller/StarshipApiController.php

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

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

class StarshipApiController extends AbstractController
{
    #[Route('/api/starships')]
    public function getCollection(): Response
    {
        $starships = [
            [
                'name' => 'USS LeafyCruiser (NCC-0001)',
                'class' => 'Garden',
                'captain' => 'Jean-Luc Pickles',
                'status' => 'taken over by Q',
            ],
            [
                'name' => 'USS Espresso (NCC-1234-C)',
                'class' => 'Latte',
                'captain' => 'James T. Quick!',
                'status' => 'repaired',
            ],
            [
                'name' => 'USS Wanderlust (NCC-2024-W)',
                'class' => 'Delta Tourist',
                'captain' => 'Kathryn Journeyway',
                'status' => 'under construction',
            ],
        ];

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

Let's try it! Find your browser and head to `/api/starships`. Dang, that was easy. If you're wondering why the JSON is styled and looks cool, that's not a Symfony thing. I have a Chrome extension installed called JSONVue.

Adding a Model Class

Now in the real world, when we start querying the database, we're going to be working with *objects*, not associative arrays. We won't add a database in this tutorial, but we *can* start using objects for our data to make things more realistic. In the `src/` directory, create a new subdirectory called `Model`.

Ok, important thing: what we're about to do has absolutely *nothing* to do with Symfony. I'm simply looking at this array and thinking:

> *"You know what? Instead of passing around this associative array with `name`, `class`, `captain`, and `status` keys, I'd rather have a `Starship` class and pass around objects."*

So entirely on my own, independent of Symfony, I've decided to create a `Model` directory - this could be called anything - and inside a new class called `Starship`. And *because* this class is just to help *us*, we get to make it look *however* we want, and it doesn't need to extend any base class.

src/Model/Starship.php

```
// ... lines 1 - 2
namespace App\Model;

class Starship
{
// ... lines 7 - 39
}
```

Create a `public function __construct()` with five properties: a `private int $id`, then four more properties for each of the four keys that we have in the array: `private string $name`, `private string $class`, `private string $captain` and `private string $status`.

src/Model/Starship.php

```
// ... lines 1 - 2
namespace App\Model;

class Starship
{
    public function __construct(
        private int $id,
        private string $name,
        private string $class,
        private string $captain,
        private string $status,
    ) {
    }
// ... lines 15 - 39
}
```

Oh, and my editor is highlighting this file because we installed PHP-CS-Fixer and that found a code style violation. I can click this to fix it or go here and hit Alt+Enter to do the fix *there*. Super nice!

Anyway, if you're not familiar with this constructor syntax, this creates a constructor with five arguments *and*, at the same time, creates five properties that will be set to whatever we pass *to* these arguments.

But, because I decided to make these properties private, if we *did* instantiate a new `Starship` object... we wouldn't be able to read any of the data! To allow that, we can create getter methods. But, I'm not going to do this by hand. Instead, go to the Code -> Generate menu option - or Cmd + N on a Mac - select getters then generate a getter for *every* property.

src/Model/Starship.php

```
// ... lines 1 - 2
namespace App\Model;

class Starship
{
    public function __construct(
        private int $id,
        private string $name,
        private string $class,
        private string $captain,
        private string $status,
    ) {
    }

    public function getId(): int
    {
        return $this->id;
    }

    public function getName(): string
    {
        return $this->name;
    }

    public function getClass(): string
    {
        return $this->class;
    }

    public function getCaptain(): string
    {
        return $this->captain;
    }

    public function getStatus(): string
    {
        return $this->status;
    }
}
```

Nice! Five shiny new, public getter method.

Creating the Model Objects

Ok, back in our controller, let's convert these arrays to objects: `new Starship()` - hit tab, so it adds the `use` statement - then give this an id of, how about, 1... and transfer the other values for `name`, `class`, `captain`, and finally `status`.

And just like that, we have our first object! I'll highlight the other two arrays and paste in the two objects to save time.

src/Controller/StarshipApiController.php

```
// ... lines 1 - 4
use App\Model\Starship;
// ... lines 6 - 9
class StarshipApiController extends AbstractController
{
// ... line 12
    public function getCollection(): Response
    {
        $starships = [
            new Starship(
                1,
                'USS LeafyCruiser (NCC-0001)',
                'Garden',
                'Jean-Luc Pickles',
                'taken over by Q'
            ),
            new Starship(
                2,
                'USS Espresso (NCC-1234-C)',
                'Latte',
                'James T. Quick!',
                'repaired',
            ),
            new Starship(
                3,
                'USS Wanderlust (NCC-2024-W)',
                'Delta Tourist',
                'Kathryn Journeyway',
                'under construction',
            ),
        ];
// ... lines 38 - 39
    }
}
```

We now have an array of 3 `Starship` *objects*... which feels nicer. And we're passing those to `$this->json()`. Is that still going to work? Totally not! We get an array of three *empty* objects!

That's because, internally, `$this->json()` uses the PHP `json_encode()` function... and that function can't handle private properties. What we need is something smarter: something that can recognize that, even though the `name` property is private, we have a public `getName()` method that can be called to read that property's value.

Hello Symfony Serializer

Is there a tool that does that? Well, remember how Symfony is a *huge* set of components that solve individual problems? One component is called *serializer*, and its *whole* job is to take objects and serialize them to JSON... or take JSON and deserialize that *back* into objects. And it can *totally* handle situations where you have private properties with public getter methods.

So let's get it installed!

```
composer require serializer
```

And once more folks, yes, this is an *alias*... and it's an alias to a *pack*. This pack installs the `symfony/serializer` package as well as a few others that make it work in a really robust way.

Now, without doing *anything* else, go back, refresh, and it works? How?

It turns out that the `$this->json()` method is smart. To peek at it, hold Command on a Mac or Ctrl on other machines and click the method name to jump into the core Symfony file where this lives.

Ah! The code here won't make total sense yet, but it detects *if* the serializer system is available.... and if it is, uses *that* to transform the object to JSON.

But, what do I mean by "serializer system" exactly? And what is the `serializer` key... inside this container thing? Or, what if we needed to transform an object to JSON somewhere *other* than our controller... where we don't have access to the `->json()` shortcut? How could we access the serializer system from *there*?

Friends, it's time to learn about *the* most important concept in Symfony: services.

Chapter 9: Services: The Backbone of Everything

Let's talk about services. These are *the* most important concept in Symfony. And once you understand them, honestly, you'll be able to do *anything*.

What is a Service?

First, a service is an object that does work. That's it. For example, if you instantiated a `Logger` object that has a `log()` method, that's a service! It does work: it logs things! Or if you created a database connection object that makes queries to the database then... yup! That's a service too.

So then... if a service is just an object that does work... what lazy objects *aren't* services? Our `Starship` class is a perfect example of a *non* service. It's main job is *not* to do work: it's to hold data. Sure, it has a few public methods... and you could even put some logic inside of these methods to do something. But ultimately, it's not a worker, it's a data holder.

What about controller classes? Yeah, they're services too. Their work is to create response objects.

Anyway, *every* bit of work that's done in Symfony is actually done by a service. Writing log messages to this file? Yeah, there's a service for that. Figuring out which route matches the current URL? That's the `router` service! What about rendering a twig template? Yep, it turns out that the `render()` method is a shortcut to find the correct service object and call a method on it.

The Container & debug:container

You may sometimes also hear that these services are organized into a big object called the "service container". You can think of the container like a giant associative array of service objects, each with a unique id. Want to see a list of every service in our app right now? Me too!

Find your terminal and run:

```
bin/console debug:container
```

That's a lot of services! Let me make this smaller so each fits on its own line... better.

On the left side, we see the *ID* of each service. And on the right, the *class* of the object that the ID corresponds to. Cool, right?

Go back to our controller and hold control or command to open up the `json()` method again. Now this makes more sense! It's checking to see if the container has a service whose ID is `serializer`. If it does, it grabs that service from the container and calls the `serialize()` method on it.

When *we* work with services, it won't look exactly like this. But the super important thing is that we now understand what's going on.

Bundles Provide Services

My next question is: where do these services come from? Like, who says there's a service whose ID is `twig`... and that when we ask the container for it, it should return a twig `Environment` object? The answer is: *entirely* from bundles. In fact, that's the main point of installing a new bundle. Bundles give us services.

Remember when we installed `twig`? It added a bundle to our app. And guess what that bundle did? Yup: it gave us new services, including the `twig` service. Bundles give us services... and services are *tools*.

Autowiring

And though there are *many* services in this list, the vast majority of these are low-level service objects that we won't ever use or care about. We also won't care about the ID of the services most of the time.

Instead, run a related command called:

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

This shows us all the services that are autowireable, which is the technique that we'll use to fetch services. It's basically a curated list of the services that we're most likely to need.

Autowiring the Logger Service

So let's do a challenge: let's log something from our controller. Here's a sneak peek into how I approach this problem in my brain:

> *"Ok, I need to log something! And... logging is work. And... services do work! Thus, there must be a logger service that I can use! Quod erat demonstrandum!"*

Forgive me latin nerds. The point is: if we want to log something, we just need to find the service that *does* that work. Okay! Rerun the command but search for log:

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

Boom! It found about 10 services, all starting with `Psr\Log\LoggerInterface`. We're going to talk about what these *other* services are in the next tutorial. For now, focus on the main one. This tells me is that there *is* a service in the container for a logger. And to get it, we can autowire it using this interface.

What does that mean? In the controller method where we want the logger, add an argument type-hinted with `LoggerInterface` - hit tab - then say `$logger`.

src/Controller/StarshipApiController.php

```
// ... lines 1 - 5
use Psr\Log\LoggerInterface;
// ... lines 7 - 10
class StarshipApiController extends AbstractController
{
// ... line 13
    public function getCollection(LoggerInterface $logger): Response
    {
// ... lines 16 - 41
    }
}
```

In this case, the *name* of the argument isn't important: it could be anything. What matters is that the `LoggerInterface` - that corresponds to this `use` statement - matches the `Psr\Log\LoggerInterface` from `debug:autowiring`.

It's that simple! Symfony will see this type-hint and say:

> *"Oh! Since that type-hint matches the autowiring type for this service, they must want me to pass them that service object."*

I don't know why Symfony sounds like a frog in my head. Anyway, let's see if this works. Add `dd($logger)`: `dd()` stands for "dump and die" and comes from Symfony.

src/Controller/StarshipApiController.php

```
// ... lines 1 - 5
use Psr\Log\LoggerInterface;
// ... lines 7 - 10
class StarshipApiController extends AbstractController
{
// ... line 13
    public function getCollection(LoggerInterface $logger): Response
    {
        dd($logger);
// ... lines 17 - 41
    }
}
```

Refresh! Yes! It printed the object beautifully then stopped execution. It's *working*! Symfony passes us a `Monolog\Logger` object, which implements that `LoggerInterface`.

The trick we just did - called autowiring - works in exactly two places: our controller methods and the `__construct()` method of any service. We'll see that second situation in the next chapter.

Controlling how Services Behave

And if you're wondering where this `Logger` service came from in the first place... we already know the answer! From a bundle. In this case, `MonologBundle`. And... how could we *configure* that service... to, I don't know, log to a different file? The answer is: `config/packages/monolog.yaml`.

This config - including this line - configures `MonologBundle`... which really means that it configures how the *services* work that MonologBundle give us. We'll learn about this percent syntax in the next tutorial, but this tells the `Logger` service to log to this `dev.log` file.

Using the Logger

Ok, now that we have the `Logger` service, let's use it! How? Well, *of course*, you can read the docs. But thanks to the type-hint, our editor will help us! `LoggerInterface` has a *bunch* of methods. Let's use `->info()` and say:

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 5
6  use Psr\Log\LoggerInterface;
↕ // ... lines 7 - 10
11 class StarshipApiController extends AbstractController
12 {
↕ // ... line 13
14     public function getCollection(LoggerInterface $logger): Response
15     {
16         $logger->info('Starship collection retrieved');
↕ // ... lines 17 - 41
42     }
43 }
```

> *“Starship collection retrieved.”*

Try it out: refresh. The page worked... but did it log anything? We could go check the `dev.log` file. *Or*, we can use the Log section of the profiler for this request.

Seeing the Profiler for an API Request

But... wait! This is an API request... so we don't have that cool web debug toolbar on the bottom! That's true... but Symfony *did* still collect all that info! To get to the profiler for this request, change the URL to `/_profiler`. This lists the most recent requests to our app, with the newest on top. See this one? That's our API request from a minute ago! If you click this token... boom! We're looking at the profiler for that API call in all its glory... including a Log section... with our message.

Ok, now that we've seen how to *use* a service, let's create our *own* service next! We're unstoppable!

Chapter 10: Creating your own Service

We know that services do work, and we know that Symfony is full of services that we can use. If you run:

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

We get the dinner menu of services, where you can order any of these by adding an argument type-hinted with the matching class or interface.

We, of course, *also* do work in *our* code... hopefully. Right now, all that work is being done inside our controller, like creating the Starship data. Sure, this is hard-coded right now, but imagine if this were *real* work: like a complex database query. Putting the logic inside a controller is "ok"... but what if we wanted to reuse this code somewhere else? What if, on our homepage, we wanted to get a dynamic count of the Starships by grabbing this same data?

Creating the Service Class

To do that, we need to move this "work" into its own service that *both* controllers could then use. In the `src/` directory, create a new `Repository` directory and a new PHP class inside called `StarshipRepository`.

src/Repository/StarshipRepository.php

```
// ... lines 1 - 2
namespace App\Repository;

class StarshipRepository
{
}
```

Just like when we built our `Starship` class, this new class has absolutely nothing to do with Symfony. It's just a class that *we've* decided to create to organize *our* work. And so, Symfony doesn't care what it's called, where it lives or what it looks like. I called it `StarshipRepository`

and put it in a `Repository` directory because that's a common programming name for a class whose "work" is to fetch a type of data, like Starship data.

Autowiring the New Service

Ok, before we even do anything in here, let's see if we can use this inside a controller. And, good news! *Just* by creating this class, it's already available for autowiring. Add a `StarshipRepository $repository` argument, and, to make sure it's working, `dd($repository)`.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 5
6  use App\Repository\StarshipRepository;
↕ // ... lines 7 - 11
12 class StarshipApiController extends AbstractController
13 {
↕ // ... line 14
15     public function getCollection(LoggerInterface $logger, StarshipRepository
   $repository): Response
16     {
17         $logger->info('Starship collection retrieved');
18         dd($repository);
↕ // ... lines 19 - 43
44     }
45 }
```

All right, spin over, click back to our endpoint, and... got it. That's so cool! Symfony saw the `StarshipRepository` type-hint, instantiated that object, then passed it to us. Delete the `dd()`... and let's move the starship data inside. Copy it... and create a new public function called, how about, `findAll()`. Inside, `return`, then paste.

src/Repository/StarshipRepository.php

```
// ... lines 1 - 4
use App\Model\Starship;

class StarshipRepository
{
    public function findAll(): array
    {
        return [
            new Starship(
                1,
                'USS LeafyCruiser (NCC-0001)',
                'Garden',
                'Jean-Luc Pickles',
                'taken over by Q'
            ),
            new Starship(
                2,
                'USS Espresso (NCC-1234-C)',
                'Latte',
                'James T. Quick!',
                'repaired',
            ),
            new Starship(
                3,
                'USS Wanderlust (NCC-2024-W)',
                'Delta Tourist',
                'Kathryn Journeyway',
                'under construction',
            ),
        ];
    }
}
```

Back over in `StarshipApiController`, delete that... and it's beautifully simple: `$starships = $repository->findAll()`.

src/Controller/StarshipApiController.php

```
// ... lines 1 - 4
use App\Repository\StarshipRepository;
// ... lines 6 - 10
class StarshipApiController extends AbstractController
{
    #[Route('/api/starships')]
    public function getCollection(LoggerInterface $logger, StarshipRepository $repository): Response
    {
        $logger->info('Starship collection retrieved');
        $starships = $repository->findAll();
// ... lines 18 - 19
    }
}
```

Done! When we try it, it *still* works... and now the code for fetching starships is nicely organized into its own class and reusable across our app.

Constructor Autowiring

With that victory under our belt, let's doing something harder. What if, from inside `StarshipRepository`, we needed access to *another* service to help us do our work? No problem! We can use autowiring! Let's try to autowire the logger service again.

The only difference this time is that we're *not* going to add the argument to `findAll()`. I'll explain why in a minute. Instead, add a new `public function __construct()` and do the auto-wiring *there*: `private LoggerInterface $logger`.

src/Repository/StarshipRepository.php

```
// ... lines 1 - 5
use Psr\Log\LoggerInterface;

class StarshipRepository
{
    public function __construct(private LoggerInterface $logger)
    {
    }
// ... lines 13 - 41
}
```

Down below, to use it, copy the code from our controller, delete that, paste it here, and update it to `$this->logger`.

src/Repository/StarshipRepository.php

```
// ... lines 1 - 5
use Psr\Log\LoggerInterface;

class StarshipRepository
{
    public function __construct(private LoggerInterface $logger)
    {
    }

    public function findAll(): array
    {
        $this->logger->info('Starship collection retrieved');
// ... lines 17 - 40
    }
}
```

Cool! Over in the controller, we can remove that argument because we're not using it anymore.

Testing time! Refresh! No error - that's a good sign. To see if it logged something, go to `/_profiler`, click on the top request, Logs, and... there it is!

So let me explain why we added the service argument to the constructor. If we want to fetch a service - like the logger, a database connection, whatever, *this* is the correct way to use autowiring: add a `__construct` method inside another service. The trick we saw earlier - where we add the argument to a normal method - yeah, that's special and *only* works for *controller* methods. It's an extra convenience that was added to the system. It's a great feature, but the constructor way... that's how autowiring really works.

And this "normal" way, it even works in a controller. You could add a `__construct()` method with an autowirable argument and that would totally work.

The point is: if you *are* in a controller method, sure, add the argument to the method - it's nice! Just remember that it's a special thing that only works here. Everywhere else, autowire through the constructor.

Using the Service on another Page

Let's celebrate our new service by using it on the homepage. Open up `MainController`. This hardcoded `$starshipCount` is *so* 30 minutes ago. Autowire `StarshipRepository $starshipRepository`, then say `$ships = $starshipRepository->findAll()` and count them with `count()`.

src/Controller/MainController.php

```
// ... lines 1 - 4
use App\Repository\StarshipRepository;
// ... lines 6 - 9
class MainController extends AbstractController
{
    #[Route('/')]
    public function homepage(StarshipRepository $starshipRepository): Response
    {
        $ships = $starshipRepository->findAll();
        $starshipCount = count($ships);
// ... lines 17 - 22
    }
}
```

While we're here, instead of this hardcoded `$myShip` array, let's grab a random `Starship` object. We can do that by saying `$myShip` equals `$ships[array_rand($ships)]`

src/Controller/MainController.php

```
// ... lines 1 - 4
use App\Repository\StarshipRepository;
// ... lines 6 - 9
class MainController extends AbstractController
{
    #[Route('/')]
    public function homepage(StarshipRepository $starshipRepository): Response
    {
        $ships = $starshipRepository->findAll();
        $starshipCount = count($ships);
        $myShip = $ships[array_rand($ships)];
// ... lines 18 - 22
    }
}
```

Let's try it! Hunt down your browser and head to the homepage. Got it! We see the randomly changing ship down here, and the correct ship number up here... because we're multiplying it by 10 in the template.

Printing Objects in Twig

And something crazy-cool just happened! A minute ago, `myShip` was an associative array. But we *changed* it to be a Starship *object*. And yet, the code on our page kept working. We just accidentally saw a superpower of Twig. Head to `templates/main/homepage.html.twig` and scroll down to the bottom. When you say `myShip.name`, Twig is really smart. If `myShip` is an associative array, it'll grab the `name` key. If `myShip` is an object, like it is now, it will grab the `name` property. But even *more* than that, if you look at `Starship`, the `name` property is *private*, so we can't access it directly. Twig realizes that. It looks at the `name` property, sees that it's private, but *also* sees that there's a public `getName()`. And so, it calls *that*.

All we need to say is `myShip.name`... and Twig handles the details of how to fetch that, which I love.

Ok, one last tiny tweak. Instead of passing the `starshipCount` into our template, we can do the count inside Twig. Delete this variable, and instead, pass a `ships` variable.

src/Controller/MainController.php

```
// ... lines 1 - 9
class MainController extends AbstractController
{
// ... line 12
    public function homepage(StarshipRepository $starshipRepository): Response
    {
        $ships = $starshipRepository->findAll();
        $myShip = $ships[array_rand($ships)];
// ... line 17
        return $this->render('main/homepage.html.twig', [
            'myShip' => $myShip,
            'ships' => $ships,
        ]);
    }
}
```

In the template, there we go, for the count, we can say `ships`, which is an array, and then use a Twig filter: `|length`.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
// ... lines 6 - 9
<div>
    Browse through {{ ships|length * 10 }} starships!

    {% if ships|length > 2 %}
// ... lines 14 - 17
    {% endif %}
</div>
// ... lines 20 - 42
{% endblock %}
```

That feels good. Let's do the same thing down here... and change it to greater than 2. Try that out. Our site just keeps working!

Next up: let's create more pages and learn how to make routes that are even smarter.

Chapter 11: Fancier Routes: Requirements, Wildcards, and More

With all the new code organization, let's celebrate by creating another API endpoint to fetch a *single* `starship`. Start like usual: create a `public function` called, how about, `get()`. I'll include the optional `Response` return type. Above this add the `#[Route]` with a URL of `/api/starships/`... hmm. This time, the last part of the URL needs to be dynamic: it should match `/api/starships/5` or `/api/starships/25`. How can we do that? How can we make a route match a wildcard?

The answer is by adding `{`, a name, the `}`.

The name inside this could be anything. No matter what, this route will now match `/api/starships/*`. But whatever you name this, you're now *allowed* to have an argument with a *matching* name: `$id`.

Below, dump this to make sure it's working.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 9
10 class StarshipApiController extends AbstractController
11 {
↕ // ... lines 12 - 19
20     #[Route('/api/starships/{id}')]
21     public function get($id): Response
22     {
23         dd($id);
24     }
25 }
```

Restricting the Wildcard to be a Number

Ok! Zoom over to `/api/starships/2` and... it *is* working!

In our app, the ID will be an integer. If I try something that is *not* an integers - like `/wharf` - the route still matches and calls our controller. And, that's almost always okay. In a real app, if we

queried the database with `WHERE ID = 'wharf'`, it wouldn't cause an error: it just wouldn't find a matching ship! And then we could trigger a 404 page, which I'll show you how to do soon.

But sometimes, we *may* want to restrict these values. We may want to say:

> *"Only match this route if the wildcard is an integer."*

To do that, inside the curly brace, after the name, add a `<`, `>` and inside, a regular expression `\d+`.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 9
10 class StarshipApiController extends AbstractController
11 {
↕ // ... lines 12 - 19
20     #[Route('/api/starships/{id<\d+>}')]
21     public function get(int $id): Response
22     {
23         dd($id);
24     }
25 }
```

This means: match a digit of any length. With this setup, if we refresh the `wharf` URL, we get a 404 error. Our route simply wasn't matched - *no* route matched - so our controller was never called. But if we go back to `/2`, that still works.

And as an added benefit, now that this only matches digits, we can add an `int` type to the argument. Now, instead of the string `2`, we get the `integer` 2. These details aren't super important, but I want you to know what options you have.

Restricting the Route HTTP Method

One thing that *is* common with APIs is to make routes only match a certain HTTP *method*, like `GET` or `POST`. For example, if you want to *fetch* all the starships, users should make a `GET` request... same if you want to fetch a single ship. If we kept building our API and created an endpoint that could be used to create a *new* `Starship`, the standard way to do that would be to use the *same* URL: `/api/starships` but with a `POST` request.

Right now, this wouldn't work. *Every* time the user requested `/api/starships` - no matter if they use a `GET` or `POST` request, it would match this *first* route.

For that reason, it's common in an API to add a `methods` option set to an array, with `GET` or `POST`. I'll do the same thing down here: `methods: ['GET']`.

src/Controller/StarshipApiController.php

```
// ... lines 1 - 9
class StarshipApiController extends AbstractController
{
    #[Route('/api/starships', methods: ['GET'])]
    public function getCollection(StarshipRepository $repository): Response
// ... lines 14 - 19
    #[Route('/api/starships/{id<\d+>}', methods: ['GET'])]
    public function get(int $id): Response
// ... lines 22 - 24
}
```

I can't easily test this in a browser, but if we made a POST request to `/api/starships/2`, it would *not* match our route.

But we *can* see the change in our terminal. Run:

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

Perfect! Most routes match *any* method... but our two API routes only match if a `GET` request is made to that URL.

Prefixing Every Route URL

Ok, I have *one* more routing trick to show you... and this is a fun one! Every route in this controller starts with the same URL: `/api/starships`. Having the full URL in each route is fine. But if we want, we can automatically *prefix* each route's URL. Above the class, add a `#[Route]` attribute with `/api/starships`.

Unlike when we put this above a *method*, this does *not* create a route. It just says: every route in this class should be prefixed with this URL. So for the first route, remove the path entirely. And for the second, we only need the wildcard part.

src/Controller/StarshipApiController.php

```
// ... lines 1 - 9
#[Route('/api/starships')]
class StarshipApiController extends AbstractController
{
    #[Route('', methods: ['GET'])]
    public function getCollection(StarshipRepository $repository): Response
// ... lines 15 - 20
    #[Route('/{id<\d+>}', methods: ['GET'])]
    public function get(int $id): Response
// ... lines 23 - 25
}
```

Try `debug:router` again... and watch these URLs:

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

They don't change!

Finishing the new API Endpoint

Okay. Let's finish our endpoint. We need to find the one ship that matches this ID. Normally we'd query the database: `select * from ship where id =` this ID. Our ships are hardcoded right now, but we can still do something that will look pretty much exactly like what it will, once we *do* have a database.

We already have a service - `StarshipRepository` - whose whole job is to fetch starship data. Let's give it a new superpower: the ability to fetch a *single* `Starship` for an id. Add `public function find()` with an `int $id` argument that will return a nullable `Starship`. So, a `Starship` if we find one for this id, else `null`.

Right now, the easiest way write this logic is to loop over `$this->findAll()` as `$starship`... then if `$starship->getId() === $id`, return `$starship`. I'll change my `uf` to `if`. Much better.

And if we didn't find anything, at the bottom, `return null`.

src/Repository/StarshipRepository.php

```
// ... lines 1 - 7
class StarshipRepository
{
// ... lines 10 - 42
    public function find(int $id): ?Starship
    {
        foreach ($this->findAll() as $starship) {
            if ($starship->getId() === $id) {
                return $starship;
            }
        }

        return null;
    }
}
```

Thanks to this, our controller is *so* simple. First, autowire the repository by adding an argument: `StarshipRepository` and just call it `$repository`. By the way, the order of arguments in a controller doesn't matter.

Then `$starship = $repository->find($id)`. Finish at the bottom with `return $this->json($starship)`.

src/Controller/StarshipApiController.php

```
// ... lines 1 - 10
class StarshipApiController extends AbstractController
{
// ... lines 13 - 21
    public function get(int $id, StarshipRepository $repository): Response
    {
        $starship = $repository->find($id);

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

I think we're ready! Refresh. It's perfect!

Triggering a 404 Page

But try an id that does *not* exist in our fake database - like `/200`. The word `null` is... *not* what we want. In this situation, we should return a response with a 404 status code.

To do that, we're going to follow a common pattern: query for an object, then check if it returned anything. If it did *not* return something, trigger a 404. Do that with throw `$this->createNotFoundException()`. I'll pass this a message.

src/Controller/StarshipApiController.php

```
// ... lines 1 - 10
class StarshipApiController extends AbstractController
{
// ... lines 13 - 21
    public function get(int $id, StarshipRepository $repository): Response
    {
        $starship = $repository->find($id);

        if (!$starship) {
            throw $this->createNotFoundException('Starship not found');
        }

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

Notice the `throw` keyword: we're throwing a special exception that triggers a 404. That's nice because, as soon as it hits this line, nothing *after* will be executed.

Try it out! Yes! A 404 response! The message - "Starship not found" - is only shown to developers in dev mode. In production, a totally different page - or JSON - would be returned. You can check the docs for details on production error pages.

Next: let's build the HTML version of this page, a page that shows details about a *single* starship. Then we'll learn how to link between pages using the route name.

Chapter 12: Generating URLs

Let's create a "show page" for ships: a page that displays the details for just *one* ship. The homepage lives in `MainController`. And so we *could* add another route and method here. But as my site grows, I'm probably going to have multiple pages related to starships: maybe to edit and delete them. So instead, in the `Controller/` directory, create a new class. Call it `StarshipController`, and, as usual, extend `AbstractController`.

Creating the Show Page

Inside, let's get to work! Add a `public function` called `show()`, I'll add the `Response` return type, then the route, with `/starships/` and a wildcard called `{id}`. And again, it's optional, but I'll be fancy and add the `\d+` so the wildcard only matches a number.

Now, *because* we have an `{id}` wildcard, we are *allowed* to have an `$id` argument down here. `dd($id)` to see how we're doing so far.

src/Controller/StarshipController.php

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

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

class StarshipController extends AbstractController
{
    #[Route('/starships/{id<\d+>}')]
    public function show(int $id): Response
    {
        dd($id);
    }
}
```

Try it. Head to `/starships/2`. Lovely!

Now we're going to do something familiar: take this `$id` and query our imaginary database for the matching `Starship`. The key to doing this is our `StarshipRepository` service and its helpful `find()` method.

In the controller, add a `StarshipRepository $repository` argument... then say `$ship` equals `$repository->find($id)`. And if *not* `$ship`, trigger a 404 page with throw `$this->createNotFoundException()` and `starship not found`.

Cool! At the bottom, instead of returning JSON, render a template: return `$this->render()` and follow the standard naming convention for templates: `starship/show.html.twig`. Pass this one variable: `$ship`.

src/Controller/StarshipController.php

```
// ... lines 1 - 4
use App\Repository\StarshipRepository;
// ... lines 6 - 9
class StarshipController extends AbstractController
{
    #[Route('/starships/{id<\d+>}')]
    public function show(int $id, StarshipRepository $repository): Response
    {
        $ship = $repository->find($id);
        if (!$ship) {
            throw $this->createNotFoundException('Starship not found');
        }

        return $this->render('starship/show.html.twig', [
            'ship' => $ship,
        ]);
    }
}
```

Creating the Template

Controller, check! Next, in the `templates/` directory, we *could* create a `starship/` directory and `show.html.twig` inside. But I want to show you a shortcut from the Symfony PhpStorm plugin. Click on the template name, press Alt+Enter and... check it out! On top it says "Twig: Create Template". Confirm the path and boom! We've got our new template! It's... hiding over here. There it is: `starship/show.html.twig`.

Pretty much every template starts the same: `{% extend 'base.html.twig' %}`... then override some blocks! Override `title`... and this time, let's use that `ship` variable: `ship.name`. Finish with `endblock`.

And for the main content, add the block `body`... `endblock` and put an `h1` inside. Print `ship.name` again and... I'll paste in a table with some info.

templates/starship/show.html.twig

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

{% block title %}{{ ship.name }}{% endblock %}

{% block body %}
    <h1>{{ ship.name }}</h1>

    <table>
        <tbody>
            <tr>
                <th>Class</th>
                <td>{{ ship.class }}</td>
            </tr>
            <tr>
                <th>Captain</th>
                <td>{{ ship.captain }}</td>
            </tr>
        </tbody>
    </table>
{% endblock %}
```

Nothing special here: we're just printing basic ship data.

When we try the page... it's alive!

Linking Between Pages

Next question: from the homepage, how could we add a link *to* the new ship show page? The most obvious option is to hardcode the URL, like `/starships/` then the id. But there's a *better* way. Instead, we're going to tell Symfony:

> *"Hey, I want to generate a URL to this route."*

The advantage is that if we decide later to change the URL of this route, every link to this will update automatically.

Let me show you. Find your terminal and run:

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

I haven't mentioned it yet, but every route has an internal name. Right now, they're being auto-generated by Symfony, which is fine. But as soon as you want to generate a URL to a route, we should take *control* of that name to make sure it never changes.

Find the show page route and add a `name` key. I'll use `app_starship_show`.

src/Controller/StarshipController.php

```
// ... lines 1 - 9
class StarshipController extends AbstractController
{
    #[Route('/starships/{id<\d+>}', name: 'app_starship_show')]
    public function show(int $id, StarshipRepository $repository): Response
// ... lines 14 - 23
}
```

The name could be anything, but this is the convention I follow: `app` because it's a route that I'm making in my app, then the controller class name and method name.

Naming a route doesn't change how it works. But it *does* let us generate a URL to it. Open up `templates/main/homepage.html.twig`. Down here, turn the ship name into a link. I'll put this onto multiple lines and add an `a` tag with `href=""`. To generate the URL, say `{{ path() }}` and pass it the *name* of the route. I'll put the closing tag on the other side.

If we stop now, this won't *quite* work. On the homepage:

> *"Some mandatory parameters are missing - `id` - to generate a URL for route `app_starship_show`."*

That makes sense! We're telling Symfony:

> *"Howdy! I want to generate a URL to this route."*

Symfony then responds:

> *"Cool... except that this route has a wildcard in it. So like... what do you want me to put in the URL for the `id` part?"*

When there's a wildcard in the route, we need to add a second argument to `path()` with `{}`. This is Twig's associative array syntax. So it's exactly like JavaScript: it's a key-value pair list. Pass `id` set to `myShip.id`.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
// ... lines 6 - 20
<div>
// ... lines 22 - 23
    <table>
        <tr>
            <th>Name</th>
            <td>
                <a href="{{ path('app_starship_show', {
                    id: myShip.id
                }) }}">{{ myShip.name }}</a>
            </td>
        </tr>
// ... lines 33 - 44
    </table>
</div>
{% endblock %}
```

And now... got it! Look at that URL: `/starships/3`.

Alrighty, our site is still *ugly*. It's time to start fixing that by bringing in Tailwind CSS and learning about Symfony's AssetMapper component.

Chapter 13: CSS & JavaScript with Asset Mapper

What about images, CSS and JavaScript? How does that work in Symfony?

Stuff in public is... Public

First off, the `public/` directory is known as your document root. Anything inside `public/` is accessible to your end user. Anything *not* in `public/` is *not* accessible, which is great! None of our top secret source files can be downloaded by our users.

So if you want to create a CSS file or an image file or anything else, life *can* be as simple as putting it in `public/`. I can now go to `/foo.txt`... and we see the file.

Hello Asset Mapper

However, Symfony has a *great* component called Asset Mapper that lets us *effectively* do the same thing... but with some important, extra features. We have a few tutorials that go deeper into this topic: one about Asset Mapper specifically and another about building things *with* Asset Mapper called LAST Stack. Check those out to go deeper.

But let's dive into the friendly waters of Asset Mapper! Commit all your changes - I already have - then install it with:

```
composer require symfony/asset-mapper
```

This recipe makes several changes... and we'll walk through each little-by-little as they're important.

But already, if we move over and refresh, our background is blue! Inspect Element in your browser and go to the console. We also have a console log!

> *"This log comes from `assets/app.js`. Welcome to asset mapper."*

Why thank you!

Asset Mapper's 2 Super Powers

Asset Mapper has two big superpowers. The first is that it helps us load CSS and JavaScript. The recipe gave us a new `assets/` directory with an `app.js` file and a `styles/app.css` file. As we saw, the console log is coming from `app.js`.

assets/app.js

```
/*
 * Welcome to your app's main JavaScript file!
 *
 * This file will be included onto the page via the importmap() Twig function,
 * which should already be in your base.html.twig.
 */
import './styles/app.css';

console.log('This log comes from assets/app.js - welcome to AssetMapper! 🎉');
```

So this file *is* being loaded. It's also apparently including `app.css`, which is what gives us that blue background.

assets/styles/app.css

```
body {
    background-color: skyblue;
}
```

We're going to talk more later about how these files are loaded and how this all works. But for right now, it's enough to know that `app.js` and `app.css` are included on the page.

The second big superpower of Asset Mapper is a bit simpler. The recipe created a `config/packages/asset_mapper.yaml` file. There's not a lot here:

config/packages/asset_mapper.yaml

```
framework:
    asset_mapper:
        # The paths to make available to the asset mapper.
        paths:
            - assets/
```

just `paths` pointing to our `assets/` directory. But because of this line, *any* file that we put in the `assets/` directory becomes available publicly. It's as if the `assets/` directory physically lives

inside `public/`. This is useful because, along the way, Asset Mapper adds asset *versioning*: an important frontend concept that we'll see in a minute.

Listing Assets & the Logical Path

But first, head to your terminal and run *another* new `debug` command:

```
php bin/console debug:asset
```

This shows *every* asset that's exposed publicly through Asset Mapper. Right now it's just two: `app.css` and `app.js`.

If you download the course code from this page and unzip it, you'll find a `tutorial/` directory with an `images/` subdirectory. I'll cut this... then paste into `assets/`.

So now we have an `assets/images/` directory with 5 files inside. And, by the way, you can organize the `assets/` directory however you want.

But now, spin back over and run `debug:asset` again:

```
php bin/console debug:asset
```

The new files are there!

Rendering an Image

On the left, see this "logical path"? That's the path we'll use to *reference* that file in Asset Mapper.

I'll show you: let's render an `img` tag to the logo. Copy the `starshop-logo.png` logical path. Then head into `templates/base.html.twig`. Right above the body block - so it's not overridden by our page content - add an `<img>` tag with `src=""`. Instead of trying to hardcode a path, say `{{` and use a new Twig function called `asset()`. Pass *this* the logical path.

That's it! Ok, I'll add an `alt` attribute... to be a good citizen of the web.

templates/base.html.twig

```
// ... line 1
<html>
// ... lines 3 - 13
    <body>
        <img src="{{ asset('images/starshop-logo.png') }}" alt="Starshop Logo">
        {% block body %}{% endblock %}
    </body>
</html>
```

Let's try this. Refresh and... it explodes!

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

Remember: our app starts tiny. And then, as we need more features, we install more Symfony components. And often, if you try to use a feature from a component that's *not* installed, it'll tell you. The Twig `asset()` function comes from another tiny component called `symfony/asset`. All *we* need to do is follow the advice. Copy the `composer require` command, spin over to your terminal and run it:

```
composer require symfony/asset
```

When it finishes, move over and refresh. There's our logo!

Automatic Asset Versioning

The most interesting part? View the page source and check out the URL: `/assets/images/starshop-logo-` then a long string of letters and numbers, `.png`. This string is called the version hash and its generated based on the *content* of the file. That means that if we update our logo later on, this hash will change automatically.

That's *super* important. Browsers like to cache images, JavaScript, and CSS files, which is great: it helps performance. But when we *change* those files, we want our users to download the *new* version: not keep using the outdated, cached version.

But because the filename will change when we update the image, the browser is going to automatically use the new one! It looks like this:

- User goes to our site and downloads `logo-abc123.png`. Their browser caches it.
- On the next visit, their browser sees the `img` tag for `logo-abc123.png`, finds the file in its cache and uses it.
- Then we come along, update that file and deploy.
- The next time the user goes to our site, the `img` tag will be pointing at a *different* filename: `logo-def456.png`. And since the browser doesn't have *that* file in its cache, it downloads it fresh.

This is kind of a small detail, but it's also *incredibly* important to make sure our users are always using the latest files. And the best part? It just works. Now that I've explained it, you'll never need to think about this again.

Ok team, let's install & start using Tailwind CSS next.

Chapter 14: Tailwind CSS

What about CSS? You're free to add whatever CSS you want to `app/styles/app.css`. That file is already loaded on the page.

Want to use Bootstrap CSS? Check out the Asset Mapper docs on how to do that. Or, if you want to use Sass, there's a symfonycasts/sass-bundle to make that easy. Though, I recommend not jumping into Sass *too* quickly. A lot of the features that Sass is famous for can now be done in native CSS, like CSS variables and even CSS nesting.

Hello Tailwind

What's my personal choice for a CSS framework? Tailwind. And part of the reason is that Tailwind is *insanely* popular. So if you're looking for resources or pre-built components, you're going to have a lot of luck if you use Tailwind.

But Tailwind *is* a bit odd in one way: it's not simply a big CSS file that you plop onto your page. Instead, it has a *build* process that scans your code for all the Tailwind classes you're using. It then dumps a final CSS file that *only* contains the code you need.

In the Symfony world, if you want to use Tailwind, there's a bundle that makes it really easy. Spin over your terminal and install a new package: `composer require symfonycasts` - hey I know them - `tailwind-bundle`:

```
composer require symfonycasts/tailwind-bundle
```

For this package, the recipe doesn't do anything other than enable the new bundle. To get Tailwind rocking, *one* time in your project, run:

```
php bin/console tailwind:init
```

This does three things. First, it downloads a Tailwind binary in the background, which you'll never really need to think about. Second, it creates a `tailwind.config.js` file at the root of our project. This tells Tailwind *where* it needs to look in our project for Tailwind CSS classes. And third, it updated our `app.css` to add these three lines. These will be replaced by the *real* Tailwind code in the background by the binary.

Running Tailwind

Finally, Tailwind needs to be *built*, so we need to run a command to do that:

```
php bin/console tailwind:build -w
```

This scans our templates and output the final CSS file in the background. The `-w` puts it in "watch" mode: instead of building *once* and exiting, it watches our templates for changes. When it notices any updates, it will automatically rebuild the CSS file. We'll see that in minute.

But we should already see a difference. Let's go to the homepage. Did you see that? The base Tailwind code did a reset. For example, our `h1` is now tiny!

Seeing Tailwind in Action

Let's try this out for real. Open `templates/main/homepage.html.twig`. Up on the `h1`, make this bigger by adding a class: `text-2xl`.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <h1 class="text-2xl">
7     Starshop: your monopoly-busting option for Starship parts!
8 </h1>
↕ // ... lines 9 - 46
47 {% endblock %}
```

As soon as we save that, you can see that tailwind *noticed* our change and rebuilt the CSS. And when we refresh, it got bigger!

Our source `app.css` file is *still* super simple - just those few lines we saw earlier. But view the page source and open the `app.css` that's being sent to our users. It's the built version from Tailwind! Behind the scenes, some magic exists that replaces those three Tailwind lines with the *real* Tailwind CSS code.

Automatically Running Tailwind with the symfony Binary

And... that's kind of it! It just works. Though there *is* an easier and more automatic way to run Tailwind. Hit Ctrl+C on the Tailwind command to stop it. Then, at the root of our project, create a file called `.symfony.local.yaml`. This is a config file for the `symfony` binary web server that we're using. Inside, add `workers`, `tailwind`, then `cmd` set to an array with each part of a command: `symfony`, `console`, `tailwind`, `build`, `--watch`, or you could use `-w`: it's the same.

I haven't talked about it yet, but instead of running `php bin/console`, we can also run `symfony console` followed by any command to get the same result. We'll talk about *why* you might want to do that in a future tutorial. But for now, consider `bin/console` and `symfony console` the same thing.

Also, by adding this `workers` key, it means that instead of *us* needing to run the command manually, when we start the `symfony` web server, *it* will run it *for* us in the background.

Watch. In your first tab, hit Ctrl+C to stop the web server... then re-run

```
symfony serve
```

so it sees the new config file. Watch: there it is! It's running the tailwind command in the background!

We can take advantage of this immediately. In `homepage.html.twig`, change this to `text-4xl`, spin over and... it works! We don't even need to *think* about the `tailwind:build` command anymore.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
<h1 class="text-4xl">
    Starshop: your monopoly-busting option for Starship parts!
</h1>
// ... lines 9 - 46
{% endblock %}
```

And since we'll be styling with Tailwind, remove the blue background.

Copying in Styled Templates

Ok, this tutorial is *not* about Tailwind or how to design a website. Trust me, you do *not* want Ryan leading the web design charge. But I *do* want to have a nice-looking site... and it's *also* important to go through the process of working with a designer.

So let's pretend that someone else has created a design for our site. And they've even given us some HTML with Tailwind classes *for* that design. If you download the course code, in a `tutorial/templates/` directory, we have 3 templates. One-by-one, I'm going to copy each file and paste it over the original. Don't worry, we'll look at what's happening in each of these files.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org/2000/svg%22 viewBox=%220 0 128 128%22><text y=%221.2em%22 font-size=%2296%22>⚫</text></svg>">
        {% block stylesheets %}
        {% endblock %}

        {% block javascripts %}
{% block importmap %}{{ importmap('app') }}{% endblock %}
        {% endblock %}
    </head>
    <body class="text-white" style="background: radial-gradient(102.21% 102.21% at 50% 28.75%, #00121C 42.62%, #013954 100%);">
        <div class="flex flex-col justify-between min-h-screen relative">
            <div>
                <header class="h-[114px] shrink-0 flex flex-col sm:flex-row items-center sm:justify-between py-4 sm:py-0 px-6 border-b border-white/20 shadow-md">
                    <a href="#">
                        <img class="h-[42px]" src="{{ asset('images/starshop-logo.png') }}" alt="starshop logo">
                    </a>
                    <nav class="flex space-x-4 font-semibold">
                        <a class="hover:text-amber-400 pt-2" href="#">
                            Home
                        </a>
                        <a class="hover:text-amber-400  pt-2" href="#">
                            About
                        </a>
                        <a class="hover:text-amber-400 pt-2" href="#">
                            Contact
                        </a>
                        <a class="rounded-[60px] py-2 px-5 bg-white/10 hover:bg-white/20" href="#">
                            Get Started
                        </a>
                    </nav>
                </header>
                {% block body %}{% endblock %}
            </div>
            <div class="p-5 bg-white/5 mt-3 text-center">
                Made with ❤️ by <a class="text-[#0086C4]" href="https://symfonycasts.com">SymfonyCasts</a>
```

```
            </div>
        </div>
    </body>
</html>
```

Do `homepage.html.twig`...

templates/main/homepage.html.twig

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

{% block title %}Starshop: Beam up some parts!{% endblock %}

{% block body %}
    <main class="flex flex-col lg:flex-row">
        <aside
            class="pb-8 lg:pb-0 lg:w-[411px] shrink-0 lg:block lg:min-h-screen text-white transition-all overflow-hidden px-8 border-b lg:border-b-0 lg:border-r border-white/20"
        >
            <div class="flex justify-between mt-11 mb-7">
                <h2 class="text-[32px] font-semibold">My Ship Status</h2>
                <button>
                    <svg xmlns="http://www.w3.org/2000/svg" width="20" height="20" viewBox="0 0 448 512"><!--!Font Awesome Pro 6.5.1 by @fontawesome - https://fontawesome.com License - https://fontawesome.com/license (Commercial License) Copyright 2024 Fonticons, Inc.--><path fill="#fff" d="M384 96c0-17.7 14.3-32 32-32s32 14.3 32 32V416c0 17.7-14.3 32-32 32s-32-14.3-32-32V96zM9.4 278.6c-12.5-12.5-12.5-32.8 0-45.3l128-128c12.5-12.5 32.8-12.5 45.3 0s12.5 32.8 0 45.3L109.3 224 288 224c17.7 0 32 14.3 32 32s-14.3 32-32 32l-178.7 0 73.4 73.4c12.5 12.5 12.5 32.8 0 45.3s-32.8 12.5-45.3 0l-128-128z"/></svg>
                </button>
            </div>

            <div>
                <div class="flex flex-col space-y-1.5">
                    <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center" style="background: rgba(255, 184, 0, .1);">
                        <div class="rounded-full h-2 w-2 bg-amber-400 blur-[1px] mr-2"></div>
                        <p class="uppercase text-xs">in progress</p>
                    </div>
                    <h3 class="tracking-tight text-[22px] font-semibold">
                        <a class="hover:underline" href="{{ path('app_starship_show', {
                            id: myShip.id
                        }) }}">{{ myShip.name }}</a>
                    </h3>
                </div>
                <div class="flex mt-4">
                    <div class="border-r border-white/20 pr-8">
                        <p class="text-slate-400 text-xs">Captain</p>
                        <p class="text-xl">{{ myShip.captain }}</p>
                    </div>

                    <div class="pl-8">
```

```
                        <p class="text-slate-400 text-xs">Class</p>
                        <p class="text-xl">{{ myShip.class }}</p>
                    </div>
                </div>
            </div>
        </aside>

        <div class="px-12 pt-10 w-full">
            <h1 class="text-4xl font-semibold mb-8">
                Ship Repair Queue
            </h1>

            <div class="space-y-5">
                <!-- start ship item -->
                    <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11 flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
                        <div class="flex justify-center min-[1174px]:justify-start">
                            <img class="h-[83px] w-[84px]" src="/images/status-in-progress.png" alt="Status: in progress">
                            <div class="ml-5">
                                <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center bg-amber-400/10">
                                    <div class="rounded-full h-2 w-2 bg-amber-400 blur-[1px] mr-2"></div>
                                    <p class="uppercase text-xs text-nowrap">in progress</p>
                                </div>
                                <h4 class="text-[22px] pt-1 font-semibold">
                                    <a
                                        class="hover:text-slate-200"
                                        href="#"
                                    >USS LeafyCruiser</a>
                                </h4>
                            </div>
                        </div>
                        <div class="flex justify-center min-[1174px]:justify-start mt-2 min-[1174px]:mt-0 shrink-0">
                            <div class="border-r border-white/20 pr-8">
                                <p class="text-slate-400 text-xs">Captain</p>
                                <p class="text-xl">Jean-Luc Pickles</p>
                            </div>

                            <div class="pl-8 w-[100px]">
                                <p class="text-slate-400 text-xs">Class</p>
                                <p class="text-xl">Garden</p>
                            </div>
                        </div>
```

```
                    </div>
                <!-- end ship item -->
            </div>

            <p class="text-lg mt-5 text-center md:text-left">
                Looking for your next galactic ride?
                <a href="#" class="underline font-semibold">Browse the {{ ships|length * 10 }} starships for sale!</a>
            </p>
        </div>
    </main>
{% endblock %}
```

and finally `show.html.twig`.

templates/starship/show.html.twig

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

{% block title %}{{ ship.name }}{% endblock %}

{% block body %}
<div class="my-4 px-8">
    <a class="bg-white hover:bg-gray-200 rounded-xl p-2 text-black" href="#">
        <svg class="inline text-black" xmlns="http://www.w3.org/2000/svg" height="16" width="14" viewBox="0 0 448 512"><!--!Font Awesome Free 6.5.1 by @fontawesome - https://fontawesome.com License - https://fontawesome.com/license/free Copyright 2024 Fonticons, Inc.--><path fill="#000" d="M9.4 233.4c-12.5 12.5-12.5 32.8 0 45.3l160 160c12.5 12.5 32.8 12.5 45.3 0s12.5-32.8 0-45.3L109.2 288 416 288c17.7 0 32-14.3 32-32s-14.3-32-32-32l-306.7 0L214.6 118.6c12.5-12.5 12.5-32.8 0-45.3s-32.8-12.5-45.3 0l-160 160z"/></svg>
        Back
    </a>
</div>
<div class="md:flex justify-center space-x-3 mt-5 px-4 lg:px-8">
    <div class="flex justify-center">
        <img class="max-h-[300px] md:max-h-[500px]" src="{{ asset('images/purple-rocket.png') }}" alt="purple ship launching">
    </div>
    <div class="space-y-5">
        <div class="mt-8 max-w-xl mx-auto">
            <div class="px-8 pt-8">
                <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center bg-amber-400/10">
                    <div class="rounded-full h-2 w-2 bg-amber-400 blur-[1px] mr-2"></div>
                    <p class="uppercase text-xs">{{ ship.status }}</p>
                </div>

                <h1 class="text-[32px] font-semibold border-b border-white/10 pb-5 mb-5">
                    {{ ship.name }}
                </h1>
                <h4 class="text-xs text-slate-300 font-semibold mt-2 uppercase">Spaceship Captain</h4>
                <p class="text-[22px] font-semibold">{{ ship.captain }}</p>

                <h4 class="text-xs text-slate-300 font-semibold mt-2 uppercase">Class</h4>
                <p class="text-[22px] font-semibold">{{ ship.class }}</p>

                <h4 class="text-xs text-slate-300 font-semibold mt-2 uppercase">Ship Status</h4>
```

```
34                 <p class="text-[22px] font-semibold">30,000 lys to next
   service</p>
35             </div>
36         </div>
37     </div>
38 </div>
39 {% endblock %}
```

Tip

If you copy the files (instead of the file contents), Symfony's cache system may not notice the change and you won't see the new design. If that happens, clear the cache by running `php bin/console cache:clear`.

I'm going to delete the `tutorial/` directory entirely so I don't get confused and edit the wrong templates.

Ok, let's see what this did! Refresh. It looks beautiful! I *love* working inside a nice design. But... some parts are broken. In `homepage.html.twig`, this is our ship repair queue... which looks nice... but there's no Twig code! The status is hardcoded, name is hardcoded and there's no loop.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <main class="flex flex-col lg:flex-row">
// ... lines 7 - 42
        <div class="px-12 pt-10 w-full">
            <h1 class="text-4xl font-semibold mb-8">
                Ship Repair Queue
            </h1>

            <div class="space-y-5">
                <!-- start ship item -->
                    <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11 flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
                        <div class="flex justify-center min-[1174px]:justify-start">
                            <img class="h-[83px] w-[84px]" src="/images/status-in-progress.png" alt="Status: in progress">
                            <div class="ml-5">
                                <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center bg-amber-400/10">
                                    <div class="rounded-full h-2 w-2 bg-amber-400 blur-[1px] mr-2"></div>
                                    <p class="uppercase text-xs text-nowrap">in progress</p>
                                </div>
                                <h4 class="text-[22px] pt-1 font-semibold">
                                    <a
                                        class="hover:text-slate-200"
                                        href="#"
                                    >USS LeafyCruiser</a>
                                </h4>
                            </div>
                        </div>
                        <div class="flex justify-center min-[1174px]:justify-start mt-2 min-[1174px]:mt-0 shrink-0">
                            <div class="border-r border-white/20 pr-8">
                                <p class="text-slate-400 text-xs">Captain</p>
                                <p class="text-xl">Jean-Luc Pickles</p>
                            </div>

                            <div class="pl-8 w-[100px]">
                                <p class="text-slate-400 text-xs">Class</p>
                                <p class="text-xl">Garden</p>
                            </div>
                        </div>
                    </div>
                <!-- end ship item -->
```

```
79              </div>
↕ // ... lines 80 - 84
85          </div>
86      </main>
87  {% endblock %}
```

Next: let's take our new design and make it *dynamic*. We'll also learn how to organize things into template partials *and* introduce a PHP enum, which are fun.

Chapter 15: Twig Partials & for Loops

We just gave our site a design makeover... which means we updated our templates to include HTML elements with a bunch of Tailwind classes. The result? A site that's easy on the eyes.

For some parts of the templates, things are still dynamic: we have Twig code to print out the captain and class. But in other parts, everything is hard-coded. And... this is pretty typical: a frontend developer might code up the site in HTML & Tailwind... but leave it for *you* to make it dynamic and bring it to life.

Organizing into a Template Partial

At the top of `homepage.html.twig`, this long `<aside>` element is the sidebar. It's fine that this code lives in `homepage.html.twig`... but it *does* take up a lot of space! And what if we want to reuse this sidebar on another page?

One great feature of Twig is the ability to take "chunks" of HTML and isolate them into their *own* templates so you can reuse them. These are called template *partials*... since they hold code for just *part* of the page.

Copy this code, and in the `main/` directory - though this could go anywhere - add a new file called `_shipStatusAside.html.twig`. Paste inside.

templates/main/_shipStatusAside.html.twig

```
<aside
    class="pb-8 lg:pb-0 lg:w-[411px] shrink-0 lg:block lg:min-h-screen text-white transition-all overflow-hidden px-8 border-b lg:border-b-0 lg:border-r border-white/20"
>
    <div class="flex justify-between mt-11 mb-7">
        <h2 class="text-[32px] font-semibold">My Ship Status</h2>
        <button>
            <svg xmlns="http://www.w3.org/2000/svg" width="20" height="20" viewBox="0 0 448 512"><!--!Font Awesome Pro 6.5.1 by @fontawesome - https://fontawesome.com License - https://fontawesome.com/license (Commercial License) Copyright 2024 Fonticons, Inc.--><path fill="#fff" d="M384 96c0-17.7 14.3-32 32-32s32 14.3 32 32V416c0 17.7-14.3 32-32 32s-32-14.3-32-32V96zM9.4 278.6c-12.5-12.5-12.5-32.8 0-45.3l128-128c12.5-12.5 32.8-12.5 45.3 0s12.5 32.8 0 45.3L109.3 224 288 224c17.7 0 32 14.3 32 32s-14.3 32-32 32l-178.7 0 73.4 73.4c12.5 12.5 12.5 32.8 0 45.3s-32.8 12.5-45.3 0l-128-128z"/></svg>
        </button>
    </div>

    <div>
        <div class="flex flex-col space-y-1.5">
            <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center" style="background: rgba(255, 184, 0, .1);">
                <div class="rounded-full h-2 w-2 bg-amber-400 blur-[1px] mr-2"></div>
                <p class="uppercase text-xs">in progress</p>
            </div>
            <h3 class="tracking-tight text-[22px] font-semibold">
                <a class="hover:underline" href="{{ path('app_starship_show', {
                    id: myShip.id
                }) }}">{{ myShip.name }}</a>
            </h3>
        </div>
        <div class="flex mt-4">
            <div class="border-r border-white/20 pr-8">
                <p class="text-slate-400 text-xs">Captain</p>
                <p class="text-xl">{{ myShip.captain }}</p>
            </div>

            <div class="pl-8">
                <p class="text-slate-400 text-xs">Class</p>
                <p class="text-xl">{{ myShip.class }}</p>
            </div>
        </div>
    </div>
</aside>
```

Back in `homepage.html.twig`, delete that, then include it with `{{` - so the say something syntax - `include()` and the name of the template: `main/_shipStatusAside.html.twig`.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <main class="flex flex-col lg:flex-row">
        {{ include('main/_shipStatusAside.html.twig') }}
// ... lines 8 - 51
    </main>
{% endblock %}
```

Try it out! And... no change! The `include()` statement is simple:

> *"Render this template and give it the same variables that I have"*

If you're wondering why I prefixed the template with an underscore... no reason! It's just a convention that helps me know that this template holds only a *part* of the page.

Looping over the Ships in Twig

In the homepage template, we can focus on the ship list below, which is this area. Right now, there's just one ship... and it's hard-coded. Our *intention* is to list every ship that we're currently repairing. And we *do* already have a `ships` variable that we're using at the bottom: it's an array of `Starship` objects.

So for the first time in Twig, we need to loop over an array! To do that, I'll remove this comment, and say `{%` - so the *do* something tag - then `for ship in ships`. `ships` is the array variable we already have and `ship` is the *new* variable name in the loop that represents a single `Starship` object. At the bottom, add `{% endfor %}`.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <main class="flex flex-col lg:flex-row">
// ... lines 7 - 8
        <div class="px-12 pt-10 w-full">
// ... lines 10 - 13
            <div class="space-y-5">
                {% for ship in ships %}
// ... lines 16 - 43
                {% endfor %}
            </div>
// ... lines 46 - 50
        </div>
    </main>
{% endblock %}
```

And already... when we try it, we get *three* hard-coded ships! That's an improvement!

Next: it's time for a plot twist that'll lead us to creating a PHP enum.

Chapter 16: PHP Enums

Inside the loop, making things dynamic is nothing new... which is great! For in progress, say `{{ ship.status }}`. When we refresh, it prints! Though, yikes! The statuses are running *way* out of their space. Our data doesn't match the design!

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <main class="flex flex-col lg:flex-row">
// ... lines 7 - 8
        <div class="px-12 pt-10 w-full">
// ... lines 10 - 13
            <div class="space-y-5">
                {% for ship in ships %}
// ... lines 16 - 43
                {% endfor %}
            </div>
// ... lines 46 - 50
        </div>
    </main>
{% endblock %}
```

Plot twist! Someone changed the project's requirements... right in the middle! That "never" happens! The new plan is this: each ship should have a status of `in progress`, `waiting`, or `completed`. Over in `src/Repository/StarshipRepository.php`, our ships *do* have a `status` - it's this argument - but it's a string that can be set to *anything*.

Creating an Enum

So we need to do some refactoring to fit the new plan. Let's think: there are exactly three valid statuses. This a *perfect* use case for a PHP *enum*.

If you're not familiar with enums, they're lovely and a great way to organize a set of statuses - like published, unpublished & draft - or sizes - small, medium or large - or anything similar.

In the `Model/` directory - though this could live anywhere... we're creating the enum for *our* own organization - create a new class and call it `StarshipStatusEnum`. As soon as I typed the word enum, PhpStorm changed the template from `class` to an `enum`. So we're not creating a class, as you can see, we created an `enum`

src/Model/StarshipStatusEnum.php

```
↕ // ... lines 1 - 2
3 namespace App\Model;
4
5 enum StarshipStatusEnum: string
6 {
↕ // ... lines 7 - 9
10 }
```

Add a `: string` to the enum to make what's called a "string-backed enum". We won't go too deep, but this allows us to define each status - like `WAITING` *and* assign that to a string, which will be handy in a minute. Add a status for `IN_PROGRESS` and finally one for `COMPLETED`.

src/Model/StarshipStatusEnum.php

```
↕ // ... lines 1 - 2
3 namespace App\Model;
4
5 enum StarshipStatusEnum: string
6 {
7     case WAITING = 'waiting';
8     case IN_PROGRESS = 'in progress';
9     case COMPLETED = 'completed';
10 }
```

That's it! That's all an enum is: a set of "states" that get centralized in one place.

Next: open up the `Starship` class. The last argument is currently a `string` status. Change it to be a `StarshipStatusEnum`. And at the bottom, the `getStatus` method will now return a `StarshipStatusEnum`.

src/Model/StarshipStatusEnum.php

```
// ... lines 1 - 2
namespace App\Model;

enum StarshipStatusEnum: string
{
    case WAITING = 'waiting';
    case IN_PROGRESS = 'in progress';
    case COMPLETED = 'completed';
}
```

Finally, in `StarshipRepository` where we create each `Starship`, my editor is angry. It says:

> *"Hey! This argument accepts a `StarshipStatusEnum`, but you're passing a string!"*

Let's calm it down. Change this to `StarshipStatusEnum::`... and it autocomplete the choices! Let's make the first one `IN_PROGRESS`. And that *did* add the `use` statement for the `enum` to the top of the class. For the next one, make it `COMPLETED`... and for the last, `WAITING`.

src/Repository/StarshipRepository.php

```
// ... lines 1 - 5
use App\Model\StarshipStatusEnum;
// ... lines 7 - 8
class StarshipRepository
{
// ... lines 11 - 14
    public function findAll(): array
    {
// ... lines 17 - 18
        return [
            new Starship(
// ... lines 21 - 24
                StarshipStatusEnum::IN_PROGRESS
            ),
            new Starship(
// ... lines 28 - 31
                StarshipStatusEnum::COMPLETED
            ),
            new Starship(
// ... lines 35 - 38
                StarshipStatusEnum::WAITING
            ),
        ];
    }
// ... lines 43 - 53
}
```

Refactoring done! Well... *maybe*. When we refresh, busted! It says:

> *"object of class `StarshipStatusEnum` could not be converted to string"*

And it's coming from the `ship.status` Twig call.

That makes sense: `ship.status` is now an enum... which can't be directly printed as a string. The easiest fix, in `homepage.html.twig`, is to add `.value`.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <main class="flex flex-col lg:flex-row">
// ... lines 7 - 8
        <div class="px-12 pt-10 w-full">
// ... lines 10 - 13
            <div class="space-y-5">
                {% for ship in ships %}
                    <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11 flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
                        <div class="flex justify-center min-[1174px]:justify-start">
// ... line 18
                            <div class="ml-5">
                                <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center bg-amber-400/10">
// ... line 21
                                    <p class="uppercase text-xs text-nowrap">{{ ship.status.value }}</p>
                                </div>
// ... lines 24 - 29
                            </div>
                        </div>
// ... lines 32 - 42
                    </div>
                {% endfor %}
            </div>
// ... lines 46 - 50
        </div>
    </main>
{% endblock %}
```

Because we made our enum *string-backed*, it has a `value` property, which will be the string that we assigned to the current status. Try it now. It looks great! In progress, completed, waiting.

Next: let's learn how we can make this *last* change a bit more elegant by creating smarter methods on our `Starship` class. Then we'll put the finishing touches on our design.

Chapter 17: Smart Model Methods & Making the Design Dynamic

Adding the `.value` to the end of the enum to print it works like a charm. But I want to show another, more elegant, solution.

Adding Smarter Model Methods

In `Starship`, it'll probably be common for us to want to get the *string* status of a `Starship`. To make that easier, why not add a shortcut method here called `getStatusString()`? This will return a `string`, and inside, return `$this->status->value`.

src/Model/Starship.php

```
// ... lines 1 - 4
class Starship
{
// ... lines 7 - 40
    public function getStatusString(): string
    {
        return $this->status->value;
    }
}
```

Thanks to this, over in the template, we can shorten to `ship.statusString`.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <main class="flex flex-col lg:flex-row">
// ... lines 7 - 8
        <div class="px-12 pt-10 w-full">
// ... lines 10 - 13
            <div class="space-y-5">
                {% for ship in ships %}
                    <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11 flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
                        <div class="flex justify-center min-[1174px]:justify-start">
// ... line 18
                            <div class="ml-5">
                                <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center bg-amber-400/10">
// ... line 21
                                    <p class="uppercase text-xs text-nowrap">{{ ship.statusString }}</p>
                                </div>
// ... lines 24 - 29
                            </div>
                        </div>
// ... lines 32 - 42
                    </div>
                {% endfor %}
            </div>
// ... lines 46 - 50
        </div>
    </main>
{% endblock %}
```

Oh, and this is *more* Twig smartness! There is *no* property called `statusString` on `Starship`! But Twig doesn't care. It sees that there *is* a `getStatusString()` method and calls that.

Watch: when we refresh, the page still works. I really like this solution, so I'll copy that... and repeat it up here for the `alt` attribute.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <main class="flex flex-col lg:flex-row">
// ... lines 7 - 8
        <div class="px-12 pt-10 w-full">
// ... lines 10 - 13
            <div class="space-y-5">
                {% for ship in ships %}
                    <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11 flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
                        <div class="flex justify-center min-[1174px]:justify-start">
                            <img class="h-[83px] w-[84px]" src="/images/status-in-progress.png" alt="Status: {{ ship.statusString }}">
                            <div class="ml-5">
                                <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center bg-amber-400/10">
// ... line 21
                                    <p class="uppercase text-xs text-nowrap">{{ ship.statusString }}</p>
                                </div>
// ... lines 24 - 29
                            </div>
                        </div>
// ... lines 32 - 42
                    </div>
                {% endfor %}
            </div>
// ... lines 46 - 50
        </div>
    </main>
{% endblock %}
```

And while we're fixing this, in `show.html.twig`, we print the status there too. So I'll make that same change... then close this.

templates/starship/show.html.twig

```
// ... lines 1 - 4
{% block body %}
// ... lines 6 - 11
<div class="md:flex justify-center space-x-3 mt-5 px-4 lg:px-8">
// ... lines 13 - 15
    <div class="space-y-5">
        <div class="mt-8 max-w-xl mx-auto">
            <div class="px-8 pt-8">
                <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center bg-amber-400/10">
// ... line 20
                    <p class="uppercase text-xs">{{ ship.statusString }}</p>
                </div>
// ... lines 23 - 34
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

Finishing our Dynamic Template

Ok: let's finish making our homepage template dynamic: it's smooth space sailing from here. For the ship name, `{{ ship.name }}`, for the captain, `{{ ship.captain }}`. And down here for the class, `{{ ship.class }}`.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <main class="flex flex-col lg:flex-row">
// ... lines 7 - 8
        <div class="px-12 pt-10 w-full">
// ... lines 10 - 13
            <div class="space-y-5">
                {% for ship in ships %}
                    <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11 flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
                        <div class="flex justify-center min-[1174px]:justify-start">
// ... line 18
                            <div class="ml-5">
// ... lines 20 - 23
                                <h4 class="text-[22px] pt-1 font-semibold">
                                    <a
// ... lines 26 - 27
                                    >{{ ship.name }}</a>
                                </h4>
                            </div>
                        </div>
                        <div class="flex justify-center min-[1174px]:justify-start mt-2 min-[1174px]:mt-0 shrink-0">
                            <div class="border-r border-white/20 pr-8">
                                <p class="text-slate-400 text-xs">Captain</p>
                                <p class="text-xl">{{ ship.captain }}</p>
                            </div>

                            <div class="pl-8 w-[100px]">
                                <p class="text-slate-400 text-xs">Class</p>
                                <p class="text-xl">{{ ship.class }}</p>
                            </div>
                        </div>
                    </div>
                {% endfor %}
            </div>
// ... lines 46 - 50
        </div>
    </main>
{% endblock %}
```

Oh, and let's fill in the link: `{{ path() }}` then the name of the route. We're linking to the show page for the ship, so the route is `app_starship_show`. And because this has an `id` wildcard, pass `id` set to `ship.id`.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <main class="flex flex-col lg:flex-row">
// ... lines 7 - 8
        <div class="px-12 pt-10 w-full">
// ... lines 10 - 13
            <div class="space-y-5">
                {% for ship in ships %}
                    <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11 flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
                        <div class="flex justify-center min-[1174px]:justify-start">
// ... line 18
                            <div class="ml-5">
// ... lines 20 - 23
                                <h4 class="text-[22px] pt-1 font-semibold">
                                    <a
                                        class="hover:text-slate-200"
                                        href="{{ path('app_starship_show', { id: ship.id }) }}"
                                    >{{ ship.name }}</a>
                                </h4>
                            </div>
                        </div>
// ... lines 32 - 42
                    </div>
                {% endfor %}
            </div>
// ... lines 46 - 50
        </div>
    </main>
{% endblock %}
```

And now, so much better! It looks nice and we can click these links.

Dynamic Image Paths

But... the image is still broken. Earlier, when we copied the images into our `assets/` directory, I included three files for the three statuses. Up here, we *are* "kind of" pointing to the in progress status... but this isn't the right way to reference images in the `assets/` directory. Instead, say `{{ asset() }}` and pass the path relative to the `assets/` directory, called the "logical" path.

If we try that now... we're closer. But the "in progress" part needs to be *dynamic* based on the status. One thing we could try is Twig concatenation: to add `ship.status` to the string. That *is* possible, though it's a bit ugly.

Instead, let's revisit the solution we used a minute ago: making all the data about our `Starship` easily accessible... *from* the `Starship` class.

Here's what I mean: add a `public function getStatusImageFilename()` that returns a string.

src/Model/Starship.php

```
// ... lines 1 - 4
class Starship
{
// ... lines 7 - 45
    public function getStatusImageFilename(): string
    {
// ... lines 48 - 52
    }
}
```

Let's do all the logic for creating the filename right here. I'll paste in a `match` function.

This says: check `$this->status` and if it's equal to `WAITING`, return this string. If it's equal to `IN_PROGRESS` return this string and so on.

src/Model/Starship.php

```
// ... lines 1 - 4
class Starship
{
// ... lines 7 - 45
    public function getStatusImageFilename(): string
    {
        return match ($this->status) {
            StarshipStatusEnum::WAITING => 'images/status-waiting.png',
            StarshipStatusEnum::IN_PROGRESS => 'images/status-in-progress.png',
            StarshipStatusEnum::COMPLETED => 'images/status-complete.png',
        };
    }
}
```

And exactly like before, because we have a `getStatusImageFilename()` method, we get to, in Twig, *pretend* like we have a `statusImageFilename` property.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <main class="flex flex-col lg:flex-row">
// ... lines 7 - 8
        <div class="px-12 pt-10 w-full">
// ... lines 10 - 13
            <div class="space-y-5">
                {% for ship in ships %}
                    <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11 flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
                        <div class="flex justify-center min-[1174px]:justify-start">
                            <img class="h-[83px] w-[84px]" src="{{ asset(ship.statusImageFilename) }}" alt="Status: {{ ship.statusString }}">
// ... lines 19 - 30
                        </div>
// ... lines 32 - 42
                    </div>
                {% endfor %}
            </div>
// ... lines 46 - 50
        </div>
    </main>
{% endblock %}
```

And now, we got it!

Last Details of Making the Design Dynamic

Final things! Let's fill in some missing links, like this logo should go to the homepage. Right now... it goes nowhere.

Remember, when we want to link to a page, we need to make sure *that* route has a name. In `src/Controller/MainController.php`... our homepage does *not* have a name. Ok, it has an auto-generated name, but we don't want to rely on that.

Add `name:` set to `app_homepage`. Or you could use `app_main_homepage`.

src/Controller/MainController.php

```
// ... lines 1 - 9
class MainController extends AbstractController
{
    #[Route('/', name: 'app_homepage')]
    public function homepage(StarshipRepository $starshipRepository): Response
// ... lines 14 - 22
}
```

To link the logo, in `base.html.twig`... here it is... Use `{{ path('app_homepage') }}`.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 13
    <body class="text-white" style="background: radial-gradient(102.21% 102.21% at 50% 28.75%, #00121C 42.62%, #013954 100%);">
        <div class="flex flex-col justify-between min-h-screen relative">
            <div>
                <header class="h-[114px] shrink-0 flex flex-col sm:flex-row items-center sm:justify-between py-4 sm:py-0 px-6 border-b border-white/20 shadow-md">
                    <a href="{{ path('app_homepage') }}">
// ... line 19
                    </a>
// ... lines 21 - 34
                </header>
// ... line 36
            </div>
// ... lines 38 - 40
        </div>
    </body>
</html>
```

Copy that and repeat it below for another home link.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 13
    <body class="text-white" style="background: radial-gradient(102.21% 102.21% at 50% 28.75%, #00121C 42.62%, #013954 100%);">
        <div class="flex flex-col justify-between min-h-screen relative">
            <div>
                <header class="h-[114px] shrink-0 flex flex-col sm:flex-row items-center sm:justify-between py-4 sm:py-0 px-6 border-b border-white/20 shadow-md">
                    <a href="{{ path('app_homepage') }}">
// ... line 19
                    </a>
                    <nav class="flex space-x-4 font-semibold">
                        <a class="hover:text-amber-400 pt-2" href="{{ path('app_homepage') }}">
                            Home
                        </a>
// ... lines 25 - 33
                    </nav>
                </header>
// ... line 36
            </div>
// ... lines 38 - 40
        </div>
    </body>
</html>
```

We'll leave these other links for a future tutorial.

Back at the browser, click that logo! All good. The final missing link is over on the show page. This "back" link should *also* go to the homepage. Open up `show.html.twig`. And up on top - there it is - I'll paste that same link.

templates/starship/show.html.twig

```
// ... lines 1 - 4
{% block body %}
<div class="my-4 px-8">
    <a class="bg-white hover:bg-gray-200 rounded-xl p-2 text-black" href="{{ path('app_homepage') }}">
        <svg class="inline text-black" xmlns="http://www.w3.org/2000/svg" height="16" width="14" viewBox="0 0 448 512"><!--!Font Awesome Free 6.5.1 by @fontawesome - https://fontawesome.com License - https://fontawesome.com/license/free Copyright 2024 Fonticons, Inc.--><path fill="#000" d="M9.4 233.4c-12.5 12.5-12.5 32.8 0 45.3l160 160c12.5 12.5 32.8 12.5 45.3 0s12.5-32.8 0-45.3L109.2 288 416 288c17.7 0 32-14.3 32-32s-14.3-32-32-32l-306.7 0L214.6 118.6c12.5-12.5 12.5-32.8 0-45.3s-32.8-12.5-45.3 0l-160 160z"/></svg>
        Back
    </a>
</div>
// ... lines 12 - 38
{% endblock %}
```

Ok team, the design is done! Congrats! Treat yourself to a tea... or latte... or donut or a walk amongst nature to celebrate. Because this is huge! Our site *looks* and feels *real*. I'm *thrilled*.

Now we can focus on the finer details. Like, when we click this link, the sidebar is *supposed* to collapse. To handle that, I want to introduce you to my favorite tool for writing JavaScript: Stimulus.

Chapter 18: Stimulus: Writing Pro JavaScript

We know how to write HTML in our templates. And we're handling CSS with Tailwind. What about JavaScript? Well, like with CSS, there's an `app.js` file, and it's already included on the page. So you can put whatever JavaScript you want right here.

But I highly recommend using a small, but mean, JavaScript library called Stimulus. It is one of my absolute favorite things on the Internet. You take a part of your existing HTML and *connect* it to a small JavaScript file, called a controller. This allows you to add behavior: like when you click this button, the `greet` method on the controller will be called.

And that's really it! Sure, Stimulus has more features, but you already understand the core of how it works. Despite its simplicity, this will let us build any JavaScript and user interface functionality we need, in a reliable and predictable way. So let's get it installed.

Installing Stimulus

Stimulus is a *JavaScript* library, but Symfony has a bundle that helps integrate it. Over at your terminal, if you want to see what the recipe does, commit your changes. I already have. Then run:

```
composer require symfony/stimulus-bundle
```

When this finishes... the recipe *did* make some changes. Let's walk through the important ones. The first is in `app.js`: our main JavaScript file. Open that up, there we go.

assets/app.js

```
import './bootstrap.js';
/*
 * Welcome to your app's main JavaScript file!
 *
 * This file will be included onto the page via the importmap() Twig function,
 * which should already be in your base.html.twig.
 */
import './styles/app.css';

console.log('This log comes from assets/app.js - welcome to AssetMapper! 🎉');
```

It added an `import` on top - `./bootstrap.js` - to a new file that lives right *next* to this.

assets/bootstrap.js

```
import { startStimulusApp } from '@symfony/stimulus-bundle';

const app = startStimulusApp();
// register any custom, 3rd party controllers here
// app.register('some_controller_name', SomeImportedController);
```

The purpose of this file is to start the Stimulus engine. Also, in `importmap.php`, the recipe added the `@hotwired/stimulus` JavaScript package along with another file that helps boot up Stimulus inside Symfony.

importmap.php

```
// ... lines 1 - 15
return [
// ... lines 17 - 20
    '@hotwired/stimulus' => [
        'version' => '3.2.2',
    ],
    '@symfony/stimulus-bundle' => [
        'path' => './vendor/symfony/stimulus-bundle/assets/dist/loader.js',
    ],
];
```

Finally, the recipe created an `assets/controllers/` directory. This is where *our* custom controllers will live. And it included a demo controller to get us started! Thanks!

assets/controllers/hello_controller.js

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

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

These controller files *do* have an important naming convention. Because this is called `hello_controller.js`, to connect this with an element on the page, we'll use `data-controller="hello"`.

How Stimulus Works

So here's how this works. As soon as Stimulus sees an element on the page with `data-controller="hello"`, it will instantiate a new instance of this controller and call the `connect()` method. So, this `hello` controller should automatically and instantly change the content of the element it's attached to.

And we can already see this. Refresh the page. Stimulus is *now* active on our site. This means it's watching for elements with `data-controller`. Let's do something wild: inspect element on the page, find *any* element - like this anchor tag - and add `data-controller="hello"`. Watch what happens when I click off to activate this change. Boom! Stimulus saw that element, instantiated our controller and called the `connect()` method. And you can do this as many times as you want on the page.

The point is: no matter *how* a `data-controller` element get on your page, Stimulus sees it. So if we make an Ajax call that returns HTML and put that onto the page... yeah, Stimulus is going to see that and our JavaScript is going to work. That's the *key*: when you write JavaScript

with Stimulus, your JavaScript will *always* work, no matter how and when that HTML is added to the page.

Creating a closeable Stimulus Controller

So let's use Stimulus to power our close button. Over in the `assets/controller/` directory, duplicate `hello_controller.js` and make a new one called `closeable_controller.js`.

I'll clear out almost everything and get down to the absolute basics: import `Controller` from stimulus... then create a class that extends it.

assets/controllers/closeable_controller.js

```
1  import { Controller } from '@hotwired/stimulus';
2
3  export default class extends Controller {
↕  // ... lines 4 - 6
7  }
```

This doesn't *do* anything, but we can already attach it to an element on the page. Here's the plan: we're going to attach the controller to the entire `aside` element. Then, when we click this button, we'll *remove* the `aside`.

That element lives over in `templates/main/_shipStatusAside.html.twig`. To attach the controller, add `data-controller="closeable"`. Oh, see that autocompletion? That comes from a Stimulus plugin for PhpStorm.

templates/main/_shipStatusAside.html.twig

```
1  <aside
↕  // ... line 2
3      data-controller="closeable"
4  >
↕  // ... lines 5 - 35
36 </aside>
```

If we move over and refresh, nothing will happen yet: the close button doesn't work. But open your browser's console. Nice! Stimulus adds helpful debugging messages: that it's starting and then - importantly `closeable initialize`, `closeable connect`.

This means that it *did* see the `data-controller` element and initialized that controller.

So back to our goal: when we click this button, we want to call code inside the closeable controller that will remove the `aside`. *In* `closeable_controller.js`, add a new method called, how about, `close()`. Inside, say `this.element.remove()`.

assets/controllers/closeable_controller.js

```
// ... lines 1 - 2
export default class extends Controller {
    close() {
        this.element.remove();
    }
}
```

In Stimulus, `this.element` will always be whatever element your controller is attached to. So, this `aside` element. But otherwise, this code is standard JavaScript: every Element has a `remove()` method.

To call the `close()` method, on the button, add `data-action=""` then the name of our controller - `closeable` - a `#` sign, and the name of the method: `close`.

templates/main/_shipStatusAside.html.twig

```
<aside
// ... line 2
    data-controller="closeable"
>
    <div class="flex justify-between mt-11 mb-7">
// ... line 6
        <button data-action="closeable#close">
            <svg xmlns="http://www.w3.org/2000/svg" width="20" height="20" viewBox="0 0 448 512"><!--!Font Awesome Pro 6.5.1 by @fontawesome - https://fontawesome.com License - https://fontawesome.com/license (Commercial License) Copyright 2024 Fonticons, Inc.--><path fill="#fff" d="M384 96c0-17.7 14.3-32 32-32s32 14.3 32 32V416c0 17.7-14.3 32-32 32s-32-14.3-32-32V96zM9.4 278.6c-12.5-12.5-12.5-32.8 0-45.3l128-128c12.5-12.5 32.8-12.5 45.3 0s12.5 32.8 0 45.3L109.3 224 288 224c17.7 0 32 14.3 32 32s-14.3 32-32 32l-178.7 0 73.4 73.4c12.5 12.5 12.5 32.8 0 45.3s-32.8 12.5-45.3 0l-128-128z"/></svg>
        </button>
    </div>
// ... lines 11 - 35
</aside>
```

Animating the Close

That's it! Testing time. Click! Gone! But I want it be fancier! I want it to animate when closing instead of being instant. Can we do that? Sure! And we don't need much JavaScript... because modern CSS is amazing.

Over on the `aside` element, add a new CSS class - it could go anywhere - called `transition-all`.

That's a Tailwind class that activates CSS transitions. This means that if certain *style* properties change - like the width suddenly being set to 0 - it will *transition* that change, instead of instantly changing it.

Also add `overflow-hidden` so that, as the width gets smaller, it doesn't create a weird scroll bar.

If we try this now, it still closes instantly. That's because there's nothing to *transition*: we're not changing the width... just removing the element.

But watch this. Inspect Element and find the `aside`: here it is. Manually change the width to 0. Cool! You go tiny, big, tiny, big, tiny! The CSS side of things *is* working.

Back in our controller, instead of removing the element, we need to change the width to zero, *wait* for the CSS transition to finish, *then* remove the element. We can do the first with `this.element.style.width = 0`.

templates/main/_shipStatusAside.html.twig

```
<aside
// ... line 2
    data-controller="closeable"
>
    <div class="flex justify-between mt-11 mb-7">
// ... line 6
        <button data-action="closeable#close">
            <svg xmlns="http://www.w3.org/2000/svg" width="20" height="20" viewBox="0 0 448 512"><!--!Font Awesome Pro 6.5.1 by @fontawesome - https://fontawesome.com License - https://fontawesome.com/license (Commercial License) Copyright 2024 Fonticons, Inc.--><path fill="#fff" d="M384 96c0-17.7 14.3-32 32-32s32 14.3 32 32V416c0 17.7-14.3 32-32 32s-32-14.3-32-32V96zM9.4 278.6c-12.5-12.5-12.5-32.8 0-45.3l128-128c12.5-12.5 32.8-12.5 45.3 0s12.5 32.8 0 45.3L109.3 224 288 224c17.7 0 32 14.3 32 32s-14.3 32-32 32l-178.7 0 73.4 73.4c12.5 12.5 12.5 32.8 0 45.3s-32.8 12.5-45.3 0l-128-128z"/></svg>
        </button>
    </div>
// ... lines 11 - 35
</aside>
```

The *tricky* part is *waiting* for the CSS transition to finish *before* removing the element. To help with that, I'm going to paste a method at the bottom of our controller.

assets/controllers/closeable_controller.js

```
// ... lines 1 - 2
export default class extends Controller {
    async close() {
        this.element.style.width = '0';
// ... lines 6 - 8
    }

    #waitForAnimation() {
        return Promise.all(
            this.element.getAnimations().map((animation) => animation.finished),
        );
    }
}
```

If you're not familiar, the `#` sign makes this a private method in JavaScript: a small detail. This code looks fancy, but it has a simple job: to ask the element to tell us when all of its CSS animations are finished.

Thanks to that, up here, we can say `await this.#waitForAnimation()`. And whenever you use `await`, you need to put `async` on the function around this. I won't go into details about

`async`, but that won't change how our code works.

assets/controllers/closeable_controller.js

```
// ... lines 1 - 2
export default class extends Controller {
    async close() {
        this.element.style.width = '0';

        await this.#waitForAnimation();
        this.element.remove();
    }

    #waitForAnimation() {
        return Promise.all(
            this.element.getAnimations().map((animation) => animation.finished),
        );
    }
}
```

Let's check the result! Refresh. And... I absolutely *love* that.

Next up, everyone wants a single page application, right? A site where there are zero full page refreshes. But to build one, don't we need to use a JavaScript framework like React? No! We're going to transform our app into a single page application in... about 3 minutes with Turbo.

Chapter 19: Turbo: Your Single Page App

When I build a UI, I want it to be beautiful, interactive, and smooth. Personally, I choose *not* to use front-end frameworks like React or Vue or Next. But you *can*... and there's nothing wrong with them: those are great tools. Also, building an API in Symfony is awesome!

But if you want to build your HTML in Twig - like I love doing - we can absolutely have a super-rich, responsive, interactive user interface!

One *big* piece of a fancy interface is removing full-page reloads. Right now, when I click around, watch: it's fast, but these are full-page reloads. Those don't happen if you use something like React or Vue.

To eliminate those, we're going to use another library from the same people that made Stimulus called Turbo. Turbo can do a *lot* of things, but its main job is to eliminate full-page refreshes. Like Stimulus, it's a JavaScript library. And also like Stimulus, Symfony has a bundle that helps integrate it.

Installing Turbo

Find your terminal and run:

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

This time, the recipe made two interesting changes. I'll show you. The first is in `importmap.php`: it added the `@hotwired/turbo` JavaScript package.

importmap.php

```
// ... lines 1 - 15
return [
// ... lines 17 - 26
    '@hotwired/turbo' => [
        'version' => '7.3.0',
    ],
];
```

The second change is in `assets/controllers.json`. We didn't talk about this file before, but it was added by the StimulusBundle recipe: it's a way to activate Stimulus controllers that live inside third-party packages.

assets/controllers.json

```
{
    "controllers": {
        "@symfony/ux-turbo": {
            "turbo-core": {
                "enabled": true,
                "fetch": "eager"
            },
            "mercure-turbo-stream": {
                "enabled": false,
                "fetch": "eager"
            }
        }
    },
    "entrypoints": []
}
```

So the `symfony/ux-turbo` PHP package we just installed has a JavaScript controller inside called `turbo-core`. And because we have `enabled: true` here, it means that controller is now registered and available: it's as if it lived in our `assets/controllers/` directory.

Now we're not going to *use* this controller directly - we're not going to attach it to an element. But the fact that it's being loaded & registered with Stimulus is enough to *activate* Turbo on our site.

Full Page Refreshes Gone

What the heck does that mean? It's like magic: give the page a refresh, and bam! Full-page reloads vanish! Watch up here: when I click back, you won't see it reload. Boom! It's super fast

and all happening via Ajax.

Here's how it works. When we click this link, Turbo intercepts the click and, instead of a full page reload, it makes an Ajax call to that page. That Ajax call returns the full HTML for that page and then Turbo puts that onto *this* page.

That small thing transforms our project into a single page application and makes a big difference with how fast our site feels.

AJAX Calls & the Web Debug Toolbar

But there's one more thing. I'll refresh so we can see it. Whenever you make an Ajax call in a Symfony app - whether it's via Turbo or any other way - the Web Debug Toolbar *notices* that. Watch right around here when I click. Check that out! We have a running list of all the Ajax calls made on this page. And if we want to see the profiler for any of those Ajax requests, we can click the link.

And yeah... there we are. Here's the Ajax request that was made for the homepage. Though with Turbo, you don't even need to rely on this trick because, as we click around, this entire bar is replaced by the new Web Debug Toolbar for the page.

Oh, and get this: in Turbo 8, which is out now, your site will feel even *faster*. That's thanks to a new feature called Instant Click. With this, when you *hover* over a link, Turbo makes an Ajax call to that page *before* you click. Then, when you *do* click, it loads instantly... or at least has a head start.

Turbo has a lot of other features, and we use a *bunch* of them in our LAST Stack Tutorial where we build a frontend with popovers, modals, toast notifications, and more.

Turbo Requires Good JavaScript

But one note about Turbo. Because full page reloads are gone, your JavaScript needs to be built in a way to handle that. A lot of JavaScript expects full page reloads... and if HTML is suddenly added to the page *without* a reload, it breaks. The good news is that if you write your JavaScript in Stimulus, you're good.

Watch. No matter how we get to the homepage, our JavaScript to close the sidebar just keeps working.

Alright squad, we're on the home stretch! Before we finish, I want to do one last bonus chapter where we play with Symfony's awesome generation tool: MakerBundle.

Chapter 20: Maker Bundle: Let's Generate Some Code!

Hats off for nearly making it through the first Symfony tutorial. You've taken a huge step toward building whatever you want on the web. To celebrate, I want to play with MakerBundle: Symfony's awesome tool for code generation.

Composer require vs require-dev

Let's get it installed:

```
composer require symfony/maker-bundle --dev
```

We haven't seen that `--dev` flag yet, but it's not *that* important. Move over and open `composer.json`. Thanks to the flag, instead of `symfony/maker-bundle` going under the `require` key, it was added down here under `require-dev`.

composer.json

```
1  {
↕  // ... lines 2 - 84
85     "require-dev": {
↕  // ... line 86
87         "symfony/maker-bundle": "^1.52",
↕  // ... lines 88 - 89
90     }
91 }
```

By default, when you run `composer install`, it will download everything under *both* `require` and `require-dev`. But `require-dev` is meant for packages that *don't* need to be available on production: packages that you only need when you're developing locally. That's because, when you do deploy, if you want, you can tell Composer:

> *"Hey! Only install the packages under my `require` key: don't install the `require-dev` stuff."*

That can give you a small performance boost on production. But mostly, it's not a big deal.

The Maker Commands

Now, we just installed a *bundle*. Do you remember the main thing that bundles give us? That's right: *services*. This time, the services that MakerBundle gave us are services that provide new *console* commands. Drumroll please. Run:

```
php bin/console
```

Or, actually, I'll start running `symfony console`, which is the same thing. Thanks to the new bundle, we have a ton of commands that start with `make`! Commands for generating a security system, making a controller, generating doctrine entities to talk to the database, forms, listeners, a registration form.... lots and lots of stuff!

Generating a Console Command

Let's use one of these to make our *own* custom console command. Run:

```
symfony console make:command
```

This will interactively ask us about our command. Let's call it: `app:ship-report`. Done!

This created exactly one file: `src/Command/ShipReportCommand.php`. Let's go check that out!

src/Command/ShipReportCommand.php

```
// ... lines 1 - 2
namespace App\Command;

use Symfony\Component\Console\Attribute\AsCommand;
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\Console\Style\SymfonyStyle;

#[AsCommand(
    name: 'app:ship-report',
    description: 'Add a short description for your command',
)]
class ShipReportCommand extends Command
{
    public function __construct()
    {
        parent::__construct();
    }

    protected function configure(): void
    {
        $this
            ->addArgument('arg1', InputArgument::OPTIONAL, 'Argument description')
            ->addOption('option1', null, InputOption::VALUE_NONE, 'Option description')
        ;
    }

    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $io = new SymfonyStyle($input, $output);
        $arg1 = $input->getArgument('arg1');

        if ($arg1) {
            $io->note(sprintf('You passed an argument: %s', $arg1));
        }

        if ($input->getOption('option1')) {
            // ...
        }

```

```
45          $io->success('You have a new command! Now make it your own! Pass --help
    to see your options.');
46
47          return Command::SUCCESS;
48      }
49  }
```

Cool! This is a normal class - it *is* a service, by the way - but with an *attribute* above: `#[AsCommand]`. This tells Symfony:

> *"Yo! See this service? It's not just a service: I would like you to include it in the list of console commands."*

The attribute includes the name of the command and a description. Then the class itself has a `configure()` method where we can add arguments and options. But the main part is that, when somebody *calls* this command, Symfony will call `execute()`.

This `$io` variable is cool. It lets us output things - like `$this->note()` or `$this->success()` - with different styles. And though we don't see it here, we can also ask the user questions interactively.

The best part? *Just* by creating this class, it's ready to use! Try it out:

```
symfony console app:ship-report
```

That's so cool! The message down here comes from the success message at the bottom of the command. And thanks to `configure()`, we have one *argument* called `arg1`. Arguments are string that we pass *after* the command, like:

```
symfony console app:ship-report ryan
```

It says:

> *"You passed an argument: ryan"*

... which comes from this spot in the command.

Building a Progress Bar

There are a *lot* of fun things you can do with commands... and I want to play with one of them. One of the superpowers of the `$io` object is to create animated progress bars.

Imagine we're building a ship report... and it requires some heavy queries. So we want to show a progress bar on the screen. To do that, say `$io->progressStart()` and pass it however many rows of data we're looping through and handling. Let's pretend we're looping over 100 rows of data for this report.

Instead of looping over real data, create a fake loop with `for`. I'm even going to include the `$i` variable in the middle! Inside, to advance the progress bar, say `$io->advance()`. Then, here is where we would do our heavy query or heavy work. Fake that with a `usleep(10000)` to create a short pause.

After the loop, finish with `$io->progressFinish()`.

src/Command/ShipReportCommand.php

```
↕ // ... lines 1 - 16
17 class ShipReportCommand extends Command
18 {
↕ // ... lines 19 - 31
32     protected function execute(InputInterface $input, OutputInterface $output):
   int
33     {
↕ // ... lines 34 - 44
45         $io->progressStart(100);
46         for ($i = 0; $i < 100; ++$i) {
47             $io->progressAdvance();
48             usleep(10000);
49         }
50         $io->progressFinish();
↕ // ... lines 51 - 54
55     }
56 }
```

That's it! Spin over and give that a try:

```
symfony console app:ship-report ryan
```

Oh, that is *so* cool.

And... that's it people! Give yourself a high five... or, better, surprise a co-worker with a jumping high five! Then celebrate with a well-deserved beer, tea, walk around the block or frisbee match with your dog. Because... you did it! You took the first big step into being dangerous with Symfony. Then, come back and try this stuff out: play with it, build a blog, create a few static pages, *anything*. That will make a huge difference.

And if you ever have any questions, we watch the comment section below each video closely and answer everything. Also keep going! In the next tutorial, we're going to become even *more* dangerous by diving deeper into Symfony's configuration and services: the systems that drive *everything* you'll do in Symfony.

Alright, friends, see you next time!

With <3 from SymfonyCasts