AssetMapper: Modern JS with Zero Build System

Chapter 1: A World without Build Systems?

Whoa, hey! Welcome to my frontend laboratory where we're going to do something that I *honestly* thought I would *never* do again. Something bold! Something... maybe just a bit crazy. We're going to write a *modern* frontend with *zero* build system.

How we got here

Back-story time! 7 years ago I was talking about how modern JavaScript *requires* a build system. I was shouting to the world that we needed to transition from creating JavaScript and CSS files in a "public" directory towards *building* them with a system, like Webpack or Vite.

These build systems were created because browsers didn't support modern features that we wanted to use. I'm talking about the `import` statement, `const`, the class syntax, and so on. If you tried to run this kind of JavaScript in a browser, you would have been greeted with sad error messages.

So, the build system would *transpile* (that's a fancy word for "convert") that *new* looking JavaScript to *old* looking JavaScript, so it could run in the browser. It would also combine JavaScript and CSS files, so we would have fewer requests, it could create versioned filenames, process TypeScript and JSX, Sass, and *much* more.

These systems are *incredibly* powerful. But they also add complexity and can slow down coding. So I'm here, 7 years later to say that... we might *not* need those build systems anymore! In this tutorial, we're going to write *all* the modern JavaScript that we know and love... but with *zero* build system, and no Node. Just you and the browser: the way the Gods of the Internet intended it.

Is this for Every Project?

Now, I admit, doing this won't be the best option for *every* project. If you want to use TypeScript, or you're using React, Vue or Next.js, you'll probably *still* want a build system... and you should probably use *their* build system. Skipping a build system *also* means no automatic tree-shaking - if you know and care about that - though we'll learn how that *can* still work.

For the most part, coding with and without a build system is identical, but I'll point out the small differences along the way. And if you're wondering about things like Sass preprocessors, or Tailwind, you can *totally* use those and we'll see how. The final site is also going to be *as* performant and fast as one built with a build system.

Project Setup

Okay, let's get to work! Coding without a build system is a *joy*: no node *or* batteries required. So you should *absolutely* download the course code from this page and code along with me. After you unzip that file, you'll have a `start/` directory with the same code that you see here. Pop open this `README.md` file. As usual, it holds all the setup details you'll need. I've done most of them already. The *last* step is to find your terminal, move into the project, and run:

```
symfony serve -d
```

to use the `symfony` binary to start the built-in web server. I'll hold "command", click and... *hello* Mixed Vinyl! But *wow* is this thing weird and ugly-looking.

This is a Symfony 6.3 project - the same project we've built in the Symfony series. It has Doctrine installed... but there's nothing particularly special about it, and right now, it has literally *zero* CSS and JavaScript. There's no `assets/` directory and nothing hiding inside the `public/` directory.

The first thing I want to explore is the reality that our browser can handle *more* modern stuff than we might realize... certainly much more than *I* realized a few months ago. Let's see what all the hype is about by taking our browser for a modern JavaScript test drive next.

Chapter 2: Doing Modern JS Right in your Browser

Before we talk about *anything* related to Symfony, we're going to strip things down to the bare minimum and prove that we *can* code modern JavaScript, right in our browser.

Directly Loading Some JavaScript

Go *directly* into the `public/` directory and create a new `app.js` file. To start, just `console.log()` a message.

This won't be processed by Symfony or *anything*. In `templates/base.html.twig`, up here in the `javascripts` block, though that doesn't make any difference, add a boring `<script>` tag for this: `<script src="{{ asset('app.js') }}">`. I *am* using the `asset()` function... but that's not doing anything either.

templates/base.html.twig

```
// ... lines 1 - 2
    <head>
// ... lines 4 - 14
        {% block javascripts %}
            <script src="{{ asset('app.js') }}"></script>
// ... line 17
        {% endblock %}
    </head>
// ... lines 20 - 69
```

Ok, head to the browser, open up your Console and... refresh. There's the log! It's snooze-worthy, but working.

Writing Modern JavaScript

Time to make things interesting! Back in `app.js`, copy the mix name. Let's create a *class*: `class MixedVinyl`, with a constructor and some properties. This uses the class syntax introduced in ES6, or ECMAScript 6... basically version "6" of JavaScript. You'll hear ES6 a lot because *most* modern features you're used to came from this version - released *way* back in 2015.

public/app.js

```
class MixedVinyl {
    constructor(title, year) {
        this.title = title;
        this.year = year;
    }

    describe() {
// ... line 8
    }
}
// ... lines 11 - 14
```

In the `describe()` method, I'm leveraging string interpolation - another modern feature from ES6 - to return the string. Below, use this: `const` - yet *another* ES6 feature - `mix = new MixedVinyl()` and pass in the mix name and year. Finally, `console.log(mix.describe())`.

public/app.js

```
class MixedVinyl {
// ... lines 2 - 6
    describe() {
        return `${this.title} was released in ${this.year}`;
    }
}

const mix = new MixedVinyl('Awesome Mix Vol. 1', 2014);
console.log(mix.describe());
```

Cool! *This* is the kind of code I like to write every day. Unfortunately, this is *also* the kind of code that browsers have historically choked on!

So, *normally*, we would have a build system like Encore that would read this modern code and rewrite it to *old* JavaScript... so it would work in our browser. But... tada! It *already* works in our browser! We don't need to do *anything*. And that's not just because I'm using a new browser. This is going to work in *every* browser.

If you're ever unsure, go to https://caniuse.com to check it out. Let's look up "ES6 class". Yup, it's basically supported by *everything*... except for IE 11, which is dead.

Using "import" in the Browser

But what about the `import` statement? Copy the `class MixedVinyl` then create another file directly inside `public/` called `vinyl.js`. Paste this in and then `export` it: `export default class`.

```
public/vinyl.js
```

```
export default class {
    constructor(title, year) {
        this.title = title;
        this.year = year;
    }

    describe() {
        return `${this.title} was released in ${this.year}`;
    }
}
// ... lines 11 - 12
```

Back over in `app.js`, `import MixedVinyl from` and, just like we do in Encore, use the relative path: `./vinyl.js`.

```
public/app.js
```

```
import MixedVinyl from './vinyl.js';
// ... lines 2 - 5
```

Though, notice that I *am* including the `.js` file extension... which you *can* do in Encore, but it's not required. More on that later - but this *was* on purpose.

Importing as a Module

So... does my browser support the `import` statement? Let's find out! Refresh. Booo:

> *“Cannot use import statement outside a module”*

Ok, not a "code red" kind of boo, more like a "code orange". Head back to `base.html.twig`. When you hear the word "module", it's referring to files that leverage `export` and `import`. And if you want your JavaScript to be able to *use* these, you need to load the original *file* "as a module". It's a simple change. Copy the `asset()` function and *now* say

`<script type="module">`. Then, instead of `src`, inside, we're going to write some JavaScript to `import` our `app.js` file.

```
templates/base.html.twig
// ... lines 1 - 2
    <head>
// ... lines 4 - 14
        {% block javascripts %}
            <script type="module">import '{{ asset('app.js') }}';</script>
// ... line 17
        {% endblock %}
    </head>
// ... lines 20 - 69
```

This may look nutty at first, but... we're simply importing the path to our `app.js` file. By doing this, `app.js` will execute *exactly* like it did before... but as a "module"... which just means that `import` and `export` statements "should" work.

Do they? They do! OMG, our browser supports the `import` statement!

Importing 3rd Party Package URLs

We can *even* import third-party packages. To find one, I'm going to use my favorite CDN: "jsDelivr". We'll be using this quite a bit throughout the tutorial. But you don't need to *use* jsDelivr's CDN in your final code. It's a mirror of *every* NPM package... and so it's a convenient place to find what we need.

Search for the popular "lodash" package. When we select it, it shows us a `<script>` tag we could use. Click on "ESM", which is short for ECMAScript modules. When you're coding with imports and exports, you want the *ESM* version of a package: it's a version that properly "exports" modules.

Now check out that `script` tag:

```
<script type="module">
import lodash from '[...]'
</script>
```

That looks *very* similar to the code we have over here! We won't use this exactly, but I *am* going to copy the URL. Now go back to `app.js`. To use `lodash` we can say `import _ from` and paste that full URL.

```
public/app.js
// ... line 1
import _ from 'https://cdn.jsdelivr.net/npm/lodash@4.17.21/+esm';
// ... lines 3 - 6
```

Yes, importing from a full URL is *totally* allowed. Or we could download this file locally: I'll talk more about that later. Below, let's say `_.camelCase()` to call one of its methods.

Let's try it! Spin over, refresh, and... look at that!. There's *no* build system here: we're just playing with files inside the `public/` directory. And yet, we're writing modern JavaScript, importing and exporting modules *and* using a third-party NPM package. That's pretty amazing.

What Features are Missing?

However, there *are* two remaining problems. First, importing packages using the full URL is annoying. I want to be able to say `import from 'lodash'` The *second* problem is asset versioning. To have a performant system, we need the final files downloaded by the browser to have version hashes in their filenames, like `app.1234abcd.js`. We need this so that we can instruct browsers to perform long-term caching. And we *can't* get this by creating & serving files directly from `public/`.

These are *precisely* the two things that Symfony's new AssetMapper component will help us solve. But I wanted to start with raw JavaScript so that we could see how... most of what we're doing is *not* solved by Symfony or AssetMapper or AI: it's solved by your browser and the modern web.

Ok, let's delete these two files so I don't get confused... and also remove the `import` inside of `base.html.twig`. Don't worry! We'll see all of that code in a different way soon.

Next: Let's install AssetMapper and get it rocking.

Chapter 3: Installing AssetMapper

We now know that we can run modern JavaScript directly in our browser. But to help smooth the process, we're going to install a new Symfony component called AssetMapper.

Find your terminal and run:

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

I'm including the second package because it gives us that nice `asset()` function in Twig. It's *already* installed in this project - just make sure you have it in yours.

Before we start: AssetMapper *is* experimental in Symfony 6.3, so there *will* likely be some backwards compatibility breaks before 6.4. But as we will see, the concepts are solid, and you can deploy a super-performant site with AssetMapper today.

Changes from the Flex Recipe

Ok, run:

```
git status
```

Oooh: the Flex recipe for AssetMapper added several things. Time for a quick tour! First, it gave us an `assets/` directory... which looks pretty much *identical* to what you would get if you installed WebpackEncore. We have an `app.js` file - this will be the main, *one* file that's executed - and also `app.css`: the main CSS file.

```
assets/app.js
```

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

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

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

In `templates/base.html.twig`, the recipe also added a `link` tag to point to `app.css`. We're going to talk more about stylesheets later, but you can already see that the CSS setup is perfectly straightforward.

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

```twig
// ... lines 1 - 2
    <head>
// ... lines 4 - 11
        {% block stylesheets %}
            <link rel="stylesheet" href="{{ asset('styles/app.css') }}">
        {% endblock %}
// ... line 15
        {% block javascripts %}
            {{ importmap() }}
// ... line 18
        {% endblock %}
    </head>
// ... lines 21 - 70
```

The recipe added one more important line to this file: `{{ importmap() }}`. That partners with a new `importmap.php` file. Those *are* important, and we'll go into detail about them soon.

The takeaway is that the recipe created a few files in the `assets/` and added a `link` tag to `base.html.twig`. But otherwise, there's not a lot going on yet.

AssetMapper "Paths"

Looking back at the terminal, the recipe also created a new `asset_mapper.yaml` file. Let's open that up: `config/packages/asset_mapper.yaml`.

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

AssetMapper has one, *main* concept: you point it at a directory or set of directories, like `assets/`, and it makes all the files inside available publicly, as *if* they lived in the `public/` directory. We'll see *how* that's accomplished in a minute.

But before we do *anything* else, refresh the page and... the background turned blue! That's coming from the `app.css` file. *And*, in the console log, we see a message that's coming from `assets/app.js`. So, somehow, magically, just by running a `composer require` command, these two files are already exposed publicly *and* are being loaded onto the page. Next, let's learn *how* this is all working.

Chapter 4: Mapping Assets

AssetMapper isn't that big of a deal. Sure it dresses cool and has good dance moves, but it's really quite simple. It has two main features.

Feature number one: we configure "paths" - like the `assets/` directory - and it makes the files inside available publicly.

Let's see this in action. If you downloaded the course code, you should have a `tutorial/` directory with an important `penguin.png` file inside. Copy that. Inside `assets/`, we can organize things *however* we want. So let's create an `images/` directory and transport our penguin there.

Now, remember, without the magic of AssetMapper, the only files that our browser should be able to access are those inside the `public/` directory. So it should be *impossible* to add an `img` tag that loads our penguin. But... it *is* possible.

Using the "Logical Path"

Head into, how about, `templates/base.html.twig`. Anywhere - I'll go above the `body` block - add an `img` with `src="{{ asset() }}"` passing this the path to our file *relative* to the `assets/` directory. So `images/penguin.png`.

templates/base.html.twig

```
// ... lines 1 - 20
    <body class="bg-gray-800 text-white">
// ... lines 22 - 49
        <img src="{{ asset('images/penguin.png') }}">
// ... lines 51 - 69
    </body>
// ... lines 71 - 72
```

That's it. This is known as the "logical path" to the asset. Because we've pointed AssetMapper at the `assets/` directory, we can refer to things inside of that via their path *relative* to that root.

And there's a great way to see *all* assets that are in the AssetMapper paths by going to the terminal and running:

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

Awesome! First, on top, it shows the AssetMapper paths, including the `assets/` directory. This project also has Pagerfanta installed. And we're already seeing how bundles can add their *own* AssetMapper paths to make their *own* files available publicly. This won't be important for us, but we could point the browser at *any* files inside that directory of the bundle.

Below, we see our image file, our CSS file, and our JavaScript file. These are their filesystem paths and these are their logical paths.

Versioned Filenames

The point is, by using the `asset()` function and the logical path to an asset, when we refresh... it works! Woh! And if we Inspect Element, check out the URL! It contains a *version* hash in the middle! I'm actually going to view the page source... it's a little easier to see.

So not only is `penguin.png` available publicly, but the path is *not* just `penguin.png`: it contains a version hash. If we *modified* the source `penguin.png` file - like gave it a cool bowtie - the version hash would automatically change forcing anyone using our site to download the fresh file. Booya!

This is also how `app.css` is loaded! Up near the top, the link tag uses `asset('styles/app.css')`, which is the *logical* path in AssetMapper to that file. And so it *also* output with a nice, versioned filename.

How Are the Files Made Public?

Okay, but how does this *work*? If you're like me, you want to know *how* the sausages are made. Well, in the `dev` environment, it works thanks to a core event listener... basically a fancy, internal Symfony controller.

For example, when the browser loads this image, that request goes through Symfony. It sees that we're trying to load `/assets/images/penguin-versionhash.png`, it finds the *source* file and serves it.

We can prove it! On the main tab, click any icon on the web debug toolbar to go into the profiler, then click "Last 10" to see the 10 most recent requests through Symfony. And there it is: the request that served the penguin image. Adorable.

In production, loading our files through Symfony would *not* be fast enough. So, instead, during deploy, you'll run a new console command:

```
php bin/console asset-map:compile
```

We're going to talk more about deployment later. But this is really cool! It copies *each* file of every AssetMapper path into the `public/assets/` directory using their versioned filename. That's it.

Suddenly, this file is no longer being served by Symfony: we're seeing a real, physical file! Over in `public/assets/`, yep! We can see the final files in all their glory.

But... while we're developing, remove that directory so that everything continues to load dynamically.

Moving favicons into AssetMapper

And actually, while we're here, see those favicons inside the `public/` directory? We're linking to them at the top of `base.html.twig`. That totally works: the `asset()` function can *still* refer to things inside the `public/` directory.

```
templates/base.html.twig
// ... lines 1 - 2
    <head>
// ... lines 4 - 7
        <link rel="apple-touch-icon" sizes="180x180" href="{{ asset('apple-touch-icon.png') }}">
        <link rel="icon" type="image/png" sizes="32x32" href="{{ asset('favicon-32x32.png') }}">
        <link rel="icon" type="image/png" sizes="16x16" href="{{ asset('favicon-16x16.png') }}">
// ... lines 11 - 19
    </head>
// ... lines 21 - 72
```

But... with almost no work, we can add free asset versioning to these! Step 1: move them into the `assets/images/` directory. Step 2: prefix each path with `images/` to get their logical path.

And... just like that, we still see the favicon up here... but more importantly, if we view the page source, those are now versioned!

Let's go a bit deeper into CSS files next. Like, how can we refer to background images from inside of CSS... if the final filename is versioned?

Chapter 5: CSS & Background Images

When we're talking about the frontend of a site, we're mostly talking about two things, CSS and JavaScript. Let's start with the CSS side of things... which is dead simple with AssetMapper. You create a CSS file inside the `assets/` directory then include it with a good old-fashioned `link` tag that uses the file's logical path. That's it. Zero magic.

This *is* a bit different than Encore. With a build system like Encore, you may be familiar with doing things like this: `import './styles/app.css`. That kind of thing will *not* work in a browser environment. Import statement are for importing JavaScript files, period. Ok, you *can* technically lazily-load CSS like this, but that's an edge-case we don't need to worry about right now.

The point is, you can't import CSS files from JavaScript files and that's ok: adding a `link` tag works great.

Referencing Images from inside CSS Files

Ok: so we know that we can refer to any file in the `assets/` directory using the `asset()` function... which we've now done twice.

But what if we need to refer to a file - like this image - from *inside* a CSS file?

templates/base.html.twig

```
// ... lines 1 - 20
    <body class="bg-gray-800 text-white">
// ... lines 22 - 49
        <img src="{{ asset('images/penguin.png') }}">
// ... lines 51 - 69
    </body>
// ... lines 71 - 72
```

Check it out. Up here, we have a little record icon in the upper-left corner. Change that to be a `span` with `class="bg-logo"` so we can include our penguin image instead. Copy that `bg-logo` class head to `app.css`, add `.bg-logo` and... I'll add some basic styles.

```
templates/base.html.twig
// ... lines 1 - 20
    <body class="bg-gray-800 text-white">
// ... lines 22 - 25
                    <a href="{{ path('app_homepage') }}" class="flex">
                        <span class="bg-logo"></span>
// ... line 28
                    </a>
// ... lines 30 - 69
    </body>
// ... lines 71 - 72
```

The *big* question is: how can we set the background image... since the final `penguin.png` will have a versioned filename? The answer is: exactly how you would *normally* do it: `url()` and then the relative path to the file: `../images/penguin.png`.

```
assets/styles/app.css
// ... lines 1 - 3
.bg-logo {
    display: inline-block;
    width: 32px;
    height: 32px;
    background-image: url('../images/penguin.png');
    background-size: contain;
    background-repeat: no-repeat;
}
```

This is *exactly* how you do it in Encore and exactly how you would do it if these files were being served directly to our browser. We simply need to write "correct" code and it'll work.

Let's add 2 more styles for the background... then testing time! Refresh and... yes, it *does* work! Inspect that image... then look at the final CSS file. Let's open this in a new tab.

Perfect! For the most part, the final files *exactly* match the source files. No magic. However, in this case, AssetMapper *did* make one small change. In the original file, we referred to `../images/penguin.png`. But over here we have `../images/penguin-versionhash.png`. Yup, AssetMapper made that tiny change to keep things working, despite the versioned filenames.

The point is: you get to code like normal... and everything just works.

Next: let's invite some third-party CSS like Bootstrap and fonts... to the party!

Chapter 6: 3rd Party CSS

We talked about adding CSS to our site, but what about *third-party* CSS like Bootstrap? With a build system such as Encore, we have a `package.json` file, and we can run:

```
npm install bootstrap
```

In AssetMapper, because there's no Node, we don't have a *such* an easy system for grabbing CSS packages. But we *can* still get them.

Finding Packages on jsDelivr

I like to use jsDelivr for this: a CDN for *all* NPM packages. Even if you don't ultimately *use* it as a CDN, it's a nice way to find and download what you need.

Search for "Bootstrap" and... there it is. A lot of times, you'll find the CSS file you need right up here, like this. I'll hit "Copy HTML + SRI". If you *don't* see the CSS file here... or you need a *different* one, you can click the "Files" tab to browse the entire package. For example - `dist/css/` and then whatever you need.

Okay, we know that CSS with AssetMapper is delightfully *boring*... so go back over to the `stylesheets` block and, above `styles/app.css`, paste the *new* `link` tag.

```
templates/base.html.twig
// ... lines 1 - 2
    <head>
// ... lines 4 - 11
        {% block stylesheets %}
            <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.cs
integrity="sha256-fx038NkLY4U1TCrBDiu5FWPEa9eiZu01EiLryshJbCo="
crossorigin="anonymous">
// ... line 14
        {% endblock %}
// ... lines 16 - 20
    </head>
// ... lines 22 - 74
```

If you want to *avoid* using the CDN, you *could* download this file directly into your project. Because there's no package system like NPM, I would probably create an `assets/vendor/` directory and put the file inside of that. Then I would *commit* that `assets/vendor/` directory to Git to keep it in your project and versioned. Committing vendor files into your project isn't *amazing*, but it's not a huge deal and is your best option right now if you want to avoid a CDN. You'll see me do this later for a JavaScript. file.

Ok, let's see if this is working! Scroll down to the middle of the page and add a `button` with `btn btn-primary` to use a few common Bootstrap styles.

```
templates/base.html.twig
// ... lines 1 - 21
    <body class="bg-gray-800 text-white">
// ... lines 23 - 50
        <button class="btn btn-primary">Primary Button</button>
// ... lines 52 - 71
    </body>
// ... lines 73 - 74
```

When we head over to the site and refresh... it works! Lovely!

Bootstrap Sass?

Ok, but what if I want to *modify* Bootstrap? Bootstrap itself is built with Sass. So, if you want, you can build Sass files that *override* Bootstrap variables - for example to change default colors.

There are two important things about this. First, you absolutely *can* use Sass with AssetMapper. There are details in the documentation about how to do that... and hopefully we'll add a bundle soon to make it even easier.

Go Deeper!

Check the Sass bundle at https://github.com/SymfonyCasts/sass-bundle

Also, in a moment, we're going to add Tailwind CSS to our site, which doesn't require Sass, but has a very similar workflow because Tailwind needs to be "built".

And, if you *do* want to use Sass with *Bootstrap*, one simple way get the Bootstrap source code is via the official Composer package for Bootstrap - so:

```
composer require twbs/bootstrap
```

If this is something you want, check out the AssetMapper docs.

Bootstrap CSS Variables

The second important thing is that, depending on what you want to do, you *may* not need to use Sass to customize Bootstrap. That's because Bootstrap also exposes *CSS* variables, though they're not *as* powerful.

We can see this down in the "Customize" "CSS Variables" section. CSS variables are a browser feature that allow you to *set* variables inside of CSS then reference them. No fancy-pants Sass needed.

For example, over in `app.css`... on top... add a `:root` pseudo selector, which is a common place to initialize variables that will be used later. Here, override a CSS variable that Bootstrap provides and uses: `--bs-border-radius`. Set it to `1rem`.

assets/styles/app.css

```
:root {
    --bs-border-radius: 1rem;
}
// ... lines 4 - 15
```

That *should* make the borders noticeably larger. Back at the browser... it works! The border radius is now larger across the site. That's one of the variables you would find in the Bootstrap documentation.

However, it's not *always* this simple. Let's say we want to override this primary color. You might think you could do that by searching for "primary"... up here... and overriding something like `--bs-primary`. That's *sort of* correct.

If you inspect our button, this color *is* the actual background color. But watch. Try to override that by changing it to a slightly lighter color. Then head back and try it. It doesn't do *anything*.

```
assets/styles/app.css
:root {
    --bs-primary: #0de1fd;
// ... line 3
}
// ... lines 5 - 16
```

Copy the CDN URL, pop that into your browser, and take off the `.min` so we can see what's going on. On top, it's setting all of those nice CSS variables. Look for `btn-primary`. I won't get too deep here, but inside of `.btn-primary`, it's setting these CSS variables to these *hard-coded* colors, instead of *using* other CSS variables that we can control.

So what do we do? In this case, we're kind of back to the basic strategy of overriding CSS... though we can at least *use* CSS variables when we do this.

Spin back over to `app.css` and I'll paste in some styling for `.btn-primary`. This overrides the variables that are set by Bootstrap to a different color. We are, at least, *using* the `bs-primary` variable: we set it up here, and can reference it in as many spots as we want. So, pretty basic CSS overriding, but with less repetition.

```
assets/styles/app.css
// ... lines 1 - 4
.btn-primary {
    --bs-btn-bg: var(--bs-primary);
    --bs-btn-border-color: var(--bs-primary);
    --bs-btn-disabled-bg: var(--bs-primary);
    --bs-btn-disabled-border-color: var(--bs-primary);
    --bs-btn-hover-bg: #0bbfd7;
    --bs-btn-hover-border-color: #0ba7d7;
}
// ... lines 13 - 24
```

And when we try it... it *does* change the color. Sweet! *So* CSS variables are *one* way to customize Bootstrap, but Sass is still an even more powerful option.

Next: let's grab one more external styling thing: an open source font!

Chapter 7: Adding Fonts

Another common CSS need is a custom *font*. My favorite source for fonts is https://fontsource.org where you can search through a *huge* number of fonts that have various open source licenses.

For example, one popular font is "Inter". Here, you can download the file, and it gives some install instructions, which are interesting: it uses the font as an `npm` package.

We're not using `npm`, but we *can* use `npm` packages: and we know how.

Head over to jsDelivr to find it. Notice that the package is called `@fontsource-variable/inter`. I'm going to search for `@fontsource/inter`. And... just like with Bootstrap, *there's* the CSS file! For the font nerds out there, if you looked inside of this file, you would see that this is the 400 weight, and it's the file you would normally use if you installed this via `npm` and imported it.

Copy that URL and paste it in the browser to see what it looks like.

Variable Fonts?

Notice that, back on FontSource, they recommend using a package starting with `@fontsource-variable`. Variable fonts are cool: instead of needing a different font file for each font *weight* like 400 vs 800, a single variable font can contain *all* the weights, while still keeping the file size reasonably small. FontSource starting offering variable fonts quite recently.

Change the URL to use `@fontsource-variable`. *This* is what we actually want. Copy it, head back over to `base.html.twig`, add `<link rel="stylesheet">`, and paste.

```
templates/base.html.twig
// ... lines 1 - 2
    <head>
// ... lines 4 - 11
        {% block stylesheets %}
            <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/@fontsource-
variable/inter@5.0.3/index.min.css">
// ... lines 14 - 15
        {% endblock %}
// ... lines 17 - 21
    </head>
// ... lines 23 - 75
```

Thanks to this, we can go over to `app.css`, inside of the `body` tag, and say `font-family:` `'Inter Variable'`, adding `sans-serif` as a backup.

```
assets/styles/app.css
// ... lines 1 - 12
body {
// ... line 14
    font-family: 'Inter Variable', sans-serif;
}
// ... lines 17 - 25
```

Let's check it! Watch this text closely. Boom! It updated thanks to that new font.

If you're wondering why I didn't just search for the `@fontsource-variable` package on jsDelivr originally, fair question: that's what I would *normally* do. jsDelivr *is* a mirror of *every* NPM package. *However*, due to a bug in the API of npmjs.com, right now, these new "variable" packages can't be found in the search. The bug has apparently been fixed - so hopefully the issue will melt away soon.

The point is, jsDelivr *does* have the package we need, we just can't find it via the search. It's kind of annoying, but should be temporary.

Next: Let's make our CSS a bit fancier by introducing Tailwind. That's going to be *especially* interesting because Tailwind requires a build step.

Chapter 8: Tailwind CSS

The HTML on our site is *already* styled with Tailwind: all the classes you see here come from it. So if we can get Tailwind installed, we should have a *much* less ugly site.

Tailwind is interesting because it's not just a CSS file you include: it requires a build step. And that's *totally* fine! Even though we don't have a build system for *everything* doesn't mean we can't choose to *add* one for some specific things.

Using TailwindBundle

Before we dive in... about a week after I recorded this, we created a bundle that makes it *super* easy to add Tailwind. It's called, creatively, TailwindBundle! Seeing how you can set up a small build system is still interesting - but if you want to skip this chapter and head over to that bundle instead, it won't hurt my feelings. The bundle basically automates what we're about to do.

Downloading the Standalone Executable

To get all of this working, we need the Tailwind binary file. As we see here, we *could* install it with Node... and that's a really flexible option. You would have a `package.json` file... but instead of it containing WebpackEncore and a ton of other stuff, it would just have Tailwind.

The other option, which avoids the need for Node entirely, is to use the standalone executable. Click the "Standalone CLI build" to go to the Tailwind release page. Find the version you need: for me, it's "tailwind-macos-arm64". You can download that here, but I'll copy the link address... so I can download it *fancily* via curl: `curl -slO` then paste!

It doesn't matter where you put this, but I'm going to move it into the `bin/` directory and rename it to `tailwindcss`... instead of that long name. Finally, because other machines - like the computers of our co-workers or the machine that deploys our site - might need a different version of this file, let's ignore it.

```
.gitignore
// ... lines 1 - 14
/bin/tailwindcss
```

So yes, this *does* mean that everyone will need to download their *own* Tailwind binary.

The very last step is to make this executable. On a Linux-based system, that's:

```
chmod +x bin/tailwindcss
```

Oh, and there is an extra, very-very last step if you're on a Mac. Run:

```
open bin/tailwindcss
```

If this is the first time you've downloaded the file, it will ask you to verify that you *do* want to open it from a security standpoint.

Initializing Tailwind

Okay! We now have the `bin/tailwindcss` executable, which does *not* require Node. From here, we can follow the normal docs. This is something I really like about the new frontend philosophy. If you *do* need a build system, you can just use Tailwind's build system directly and follow their instructions: no need for a Symfony-specific solution.

Here, it says that we need to run:

```
tailwindcss init
```

So let's do that!

```
./bin/tailwindcss init
```

This creates a shiny new `tailwind.config.js` file. Let's go check it out!

```
tailwind.config.js
/** @type {import('tailwindcss').Config} */
module.exports = {
  content: [],
  theme: {
    extend: {},
  },
  plugins: [],
}
// ... lines 9 - 10
```

The most important thing is to configure the `content` key. This tells Tailwind *where* it should look for HTML that may contain Tailwind classes. Search for their Symfony-specific documentation. Down here, they have exactly what we want! Copy the `content` key... then paste! I mean... paste it in the correct spot!

```
tailwind.config.js
// ... line 1
module.exports = {
  content: [
    "./assets/**/*.js",
    "./templates/**/*.html.twig",
  ],
// ... lines 7 - 10
}
// ... lines 12 - 13
```

The last step is to copy the three base directive lines for Tailwind... and put those inside `app.css`. I'll remove the Bootstrap stuff... but keep a little bit of our custom code down here. Nice!

```
assets/styles/app.css
@tailwind base;
@tailwind components;
@tailwind utilities;
// ... lines 4 - 17
```

Building the CSS File

Finally, we're ready to build! At your command line, run `bin/tailwind`, use `-i` to point to the input `assets/styles/app.css` file, then `-o` to tell it where to *output* the final code. Use `assets/styles/app.tailwind.css` so it's in the same directory, which is important so that any relative image paths will still work. At the end, add `-w` so it will keep running and watching for changes:

```
./bin/tailwindcss -i assets/styles/app.css -o assets/styles/app.tailwind.css -w
```

And that's it! Built! Over here, we have an `app.tailwind.css` file containing *all* the goodies. Awesome!

In `base.html.twig`, instead of pointing at `app.css` - which is now kind of an "internal" source file - point this at `app.tailwind.css`.

templates/base.html.twig

```
// ... lines 1 - 2
    <head>
// ... lines 4 - 11
        {% block stylesheets %}
// ... lines 13 - 14
            <link rel="stylesheet" href="{{ asset('styles/app.tailwind.css') }}">
        {% endblock %}
// ... lines 17 - 21
    </head>
// ... lines 23 - 75
```

Moment of truth. Back to the browser! Refresh. Our site is styled! That means we can get rid of the Bootstrap stuff: remove the Bootstrap CDN link... since we were just demonstrating how that works... and also the button down here.

That looks good!

Ignoring the Built File

But what about this `app.tailwind.css` built file? Do we *ignore* that from git? Do we *commit* it? It's up to you! We *can* commit it - it would make deploying easier, but we generally *don't* want to commit built stuff. I *will* ignore it... then we'll see how that works into our deployment process a bit later.

```
.gitignore
// ... lines 1 - 15
/assets/styles/app.tailwind.css
```

Ok, done! Next: Let's turn to *JavaScript*.

Chapter 9: JavaScript & importmap

Remove the `<img>` tag, so we can see our normal page. Don't worry about our little penguin guy: we still have him up here in the logo.

When we refresh the page, notice that we *do* have a `console.log()` message... which says it's coming from `assets/app.js`. If we head over to `assets/app.js`... yup! There it is!

```
assets/app.js
// ... lines 1 - 6
console.log('This log comes from assets/app.js - welcome to AssetMapper! 🎉')
```

How assets/app.js is Loaded

We know that we *can* write modern ES6 code in here, as well as import other files. We're going to do *all* of that. But first: *How* and *why* is *this* file even being executed? Our CSS is being loaded thanks to this nice, boring `<link>` tag. We *don't* see a `<script>` tag for `app.js`... but we *do* see this `importmap()` function. And *that's* the key.

Back over on the site, View the page source. Down here... *this* is what `importmap` adds. We're going to talk about each part, but the most important thing right now is at the bottom:

```
<script type="module">import 'app';</script>
```

Earlier, when we created an `app.js` file inside the `public/` directory, this is almost *exactly* the code we wrote to load it. We used `import` and then the *path* to that file. But... this time, it just says `app`. Shouldn't it say something like `/assets/app.12345.js`"? How does it know that `app` refers to the final version of this file? *This* is where the `importmap` part, up here, shines.

The Wonderful importmap

This section is generated from an `importmap.php` file inside our project. The file isn't super-interesting *yet*: it'll be more useful soon when we talk about third party JavaScript. But it *does* have this `app` key that points to our `assets/app.js` file using its logical path.

importmap.php

```php
// ... lines 1 - 15
return [
    'app' => [
        'path' => 'app.js',
        'preload' => true,
    ],
];
```

Thanks to that, this `<script type="importmap">` dumps onto the page. When you import something that doesn't start with a ".", "/", or "../", that's called a *bare* import. We usually see this for third-party libraries. In the browser environment, when it sees a "bare import", your browser looks for an `importmap` on the page to find a matching entry. Our browser sees `import 'app'`, finds this key here, and *that's* the path it downloads. It effectively copies this path here and pastes it down there. *That's* why our `app.js` file is being executed: it's team work between the `importmap` and the extra `<script type="module">` that bootstraps our app!

The *greatest* thing about `importmap` is that it's *not* a Symfony thing: it's just an *internet* thing. It's how your browser works. We *do* have this `importmap.php` file, which *is* a Symfony thing. But once this is on the page, your *browser* is the star.

The importmap shim + Older Browsers

And `importmap` works in... *most* browsers. If you go to "caniuse.com" and search for "importmap"... it currently works in about 81% of browsers. That *would* be a huge problem, except that the `importmap()` function also dumps a *shim*. You can see that here. Thanks to this, if a browser *doesn't* support `importmap`, this *adds* that functionality. So, it's *just* going to work.

Importing Relative JavaScript Files

Head into `app.js`: let's write some modern code. In `assets/`, first create a new directory called `lib/`. And inside *that*, a new file called `vinyl.js`. You can organize things however you want, and this is *one* example of isolating some code into its own file.

I'll paste in the same class we had earlier. Back over in `app.js`, import that: `import Vinyl` and I can hit "tab" to autocomplete the `from './lib/vinyl'` part. Instantiate this using the same code as before... and then `console.log(mix.describe())`.

assets/lib/vinyl.js

```
export default class {
    constructor(title, year) {
        this.title = title;
        this.year = year;
    }

    describe() {
        return `${this.title} was released in ${this.year}`;
    }
}
// ... lines 11 - 12
```

assets/app.js

```
import Vinyl from './lib/vinyl';

const mix = new Vinyl('Awesome Mix Vol. 1', 2014);
console.log(mix.describe());
```

Using .js when Importing

I love it! We're coding like normal and using `./` to import. But when we go over and refresh... it *doesn't* work. Check out the 404: `/assets/lib/vinyl` coming from `app.js`.

So... what's going on here? We'll talk more later about debugging, but here's a hint: if you ever notice that your browser is trying to download a path that *doesn't* include the "version" part in the filename, *something is wrong with your path*... and you should check for typos.

Our problem is that we need to add the `.js`. It turns out that leaving the `.js` off is a Node thing... and it works if you're programming in Node. But in true JavaScript environments, like in your browser, you *do* need to include it.

```
assets/app.js
import Vinyl from './lib/vinyl.js';
// ... lines 2 - 5
```

If we refresh now... that was it! It was really my editor's fault that the `.js` was missing when it autocompleted it. Fortunately, we can fix that! Go into your PhpStorm settings and search for "use file extension". Under "Code Style" and "JavaScript", change "Use file extension" to "Always".

This time... if we say `import Vinyl` and hit "tab", nice! We get the `.js`.

Automatic Importmap Entries

But the fun doesn't stop: there's something interesting happening behind the scenes. Click into this `console.log()`... just as an easy way to see the source of the final `app.js` file.

Yup, its contents look exactly like the original file, including the `import from './lib/vinyl.js'`. There's just one problem: that's *not* the final filename for `vinyl.js`!

Pop over to the Network tools, select "JS", and search for "vinyl". *All* files served by AssetMapper have a versioned part of their file name, and we see that for `vinyl.js`. But then... how the heck does our browser read `./lib/vinyl.js` and *know* that it should download this long filename?

The *answer*, if you view the page source, is... dramatic drumroll... the `importmap`. And I *love* this. The `importmap` is constructed from two sources. The *first* source is obvious: `importmap.php`. And we'll add more entries to it soon. The *second* source is more subtle. Whenever our JavaScript imports another JavaScript file using a relative path, that imported file is automatically added.

This is powerful. It means that our final code can look like it originally does: `./lib/vinyl.js`. But thanks to the `importmap`, our browser will smartly download the real file with the long version part in the name. This is really an internal detail, but it's cool to see how it works.

Okay, we've talked about `importmaps` a little... but we haven't seen its *biggest* superpower: using third party packages. Let's explore that *next*.

Chapter 10: importmap:require - 3rd Party JS Libs

In our code, we get to use `import` statements with relative paths, ES6 classes: everything we're used to. It's business as usual. Except, how can we use third-party packages?

As we saw earlier, we *could* import things via a full URL, like `import _ from`, and I'll paste in the CDN URL that we used earlier. With that done, the rest is normal: add `_.camelCase()` to the log.

```
assets/app.js
// ... line 1
import _ from 'https://cdn.jsdelivr.net/npm/lodash@4.17.21/+esm';
// ... lines 3 - 4
console.log(_.camelCase(mix.describe()));
```

If we refresh and check the console... that *works*. But I don't like it! I don't want to have to include this crazy URL everywhere I use `lodash`. And what happens if we upgrade lodash... and I need to change the URL in 10 different files? Lame!

importmap:require to Fetch Node Packages

If we were using a build system, like Webpack, we could just say:

```
yarn add lodash
```

or

```
npm install lodash
```

We're *not* using `yarn` or `npm`, but we can do *nearly* the same thing. Over in the terminal, open a new tab, and run `php bin/console importmap:require` followed by the name of the NPM package we want: `lodash`:

```
php bin/console importmap:require lodash
```

Done! It added `lodash` to `importmap.php` and tells us we can use the package as usual. This means we can say `import _ from 'lodash'`... and everything will work *fine*.

```
assets/app.js
// ... line 1
import _ from 'lodash';
// ... lines 3 - 6
```

How? When we ran the command, it made one *tiny* change: it added this section to `importmap.php`. And as cool as this is, it's not magic. Behind the scenes, the command went to the JSDelivr CDN, found the latest version of `lodash`, then added the `lodash` key set to that URL.

If you head over and look at the page source... no surprise! We have a new `lodash` entry inside the `importmap`! When our browser sees `import _ from 'lodash'`, it looks inside the `importmap` for `lodash`, finds this URL, and downloads it from *there*. Our browser is the hero!

Telling your Editor about the Packages

One bummer is that we don't get autocompletion in our editor. It says "Module not installed". And if I say `_.`... it doesn't really work. It's autocompleting `camelCase`... but only because I'm using that down here.

I hope this will be better-supported in PhpStorm soon. There *is* a workaround, but it's a bit manual. Copy the package, go into `base.html.twig` and add a temporary `<script>` tag that points to this. Hit "alt" + "enter" and select "Download library". This downloads that into the "External Libraries" section down here: `/lodash`.

Ok, remove that `script` tag. Back in `app.js`, it's *still* going to underline the import as if it doesn't know what it is, but it *does* autocomplete when we use `_.` something. For example, `tail()` is from `lodash`.

Updating Packages

What about *updating* the versions of packages inside `importmap`? Whelp, there's a command for that!

```
php bin/console importmap:update
```

That will loop through every package and update its URL to the latest version. This is *already* the latest version... but if we change it to `.19`... then run the `update` command... it moves back up to `.21`. The command *could* be more flexible - like by allowing you to update just one package, or by having some version constraints - and those things may be added in the future.

```
importmap.php
// ... lines 1 - 15
return [
// ... lines 17 - 20
    'lodash' => [
        'url' => 'https://cdn.jsdelivr.net/npm/lodash@4.17.21/+esm',
    ],
];
```

Downloading Packages Locally

Finally, if you don't want to rely on the CDN, you don't *have* to. To avoid it, when you require the package - or any time later - pass the `--download` option:

```
php bin/console importmap:require lodash --download
```

In `importmap.php`, this *still* shows the source URL to the CDN, but it downloaded that file into an `assets/vendor/` directory. This `downloaded_to` points to the logical path for that file.

```
importmap.php
// ... lines 1 - 15
return [
// ... lines 17 - 20
    'lodash' => [
        'downloaded_to' => 'vendor/lodash.js',
// ... line 23
    ],
];
```

The result? When we go over and refresh.... and "View Page Source"... the `importmap` now points to the *local* file! We're no longer relying on the CDN.

But... now what? Do we commit this `vendor/lodash.js` file? The answer is... *yes*. At least at this moment, that's the only way to version that file and keep it in your repository.

So even *without* npm or yarn, we *can* use *any* npm package we want. Woo! But *sometimes*, instead of importing an entire package, we may only want to import a specific *file*. Let's talk about how we can do that *next*.

Chapter 11: Importing Specific Package Files

Sometimes, instead of importing a package itself, you may want to import only *part* of it: like a specific *file*. Lodash is a good example of this.

No Tree Shaking in the Browser

But before we get there, instead of importing *everything* from `lodash`, you should be able to say `import { camelCase } from 'lodash'`. Then, down here, you would use `camelCase` directly.

```
assets/app.js
// ... line 1
import { camelCase } from 'lodash';
// ... lines 3 - 4
console.log(camelCase(mix.describe()));
```

However, when we move over and try this... error!

> *“The requested module `lodash` does not provide an export named `camelCase`.”*

This *should* work... and the reason it doesn't is *complicated*. Basically, due to the way that *this* specific library packages their module, you *can't* import specific functions like this. It *will* work with most other packages, however.

For example, if you say `import { Modal } from 'bootstrap'` (if you're using Bootstrap), that *will* work. Bootstrap packages their files correctly.

However, using this syntax *may* not always be ideal with AssetMapper.

Here's the problem. If we ran this code through Encore, Encore would do something called "tree shaking". This is where it would see that we're only importing `camelCase` from `lodash`. And so, in the final JavaScript, it would only give us the code for `camelCase`, not the *entire* `lodash` package.

In a browser environment, if you `import` from `lodash`, you're going to get *all* of `lodash`... even if you're only importing one part of it. Now, that *might* not be that big of a deal. The *full* build of `lodash` is still only 24 kilobytes. But what if we *are* using a big package... but only need to import one specific thing?

Importing a Specific File

A lot of times, there's a specific *file* that we can import, like `/camelCase`. You'll usually find details about these files in the docs... though you can also go look for them. Head back to JSDelivr... and down here, search for "lodash". Below, click "Files" to see all the files that are part of this package.

For `lodash`, it's a *huge* list... because this is a *huge* library. One of these is `camelCase.js`.

Ok! So let's try importing `lodash/camelCase`.

```
assets/app.js
// ... line 1
import camelCase from 'lodash/camelCase';
// ... lines 3 - 6
```

I'm not including the `.js`... but it's not going to work anyway. Watch: when we refresh... error!

> *"Failed to resolve module specifier `lodash/camelCase`. Relative references must start with either "/", "./", or "../""*

This error means that we're importing something using a "bare" import, and it was *not* found in the `importmap`. If we "View Page Source", we *do* have an `importmap` for `lodash`, but not `lodash/camelCase`. Yup, that matching is done *exactly*. Ok, there *is* a way to do a, sort of, "fuzzy" matching - `lodash/*` - but I don't use that.

The point is: if you want to use `lodash/camelCase`, you should add *that* to your importmap, not `lodash`.

Watch: find your terminal and run:

```
php bin/console importmap:remove lodash
```

That will remove `lodash` from `importmap.php` and delete the file from `assets/vendor/`.

```
importmap.php
// ... lines 1 - 15
return [
    'app' => [
        'path' => 'app.js',
        'preload' => true,
    ],
];
```

Nice! Now run `./bin/console importmap:require` with the package name / the path that you want: `lodash/camelCase.js`.

```
php bin/console importmap:require lodash/camelCase.js
```

`camelCase.js` is the name of the file over on the CDN. But you'll notice that, a lot of times in the docs, they'll reference `lodash/camelCase` without the `.js`. And in this case, you *can* leave the `.js` off: it's up to you. That works because jsDelivr is friendly and makes both versions of the URL work.

The result of the command? The same as before! We get a new entry in `importmap.php` matching what we want to import and set to a URL.

```
importmap.php
// ... lines 1 - 15
return [
// ... lines 17 - 20
    'lodash/camelCase' => [
        'url' =>
'https://cdn.jsdelivr.net/npm/lodash@4.17.21/camelCase/+esm',
    ],
];
```

Copy that URL so we can see it. There we go! It's the code from *just* `camelCase.js`.

And when we try the page... it works!

Here's the takeaway: if you need to import a specific file from a package, you can do that: just pass the package name + file path to `importmap:require`.

Next, let's add Stimulus to our app!

Chapter 12: Adding Stimulus

We can write modern JavaScript in this file, we can import third-party packages: we're free to dream up whatever code we want. *But*, if you're like me, you probably want to use Stimulus. So let's get that installed.

Stimulus is just a JavaScript library, so we *could* say

```
php bin/console importmap:require '@hotwired/stimulus'
```

Then follow their docs on how to get things set up.

Installing StimulusBundle

But Symfony has special integration with Stimulus. So instead, run:

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

StimulusBundle is a relatively new package that houses some Twig shortcuts that we'll use, like `stimulus_controller()`. But, more deliciously, it has a recipe that will set our app up to load Stimulus controllers *effortlessly*.

Check it out: thanks to the recipe, we now have an `assets/controllers/` directory with `hello_controller.js` inside.

```
assets/controllers/hello_controller.js
import { Controller } from '@hotwired/stimulus';
// ... lines 2 - 11
export default class extends Controller {
    connect() {
        this.element.textContent = 'Hello Stimulus! Edit me in
assets/controllers/hello_controller.js';
    }
}
```

Without touching *anything* else, open up `templates/vinyl/homepage.html.twig` and, right after the `<h1>`, add a new `<div>`. Let's *attach* the new `hello` controller to this element. Do that with: `stimulus_controller()` - that's one of the new functions that comes from StimulusBundle - passing `hello`.

```
templates/vinyl/homepage.html.twig
// ... lines 1 - 4
{% block body %}
<div class="px-4">
// ... line 7
    <div {{ stimulus_controller('hello') }}></div>
// ... lines 9 - 33
</div>
{% endblock %}
```

That can't *possibly* work already... right? Refresh. It *does*. That's bananas! And down in the console, we see logs about Stimulus initializing and our `hello` controller connecting. With just one `composer require` line, Stimulus is alive!

How Stimulus Loads

Let's put on our detective hats and delve a bit deeper into *how* this works and what the recipe actually did. In `templates/base.html.twig`, this is probably the least important change: it added `ux_controller_link_tags()`. We'll talk about that in the next chapter when we explore UX packages. But, in short, if a UX package come with their own CSS, this outputs that. Right now, it's not doing anything.

```
templates/base.html.twig
// ... lines 1 - 2
    <head>
// ... lines 4 - 11
        {% block stylesheets %}
            {{ ux_controller_link_tags() }}
// ... lines 14 - 15
        {% endblock %}
// ... lines 17 - 21
    </head>
// ... lines 23 - 72
```

More importantly, the recipe added a new `assets/bootstrap.js` file. And, in `assets/app.js`, it sprinkled in some code to *import* that file. So, `app.js` loads, that imports `bootstrap.js`, and then *that* imports `@symfony/stimulus-bundle`.

```
assets/bootstrap.js
import { startStimulusApp } from '@symfony/stimulus-bundle';

const app = startStimulusApp();
// ... lines 4 - 6
```

Ooh, that's a bare import! It doesn't start with "../" or "./"! That means our browser will look for it in the `importmap` to figure out which file to load.

Ok! Go open `importmap.php`. Surprise! The recipe added two new entries: One for the `@hotwired/stimulus` library itself and *another* for `@symfony/stimulus-bundle`, which points to this weird looking `path`.

```
importmap.php
// ... lines 1 - 15
return [
// ... lines 17 - 23
    '@hotwired/stimulus' => [
        'url' =>
'https://cdn.jsdelivr.net/npm/@hotwired/stimulus@3.2.1/+esm',
    ],
    '@symfony/stimulus-bundle' => [
        'path' => '@symfony/stimulus-bundle/loader.js',
    ],
];
```

Up here, when using a CDN, the entry will have a `url` key. When pointing to a *local* file, the entry will have a `path` key, which will be the *logical* path to a file in AssetMapper.

But, what does this weird path point to? Spin over to your terminal and run:

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

If you take an elevator to the top... voilà! When we installed StimulusBundle, it added a new "asset path" to our system, which points to `vendor/symfony/stimulus-bundle/assets/dist` and it has a "Namespace prefix". This means that, to point to a file in this directory, the logical path will start with `@symfony/stimulus-bundle`.

So over here, when we say `@symfony/stimulus-bundle/loader.js`, we're referring to this file right here: `vendor/symfony/stimulus-bundle/assets/dist/loader.js`. That's a long way of saying, when we import `@symfony/stimulus-bundle`, it's actually importing this `vendor/symfony/stimulus-bundle/assets/dist/loader.js` file. The bundle exposes that file by adding the AssetMapper "asset path", which allows the recipe to add an entry to `importmap.php` that points to it.

How our Controllers are Registered

Okay, so we're loading this `loader.js` file, and we can see that over here. In your browser, refresh... go to your Network tools, and search for "loader". There it is! Open this up in a new tab.

This code has functions to start the Stimulus application and register the controllers from our app - like `hello_controller.js`. But... wait. This is just a hard-coded file. How is it able to *dynamically* find and load the files that live inside our `assets/controllers/` directory?

The key is on top: `import`, `isApplicationDebug`, `eagerControllers`, `lazyControllers` from `./controllers.js`. This... is a bit of *magic*. Go back to the Network tools and search for "controllers"... there it is - `controllers.js`. Open *this* new tab. Woh! It has `import controller_0` from `../../controllers/hello_controller.js`, which it then exports to a variable called `eagerControllers`.

This file is crafted *dynamically* by the bundle. If we look down in the `vendor/` directory, `loader.js` is a nice static file. But if you look at `controllers.js`, it doesn't look like at *all* like what we have in the browser! When this file is served, AssetMapper intercepts it, looks inside of our `assets/controllers/` directory, finds all the controllers there, and then returns *dynamic* contents based on these.

Watch. Create another file called `goodbye-controller.js` (you can use dashes or underscores). Change the text to `Goodbye controller!`.

```
assets/controllers/goodbye_controller.js
```

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

export default class extends Controller {
    connect() {
        this.element.textContent = 'Goodbye controller!';
    }
}
```

You might expect that, when we refresh the file, we'll see the new controller pop in here. And you're *almost* right. What really happens is... *nothing*! No change! Or you might even get a 404 error. That's because the *content* of this file just changed and so the *hash* will also change. We're looking at an out-of-date version of the file!

Back on the site, if we refresh, we *should* see a new file with a new hash. We *don't*... due to a caching bug which has already been fixed. To work around that, I'll run:

```
php bin/console cache:clear
```

Then go refresh. *Now* I see that this has a different file name, and the contents *have* dynamically changed to include `goodbye-controller.js`!

So there you have it, the thrilling journey into the heart of how Stimulus and AssetMapper became best friends. `bootstrap.js` loads a file that starts Stimulus... and that automatically loads everything inside the `assets/controllers/` directory... as well as any 3rd party UX packages in `assets/controllers.json`. Let's talk about those 3rd party packages next.

Chapter 13: Symfony UX Stimulus Packages

We can now create custom Stimulus controllers with ease. The *other* half of StimulusBundle is the ability to get more *free* Stimulus controllers by installing a UX package. Let's add one and see how it works!

Installing Turbo

Let's start by adding Turbo. At your terminal, say:

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

Here's the juiciest part: just like when we added Stimulus, there's absolutely *nothing* else you need to do to set this up. Refresh and... it just works! Turbo eliminates the need for full page refreshes. Head over to your Network tools and click on "Fetch/XHR". Let's actually clear this out so we can see everything. Perfect. Then, if we click up here... you can see that this is coming from an AJAX call! Yup, those full page refreshes are *gone*. So Turbo *just works*. There's no build system to get in the way, and that's *beautiful*.

UX Packages Often add Importmap Entries

In practice, this works because a new JavaScript file is being loaded called `turbo_controller.js`. Filter the network calls to JavaScript... and refresh, because I cleared them. There we go! Our page loads `turbo_controller.js` and *that* imports `@hotwired/turbo`, which *starts* Turbo.

Open up `importmap.php`. When we installed the UX Turbo package, its recipe *added* this new `@hotwired/turbo` entry.

```
importmap.php
// ... lines 1 - 15
return [
// ... lines 17 - 29
    '@hotwired/turbo' => [
        'url' =>
'https://cdn.jsdelivr.net/npm/@hotwired/turbo@7.3.0/+esm',
    ],
];
```

This is a *really* common pattern with UX packages: if a UX package depends on a third-party package, its recipe will add that package to your `importmap` *automatically*. The result is that, when that package is referenced - like `import '@hotwired/turbo'` - it just works.

How UX Controllers are Loaded

The *real* question is: who's loading `turbo_controller.js`, which lives deep inside the `symfony/ux-turbo` PHP package?

The answer is: the same trick we learned a moment ago. Search for `controllers` and open that file in a new tab. This is the dynamic file that StimulusBundle builds. As it turns out, it looks for packages in our `assets/controllers/` directory, which is these two, *and* it reads the `assets/controllers.json` file. When we installed UX Turbo, it added this new section here, which is where we activate different controllers. It activated one called `turbo-core` with `"enabled": true` and added another deactivated one with `"enabled": false`. So when this file is built, it parses the `assets/controllers.json` file, finds the controllers that we've enabled, and adds them here.

assets/controllers.json

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

The final result is that it imports that controller file here and exports it so that the `loader.js` file can register it in Stimulus. So any controllers in `assets/controllers/` *or* in this file are registered automatically.

autoimport & CSS Files

Head back into `base.html.twig`. When we installed StimulusBundle, its recipe came bearing gifts - one of which was this `ux_controller_link_tags()`. Right now, that does nothing. *However*, some UX packages come with CSS files. You'll find them under a key called `autoimport`, which the recipe will add under the controller. This `ux_controller_link_tags()` finds all the CSS files for all the controllers you have activated, and it outputs them. Nothing too fancy.

Next: let's learn one more thing about Stimulus, which just happens to be one of my *favorite* things: how to make our controllers *lazy*.

Chapter 14: Lazy Stimulus Controllers

It's getting messy in here: let me close a few files... then crack open `assets/controllers/goodbye-controller.js`. Pretend, for a moment, that this controller is huge. Or, more likely, it imports a big third-party package like `d3` for charts. *But*, we're only using this controller on *some* pages.

Here's the deal. In order to register your controllers with Stimulus, all of these files are downloaded immediately. So the page loads, Stimulus starts up, all of these files are downloaded, and any files they import are *also* downloaded. That's often ok, but if you're importing something *big*, that can be wasteful.

To fix that, above the class, you can add a very special syntax - `/* stimulusFetch: 'lazy' */`.

assets/controllers/goodbye_controller.js

```
// ... lines 1 - 2
/* stimulusFetch: 'lazy' */
export default class extends Controller {
// ... lines 5 - 7
}
```

This works thanks to StimulusBundle. When it spots this, it tells Stimulus to hold its horses and *not* download this JavaScript file or anything it imports *until* an element that matches this is on the page.

Watch. Before making that change, if we searched for "goodbye", that controller *was* being loaded, even though it's not used on this page. But *now*, refresh and search for "goodbye". It's *not* there! Inspect the `data-controller="hello"` element. Change that to `goodbye` and... boom! It works! You can see that it activated (that's what our `Goodbye controller!` does), and if we look at the Network tab, *now* it downloaded. I *love* this feature.

This can also be done for third-party packages. If you look in `assets/controllers.json`... Turbo isn't a very good example of this, but if we said `"fetch": "lazy"` on any of these, they would have the *same* behavior that we just saw.

```
assets/controllers.json
{
    "controllers": {
        "@symfony/ux-turbo": {
            "turbo-core": {
                "enabled": true,
                "fetch": "eager"
            },
// ... lines 8 - 11
        }
    },
// ... line 14
}
```

That's it! Easiest chapter *ever*! Use this to keep your initial page lightweight if you have some heavy Stimulus controllers that are only used on certain page.

Next: sometimes, deep sigh, the tech gods frown upon us and things don't work. Let's learn a few tips to help debug when that happens.

Chapter 15: Debugging

Sometimes things won't work. But, with AssetMapper, there are some telltale signs when things go wrong. Let's see a few of the most common ways that things can... get weird.

Typo in Logical Path? Missing Versioned Path

One of the fastest ways to mess things up is in `/templates/vinyl/homepage.html.twig`: use the `asset()` function and pass an invalid path. Remember, in `assets/images/`, we have `penguin.png`. So, `images/penguin.png` is its "logical path" in AssetMapper. Let's say `images/`, but then `duck.png`.

templates/vinyl/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
<div class="px-4">
// ... lines 7 - 8
    <img src="{{ asset('images/duck.png') }}">
// ... lines 10 - 34
</div>
{% endblock %}
```

This is obviously not the right path... or even the right animal. So no surprise that, on the homepage, we get a 404. The key thing about this 404, if we look at the console, is its suspicious-looking path. Look closely: there's no *version* in the filename! This tells us that this path was *not* found in any of the AssetMapper directories: it's an *invalid* logical path. And so, AssetMapper ignored it and returned the raw, unversioned filename.

To help visualize the *valid* logical paths, remember that you can run:

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

Up here, this is *everything* that you're allowed to pass to the `asset()` function. And *there's* `images/penguin.png`. If we put `images/penguin.png` here instead... now it *works*.

```
templates/vinyl/homepage.html.twig
// ... lines 1 - 4
{% block body %}
<div class="px-4">
// ... lines 7 - 8
    <img src="{{ asset('images/penguin.png') }}">
// ... lines 10 - 34
</div>
{% endblock %}
```

The key thing to look for is the version hash in the filename. If it's not there, AssetMapper couldn't find your path.

Invalid Import Paths

Another common mistake is to mess up an *import*. Like... maybe in `styles/app.css`, we mistype a part of this image `url()`. Or, in `app.js`, when importing `vinyl.js`, we forget the `.js` at the end.

```
assets/app.js
// ... line 1
import Vinyl from './lib/vinyl';
// ... lines 3 - 7
```

Accidents like this give us the same result as the first mistake! When we refresh, we get a 404. You can see that here. But again, the *key* thing is the missing *version hash*. That's a sign that the path couldn't be found, so it couldn't be handled by AssetMapper.

In this case, the invalid path lives in `app.js`. When we're inside a *template* and use the `asset()` function, we pass the *logical* path to a file. But if we're inside of `app.js` or `app.css`, instead of the logical path, we use the *relative* path. This is by design. We get to code inside of these files as if AssetMapper doesn't exist. We don't need to think about logical paths, we just think:

> *“What relative path would I use if these files were all just being served directly to my browser?”*

Anyway, if the version hash is missing, we have an invalid path, which could be an invalid logical path in a template *or* an invalid *relative* path in some import somewhere.

Seeing a List of Invalid Paths

By the way, there's an almost hidden way to see if any invalid imports appear *anywhere* in your code. First, run:

```
php bin/console cache:clear
```

That clears Symfony's cache, of course, but it also clears an internal cache in AssetMapper. Now when we run

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

it re-builds the cache for all of those assets internally. When it does that, it parses our files and reports any missing imports. See?

> *“WARNING Unable to find asset `./lib/vinyl` imported from `assets/app.js`.”*

And in this case, we even get an extra message:

> *“Try adding ".js" to the end of the import”*

Good idea. If we add that `.js` back... things work again.

Importing Missing "Packages"

The last common way to mess things up is to use a bare import - an import that doesn't start with `./` or `../` - for something that does *not* appear in your `importmap`. Here, the intention is to use the `bootstrap` library... but we don't have it in our `importmap`. The exact error will vary based on your browser, but for me it says:

> *“Failed to resolve module specifier "bootstrap". Relative references must start with either "/", "./", or "../".”*

Translation:

> *“Hey! If you're trying to refer to a relative path, you forgot the `./` or `../` part, you goofball! But if you're trying to import a package, you forgot to add it to your importmap!”*

The solution is usually to run: `php bin/console importmap:require` to add that package.

Next up: what if you have a bunch of CSS or JavaScript that you *only* want to load on a single page or section of your site - like an admin section? How can we organize things so that we don't have to load all of that code on every page?

Chapter 16: Page-Specific CSS & JS

Head over to `/admin`. Surprise! We *do* have an admin section on our site. Well... *sort of*. It's only a big rectangle, but it represents a make-believe admin. Why? Well, suppose we have some CSS and JS that are *only* needed here. If we write that in the normal way and in the normal files, that code is going to be downloaded everywhere, including when someone goes to the frontend of our site. That, at the very least, is wasteful. A better way is to *only* download the admin CSS and JS when you visit the admin area!

My favorite way to do this is with lazy Stimulus controllers, and we've already talked about those. But another option is to create an extra set of CSS and JavaScript that are explicitly loaded *only* on these pages. Let's see how to do that with AssetMapper.

If we were using Webpack Encore, we'd open the `webpack.config.js` file and add a second *entry*. That would result in a new CSS and JavaScript file. In AssetMapper, we can do something really similar.

Creating a new CSS File

Let's start with CSS, which is pretty darn simple. In the `assets/styles/` directory, create an `admin.css` file and, to see if things are working, add `.admin-wrapper` with some X-Y padding.

```
assets/styles/admin.css
.admin-wrapper {
    padding: 0 3rem;
}
```

That'll add a little space right here. *Then*, go into the template for this page - `templates/admin/dashboard.html.twig` - and, right here, add that class: `class="admin-wrapper"`.

```
templates/admin/dashboard.html.twig
// ... lines 1 - 8
{% block body %}
    <div class="admin-wrapper">
// ... lines 11 - 14
{% endblock %}
```

At this point, the new `admin.css` file *is* technically available publicly... because it's in the `assets/` directory. But, we're not *using* it yet. To do that, we need a link tag.

There's nothing special about this. Say `{% block stylesheets %}` and `{% endblock %}` to override the block from the parent template. Then call `{{ parent() }}` to include the normal stuff and, down here, add `<link rel="stylesheet"` pointing to `asset('styles/admin.css')`. And... let me fix my typo up here. *That's* what we want.

```
templates/admin/dashboard.html.twig
// ... lines 1 - 2
{% block stylesheets %}
    {{ parent() }}

    <link rel="stylesheet" href="{{ asset('styles/admin.css') }}">
{% endblock %}
// ... lines 8 - 16
```

Back on the site... yup! The CSS *is* being applied: we've got extra padding. Refreshingly simple.

Creating a Page-Specific JavaScript File

But... what about JavaScript? Once again, we'll start a lot like Encore. Create a new file... maybe next to `app.js` called `admin.js`. Add `console.log('admin.js file')` so we can see if it's loading.

```
assets/admin.js
console.log('admin.js file!');
```

Like with the CSS file, this file *is* now publicly available... but nothing is *loading* it. Remember: the `app.js` file is loaded thanks to this `<script type="module">` line down here that imports `app`. We *automatically* get this, over in `base.html.twig`, via the `importmap()` Twig function.

So... is there a way to tell this function to *also* import our `admin.js` file? Actually, no! Why? Mostly because... it's just so easy to add ourselves!

Watch: back in `dashboard.html.twig`, say `{% block javascripts %}`, `{% endblock%}`, then `{{ parent() }}`. Below that, add a `<script>` tag with `type="module"`. Now we're going to code as if we're in a JavaScript file. Say `import` and then the *path* to the JavaScript file. Effectively, we want something like - `/assets/admin.js`. But, of course, to get the real path we use the `asset()` function and pass the logical path: `admin.js`.

```
templates/admin/dashboard.html.twig
// ... lines 1 - 8
{% block javascripts %}
    {{ parent() }}

    <script type="module">
        import '{{ asset('admin.js') }}';
    </script>
{% endblock %}
// ... lines 16 - 24
```

That's it! Let's try this thing! Refresh and check the console. Got it! Our `admin.js` file *is* being loaded! If you check out the page source... down here... *yep*. You can see `<script type="module">` from the `importmap()` function where it says `import 'app'`. And, after, we import `admin.js` via its path.

The original is just `import 'app'`... because we rely on the `importmap` to *map* that to its URL. That's *nice*... but it's not actually necessary. Putting the path right here works fine too. That's what we're doing for simplicity.

One of the things we saw in this chapter is that *everything* in the `assets/` directory is exposed publicly... which is the whole point of AssetMapper! But sometimes you may have a few files that you want to put in that directory, but keep private. Let's check into AssetMapper's exclude feature and other config options next.

Chapter 17: Excluding Files

We now have CSS - which we're building with Tailwind - we have JavaScript, we're bringing in third-party JavaScript, *and* we're using *modern* JavaScript syntax. Our app has everything that a real app has! Sure, it's kind of small, but we're *almost* ready to deploy it.

Checking your Exposed Files

Before we do, let's do a quick audit on the assets that are inside AssetMapper. Find your terminal and run:

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

This lists all of our asset paths, which includes our *main* asset path - `assets/` - plus a few from bundles that have exposed their own directories. Below is a list of *every* file that will be exposed publicly.

We're running this command to see if there's anything in this list that we do *not* want to publicly expose. For example, this `assets/styles/app.css` file. This is really a *source* file: it's not meant for the user to download directly. We're using Tailwind to build that *into* `app.tailwind.css`, and *that's* what the user will download. It's not a *huge* deal that this is available publicly, but it's a good example of how we can hide "source" files that we *don't* want to expose.

Asset Mapper Config

Start by running

```
php bin/console config:dump framework asset_mapper
```

We're asking the system to give us example configuration for everything that can be configured under `framework`, `asset_mapper`. When we first installed AssetMapper, its recipe gave us a `config/packages/asset_mapper.yaml` file. Here, we have `framework`, `asset_mapper`, and a key called `paths`. When we run this command... sure enough, up here on top, it shows `paths`. Below that, we have some other interesting things.

The first is `excluded_patterns`. *This* is how we're going to hide certain files or paths - and we'll talk more about that in a minute. You can also control the `public_prefix`, which is where your files are output to in the `public/` directory.

This `extensions` isn't *super* important... it's mostly just for the `dev` environment... and there are a few other things like your `importmap_path`, and even some attributes you can put on the `<script>` tag that's dumped by the `importmap()` function.

Excluding Files / Patterns

So there's some good stuff in here... but you won't need to worry about most of it, aside from `excluded_patterns`.

Copy that key, spin over to `asset_mapper.yaml`, and on the same level as `paths`, paste. We want to exclude `assets/styles/app.css`.

config/packages/asset_mapper.yaml

```yaml
framework:
    asset_mapper:
// ... lines 3 - 5
        excluded_patterns:
            - 'assets/styles/app.css'
```

But this isn't *quite* correct. To prove it, run

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

again. If you look up... `assets/styles/app.css` is still there! That's because `excluded_patterns` is meant to be a *glob*. In other words, change this to `*/assets/styles/app.css`... and surround it by quotes.

```yaml
config/packages/asset_mapper.yaml
framework:
    asset_mapper:
// ... lines 3 - 5
        excluded_patterns:
            - '*/assets/styles/app.css'
```

This says that any "filesystem path" that ends with `/assets/styles/app.css` will be ignored. And when we try the command again...

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

Awesome. *This* is what we want to see. Every file here will be dumped into the `/public/assets` directory. The fact that `assets/styles/app.css` is *not* here means that it will *not* be dumped into the `public/` directory.

I think it's time to deploy our site! Let's get a deploy set up next on platform.sh.

Chapter 18: Deploying to Platform.sh

I have a *wild* idea. Let's deploy this site *for real*.

AssetMapper Deploy Requirements

You can deploy your code however you want... using any service or web server. It doesn't matter with AssetMapper. The only requirement is that your web server supports HTTP/2 so that our assets - the JavaScript and CSS files - can be downloaded in *parallel* super fast. HTTP/2 is the reason why it's not terribly important that our files *aren't* being combined to minimize requests.

All web servers - nginx, Caddy, whatever - *do* support HTTP/2. *Or* you could add Cloudflare or a similar service in *front* of your site which gives you this for *free*... along with some other benefits.

platform.sh Config File Setup

Anyway, we're going to deploy with Platform.sh, which is a "Platform as a Service". That means we can deploy just by creating a few config files. And this first section is *all about* getting that set up. Once we're done, we'll talk about some specifics of deploying with AssetMapper.

So, let's get started! We're going to do most of this in the command line with the Symfony binary. Start by running:

```
symfony project:init
```

This bootstraps a few platform.sh files, which you can see right here. `.platform.app.yaml` contains instructions for how to deploy - like which commands to run, what version of PHP to use, web server configuration and more. `services.yaml` is where you set up the services you need - like databases, queues, etc - and `routes.yaml` sets up your domains, and is a bit less important. Oh, and you can also add any custom `php.ini` config with this file.

I've been committing my progress along the way. So when I run

```
git status
```

it just shows these new files. Let's commit these and... great!

Registering the platform.sh Project

Now that we have those local files, we need to dial up the folks at platform.sh and tell them that we want to create a new project. We'll do that with:

```
symfony cloud:project:create
```

I already have some projects under Platform.sh.... which means I already have an organization... and it already has my credit card. Thieves! If you're doing this for the first time, you'll have a few extra steps.

Give your project a title, select a region - I'm using "eu-5" - and then it asks which branch will be your production environment. I'm using the default `main` branch.

Next, it asks if we want to set "Mixed Vinyl" as the *remote* for this repository. This is kinda cool because it exposes *how* platform.sh works. To deploy with platform.sh, we actually push our git repository to a remote repository on their services. They see this, take the code, and deploy!

Anyway, I'm going to say "no" - but you can say "yes". Because I'm saying no, you'll see me do this manually in a minute - and I'll explain more about it.

Finally, it asks to confirm pricing. This $12 USD per month is the developer rate that you can pay to play around with stuff. It *will* be more expensive when you decide to deploy your site to production for real. I love platform.sh because of how easy it makes my life, but there *are* cheaper options.

This will take a minute or two to set everything up behind the scenes. When it finishes... ding! We get some info, including the new Project ID.

Side Note: There *is* also a web interface on Platform.sh, so not *everything* needs to be done via the command line. But, yea know, nerds like me prefer the command line.

Linking the Local Code to the Remote Project

Copy the Project ID. At this point, we have some local config files - like `.platform.app.yaml` - *and* we created a new "project" on platform.sh. But the two aren't linked together yet: our local code doesn't know that it "belongs" to this project up on platform.sh.

To link them, run

```
symfony project:set-remote
```

and paste the Project ID.

Our First Deploy

Done! Ready to deploy? Do it with:

```
symfony deploy
```

We're currently on the "main" branch, so it's pushing to, basically our "production" machine, which is called an *environment*. One of the coolest parts of platform.sh is that you will deploy your `main` branch to the "production" environment, but you can also deploy other git branches to other platform.sh environments, which you can think of as other platform.sh servers.

Anyway, as I mentioned, behind the scenes, this command is just a shortcut to `git push` our branch up to a git remote on the platform.sh servers. And doing that kicks off the deploy!

Oooh, this looks fancy and geeky. Tons of details here, including a warning about using an old version of Composer. We'll fix that in a minute. Down here, we see that it's running `symfony composer install` and doing some other steps: all the basic stuff you need to deploy any Symfony app.

At the bottom, it gives us an SSL certificate, and if we keep scrolling... oooh, we have a message about a database error! Ignore that for now because... when it finishes, it gives us a URL!

Because we haven't configured any domains for the site, it gives us a *temporary* URL. Copy that, spin over and... our site is *alive*! It doesn't have any styling... since we haven't talked about AssetMapper, but it at least kinda works!

But how? How did it know to run `composer install` and those other things? What about that Composer warning and the database error? Let's dive into all of that next.

Chapter 19: Configuring the Platform.sh Deploy

We just deployed a semi-working version of our site to platform.sh! All it took was one command to bootstrap a few config files and another to create the project inside of platform.sh.

But... we had some errors and warnings along the way. On top of the output, we see a warning about using an old composer version. In a minute, we'll see how and *why* composer is used during the deploy. But when it is, for *some* reason, it's currently using an old version of Composer.

Fortunately, it warns us *and* tells us how to fix it.

.platform.app.yaml and How Deploying Works

Copy this `dependencies` line. Then, open `.platform.app.yaml`. *This* is the main deploy file: almost every deploy tweak you'll make will be made here.

There are two steps to the deploy process. The first is the `build` step where it's *building* your code: you can think of this as the step that prepares all the physical *files* that your project needs. Once the `build` step is done, it spins up a container, puts the files inside and then runs the second and final part of the process: the `deploy` step. This is where you can run some final commands.

See these `symfony-build` and `symfony-deploy` scripts? These are pre-made scripts that contain *most* of what your app needs to deploy. If you downloaded them and opened them up, you'd see things like running `composer install`, warming up the cache and running database migrations. We can add custom stuff above or below.

The config has `mounts` - for directories that you want to keep persistent between deploys - PHP extensions, your PHP version, and quite a bit more.

Using a Newer Composer Version

Anywhere inside, paste the `dependencies` line... and make sure it's not indented.

```
.platform.app.yaml
// ... lines 1 - 4
dependencies:
    php:
        composer/composer: "^2"
// ... lines 8 - 53
```

And just like that, we're using Composer version 2.

Setting up the Database Serve

The second error that we had, down near the bottom, was,

> *“could not find driver”*

This come from when the `symfony-deploy` script tries to run our database migrations. Locally, we're using Postgres. You can see that in `docker-compose.yml`. Do we have a database up on Platform.sh yet? The answer is... actually *yes*.

In addition to the main deploy file - `.platform.app.yaml` - we have a `/.platform` directory with a few other files. The most important is `services.yaml`. This is where we define *services* like databases, Redis, Elasticsearch and others. When we initialized the project, it noticed that we're using Postgres and added a database for us!

```
.platform/services.yaml
// ... line 1
database:
    type: postgresql:15
    disk: 1024
```

The error we're getting isn't because it can't find the database: it's because our PHP install is missing the `PDO_PGSQL` driver! And thanks to `.platform.app.yaml`, adding that is easy.

Find `extensions`, and add `pdo_pgsql`.

```
.platform.app.yaml
// ... lines 1 - 8
runtime:
    extensions:
// ... lines 11 - 15
        - pdo_pgsql
// ... lines 17 - 53
```

Ok, ready to re-deploy? Remember: deploying happens via a *push*, so we need to commit these files. Run `git commit -m` with an inspirational message.

```
git commit -m "tweaking deploy script"
```

Now run:

```
symfony deploy
```

This runs the same steps as our first deploy, which we now understand include a `build` step then a `deploy` step. It'll take a minute or two, but should be a *bit* faster because it doesn't need to re-provision the SSL certificate.

Viewing the Logs

At the very end... the migration command *still* failed, but with a *different* error:

> *“Connection refused”*

Ah, ignore that for a minute. Instead, go back to the site and refresh. The homepage still works. But... that's because our homepage *doesn't* use the database. If you hit "Browse Mixes"... 500 error! That 500 error is *probably* due to a database connection problem. But let's pretend that we have *no* idea what's causing this. How could we figure that out? This is where the

```
symfony logs
```

command comes in handy. This connects to whatever platform.sh "environment" - or "server" - that our current git branch is connected to and sends back log info. There are a bunch of choices - but hit `2` to go to the "app" log. This represents the Symfony logs coming from our app. And... oooh. I see several:

> *“connection to server [..] failed: Connection refused”*

Adding the Database "Relationship"

Let's think about this. We apparently *do* have Postgres set up, thanks to the `services.yaml` file. But we never configured our app to *talk* to it. Remember, in `.env`, we have a `DATABASE_URL` env var that's supposed to point our database. We never configured that on our production site, so it's just using this default value. And no surprise, that's not working.

How *can* we configure `DATABASE_URL` to point to... wherever this database server is? The answer is... we... uh... don't? And it's pretty cool.

Platform.sh has this idea of *relationships*. You have a number of services in `services.yaml`. But your app can't *talk* to these until you link them together using what's called a "relationship".

Search `.platform.app.yaml` for "relationships". It's not here yet, so let's add that. Each "relationship" has an internal name. It could technically be anything, but, in practice, you should use `database`. We'll see the *significance* of that in a moment. Set this to the word `database`, because that's the key we have here, then `:` followed by the *type* of the service, which is `postgresql`.

```
.platform.app.yaml
// ... lines 1 - 17
relationships:
    database: "database:postgresql"
// ... lines 20 - 56
```

This syntax has always looked weird to me. The *important* thing is that the key could be anything, like `banana`, but this `database` refers to this `database` over in `services.yaml`, and `postgresql` refers to *this* `postgresql`.

But though the first `database` key *could* be anything, I used `database` on purpose. Symfony does a really nice thing when it deploys via Platform.sh. It sees this relationship, notices it's for a database, and then automatically exposes an environment variable containing the connection info *to* that database!

What's this environment variable called? Since we used the key "database", it will be called `DATABASE_URL`. In other words, it's going to set this environment variable *for* us. I'll prove it!

SSHing onto the Container

One of my favorite things about Platform.sh is that you can SSH onto your container. Watch:

```
symfony ssh
```

There we go. Once here, if you want to see *every* environment variable, run:

```
printenv
```

Look at that! You *won't* see anything that starts with "database", but we *should* after we deploy this next change. Type `exit`, run

```
git status
```

and then

```
git add -p
```

That's what we want! Commit with

```
git commit -m "adding database relation"
```

And

```
symfony deploy
```

This time, it deploys *way* faster. Because we didn't change any *application* code, platform.sh was smart enough to use our old app code, instead of doing all that building again. We can see that:

> *“Reusing existing build for this tree ID”*

And hey! This time, we see that it `Successfully migrated`! Yea! it ran our migration with zero problems. When we spin over and check the site... it *works*. It's still missing all of our styling... but we'll fix that next. The important thing is that the database *is* working.

You can see the difference that made if you run

```
symfony ssh
```

and

```
printenv
```

This time, there are several `DATABASE_` variables, including the most important `DATABASE_URL`.

Ok, the final missing piece from our deployed site is... all of its assets! Let's see what's needed to deploy an AssetMapper site next.

Chapter 20: Deploying the Assets

How do we get our assets onto the site? If you "View Page Source", it *looks* like things are working. We see the `importmap` and... down here, these paths look correct: they even have the version part in their filenames.

Compiling Assets for Production

Unfortunately, all of these files return a 404. Boo. In the `dev` environment, when we're working locally, these files don't physically exist. But an internal Symfony listener intercepts the request, finds the file, and serves them.

But in the `prod` environment, that system isn't even active. It's too slow to run on production... so everything just 404s. And that's okay! A long time ago, we learned about the command to fix this:

```
php bin/console asset-map:compile
```

This command's job is simple: it takes all the assets that AssetMapper knows about and move them into the `public/assets/` directory. It's not a command you need to run locally, but it *is* something you need to run when you deploy.

Copy this, head over to `.platform.app.yaml`, and go down to the `build` step. This is pretty cool! We're going to let Symfony do its `build` thing, and afterward, add our own stuff. Right here, add `php bin/console asset-map:compile`. That should do it!

Tip

Symfony + Platform.sh now detects AssetMapper and automatically runs this for you doing build. Woo!

```
.platform.app.yaml
// ... lines 1 - 43
hooks:
    build: |
// ... lines 46 - 49
        NODE_VERSION=18 symfony-build

        php bin/console asset-map:compile
// ... lines 53 - 58
```

Why are we running this during `build` and not `deploy`? As a rule of thumb, if a command's job is to "prepare" *files*, it should be in the `build` step. Or, another way to think about it is: if a command does *not* require a connection to the database or any other running services, there's a good chance it's a "build" thing.

Tip

Keep your "deploy" step as fast as possible because the incoming requests are held until it finishes. You can find more information here: https://docs.platform.sh/overview/build-deploy.html#deploy-steps

Head back over here and run:

```
git add -p
```

That whitespace bothers me... so I'll fix it and preserve my sanity. Then run `git commit -m` with a fancy message.

```
git commit -m "asset-map:compile"
```

You know what's next. Punch it!

```
symfony deploy
```

Let's fast-forward to the good part. Here it is! We see it running the command!

> *“Compiling assets to `/app/public/assets/` Compiled 16 assets”*

It also writes a few other files inside the `public/assets/` directory: `manifest.json` and `importmap.json`. They help Symfony dump the `importmap` and other things onto the page *even faster*.

And... done! Spin over, refresh, and... it *still* looks bad!? Ah, but things are not as bad as you think! Head to the homepage and open your Console. Hey! Our JavaScript *is* running! We see the `console.log()`!

Building Tailwind on Deploy

So JavaScript, check! CSS... not so much. We still have a 404 on `app.tailwind.css`.

Remember: when you see a 404 to a filename that does *not* include a version part, like here, it means that AssetMapper can't find that file. Can you spot the problem? This `app.tailwind.css` is a file that we're *building*... and it's *not* committed to the repository! I'll stop this command and re-run it so we can see the details. Yup, we're *building* the `app.tailwind.css` file, ignoring it from Git, and since Platform.sh deploys using our files *from* Git, that file is simply *missing*.

No big deal. This is just another thing we need to add to our `build` step... *before* we run `asset-map:compile` so that the file is available.

I'll paste in the code for this. This is basically the same code we ran earlier to set things up locally, except that we're using the `linux-x64` build. We're downloading that, moving it into the `/bin` directory (it doesn't really matter where it goes), making it executable, and then running that same command so that the output file *is* there by the time `asset-map:compile` runs.

```
.platform.app.yaml
// ... lines 1 - 43
hooks:
    build: |
// ... lines 46 - 51
        curl -sLO
https://github.com/tailwindlabs/tailwindcss/releases/download/v3.3.2/tailwin
linux-x64
        mv tailwindcss-linux-x64 bin/tailwindcss
        chmod +x bin/tailwindcss
        ./bin/tailwindcss -i assets/styles/app.css -o
assets/styles/app.tailwind.css
// ... lines 56 - 63
```

Oh, and don't forget about the TailwindBundle which makes using Tailwind - including this deploy step - a bit easier.

Back over here, let's commit that new config change.... then deploy again:

```
symfony deploy
```

Even while it's deploying, we can see that this working. Last time, there were 16 files, now there are 17. When it finishes, spin over, refresh and... it's *alive*! All the pages have CSS. I love it!

Now that we're on production, let's talk about the things we need to check to make sure our assets are served blazingly fast.

Chapter 21: Long-Term Caching, Compression & File Combining

It's time to celebrate! We're on production, and it was really just as simple as making sure Tailwind was being built and running the `asset-map:compile` command.

Now that we're here, there *are* a few things we need to check on, to make sure our site is *fast*.

1) HTTP/2: You Definitely Need It

The *first* is to check that your web server is using HTTP/2. We talked about that earlier, so hopefully you already got that rocking.

2) Combining Files

The *second* thing is... well.. not really a thing at all. I just want to point out that nothing in AssetMapper ever *combined* our files to reduce the number of HTTP requests. We're going to talk more about this in a few minutes, but thanks to HTTP/2, you almost definitely do *not* need to combine your files together. So if you were thinking that this was missing, it's not! It's by design. That's good! One less thing.

3) File Compression / Minification

But what about *minifying* our files? It's true: right now, our files are being served *without* minification, and that *is* a problem. We *do* want our CSS and JavaScript files to be minified... or at least *compressed*. And this is thing number three to think about.

But... this is something that can be done by our web server. Yup, if you ask kindly, you should be able to convince your web server to compress your files so they're smaller when being sent across the network. This is something that all web servers support, and it's done *automatically* by Platform.sh.

Here's how we can see it. Go to "Network" tools and select JavaScript. Select one of the files then go to "Headers". I'll make this a bit bigger. Okay, see this "Content-Encoding" response header? *That's* compression. This "br" stands for something called Brotli, which is more delicious than it sounds. Brotli is an advanced compression format. The other common value is `gzip`. So all of our static files *are* already being compressed! We get that for *free* with Platform.sh, so we can check that item off our list. Check your deploy system or web server docs for details on how to do this in *your* situation.

But wait, are minifying and compressing the same thing or different? Actually, they're a bit different. Both minifying and compressing *greatly* reduce the size of a file. We're using *compression*. Minification *can* result in slightly smaller files - in part because it will remove code comments - but it's not significant. Web servers themselves don't support minification, but if you use Cloudflare, there *is* a way to enable auto-minification. It probably won't make a *huge* difference, but you can try it.

4) Long-Term File Caching

The fourth and final thing that we need to do is make sure that all of our static files are set up for *long-term* expiration. Because we have these nice versioned file names, when a user visits our site, we want them to download this file *one* time and never, *ever* download it again. We want them to cache it forever. Because, if we change anything inside of this file, the whole filename will change! And the user's browser will naturally download the new version the next time they visit the site.

Over here, under "Headers", we *do* see an "Expires" response header. Out of the box, Platform.sh adds an "Expires" header for static assets, set to one hour. We can do *much* better than that.

Over in `.platform.app.yaml`, under `locations`... there we go... we see... `expires: 1h`. That's *fine* as a default for *other* static assets that we may have on our site. So I'm going to leave that alone. But add another rule to be more specific. For the regex, if the URL matches `/assets/` anything, then set the `expires` header to `365d`. Yes, 1 year - that's *forever* in Internet time!

```
.platform.app.yaml
// ... lines 1 - 30
web:
    locations:
        "/":
// ... lines 34 - 36
                rules:
                    '^\/assets\/.*':
                        expires: 365d
// ... lines 40 - 64
```

Cool! Lets commit that... and deploy, deploy!

```
symfony deploy
```

When that finishes, *refresh*. We won't see anything obvious... but let's check out one of our JavaScript files, like `app.js`. We're looking for "Expires". If you don't see it - or it's still short - do a force refresh. This is weird case where the file didn't change, but we might still have the cached version from a minute ago with the old header. And if you *do* see this, *congratulations*! That's the goal.

So, our assets are being compressed, and they have long-term "Expires" headers. We've checked *all* of our boxes for a performant site!

Next: We're going to *prove* the site is fast by using Lighthouse to *profile* the site's performance. We'll learn about how files are downloaded, how pages are built, and we'll make our frontend even *more* efficient.

Chapter 22: Optimizing & Profiling

Instead of me *telling* you the site is fast, let's prove it! In Chrome, there's a tool called "Lighthouse", which you can also get for some other browsers. Run this for just performance and select "Analyze page load".

This is the *best* way to see if you have any frontend performance problems. Our score will likely be pretty high - simply because our site is small and quick - but we can use the report to zero-in on a few possible problems. And... yep! We got a 98 with *no build system*! That's *amazing*. But we can do even better.

Eliminate Render-Blocking Resources

If we scroll down, we can see where our problem areas are. The first is "Eliminate render-blocking resources", which points to our font file. A lot of what we're going to talk about has nothing to do with AssetMapper: it's just frontend performance in general. If you open `templates/base.html.twig`, we have a `<link>` tag that points to this font file.

```
templates/base.html.twig
// ... lines 1 - 2
    <head>
// ... lines 4 - 11
        {% block stylesheets %}
// ... line 13
            <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/@fontsource-variable/inter@5.0.3/index.min.css">
// ... line 15
        {% endblock %}
// ... lines 17 - 21
    </head>
// ... lines 23 - 72
```

When your site sees a `<link rel="stylesheet">` tag, it downloads it before it renders the page. So it basically *freezes* the rendering of the page until the download finishes.

But this is interesting. Open that file... and let's get a non-minified version. It has a bunch of potential font faces. Here's how this works: our browser downloads *this* file immediately... but the font files themselves won't be downloaded *until* and *unless* we *use* this font. Additionally, `font-display: swap` tells the browser:

> *“Hey, it's ok to render some text that's supposed to use this font, even if the font isn't downloaded yet. You can use the default system font first, show the text, finish downloading this font file, and then use it.”*

Essentially, this CSS file is written in a way where all of these font files are going to be downloaded *lazily*. The *problem*, which isn't really a *big* problem, is that, at this point, our browser just sees a CSS file and thinks:

> *“I need to download that CSS file right now and I can't render the page until that finishes!”*

Once it *does* finally see the CSS contents, it discovers that there are a bunch of font files that it can lazily download.

So, CSS files are render-blocking resources... which is *normally* great, because we *don't* want the page to render unstyled for a half second before the CSS downloads. But *this* particular file is funny because it is a "render-blocking" resource... but doesn't contain anything critical.

If we care enough to eliminate this render-blocking resource, we can move it into `app.css`. Start by copying this file... or really just the font faces we need: a lot of these are for languages that we're not using. I'll copy the two Latin fonts... though we likely don't even need this Latin extension one. Then delete this CSS file entirely, go to `assets/styles/app.css`, and paste. These aren't real URLs... so go copy the URL... paste, take off the `index.css`... and that should be fine. Copy the URL again... and do the same thing down here.

```
assets/styles/app.css
// ... lines 1 - 4
/* inter-latin-ext-wght-normal */
@font-face {
  font-family: 'Inter Variable';
  font-style: normal;
  font-display: swap;
  font-weight: 100 900;
  src: url(https://cdn.jsdelivr.net/npm/@fontsource-variable/inter@5.0.3/files/inter-latin-ext-wght-normal.woff2) format('woff2-variations');
  unicode-range: U+0100-02AF,U+0300-0301,U+0303-0304,U+0308-0309,U+0323,U+0329,U+1E00-1EFF,U+2020,U+20A0-20AB,U+20AD-20CF,U+2113,U+2C60-2C7F,U+A720-A7FF;
}

/* inter-latin-wght-normal */
@font-face {
  font-family: 'Inter Variable';
  font-style: normal;
  font-display: swap;
  font-weight: 100 900;
  src: url(https://cdn.jsdelivr.net/npm/@fontsource-variable/inter@5.0.3/files/inter-latin-wght-normal.woff2) format('woff2-variations');
  unicode-range: U+0000-00FF,U+0131,U+0152-0153,U+02BB-02BC,U+02C6,U+02DA,U+02DC,U+0300-0301,U+0303-0304,U+0308-0309,U+0323,U+0329,U+2000-206F,U+2074,U+20AC,U+2122,U+2191,U+2193,U+2212,U+2215,U+FEFF,U+FFFD;
}
// ... lines 24 - 37
```

Perfect. This *is* adding some complexity to our code for only a small gain, so I'd say this is a lower priority. We have basically the same amount of CSS as before, but we've eliminated a small, unnecessary blocking resource.

The other failure we have is similar. It's for FontAwesome - specifically, this JavaScript file. That's *also* in `base.html.twig`. Since this `<script>` tag doesn't have `defer` or `async` on it, this will *also* block the rendering of the page. If we want, we can add `defer` to this, which says:

> *“Start download this immediately, but don't block the page while it's finishing.”*

```
templates/base.html.twig
// ... lines 1 - 2
    <head>
// ... lines 4 - 16
        {% block javascripts %}
// ... line 18
            <script defer src="https://kit.fontawesome.com/5a377fab5b.js" crossorigin="anonymous"></script>
        {% endblock %}
    </head>
// ... lines 22 - 71
```

Because this is for FontAwesome fonts, the worst-case scenario is that the page loads and then our font icons show up just a moment later.

Profiling Again!

Okay, now that we've changed a couple of things, let's *test* it. To save time redeploying, I'll go back to my local site and run Lighthouse again. "Analyze page load"... make this a bit bigger, and... awesome! We're getting 100 locally!

But if you look down here... we *do* still have some opportunities to improve. We see "Serve images in next gen formats", which is a good thing to check on later, but not related to Symfony or AssetMapper. This "Avoid serving legacy JavaScript to modern browsers" - I believe that's referring to the `importmap` *shim*: the code that makes the importmap work on all browsers. That's small & necessary, so not a big deal.

Avoiding Chaining Critical Requests

But below *that*, we see "Avoid chaining critical requests". This is probably *the* most important item on this list.

Here's what's happening. As you can see, it downloads the HTML first. Once it does, it realizes that it needs to download this CSS file. Once it downloads the CSS file, it realizes that it needs to download this font file. See the problem? Instead of knowing - from the start - that it needs these and downloading them all at once in parallel, our browser is finding out about them little by little. Ultimately, it means this font file will take *longer* to load because it can't *start* downloading it until it downloads a few other files.

How can we fix this? Preloading. Let's talk about this important topic next.

Chapter 23: Preloading

We just discovered a problem: our browser needs to download the page... *and* a CSS file before it even *realizes* that it needs to download this font. This may not be a huge deal, but there's a cool solution: preloading.

Manually Adding a link rel="preload"

Go find the font URL - it's this "normal" one - and copy it. Next, open `base.html.twig` and, up here, add a `<link>` tag. Unlike a normal `<link rel="stylesheet"` that points to a CSS file, the purpose of *this* link tag will be to yell to the browser:

> *“Hey, you don't know it yet, but you should download this font file.”*

To do that, say `rel="preload"` then `href=""` and paste that long URL. And when preloading fonts, we need to add `as="font"`, `type="font/woff2"`, and `crossorigin=""` at the end.

templates/base.html.twig

```
// ... lines 1 - 2
    <head>
// ... lines 4 - 10
        <link rel="preload"
href="https://cdn.jsdelivr.net/npm/@fontsource-
variable/inter@5.0.3/files/inter-latin-wght-normal.woff2" as="font"
type="font/woff2" crossorigin>
// ... lines 12 - 21
    </head>
// ... lines 23 - 72
```

Ok, let's see what Lighthouse thinks of this! After... we *still* score 100 - yay! - and under "Avoid chaining critical requests", that font file is *gone*.

Preloading via a Header

But... what about `app.tailwind.css`? The browser downloads the HTML and then immediately sees the `link` tag for this. Is there a way to *hint* to the browser that it needs to download `app.tailwind.css` even *before* it downloads the HTML? The answer, surprisingly, is *yes*!

But we need a Symfony component called "WebLink". At your terminal, run:

```
composer require symfony/web-link
```

Once that's done, back in `base.html.twig`, add *another* preload down here that looks similar: `<link rel="preload" href="">`. This time, use a Twig function called `preload()` passing the normal `asset()` function to point to `styles/app.tailwind.css`. In this case, this `preload` function needs another option called `as: 'style'`.

templates/base.html.twig

```
// ... lines 1 - 2
    <head>
// ... lines 4 - 10
        <link rel="preload" href="{{
preload(asset('styles/app.tailwind.css'), { as: 'style' }) }}">
// ... lines 12 - 22
    </head>
// ... lines 24 - 73
```

That's it! Go refresh the page and "View page source". No surprise: this outputs a `preload` tag. And... so far, the `preload()` Twig function looks like nothing more than a semi-worthless shortcut!

Even more, for the `app.tailwind.css` file, this `link` tag is pointless! This basically tells the browser:

> *"Hey, you should start downloading this CSS file."*

But... one line later, it would have found it anyway! So why did we do this? It turns out that the `preload()` function does two things: it outputs the `link` tag `href`... but it *also* tells Symfony that it should add a `preload` *header* to the response.

Go to the Network tools, select "All", find our main page, go to "Headers", and look under "Response Headers" Woh! We have a new "Link" header called "preload" that points to our

CSS file! So as the browser starts downloading the response, at the *very* top it sees a *hint* that it should start downloading that CSS file!

If we go back over to Lighthouse and analyze the page load again... down here... beautiful! That entire section is *gone*.

preload JavaScript in importmap.php

There are a few other things here like "Keep request counts low and transfer sizes small", but these aren't really warnings: just something to keep in mind.

But on the topic of preloading, even though we don't have any more warnings, there *is* another spot where preloading can improve performance, and it has to do with JavaScript.

Over in `importmap.php`, there's a key called `preload` that we haven't talked about. It's set to `true` for `app`. Set that to `false`.

importmap.php

```php
// ... lines 1 - 15
return [
    'app' => [
// ... line 18
        'preload' => false,
    ],
// ... lines 21 - 32
];
```

Now, move over and run another Lighthouse report. We still get a score of 100, but if we go down here, ah: "Avoid chaining critical requests" is back! And check it out! We have the HTML page, down to `app.js`, then `bootstrap.js`, the Stimulus loader, Stimulus itself, controllers... *wow*. A *bunch* of stuff is chaining.

This is the same problem we saw with the CSS and font files. First, our browser downloads the HTML. *Then* it sees that it needs to download `app.js`. Once it downloads *that* file, it sees that it needs to download `bootstrap.js`. Then, it realizes it needs to download the `stimulus-bundle/loader`, and so on: one-by-one-by-one. Instead of downloading *all* of those things in parallel, it has to *discover* them little-by-little.

`preload` fixes that. Change this back to `true`, refresh the page, and view page's source.

```php
importmap.php
// ... lines 1 - 15
return [
    'app' => [
// ... line 18
        'preload' => true,
    ],
// ... lines 21 - 32
];
```

We know that the `importmap()` Twig function dumps the `importmap` and the `<script type="module">`. But it *also* dumps these `modulepreload` things. These are cool. Because we said `'preload' => true` for `app`, it adds a `<link rel="modulepreload">` for `app.js`. That hints to the browser that it should start downloading `app.js` *immediately*. Though, that's not really important because it would have figured that out in about 1 microsecond anyway.

The *real* power is that AssetMapper *then* sees that `assets/app.js` imports `bootstrap.js`. And because `app.js` is preloaded, it *also* preloads `bootstrap.js`. And since this imports `./lib/vinyl.js`, it *also* preloads `./lib/vinyl.js`. So it will download all three of these files immediately.

At this point, if we ran Lighthouse again, it wouldn't complain about any of these chained requests. But we *still* have room for improvement. On the network tools, for JavaScript, check out the waterfall column. We see that a few files *start* downloading, and then a few more... and a few more. So we still have this chaining problem, though it's apparently not a big enough deal for Lighthouse to report on.

We know that `bootstrap.js` is being preloaded, but `@symfony/stimulus-bundle` *isn't*... even though it's imported from `bootstrap.js`. Why doesn't the preload "follow" that import like the others?

The key thing to understand is that, because we're preloading `app.js`, anything that `app.js` imports with a *relative* import will automatically be preloaded as well. But anything we import with a *bare* import, like `lodash/camelCase` or `@symfony/stimulus-bundle`, *won't* be preloaded automatically. Perhaps they should, but they have their own entries inside of `importmap.php`, so you control the preloading for those independently.

At this point, we're *really* optimizing performance - maybe over-optimizing. But if you want to avoid this chaining problem, you could add `preload` to the items we *know* will be critical to the page rendering. For example, `@hotwired/stimulus` is critical, `stimulus-bundle` is critical because that's what loads our controllers, and `@hotwired/turbo` is also critical.

```
importmap.php
// ... lines 1 - 15
return [
// ... lines 17 - 23
    '@hotwired/stimulus' => [
// ... line 25
        'preload' => true,
    ],
    '@symfony/stimulus-bundle' => [
// ... line 29
        'preload' => true,
    ],
    '@hotwired/turbo' => [
// ... line 33
        'preload' => true,
    ],
];
```

When we refresh... nothing changes: we just have more `modulepreload` items in the HTML. If we run Lighthouse one more time, we're *still* scoring 100, and you can see that there are no major problems down here. *Fantastic*.

Preload Everything

By the way, if you're now thinking:

> *"Why don't we just preload everything?"*

That's a good thought! But, don't! Your browser is smart, and without any preloads, it has a highly-intelligent algorithm to determine the best order to download files to load things as fast as possible. If you preloaded everything, the loading order *probably* wouldn't be as good. Just use preloads for *critical* assets.

Ok! We made it! I think AssetMapper is a breath of fresh air - and I hope you feel the same! There are some things it *doesn't* do, like tree-shaking or handling TypeScript. But for a large number of projects, it's a great fit! And the cool thing is, you're still writing normal JavaScript. So if you ever *did* need to move to a build system later, you could do that.

Let us know what you think, and hopefully we can make more improvements for Symfony 6.4.

Alright friends, see you next time!