Webpack Encore: Frontend like a Pro!

Chapter 1: Hello Webpack Encore

Yo friends! It's Webpack time! Yeeeeeeah! Well, maybe not *super* "yeeeeeeah!" if *you* are the person responsible for installing and *configuring* Webpack... Cause, woh, yea, that can be tough! Unless... you're using Webpack Encore! More about that in a few minutes.

Why all the Webpack Buzz?

But first, I want you to know why you should *care* about Webpack... like super please-let-me-start-using-webpack-right-now-and-never-stop-using-it kind of care. Sure, *technically*, Webpack is just a tool to build & compile your JavaScript and CSS files. But, it will *revolutionize* the way you write JavaScript.

The reason is *right* on their homepage! In PHP, we organize our code into small files that work together. But then, traditionally, in JavaScript, we just *smash* all our code into one gigantic file. Or, if we *do* split them up, it's still a pain because we *then* need to remember to go add a new `script` tag to every page that needs it... and those script tags need to be in *just* the right order. If they're not, kaboom! And even if you *do* have a build system like Gulp, you *still* need to manage keeping all of the files listed there and in the right order. How can our code be *so* nicely organized in PHP, but such a disaster in JavaScript?

Webpack changes this. Suppose we have an `index.js` file but we want to organize a function into a different, `bar.js` file. Thanks to Webpack, you can "export" that function as a value from `bar.js` and then *import* and use it in `index.js`. Yes, we can *organize* our code into small pieces! Webpack's job is to read `index.js`, parse through *all* of the `import` statements it finds, and output *one* JavaScript file that has everything inside of it. Webpack is a huge over-achiever.

So let's get to it! To import or... *Webpack* the *maximum* amount of knowledge from this tutorial, download the course code from this page and code along with me. After you unzip the file, you'll find a `start/` directory that has the same code I have here: a Symfony 4 app. Open up the `README.md` file for all the setup details.

The last step will be to open a terminal, move into the project and start a web server. I'm going to use the Symfony local web server, which you can get from https://symfony.com/download.

Run it with:

```
symfony serve
```

Then, swing back over to your browser and open up https://localhost:8000 to see... The Space Bar! An app we've been working on throughout our Symfony series. And, we *did* write some JavaScript and CSS in that series... but we kept it *super* traditional: the JavaScript is pretty boring, and there are multiple files but each has its own `script` tag in my templates.

This is *not* the way I really code. So, let's do this correctly.

Installing WebpackEncoreBundle + Recipe

So even though both Webpack and Encore are *Node* libraries, if you're using Symfony, you'll install Encore via composer... well... sort of. Open a new terminal tab and run:

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

This downloads a small bundle called WebpackEncoreBundle. Actually, Encore *itself* can be used with any framework or any language! But, it works *super* well with Symfony, and this thin bundle is part of the reason.

This bundle *also* has a Flex *recipe* - oooooOOOOooo - which gives us some files to get started! If you want to use Webpack from *outside* of a Symfony app, you would just need these files in your app.

Back in the editor, check out `package.json`:

package.json

```
{
    "devDependencies": {
        "@symfony/webpack-encore": "^0.27.0",
        "core-js": "^3.0.0",
        "webpack-notifier": "^1.6.0"
    },
    "license": "UNLICENSED",
    "private": true,
    "scripts": {
        "dev-server": "encore dev-server",
        "dev": "encore dev",
        "watch": "encore dev --watch",
        "build": "encore production --progress"
    }
}
```

This is the `composer.json` file of the Node world. It requires Encore itself plus two optional packages that we'll use:

package.json

```
{
    "devDependencies": {
        "@symfony/webpack-encore": "^0.27.0",
        "core-js": "^3.0.0",
        "webpack-notifier": "^1.6.0"
    },
// ... lines 7 - 14
}
```

Installing Encore via Yarn

To actually *download* these, go back to your terminal and run:

```
yarn
```

Or... `yarn install` if you're less lazy than me - it's the same thing. Node has *two* package managers - "Yarn" and "npm" - I know, kinda weird - but you can install and use whichever you want. Anyways, this is downloading our 3 libraries and their dependencies into Node's version of the `vendor/` directory: `node_modules/`.

And... done! Congrats! You now have a gigantic `node_modules/` directory... because JavaScript has tons of dependencies. Oh, the recipe *also* updated our `.gitignore` file to *ignore* `node_modules/`:

.gitignore

```
// ... lines 1 - 22
###> symfony/webpack-encore-bundle ###
/node_modules/
/public/build/
// ... lines 26 - 27
###< symfony/webpack-encore-bundle ###
// ... lines 29 - 31
```

Just like with Composer, there is *no* reason to commit this stuff. This *also* ignores `public/build/`, which is where Webpack will *put* our final, built files.

Hello webpack.config.js

In fact, I'll show you why. At the root of your app, the recipe added the most important file of all `webpack.config.js`:

webpack.config.js

```
var Encore = require('@symfony/webpack-encore');

Encore
    // directory where compiled assets will be stored
    .setOutputPath('public/build/')
    // public path used by the web server to access the output path
    .setPublicPath('/build')
    // only needed for CDN's or sub-directory deploy
    //.setManifestKeyPrefix('build/')

    /*
     * ENTRY CONFIG
     *
     * Add 1 entry for each "page" of your app
     * (including one that's included on every page - e.g. "app")
     *
     * Each entry will result in one JavaScript file (e.g. app.js)
     * and one CSS file (e.g. app.css) if you JavaScript imports CSS.
     */
    .addEntry('app', './assets/js/app.js')
    //.addEntry('page1', './assets/js/page1.js')
    //.addEntry('page2', './assets/js/page2.js')

    // When enabled, Webpack "splits" your files into smaller pieces for greater optimization.
    .splitEntryChunks()

    // will require an extra script tag for runtime.js
    // but, you probably want this, unless you're building a single-page app
    .enableSingleRuntimeChunk()

    /*
     * FEATURE CONFIG
     *
     * Enable & configure other features below. For a full
     * list of features, see:
     * https://symfony.com/doc/current/frontend.html#adding-more-features
     */
    .cleanupOutputBeforeBuild()
    .enableBuildNotifications()
    .enableSourceMaps(!Encore.isProduction())
    // enables hashed filenames (e.g. app.abc123.css)
    .enableVersioning(Encore.isProduction())

    // enables @babel/preset-env polyfills
    .configureBabel(() => {}, {
```

```
        useBuiltIns: 'usage',
        corejs: 3
    })

    // enables Sass/SCSS support
    //.enableSassLoader()

    // uncomment if you use TypeScript
    //.enableTypeScriptLoader()

    // uncomment to get integrity="..." attributes on your script & link tags
    // requires WebpackEncoreBundle 1.4 or higher
    //.enableIntegrityHashes()

    // uncomment if you're having problems with a jQuery plugin
    //.autoProvidejQuery()

    // uncomment if you use API Platform Admin (composer req api-admin)
    //.enableReactPreset()
    //.addEntry('admin', './assets/js/admin.js')
;

module.exports = Encore.getWebpackConfig();
```

This is the configuration file that Encore reads. Actually, if you use Webpack by itself, you would have this *exact* same file! Encore is basically a configuration generator: you tell it how you want Webpack to behave and then, *all* the way at the bottom, say:

> *“Please give me the standard Webpack config that will give me that behavior.”*

Encore makes things easy, but it's *still* true Webpack under-the-hood.

Most of the stuff in this file is for configuring some optional features that we'll talk about along the way - so ignore it all for now. The three *super* important things that *we* need to talk about are output path, public path and this `addEntry()` thing:

webpack.config.js

```
var Encore = require('@symfony/webpack-encore');

Encore
    // directory where compiled assets will be stored
    .setOutputPath('public/build/')
    // public path used by the web server to access the output path
    .setPublicPath('/build')
// ... lines 8 - 10
    /*
     * ENTRY CONFIG
     *
     * Add 1 entry for each "page" of your app
     * (including one that's included on every page - e.g. "app")
     *
     * Each entry will result in one JavaScript file (e.g. app.js)
     * and one CSS file (e.g. app.css) if you JavaScript imports CSS.
     */
    .addEntry('app', './assets/js/app.js')
    //.addEntry('page1', './assets/js/page1.js')
    //.addEntry('page2', './assets/js/page2.js')
// ... lines 23 - 65
;
// ... lines 67 - 68
```

Let's do that next, *build* our first Webpack'ed files and include them on the page.

Chapter 2: Webpacking our First Assets

So, Webpack only needs to know three things. The first - `setOutputPath()` - tells it *where* to put the final, built files and the second - `setPublicPath()` - tells it the public path to this directory:

webpack.config.js

```
// ... lines 1 - 2
Encore
    // directory where compiled assets will be stored
    .setOutputPath('public/build/')
    // public path used by the web server to access the output path
    .setPublicPath('/build')
// ... lines 8 - 65
;
// ... lines 67 - 68
```

The Entry File

The *third* important piece, and where *everything* truly starts, is `addEntry()`:

Tip

The Encore recipe *now* puts `app.js` in `assets/app.js`. But the purpose of the file is exactly the same!

webpack.config.js

```
// ... lines 1 - 2
Encore
// ... lines 4 - 10
    /*
     * ENTRY CONFIG
     *
     * Add 1 entry for each "page" of your app
     * (including one that's included on every page - e.g. "app")
     *
     * Each entry will result in one JavaScript file (e.g. app.js)
     * and one CSS file (e.g. app.css) if you JavaScript imports CSS.
     */
    .addEntry('app', './assets/js/app.js')
    //.addEntry('page1', './assets/js/page1.js')
    //.addEntry('page2', './assets/js/page2.js')
// ... lines 23 - 65
;
// ... lines 67 - 68
```

Here's the idea: we point Webpack at just *one* JavaScript file - `assets/js/app.js`. Then, it parses through *all* the import statements it finds, puts all the code together, and outputs one file in `public/build` called `app.js`. The first argument - `app` - is the entry's name, which can be anything, but it controls the final filename: `app` becomes `public/build/app.js`.

And the recipe gave us a few files to start. Open up `assets/js/app.js`:

Tip

The recipe now puts CSS files into an `assets/styles/` directory. So, `assets/styles/app.css` - but the purpose of all these files is the same.

assets/js/app.js

```
/*
 * Welcome to your app's main JavaScript file!
 *
 * We recommend including the built version of this JavaScript file
 * (and its CSS file) in your base layout (base.html.twig).
 */

// any CSS you require will output into a single css file (app.css in this case)
require('../css/app.css');

// Need jQuery? Install it with "yarn add jquery", then uncomment to require it.
// const $ = require('jquery');

console.log('Hello Webpack Encore! Edit me in assets/js/app.js');
```

This is the file that Webpack will start reading. There's not much here yet - a `console.log()` and... woh! There *is* one cool thing: a `require()` call to a CSS file! We'll talk more about this later, but in the same way that you can import other JavaScript files, you can import CSS too! And, by the way, this `require()` function and the `import` statement we saw earlier on Webpack's docs, do basically the same thing. More on that soon.

To make the CSS a bit more obvious, open `app.css` and change the background to `lightblue` and add an `!important` so it will override my normal background:

assets/css/app.css

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

`disableSingleRuntimeChunk()`

Before we execute Encore, back in `webpack.config.js`, we need to make one other small tweak. Find the `enableSingleRuntimeChunk()` line, comment it out, and put `disableSingleRuntimeChunk()` instead:

webpack.config.js

```
// ... lines 1 - 2
Encore
// ... lines 4 - 26
    // will require an extra script tag for runtime.js
    // but, you probably want this, unless you're building a single-page app
    //.enableSingleRuntimeChunk()
    .disableSingleRuntimeChunk()
// ... lines 31 - 66
;
// ... lines 68 - 69
```

Don't worry about this yet - we'll see *exactly* what it does later.

Running Encore

Ok! We've told Webpack *where* to put the built files and which *one* file to start parsing. Let's do this! Find your terminal and run the Encore executable with:

```
./node_modules/.bin/encore dev
```

Tip

For Windows, your command may need to be `node_modules\bin\encore.cmd dev`

Because we want a development build. And... hey! A nice little notification that it worked!

And... interesting - it built *two* files: `app.js` and `app.css`. You can see them inside the `public/build` directory. The `app.js` file... well... basically just contains the code from the `assets/js/app.js` file because... that file didn't import any other JavaScript files. We'll change that soon. But our `app` entry file *did* require a CSS file:

assets/js/app.js

```
// ... lines 1 - 7
// any CSS you require will output into a single css file (app.css in this case)
require('../css/app.css');
// ... lines 10 - 15
```

And yea, Webpack understands this!

Here's the *full* flow. First, Webpack looks at `assets/js/app.js`. It then looks for *all* the `import` and `require()` statements. Each time we import a JavaScript file, it puts those contents into the final, built `app.js` file. And each time we import a CSS file, it puts *those* contents into the final, built `app.css` file.

Oh, and the final filename - `app.css`? It's `app.css` because our *entry* is called app. If we changed this to `appfoo.css`, renamed the file, then ran Encore again, it would *still* build `app.js` and `app.css` files thanks to the first argument to `addEntry()`:

webpack.config.js

```
↕ // ... lines 1 - 2
3 Encore
↕ // ... lines 4 - 19
20     .addEntry('app', './assets/js/app.js')
↕ // ... lines 21 - 66
67 ;
↕ // ... lines 68 - 69
```

Adding the Script & Links Tags

What this means is... we now have *one* JavaScript file that contains *all* the code we need and one CSS file that contains all the CSS! All *we* need to do is add them to our page!

Open up `templates/base.html.twig`. Let's keep the existing stylesheets for now and add a new one: `<link rel="stylesheet" href="">` the `asset()` function and the public path: `build/app.css`:

templates/base.html.twig

```
1  <!doctype html>
2  <html lang="en">
3
4      <head>
↕  // ... lines 5 - 8
9          {% block stylesheets %}
10             <link rel="stylesheet" href="{{ asset('build/app.css') }}">
↕  // ... lines 11 - 14
15         {% endblock %}
16     </head>
↕  // ... lines 17 - 107
108 </html>
```

At the bottom, add the `script` tag with `src="{{ asset('build/app.js') }}"`.

> **Tip**
>
> In new Symfony projects, the `javascripts` block is at the top of this file - inside the `<head>` tag. We'll learn more about why in a few minutes.

templates/base.html.twig

```
<!doctype html>
<html lang="en">
// ... lines 3 - 17
    <body>
// ... lines 19 - 90
        {% block javascripts %}
            <script src="{{ asset('build/app.js') }}"></script>
// ... lines 93 - 105
        {% endblock %}
    </body>
</html>
```

If you're not familiar with the `asset()` function, it's not doing anything important for us. Because the `build/` directory is our document root, we're literally pointing to the public path.

Let's try it! Move over, refresh and... hello, weird blue background. And in the console... yes! There's the log!

> **Tip**
>
> If you're coding along with a fresh Symfony project, you likely will *not* see the `console.log()` being printed. That's ok! In the next chapter, you'll learn about a Twig function that will render some `<script>` tags that you're currently missing.

We've only started to scratch the surface of the possibilities of Webpack. So if you're still wondering: "why is going through this build process so useful?". Stay tuned. Because next, we're going to talk about the `require()` and `import` statements and start organizing our code.

Chapter 3: Twig Helpers, entrypoints.json & yarn Scripts

Encore is outputting `app.css` and `app.js` thanks to the `app` entry:

webpack.config.js

```
↕ // ... lines 1 - 2
3 Encore
↕ // ... lines 4 - 19
20     .addEntry('app', './assets/js/app.js')
↕ // ... lines 21 - 66
67 ;
↕ // ... lines 68 - 69
```

And we successfully added the `<link>` tag for `app.css` and, down here, the `<script>` for `app.js`:

templates/base.html.twig

```
1  <!doctype html>
2  <html lang="en">
3
4      <head>
↕  // ... lines 5 - 8
9          {% block stylesheets %}
10             <link rel="stylesheet" href="{{ asset('build/app.css') }}">
↕  // ... lines 11 - 14
15         {% endblock %}
16     </head>
17
18     <body>
↕  // ... lines 19 - 90
91         {% block javascripts %}
92             <script src="{{ asset('build/app.js') }}"></script>
↕  // ... lines 93 - 105
106        {% endblock %}
107    </body>
108 </html>
```

The Twig Helper Functions

But when you use Encore with Symfony, you *won't* render script and link tags by hand. No way! We're going to be *way* lazier, and use some helper functions from WebpackEncoreBundle. For the stylesheets, use `{{ encore_entry_link_tags() }}` and pass it `app`, because that's the name of the entry:

templates/base.html.twig

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

    <head>
// ... lines 5 - 8
        {% block stylesheets %}
            {{ encore_entry_link_tags('app') }}
// ... lines 11 - 14
        {% endblock %}
    </head>
// ... lines 17 - 107
</html>
```

At the bottom, replace the script tag with almost the same thing: `{{ encore_entry_script_tags('app') }}`:

Tip

In new Symfony projects, the `javascripts` block is at the top of this file - inside the `<head>` tag. Also, Encore will render a `defer` attribute on each `script` tag. To follow this tutorial, in `config/packages/webpack_encore.yaml`, comment-out the `defer: true` key to avoid this. For more info about `defer` and its performance benefits, check out https://symfony.com/blog/moving-script-inside-head-and-the-defer-attribute

templates/base.html.twig

```
<!doctype html>
<html lang="en">
// ... lines 3 - 17
    <body>
// ... lines 19 - 90
        {% block javascripts %}
            {{ encore_entry_script_tags('app') }}
// ... lines 93 - 105
        {% endblock %}
    </body>
</html>
```

Move over and refresh to try this. Wow! This made absolutely *no* difference! The `<link>` tag on top looks *exactly* the same. And... if I search for "script"... yep! That's *identical* to what we had before.

So... why? Or maybe better, *how*? Is it just taking the `app` and turning it into `build/app.js`? Not quite... it's a *bit* more interesting than that.

In the `public/build/` directory, Encore generates a very special file called `entrypoints.json`. This is a map from each entry name to the CSS and JS files that are needed to make it run. If you were listening closely, I just said *two* strange things. First, we only have one entry right now. But yes, we *will* eventually have *multiple* entries to power page-specific CSS and JS. Second, for performance, *eventually* Webpack *may* split a single entry into *multiple* JavaScript and CSS files and we will need *multiple* script and link tags. We'll talk more about that later.

The important thing right now is: we have these handy helpers that output the exact link and script tags we need... even if we need multiple.

Using --watch

Ok, back to Encore. Because it's a *build* tool, each time you make a change to anything, you need to rebuild:

```
./node_modules/.bin/encore dev
```

That's lame. So, of course, Webpack *also* has a "watch" mode. Re-run the same command but with `--watch` on the end:

```
./node_modules/.bin/encore dev --watch
```

Encore boots up, builds and... just chills out and waits for more changes. Let's test this. In `app.js`, I think we need a few more exclamation points:

assets/js/app.js

```
// ... lines 1 - 13
console.log('Hello Webpack Encore! Edit me in assets/js/app.js!!!');
```

Save, then check out the terminal. Yea! It already rebuilt! In your browser, refresh. Boom! Extra exclamation points. If that doesn't work for some reason, do a force refresh.

Shortcut "scripts"

But even *that* is too much work. Press `Ctrl`+`C` to stop Encore. Instead, just run:

```
yarn watch
```

That's a shortcut to do the *same* thing. You can even see it in the output: `encore dev --watch`. But there's *no* magic here. Open up `package.json`. We got this file from the recipe when we installed the WebpackEncoreBundle via Composer. See this `scripts` section?

package.json

```
{
// ... lines 2 - 8
    "scripts": {
        "dev-server": "encore dev-server",
        "dev": "encore dev",
        "watch": "encore dev --watch",
        "build": "encore production --progress"
    }
}
```

This is a feature of Yarn and npm: you can add "shortcut" commands to make your life easier. `yarn watch` maps to `encore dev --watch`. Later, we'll use `yarn build` to generate our assets for *production*.

With all this done, let's get back to the *core* of why Webpack is awesome: being able to import and require other JavaScript. That's next.

Chapter 4: Modules: require() & import()

Let's get back to talking about the *real* power of Webpack: the ability to import or require JavaScript files. Pretend that building this string is actually a lot of work. Or maybe it's something we need to re-use from somewhere else in our code:

assets/js/app.js

```
↕ // ... lines 1 - 13
14 console.log('Hello Webpack Encore! Edit me in assets/js/app.js!!!');
```

So, we want to isolate it into its own file. If this were PHP, we would create a new file to hold this logic. In JavaScript, we're going to do the same thing.

In `assets/js/`, create a new file called `get_nice_message.js`. *Unlike* PHP, in JavaScript, each file that you want to use somewhere else needs to *export* something, like a function, object, or even a string. Do that by saying `module.exports =` and then the thing you want to export. Let's create a `function()` with one argument `exclamationCount`:

assets/js/get_nice_message.js

```
1 module.exports = function(exclamationCount) {
↕ // ... line 2
3 };
```

Inside, let's go steal our string... then return that string and, to increase our fanciness, add `'!'.repeat(exclamationCount)`:

assets/js/get_nice_message.js

```
1 module.exports = function(exclamationCount) {
2     return 'Hello Webpack Encore! Edit me in
  assets/js/app.js'+'!'.repeat(exclamationCount);
3 };
```

Yes. Because strings are *objects* in JavaScript, this works - it's kinda cool. By the way, when a JavaScript file exports a value like this, it's known as a "module". That's not a big deal, but you'll hear this term a lot: JavaScript *modules*. OooOOOoo. It just refers to what we're doing here.

Now go back to `app.js`. At the top, well... it doesn't need to be on top, but usually we organize the imports there, add `const getNiceMessage = require('./get_nice_message');`:

assets/js/app.js

```
/*
 * Welcome to your app's main JavaScript file!
 *
 * We recommend including the built version of this JavaScript file
 * (and its CSS file) in your base layout (base.html.twig).
 */

// any CSS you require will output into a single css file (app.css in this case)
require('../css/app.css');

// Need jQuery? Install it with "yarn add jquery", then uncomment to require it.
// const $ = require('jquery');

const getNiceMessage = require('./get_nice_message');
// ... lines 15 - 17
```

Notice the `.js` extension is optional, you can add it or skip it - Webpack knows what you mean. And because, key strokes are expensive... and programmers are lazy, you usually don't see it.

Also, that `./` at the beginning *is* important. When you're pointing to a file *relative* to the current one, you need to start with `./` or `../`. If you *don't*, Webpack will think you're trying to import a third-party package. We'll see that soon.

And now that we have our `getNiceMessage()` function, let's call it! Pass it 5 for *just* the right number of excited exclamation points:

assets/js/app.js

```
// ... lines 1 - 13
const getNiceMessage = require('./get_nice_message');

console.log(getNiceMessage(5));
```

And because we're running the `watch` command in the background, when we refresh, it just works!

import Versus require

But! When we originally looked at the Webpack docs, they weren't using `require()` and `module.exports`! Nope, they were using `import` and `export`. It turns out, there are *two* valid ways to export and import values from other files... and they're *basically* identical.

To use the *other* way, remove `module.exports` and say `export default`:

assets/js/get_nice_message.js

```
export default function(exclamationCount) {
// ... line 2
};
```

That does the *same* thing. The `default` is important. With this syntax, a module, so, a file, can export *more* than one thing. We're not going to talk about that here, but most of the time, you'll want to export just *one* thing, and this `default` keyword is how you do that.

Next, back in `app.js`, the `require` changes to `import getNiceMessage from './get_nice_message'`:

assets/js/app.js

```
// ... lines 1 - 13
import getNiceMessage from './get_nice_message';
// ... lines 15 - 17
```

That's it! That is 100% the same as what we had before. So, which should you use? Use *this* syntax. The `require()` function comes from Node. But the `import` and `export` syntax are the *official* way to do module loading in ECMAScript, which is the actual name for the JavaScript language specification.

You can - and *should* - also use this for CSS. Just `import`, then the path:

assets/js/app.js

```
// ... lines 1 - 7
// any CSS you require will output into a single css file (app.css in this case)
import '../css/app.css';
// ... lines 10 - 17
```

There's no `from` in this case because we don't need it to return a value to us.

Make sure *all* this coolness works: refresh! Yes!

Woh! Hey! Shut the front door! Did we just organize our JavaScript without global variables? Yes! We totally did! And that is *no* small thing. Heck, we could stop the tutorial right now, and you would *still* have this amazing superpower.

But... we won't! There is still *so* much cool stuff to talk about. Like, how we can now *super* easily install third-party libraries via Yarn and import them in our code. Let's do it!

Chapter 5: Importing External Libraries & Global Variables

We already added the `app` entry files to our base layout: the `<script>` tag and the `<link>` tag both live here:

templates/base.html.twig

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

    <head>
// ... lines 5 - 8
        {% block stylesheets %}
            {{ encore_entry_link_tags('app') }}
// ... lines 11 - 14
        {% endblock %}
    </head>

    <body>
// ... lines 19 - 90
        {% block javascripts %}
            {{ encore_entry_script_tags('app') }}
// ... lines 93 - 105
        {% endblock %}
    </body>
</html>
```

This means that *any* time we have some CSS or JavaScript that should be included on every page, we can put it in `app.js`.

Look down at the bottom:

templates/base.html.twig

```
<!doctype html>
<html lang="en">
// ... lines 3 - 17
    <body>
// ... lines 19 - 90
        {% block javascripts %}
            {{ encore_entry_script_tags('app') }}

            <script src="https://code.jquery.com/jquery-3.2.1.min.js" integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4=" crossorigin="anonymous"></script>
            <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.3/umd/popper.min.js" integrity="sha384-vFJXuSJphROIrBnz7yo7oB41mKfc8JzQZiCq4NCceLEaO4IHwicKwpJf9c9IpFgh" crossorigin="anonymous"></script>
            <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/js/bootstrap.min.js" integrity="sha384-alpBpkh1PFOepccYVYDB4do5UnbKysX5WZXm3XxPqe5iKTfUKjNkCk9SaVuEZflJ" crossorigin="anonymous"></script>
            <script>
                $('.dropdown-toggle').dropdown();
                $('.custom-file-input').on('change', function(event) {
                    var inputFile = event.currentTarget;
                    $(inputFile).parent()
                        .find('.custom-file-label')
                        .html(inputFile.files[0].name);
                });
            </script>
        {% endblock %}
    </body>
</html>
```

Ah... we have a few script tags for external files *and* some inline JavaScript. Shame on me! Let's refactor *all* of this into our new Encore-powered system.

The first thing we include is jQuery... which makes sense because we're using it below. Great! Get rid of it. Not surprisingly... this gives us a nice, big error:

> *"$ is not defined"*

Installing a Library (jQuery)

No worries! One of the most *wondrous* things about modern JavaScript is that we can install third-party libraries properly. I mean, with a package manager. Find your terminal and run:

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

The `--dev` part isn't important. *Technically* we only need these files during the "build" process... they don't need to be included on production... which is why the `--dev` makes sense. But in 99% of the cases, it doesn't matter. We'll talk about production builds and deploying at the end of the tutorial.

And... that was *painless*! We now have jQuery in our app.

Importing a Third-Party Library

We already know how to import a file that lives in a directory next to us. To import a *third* party library, we can say `import $ from`, and then the name of the package: `jquery`:

assets/js/app.js

```
// ... lines 1 - 7
// any CSS you require will output into a single css file (app.css in this case)
import '../css/app.css';

import $ from 'jquery';
// ... lines 12 - 15
```

The critical thing is that there is no `.` or `./` at the start. If the path starts with a `.`, Webpack knows to look for that file relative to this one. If there is *no* `.`, it knows to look for it inside the `node_modules/` directory.

Check it out: open `node_modules/` and ... there's it is! A `jquery` directory! But how does it know exactly *which* file in here to import? I'm *so* glad you asked! Open jQuery's `package.json` file. Every JavaScript package you install... unless it's *seriously* ancient, will have a `main` key that tells Webpack *exactly* which file it should import. *We* just say `import 'jquery'`, but it *really* imports this specific file.

Global Variables inside Webpack

Cool! We've imported jQuery in `app.js` and set it to a `$` variable. And because that `<script>` tag is included *above* our inline code in `base.html.twig`:

templates/base.html.twig

```
<!doctype html>
<html lang="en">
// ... lines 3 - 17
    <body>
// ... lines 19 - 90
        {% block javascripts %}
            {{ encore_entry_script_tags('app') }}

            <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.3/umd/popper.min.js" integrity="sha384-vFJXuSJphROIrBnz7yo7oB41mKfc8JzQZiCq4NCceLEaO4IHwicKwpJf9c9IpFgh" crossorigin="anonymous"></script>
            <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/js/bootstrap.min.js" integrity="sha384-alpBpkh1PFOepccYVYDB4do5UnbKysX5WZXm3XxPqe5iKTfUKjNkCk9SaVuEZflJ" crossorigin="anonymous"></script>
            <script>
                $('.dropdown-toggle').dropdown();
                $('.custom-file-input').on('change', function(event) {
                    var inputFile = event.currentTarget;
                    $(inputFile).parent()
                        .find('.custom-file-label')
                        .html(inputFile.files[0].name);
                });
            </script>
        {% endblock %}
    </body>
</html>
```

The `$` variable should be available down here, right?

Nope! `$` is *still* not defined! Wait, the *second* error is more clear. Yep, `$` is not defined, coming from our code in `base.html.twig`.

This uncovers a *super* important detail. When you import a file from a 3rd party library, that file behaves *differently* than if you add a `<script>` tag on your page that points to the *exact* same file! Yea!

That's because a well-written library will contain code that detects *how* it's being used and then changes its behavior.

Check it out: open `jquery.js`. It's not *super* easy to read, but look at this: if `typeof module.exports === "object"`. That's *key*. *This* is jQuery detecting if it's being used from within an environment like Webpack. If it *is*, it *exports* the jQuery object in the same way that we're exporting a function from the `get_nice_message.js` file:

assets/js/get_nice_message.js

```
1  export default function(exclamationCount) {
↕  // ... line 2
3  };
```

But if we are *not* in a module-friendly environment like Webpack... specifically, if jQuery is being loaded via a script tag in our browser, it's not too obvious, but this code is creating a *global* variable.

So, *if* jQuery is in a script tag, we get a global `$` variable. But if you *import* it like we're doing here:

assets/js/app.js

```
↕  // ... lines 1 - 10
11 import $ from 'jquery';
↕  // ... lines 12 - 15
```

It does *not* create a global variable. It *returns* the jQuery object, which is then set on this *local* variable. Also, all modules... or "files", in Webpack live in "isolation": if you set a variable in one file, it *won't* be available in any other file, regardless of what order they're loaded.

That is probably the *biggest* thing to re-learn in Webpack. Global variables are dead. That's *awesome*. But it *also* changes *everything*.

Forcing a Global jQuery Variable

The *ultimate* solution is to refactor all of your code from your templates and un-Webpack-ified JavaScript files *into* Encore. But... if you're upgrading an *existing* site, phew! You probably have a *ton* of JavaScript that expects there to be global `$` or `jQuery` variables. Moving *all* of that into Encore *all* at once... it's, uh... not very realistic.

So, if you *really* want a global variable, you can add one with `global.$ = $`:

assets/js/app.js

```
↕ // ... lines 1 - 10
11 import $ from 'jquery';
12 global.$ = $;
↕ // ... lines 13 - 16
```

That `global` keyword is special to Webpack. Try it now: refresh! It works!

But... don't do this unless you *have* to. I'll remove it and add some comments to explain that this is useful for legacy code:

assets/js/app.js

```
↕ // ... lines 1 - 10
11 import $ from 'jquery';
12 // uncomment if you have legacy code that needs global variables
13 //global.$ = $;
↕ // ... lines 14 - 17
```

Let's *properly* finish this next by refactoring all our code into `app.js`, which will include installing *two* more libraries and our first jQuery plugin... It turns out that jQuery plugins are a special beast.

Chapter 6: Bootstrap & the Curious Case of jQuery Plugins

The inline code in `base.html.twig` isn't working anymore because we've *eliminated* the `$` global variable:

templates/base.html.twig

```
<!doctype html>
<html lang="en">
// ... lines 3 - 17
    <body>
// ... lines 19 - 90
        {% block javascripts %}
// ... lines 92 - 95
            <script>
                $('.dropdown-toggle').dropdown();
                $('.custom-file-input').on('change', function(event) {
                    var inputFile = event.currentTarget;
                    $(inputFile).parent()
                        .find('.custom-file-label')
                        .html(inputFile.files[0].name);
                });
            </script>
        {% endblock %}
    </body>
</html>
```

Woo! To make it work, let's move *all* this code into `app.js`:

assets/js/app.js

```
// ... lines 1 - 15
console.log(getNiceMessage(5));

$('.dropdown-toggle').dropdown();
$('.custom-file-input').on('change', function(event) {
    var inputFile = event.currentTarget;
    $(inputFile).parent()
        .find('.custom-file-label')
        .html(inputFile.files[0].name);
});
```

Instead of global variables, we're importing `$` and that's why it's called `$` down here:

assets/js/app.js

```
↕ // ... lines 1 - 10
11 import $ from 'jquery';
↕ // ... lines 12 - 17
18 $('.dropdown-toggle').dropdown();
19 $('.custom-file-input').on('change', function(event) {
20     var inputFile = event.currentTarget;
21     $(inputFile).parent()
22         .find('.custom-file-label')
23         .html(inputFile.files[0].name);
24 });
```

It's all just local variables.

Try it now. Ok, it *sorta* works. It logs... then explodes. The error has some Webpack stuff on it, but it ultimately says:

> *"dropdown is not a function"*

Click the `app.js` link. Ah, it's having trouble with the `dropdown()` function. *That* is one of the functions that *Bootstrap* adds to jQuery. And... it makes sense why it's missing: we're running all of our code here, and *then* including Bootstrap:

templates/base.html.twig

```
1  <!doctype html>
2  <html lang="en">
↕  // ... lines 3 - 17
18     <body>
↕  // ... lines 19 - 90
91         {% block javascripts %}
92             {{ encore_entry_script_tags('app') }}
↕  // ... lines 93 - 94
95             <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
   beta.2/js/bootstrap.min.js" integrity="sha384-
   alpBpkh1PFOepccYVYDB4do5UnbKysX5WZXm3XxPqe5iKTfUKjNkCk9SaVuEZflJ"
   crossorigin="anonymous"></script>
96         {% endblock %}
97     </body>
98 </html>
```

It's simply *not* adding the function in time! Well actually, it's a *bit* more than that. *Even* if we moved this script tag up, it *still* wouldn't work. Why? Because when you include Bootstrap via a

script tag, it *expects* jQuery to be a global variable... and that - wonderfully - doesn't exist anymore.

Let's do this properly.

Installing Bootstrap

Oh, by the way, this `popper.js` thing is here because it's needed by Bootstrap:

templates/base.html.twig

```
<!doctype html>
<html lang="en">
// ... lines 3 - 17
    <body>
// ... lines 19 - 90
        {% block javascripts %}
            {{ encore_entry_script_tags('app') }}

            <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.3/umd/popper.min.js" integrity="sha384-vFJXuSJphROIrBnz7yo7oB41mKfc8JzQZiCq4NCceLEaO4IHwicKwpJf9c9IpFgh" crossorigin="anonymous"></script>
            <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/js/bootstrap.min.js" integrity="sha384-alpBpkh1PFOepccYVYDB4do5UnbKysX5WZXm3XxPqe5iKTfUKjNkCk9SaVuEZflJ" crossorigin="anonymous"></script>
        {% endblock %}
    </body>
</html>
```

You'll see how this works in Webpack in a moment. Delete both of the script tags:

templates/base.html.twig

```
<!doctype html>
<html lang="en">
// ... lines 3 - 17
    <body>
// ... lines 19 - 90
        {% block javascripts %}
            {{ encore_entry_script_tags('app') }}
        {% endblock %}
    </body>
</html>
```

Then, find your terminal and run:

```
yarn add "bootstrap@^4" --dev
```

Oh, and how did I know that the package name was `bootstrap`? Just because I cheated and searched for it before recording. Go to https://yarnpkg.com/ and search for "Bootstrap". 9.7 million downloads... in the last 30 days... that's probably the right one.

And... it's done! Oh, and there's a little notice:

> *"bootstrap has an unmet peer dependency popper.js"*

We'll come back to that in a minute.

Importing jQuery Plugins

Back in `app.js` installing Bootstrap isn't enough. On top, add `import 'bootstrap'`:

assets/js/app.js

```
// ... lines 1 - 10
import $ from 'jquery';
import 'bootstrap'; // adds functions to jQuery
// ... lines 13 - 26
```

Nope, we *don't* need to say `import $ from` or anything like that. Bootstrap is a jQuery plugin and jQuery plugins are... super weird. They do *not* return a value. Instead, they *modify* jQuery and add functions to it. I'll add a note here because... it just *looks* strange: it's weird that adding *this* allows me to use the `tooltip()` function, for example.

How Bootstrap Finds jQuery

But wait a second. If Bootstrap *modifies* jQuery... internally, how does it *get* the jQuery object in order to do that? I mean, jQuery is no longer global: if *we* need it, we need to import it. Well... because Bootstrap is a well-written library, it does the *exact* same thing. It *detects* that it's in a Webpack environment and, instead of expecting there to be a *global* `jQuery` variable, it *imports* `jquery`, *just* like we are.

And, fun fact, when two different files import the *same* module, they get back the same, *one* instance of it - a lot like Symfony's container. *We* import jQuery and assign it to `$`. Then, a microsecond later, Bootstrap imports that *same* object and modifies it:

assets/js/app.js

```
// ... lines 1 - 10
import $ from 'jquery';
import 'bootstrap'; // adds functions to jQuery
// ... lines 13 - 26
```

By the time we get past line 12, the `$` variable has the new `tooltip()` function.

Installing popper.js

But... you may have noticed that, while I was talking about how awesome this is all going to work... my build was failing!

> *"This dependency was not found:* `popper.js` *in* `bootstrap.js`*"*

This is awesome! Bootstrap has *two* dependencies: jQuery but *also* another library called `popper.js`. Internally, it tries to import *both* of them. But, because this is not installed in our project, it fails. By the way, if you're wondering:

> *"Why doesn't Bootstrap just list this as a dependency in its* `package.json` *so that it's automatically downloaded for us?"*

Excellent question! And that's *exactly* how we would do it in the PHP world. Short answer: Node dependencies are complicated, and so *sometimes* it will work like this, but *sometimes* it's a better idea for a library to force *us* to install its dependency manually. That's called a "peer" dependency.

Anyways, this is a great error, and it even suggests how to fix it: `npm install --save popper.js`. Because we're using Yarn, we'll do our version of that command. Back in your open terminal tab, run:

```
yarn add popper.js --dev
```

When that finishes... ah. Because we haven't modified any files, Webpack doesn't know it should re-build. Let's go over here and just add a space. That triggers a rebuild which is... successful!

Try it out - refresh! No errors.

Next! I have a surprise! Webpack has *already* started to silently optimize our build through a process called code splitting. Let's see what that means and learn how it works.

Chapter 7: The Magic of Split Chunks

View the HTML source and search for `app.js`. Surprise! We have *multiple* script tags! Actually, let me go to the inspect - it's a bit prettier. Black magic! We have *two* script tags - one for `app.js` but *also* one for `vendors~app.js`. What the heck? Go look at the `public/build/` directory. Yeah, there *is* a `vendors~app.js` file.

I *love* this feature. Check out `webpack.config.js`. One of the optional features that came pre-enabled is called `splitEntryChunks()`:

webpack.config.js

```
↕ // ... lines 1 - 2
3 Encore
↕ // ... lines 4 - 23
24     // When enabled, Webpack "splits" your files into smaller pieces for greater optimization.
25     .splitEntryChunks()
↕ // ... lines 26 - 66
67 ;
↕ // ... lines 68 - 69
```

Here's how it works. We tell Webpack to read `app.js`, follow *all* the imports, then eventually create one `app.js` file and one `app.css` file. But internally, Webpack uses an algorithm that, in this case, determines that it's more efficient if the one `app.js` file is split into *two*: `app.js` and `vendors~app.js`. And then, yea, we need *two* script tags on the page.

The Logic of Splitting

That may sound... odd at first... I mean, isn't part of the point of Webpack to combine all our JavaScript into a single file so that users can avoid making a ton of web requests?

Yes... but not always. The `vendors~app.js` file has some Webpack-specific code on top, but *most* of this file contains the *vendor* libraries that we imported. Stuff like `bootstrap` & `jquery`.

When Webpack is trying to figure out how to split the `app.js` file, it looks for code that satisfies several conditions. For example, *if* it can find code from the `node_modules/` directory *and* that

code is bigger than 30kb *and* splitting it into a new file would result in 3 or fewer final JavaScript files for this entry, it will split it. That's *exactly* what's happening here. Webpack especially likes splitting "vendor" code - that's the stuff in `node_modules/` - into its own file because vendor code tends to *change* less often. That means your user's browser can cache the `vendors~app.js` file for a longer time... which is cool, because those files tend to be pretty big. Then, the `app.js` file - which contains *our* code that probably changes *more* often, is smaller.

The algorithm *also* looks for code re-use. Right now, we only have *one* entry. But in a little while, we're going to create *multiple* entries to support page-specific CSS and JavaScript. When we do that, Webpack will automatically start analyzing which modules are *shared* between those entries and isolate them into their own files. For example, suppose our `get_nice_message.js` file is imported from *two* different entries: `app` and `admin`. Without code splitting, that code would be *duplicated* inside the final built `app.js` *and* `admin.js`. *With* code splitting, that code *may* be split into its own file. I say "may" because Webpack is smart: if the code is *tiny*, splitting it into its own file would be *worse* for performance.

SplitChunksPlugin

All of this craziness happens without us even knowing or caring. This feature comes from a part of Webpack called the SplitChunksPlugin. On top, it explains the logic it uses to split. But you can configure *all* of this.

Oh, see this big example config? *This* is a small piece of what Webpack's config *normally* looks like without Encore: your `webpack.config.js` would be a big config object like this. So, if we wanted to apply some of these changes, how could we do that in Encore?

The answer lives at the bottom of `webpack.config.js`. At the end, we call `Encore.getWebpackConfig()`, which *generates* standard Webpack config:

webpack.config.js

```
↕ // ... lines 1 - 68
69 module.exports = Encore.getWebpackConfig();
```

If you need to, you can always set this to a variable, *modify* some keys, then export the final value when you're finished:

```
// webpack.config.js
```

```
// ...

const config = Encore.getWebpackConfig();
config.optimization.splitChunks.minSize = 20000;

module.exports = config;
```

But for most things, there's an easier way. In this case, you can say `.configureSplitChunks()` and pass it a callback function. Encore will pass you the *default* split chunks configuration and then you can tweak it:

```
// webpack.config.js

// ...

Encore.
    // ...
    .splitEntryChunks()
    .configureSplitChunks(function(splitChunks) {
        splitChunks.minSize = 20000;
    })
    // ...
;

module.exports = Encore.getWebpackConfig();
```

This is a common way to extend things in Encore.

But... Webpack does a pretty great job of splitting things out-of-the-box. And... if you look at the `entrypoints.json` file, Encore makes sure that this file stays up-to-date with exactly *which* script and link tags each entry requires. The Twig helpers are already reading this file and taking care of everything:

templates/base.html.twig

```
<!doctype html>
<html lang="en">
// ... lines 3 - 17
    <body>
// ... lines 19 - 90
        {% block javascripts %}
            {{ encore_entry_script_tags('app') }}
        {% endblock %}
    </body>
</html>
```

Basically, code splitting is free performance.

Oh, and *all* of this applies to CSS too. In a few minutes, after we've made our CSS a bit fancier, you'll notice that we'll suddenly have multiple `link` tags.

Next, let's do that! Let's take our CSS up a level by removing the extra link tags from our base layout and putting everything into Encore. To do this, we'll start importing CSS files from third-party libraries in `node_modules/`.

Chapter 8: Importing 3rd Party CSS + Image Paths

We're on a mission to refactor all the old `<script>` and `<link>` tags *out* of our templates. For the base layout, we're half way done! There is only *one* script tag, which points to the `app` entry:

templates/base.html.twig

```
<!doctype html>
<html lang="en">
// ... lines 3 - 17
    <body>
// ... lines 19 - 90
        {% block javascripts %}
            {{ encore_entry_script_tags('app') }}
        {% endblock %}
    </body>
</html>
```

That's *perfect*.

Back on top, we *do* still have multiple link tags, including Bootstrap from a CDN, FontAwesome, which I apparently just committed into my `public/css` directory, and some custom CSS in `styles.css`:

templates/base.html.twig

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

    <head>
// ... lines 5 - 8
        {% block stylesheets %}
            {{ encore_entry_link_tags('app') }}

            <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
integrity="sha384-
Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
crossorigin="anonymous">
            <link rel="stylesheet" href="{{ asset('css/font-awesome.css') }}">
            <link rel="stylesheet" href="{{ asset('css/styles.css') }}">
        {% endblock %}
    </head>
// ... lines 17 - 94
</html>
```

First, eliminate Bootstrap! In the *same* way that we can properly install JavaScript libraries with Yarn, we can *also* install CSS libraries! Woo!

In `app.js`, we're already importing a single `app.css` file:

assets/js/app.js

```
// ... lines 1 - 7
// any CSS you require will output into a single css file (app.css in this case)
import '../css/app.css';
// ... lines 10 - 25
```

We *could* add another import *right* here for the Bootstrap CSS. Instead, I *prefer* to import just *one* CSS file per entry. Then, from *within* that CSS file, we can use the standard `@import` CSS syntax to import other CSS files. To Webpack, these two approaches are identical.

Now, you *might* be thinking:

> *"Don't we need to install the bootstrap CSS library?"*

And... yes! Well, I mean, no! Um, I mean, we already did it! In `node_modules/`, look for `bootstrap/`. This directory contains JavaScript but it *also* contains the Bootstrap CSS.

Importing CSS from node_modules

But... hmm... In JavaScript, we can say `import` then simply the name of the package and... it just works! But we *can't* repeat that same trick for CSS.

Instead, we'll point directly to the path we want, which, in this case is probably `dist/css/bootstrap.css`. Here's how: `@import`, `~bootstrap` and the path: `/dist/css/bootstrap.css`:

assets/css/app.css

```
1 @import '~bootstrap/dist/css/bootstrap.css';
```

The `~` part is special to CSS and Webpack. When you want to reference the `node_modules/` directory from within a CSS file, you need to start with `~`. That's *different* than JavaScript where any path that *doesn't* start with `.` is assumed to live in `node_modules/`. After the `~`, it's just a normal, boring path.

But yea... that's all we need! Move over and refresh. This looks exactly the same!

Referencing *just* the Package Name

And... remember how I said that we *can't* simply import CSS by referencing *only* the package name? That was... kind of a lie. Shorten this to just `~bootstrap`:

assets/css/app.css

```
1 @import '~bootstrap';
```

Go try it! Refresh and... the same!

This works thanks to a little extra feature we added to Encore... which may become a more standard feature in the future. We already know that when we import a package by its name in JavaScript, Webpack looks in `package.json`, finds the `main` key.... there it is and uses *this* to know that it should *finally* import the `dist/js/bootstrap.js` file.

Some libraries *also* include these `style` or `sass` keys. And when they *do*, you only need to `@import` `~` and the package name. Because we're doing this from inside a CSS file, it knows to look inside `package.json` for a `style` key.

This is *just* a shortcut to do the exact same thing we had before.

Installing & Importing Font Awesome

Bootstrap, check! Let's keep going: the next link tag is for FontAwesome. Get rid of that and celebrate by deleting the `public/css/font-awesome.css` file *and* this entire `fonts/` directory. This feels great! We're deleting things that I never should have committed in the first place.

Next, download FontAwesome with:

```
yarn add font-awesome --dev
```

When it finishes, go back to `node_modules/` and search for `font-awesome/`. Got it! Nice! It has directories for `css/`, `less/`, `scss/` whatever format we want. And fortunately, if you look inside `package.json`, it *also* has a `style` key.

Easy peasy! In `app.css`, add `@import '~font-awesome'`:

assets/css/app.css

```
1 @import '~bootstrap';
2 @import '~font-awesome';
```

Done. Find your browser and refresh. Let's see... down here, yes! *This* is a FontAwesome icon. It still works!

Image & Font Handling

But this is *way* cooler than it seems! Internally, the FontAwesome CSS file references some *font* files that the user's browser needs to download: these files here. But... these files aren't in our `public` directory... so shouldn't the paths to these be broken?

Close up `node_modules/` and check out the `public/build/` directory. Whoa! Where did this `fonts/` directory come from? When Webpack sees that a CSS file *refers* to a font file, it *copies* those fonts into this `fonts/` directory and *rewrites* the code in the final `app.css` file so that the font paths point *here*. Yes, it just *handles* it.

It also automatically adds a hash to the filename that's based on the file's *contents*. So if we ever update the font file, that hash would automatically change and the CSS would automatically point to it. That's *free* browser cache busting.

Moving our CSS into Encore

Ok *one* more link tag to go:

templates/base.html.twig

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

    <head>
// ... lines 5 - 8
        {% block stylesheets %}
// ... lines 10 - 11
            <link rel="stylesheet" href="{{ asset('css/styles.css') }}">
        {% endblock %}
    </head>
// ... lines 15 - 92
</html>
```

Remove it! Then, open `css/styles.css`, copy *all* of this, delete that file, and, in `app.css`, highlight the blue background and paste!

assets/css/app.css

```
@import '~bootstrap';
@import '~font-awesome';

body {
    position: relative;
    background: #efefee;
    min-height: 45rem;
    padding-bottom: 80px;
}

html {height:100%}

/* NAVIGATION */

.navbar-bg {
    background: url('../images/space-nav.jpg');
    background-size: 80%;
}

.dropdown-menu, .dropdown-menu.show {
    right: 0;
}

.space-brand {
    color: #fff;
    font-weight: bold;
}

.nav-profile-img {
    width: 50px;
    border: 1px solid #fff;
}

.nav-tabs .nav-link:focus, .nav-tabs .nav-link:hover {
    color: #efefee;
}

/* ADVERTISEMENT */

.ad-space {
    background: #fff;
    border-radius: 5px;
    border-top: 5px solid green;
}

.advertisement-img {
```

```
    width: 150px;
    height: auto;
    border: 2px solid #efefee;
    border-radius: 5px;
}

.advertisement-text {
    font-weight: bold;
}

.quote-space {
    background: #fff;
    margin-top: 30px;
    border-radius: 5px;
    border-top: 5px solid hotpink;
}

/* ARTICLES */

.main-article {
    border: 2px solid #efefee;
    Background: #fff;
    border-top-left-radius: 6px;
    border-top-right-radius: 6px;
}

.main-article img {
    width: 100%;
    height: 250px;
    border-top-right-radius: 5px;
    border-top-left-radius: 5px;
    border-top: 5px solid lightblue;
}

.article-container {
    border: 1px solid #efefee;
    border-top-left-radius: 5px;
    border-bottom-left-radius: 5px;
    background: #fff;
}

.main-article-link, .article-container a {
    text-decoration: none;
    color: #000;
}

.main-article-link:hover {
```

```
    text-decoration: none;
    color: #000;
}

.article-title {
    min-width: 300px;

}

@media (max-width: 440px) {
    .article-title {
        min-width: 100px;
        max-width: 245px;
    }
}

.article-img {
    height: 100px;
    width: 100px;
    border-top-left-radius: 5px;
    border-bottom-left-radius: 5px;
}

.article-author-img {
    height: 25px;
    border: 1px solid darkgray;
}

.article-details {
    font-size: .8em;
}

/* PROFILE */

.profile-img {
    width: 150px;
    height: auto;
    border: 2px solid #fff;
}

.profile-name {
    font-size: 1.5em;
}

.my-article-container {
    background: #FFBC49;
    border: solid 1px #efefee;
```

```
    border-radius: 5px;
}


/* CREATE ARTICLE */

.create-article-container {
    min-width: 400px;
    background-color: lightblue;
    border-radius: 5px;
}

/* ARTICLE SHOW PAGE */

.show-article-container {
    width: 100%;
    background-color: #fff;
}

.show-article-container.show-article-container-border-green {
    border-top: 3px solid green;
    border-radius: 3px;
}

.show-article-img {
    width: 250px;
    height: auto;
    border-radius: 5px;
}

.show-article-title {
    font-size: 2em;
}

.like-article, .like-article:hover {
    color: red;
    text-decoration: none;
}

@media (max-width: 991px) {
    .show-article-title {
        font-size: 1.5em;
    }

    .show-article-title-container {
        max-width: 220px;
    }
```

```
}

.article-text {
    margin-top: 20px;
}

.share-icons i {
    font-size: 1.5em;
}

.comment-container {
    max-width: 600px;
}

.comment-img {
    width: 50px;
    height: auto;
    border: 1px solid darkgray;
}

.commenter-name {
    font-weight: bold;
}

.comment-form {
    min-width: 500px;
}

@media (max-width: 767px) {
    .comment-form {
        min-width: 260px;
    }
    .comment-container {
        max-width: 280px;
    }
}



/* FOOTER */

.footer {
  position: absolute;
  bottom: 0;
  width: 100%;
  height: 60px; /* Set the fixed height of the footer here */
  line-height: 60px; /* Vertically center the text there */
```

```
235     background-color: #fff;
236     margin-top: 10px;
237 }
238 
239 /* Sortable */
240 .sortable-ghost {
241     background-color: lightblue;
242 }
243 .drag-handle {
244     cursor: grab;
245 }
```

That's a simple step so... it should work, right? Nope! Check out the build failure:

> *"Module not found: Can't resolve `../images/space-nav.jpg` in our `assets/css/` directory."*

It doesn't show the exact file, but we only have one. Ah, *here's* the problem:

assets/css/app.css

```
↕ // ... lines 1 - 12
13 /* NAVIGATION */
14 
15 .navbar-bg {
16     background: url('../images/space-nav.jpg');
↕ // ... line 17
18 }
↕ // ... lines 19 - 246
```

PhpStorm is super angry about it too! This background image references `../images/`, which was *perfect* when the code lived in the `public/css/` directory. But when we moved it, we broke that path!

This is awesome! Instead of us silently not realizing we did this, we get a *build* error. Amazing! We can't break paths without Webpack *screaming*.

To fix this, let's "cut" the entire `images/` directory and move it into the `assets/` folder. Yep, it's gone. But Encore doesn't know to re-compile... so make a small change and save. Build successful!

Go check it out. Refresh! It works! And even *better*, look at the `build/` folder. We have an `images/` directory with `space-nav.jpg` inside. *Just* like with fonts, Webpack sees our path,

realizes that `space-nav.jpg` needs to be public, and so moves it into the `build/images/` directory and rewrites the `background-image` code in the final CSS to point here.

The moral is this: all *we* need to do is worry about writing our code correctly: using the proper relative paths from source CSS file to source image file. Webpack handles the ugly details.

Now, this *did* break a few `<img>` tags on our site that are referencing some of these files. Now that they're not in the `public/` directory... they don't work. We'll handle that soon.

But next, let's get more from our CSS by using Sass.

Chapter 9: Sass & Overriding Bootstrap Variables

What if I want to use Sass instead of normal CSS, or maybe Less or Stylus? *Normally*, that takes some setup: you need to create a system that can compile all of your Sass files into CSS. But with Encore, we get this for free!

Rename `app.css` to `app.scss`. Of course, when we do that, the build fails because we need to update the `import` in `app.js`:

assets/js/app.js

```
↕ // ... lines 1 - 7
8 // any CSS you require will output into a single css file (app.css in this case)
9 import '../css/app.scss';
↕ // ... lines 10 - 26
```

But the build *still* fails. Go check out the error. Woh! That's awesome! It basically says:

> *"Hey! How are you? Great weather lately, right? Listen, it looks like you're trying to load a Sass file. That's super! To do that, enable the feature in Encore and install these libraries."*

This is the philosophy of Encore: give you a really solid, but small-ish core, and then offer a *ton* of optional features.

Enabling Sass

Go back to `webpack.config.js`. The `enableSassLoader()` line is already here. Uncomment it:

webpack.config.js

```
↕ // ... lines 1 - 2
3 Encore
↕ // ... lines 4 - 50
51     // enables Sass/SCSS support
52     .enableSassLoader()
↕ // ... lines 53 - 66
67 ;
↕ // ... lines 68 - 69
```

Back at the terminal, copy the `yarn add` command, go to the open tab, and run it!

> **Tip**
>
> Instead of `node-sass`, install `sass`. It's a pure-JavaScript implementation that is easier to install and is now recommended.

```
yarn add sass-loader@^7.0.1 sass --dev
```

This *could* take a minute or two: `node-sass` is a C library and it may need to *compile* itself depending on your system. Ding!

Thanks to the watch script, we *normally* don't need to worry about stopping or restarting Encore. There is *one* notable exception: when you make a change to `webpack.config.js`, you *must* stop and restart Encore. That's just a limitation of Webpack itself: it can't re-read the fresh configuration until you restart.

Hit `Control+C` and then run `yarn watch` again.

```
yarn watch
```

And this time... yes! We just added Sass support in like... two minutes - how awesome is that?

Organizing into Partials

This next part is optional, but I want to get organized... instead of having one big file, create a new directory called `layout/`. And for this top stuff, create a file called `_header.scss`. Little-by-little, we're going to move *all* of this code into different files. Grab the first section and put it into header:

assets/css/layout/_header.scss

```
body {
    position: relative;
    background: #efefee;
    min-height: 45rem;
    padding-bottom: 80px;
}

html {height:100%}

/* NAVIGATION */

.navbar-bg {
    background: url('../images/space-nav.jpg');
    background-size: 80%;
}

.dropdown-menu, .dropdown-menu.show {
    right: 0;
}

.space-brand {
    color: #fff;
    font-weight: bold;
}

.nav-profile-img {
    width: 50px;
    border: 1px solid #fff;
}

.nav-tabs .nav-link:focus, .nav-tabs .nav-link:hover {
    color: #efefee;
}
```

We'll import the new files when we finish.

Next is the "advertisement" CSS. Create another folder called `components/`. And inside, a new `_ad.scss` file. I'll delete the header... then move the code there:

assets/css/components/_ad.scss

```
.ad-space {
    background: #fff;
    border-radius: 5px;
    border-top: 5px solid green;
}

.advertisement-img {
    width: 150px;
    height: auto;
    border: 2px solid #efefee;
    border-radius: 5px;
}

.advertisement-text {
    font-weight: bold;
}

.quote-space {
    background: #fff;
    margin-top: 30px;
    border-radius: 5px;
    border-top: 5px solid hotpink;
}
```

Let's keep going! For the article stuff, create `_articles.scss`, and move the code:

assets/css/components/_articles.scss

```
.main-article {
    border: 2px solid #efefee;
    Background: #fff;
    border-top-left-radius: 6px;
    border-top-right-radius: 6px;
}

.main-article img {
    width: 100%;
    height: 250px;
    border-top-right-radius: 5px;
    border-top-left-radius: 5px;
    border-top: 5px solid lightblue;
}

.article-container {
    border: 1px solid #efefee;
    border-top-left-radius: 5px;
    border-bottom-left-radius: 5px;
    background: #fff;
}

.main-article-link, .article-container a {
    text-decoration: none;
    color: #000;
}

.main-article-link:hover {
    text-decoration: none;
    color: #000;
}

.article-title {
    min-width: 300px;

}

@media (max-width: 440px) {
    .article-title {
        min-width: 100px;
        max-width: 245px;
    }
}

.article-img {
    height: 100px;
```

```
    width: 100px;
    border-top-left-radius: 5px;
    border-bottom-left-radius: 5px;
}

.article-author-img {
    height: 25px;
    border: 1px solid darkgray;
}

.article-details {
    font-size: .8em;
}
// ... lines 60 - 140
```

Then, `_profile.scss`, copy that code... and paste:

assets/css/components/_profile.scss

```
.profile-img {
    width: 150px;
    height: auto;
    border: 2px solid #fff;
}

.profile-name {
    font-size: 1.5em;
}

.my-article-container {
    background: #FFBC49;
    border: solid 1px #efefee;
    border-radius: 5px;
}
```

For the "Create Article" and "Article Show" sections, let's copy *all* of that and put it into `_article.scss`:

assets/css/components/_articles.scss

```
.main-article {
    border: 2px solid #efefee;
    Background: #fff;
    border-top-left-radius: 6px;
    border-top-right-radius: 6px;
}

.main-article img {
    width: 100%;
    height: 250px;
    border-top-right-radius: 5px;
    border-top-left-radius: 5px;
    border-top: 5px solid lightblue;
}

.article-container {
    border: 1px solid #efefee;
    border-top-left-radius: 5px;
    border-bottom-left-radius: 5px;
    background: #fff;
}

.main-article-link, .article-container a {
    text-decoration: none;
    color: #000;
}

.main-article-link:hover {
    text-decoration: none;
    color: #000;
}

.article-title {
    min-width: 300px;

}

@media (max-width: 440px) {
    .article-title {
        min-width: 100px;
        max-width: 245px;
    }
}

.article-img {
    height: 100px;
```

```
    width: 100px;
    border-top-left-radius: 5px;
    border-bottom-left-radius: 5px;
}

.article-author-img {
    height: 25px;
    border: 1px solid darkgray;
}

.article-details {
    font-size: .8em;
}

/* CREATE ARTICLE */

.create-article-container {
    min-width: 400px;
    background-color: lightblue;
    border-radius: 5px;
}

/* ARTICLE SHOW PAGE */

.show-article-container {
    width: 100%;
    background-color: #fff;
}

.show-article-container.show-article-container-border-green {
    border-top: 3px solid green;
    border-radius: 3px;
}

.show-article-img {
    width: 250px;
    height: auto;
    border-radius: 5px;
}

.show-article-title {
    font-size: 2em;
}

.like-article, .like-article:hover {
    color: red;
    text-decoration: none;
```

```
}

@media (max-width: 991px) {
    .show-article-title {
        font-size: 1.5em;
    }

    .show-article-title-container {
        max-width: 220px;
    }
}

.article-text {
    margin-top: 20px;
}

.share-icons i {
    font-size: 1.5em;
}

.comment-container {
    max-width: 600px;
}

.comment-img {
    width: 50px;
    height: auto;
    border: 1px solid darkgray;
}

.commenter-name {
    font-weight: bold;
}

.comment-form {
    min-width: 500px;
}

@media (max-width: 767px) {
    .comment-form {
        min-width: 260px;
    }
    .comment-container {
        max-width: 280px;
    }
}
```

And for the footer, inside `layout/`, create one more file there called `_footer.scss` and... move the footer code:

assets/css/layout/_footer.scss

```
.footer {
  position: absolute;
  bottom: 0;
  width: 100%;
  height: 60px; /* Set the fixed height of the footer here */
  line-height: 60px; /* Vertically center the text there */
  background-color: #fff;
  margin-top: 10px;
}
```

And finally, copy the sortable code, create another components partial called `_sortable.scss` and paste:

assets/css/components/_sortable.scss

```
.sortable-ghost {
    background-color: lightblue;
}
.drag-handle {
    cursor: grab;
}
```

Now we can import all of this with `@import './layout/header'` and `@import './layout/footer'`:

assets/css/app.scss

```
@import '~bootstrap';
@import '~font-awesome';

@import './layout/header';
@import './layout/footer';
// ... lines 6 - 11
```

Notice: you don't need the `_` or the `.scss` parts: that's a Sass thing. Let's add a few more imports for the components: `ad`, `articles`, `profile` and `sortable`:

assets/css/app.scss

```
↕ // ... lines 1 - 3
4 @import './layout/header';
5 @import './layout/footer';
6
7 @import './components/ad';
8 @import './components/articles';
9 @import './components/profile';
10 @import './components/sortable';
```

Phew! That took some work, but I like the result! But, *of course*, Encore is here to ruin our party with a build failure:

> *"Cannot resolve `./images/space-nav.jpeg`"*

We know that error! In `_header.scss`... ah, there it is:

assets/css/layout/_header.scss

```
↕ // ... lines 1 - 11
12 .navbar-bg {
13     background: url('../images/space-nav.jpg');
↕ // ... line 14
15 }
↕ // ... lines 16 - 34
```

The path needs to go *up* one more level now:

assets/css/layout/_header.scss

```
↕ // ... lines 1 - 11
12 .navbar-bg {
13     background: url('../../images/space-nav.jpg');
↕ // ... line 14
15 }
↕ // ... lines 16 - 34
```

And... it works.

Move over and make sure nothing looks weird. Brilliant!

Adding Variables

To celebrate that we're processing through Sass, let's at *least* use *one* of its features. Create a new directory called `helper/` and a new file called `_variables.scss`.

At the top of `_header.scss`, we have a gray `background` color:

assets/css/layout/_header.scss

```
1  body {
↕  // ... line 2
3      background: #efefee;
↕  // ... lines 4 - 5
6  }
↕  // ... lines 7 - 34
```

Just to prove we can do it, in `_variables`, create a new variable called `$lightgray` set to `#efefee`:

assets/css/helper/_variables.scss

```
1  $lightgray: #efefee;
```

And back in headers, reference that: `$lightgray`:

assets/css/layout/_header.scss

```
1  body {
↕  // ... line 2
3      background: $lightgray;
↕  // ... lines 4 - 5
6  }
↕  // ... lines 7 - 34
```

We even get auto-completion on that! As *soon* as we save, the build fails!

> *"Undefined variable: "$lightgray""*

Perfect! Because... inside of `app.scss`, all the way on top, we still need to `@import` the `helper/variables` file:

assets/css/app.scss

```
1  @import './helper/variables';
↕  // ... lines 2 - 13
```

About a second later... ding! It builds and... the background is still there.

Overriding Bootstrap Sass Variables

But wait, there's more! When we import `bootstrap`, Encore has some logic to find the right CSS file in that package. But now that we're inside a *Sass* file, it's smart enough to *instead* import the `bootstrap.scss` file! Woh!

Check it out. Hold Command or `Ctrl` and click `~bootstrap` to jump to that directory. Then open up `package.json`. This has a `style` key, but it *also* has a `sass` key! Because we're importing from inside a Sass file, Encore *first* looks for the `sass` key and loads that file. If there isn't a `sass` key, it falls back to using `style`.

Now look at the `font-awesome/` directory and find *its* `package.json` file. It actually does *not* have a `sass` key! And so, it's *still* loading the `font-awesome.css` file, which is fine. If you *did* want to load the Sass file, you would just need to point at the file path directly.

Anyways, to *prove* that the Bootstrap Sass file is being loaded, we can override some of its variables. See this search button? It's blue because it has the `btn-info` class. It's color hash is... here: `#1782b8`.

Suppose you want to change the info color *globally* to be a bit darker. Bootstrap lets you do that in Sass by overriding a variable called `$info`.

Try it: inside the variables file, set `$info:` to `darken()`, the hash, and `10%`:

assets/css/helper/_variables.scss

```
1  $info: darken(#17a2b8, 10%);
↕  // ... lines 2 - 5
```

Once the build finishes... watch closely. It got darker! How cool is that?

Next, let's fix our broken `img` tags thanks to one of my favorite new Encore features called `copyFiles()`.

Chapter 10: Copying Files

Do a force refresh on the homepage. Ok, we've got some broken images. Inspect that. Of course: this points to `/images/meteor-shower.jpg`.

Open this template: `article/homepage.html.twig`. There it is:

templates/article/homepage.html.twig

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

            <!-- Article List -->

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

                <!-- H1 Article -->
                <a class="main-article-link" href="#">
                    <div class="main-article mb-5 pb-3">
                        <img src="{{ asset('images/meteor-shower.jpg') }}" alt="meteor shower">
// ... line 15
                    </div>
                </a>
// ... lines 18 - 40
            </div>
// ... lines 42 - 61
        </div>
    </div>
{% endblock %}
```

A normal `asset()` function pointing to `images/meteor-shower.jpg`. That's broken because we moved our entire `images/` directory out of `public/` and into `assets/`.

There's a nice side-effect of using a build system like Webpack: you don't need to keep your CSS, JavaScript or assets in a public directory anymore! You put them in `assets/`, organize them *however* you want, and the end-user will only ever see the final, built version.

But unless you're building a single page application, you'll probably still have some cases where you want to render a good, old-fashioned `img` tag. And because this image is *not* being processed through Webpack, it's not being copied into the final `build/` directory.

Hello `copyFiles()`

To make life more joyful, Encore has a feature for *exactly* this situation. Open up `webpack.config.js`. And, anywhere in here, say `.copyFiles()` and pass this a configuration object:

webpack.config.js

```
↕ // ... lines 1 - 2
3 Encore
↕ // ... lines 4 - 53
54     .copyFiles({
↕ // ... line 55
56     })
↕ // ... lines 57 - 70
71 ;
↕ // ... lines 72 - 73
```

Tip

If you're using Encore 1.0 or later, you'll also need to install `file-loader`. As soon as you use `copyFiles()`, check your Encore terminal tab: it will have the exact command you need to run.

Obviously... this function helps you copy files from one place to another. Neato! But... how exactly do we use it? One of the nicest things about Encore is that its *code* is *extremely* well-documented. Hold `Command` or `Ctrl` and click `copyFiles()`. It jumps us *straight* to the `index.js` file of Encore... which is almost *entirely* small methods with HUGE docs above them! This is a *great* resource for finding out, not only *how* you can use a function, but what functions and features are even available!

For `copyFiles()`, it can be as simple as:

> *"I want to copy everything from `assets/images` into my build directory."*

Yea, that sounds about right. If we did that, we could *then* reference those images from our `img` tags. Copy that config, go back to `webpack.config.js` and paste. Oh, I have an extra set of curly braces:

webpack.config.js

```
// ... lines 1 - 2
Encore
// ... lines 4 - 53
    .copyFiles({
        from: './assets/images'
    })
// ... lines 57 - 70
;
// ... lines 72 - 73
```

And because we just made a change to `webpack.config.js`, find your terminal, press `Ctrl`+`C`, and re-run Encore. When that finishes... go check it out. In the `public/build/` directory, there they are: `meteor-shower.jpg`, `space-ice.png` and so on.

Controlling the copy Destination

Um, but it *is* kind of lame that it just dropped them directly into `build/`, I'd rather, for my *own* sanity, copy these into `build/images/`.

Let's see... go back to the docs. Here it is: you *can* give it a destination... and this has a few *wildcards* in it, like `[path]`, `[name]` and `[ext]`. Oh, but use this second one instead: it gives us built-in file versioning by including a hash of the contents in the filename.

Back in our config, paste that:

webpack.config.js

```
// ... lines 1 - 2
Encore
// ... lines 4 - 53
    .copyFiles({
        from: './assets/images',
        to: 'images/[path][name].[hash:8].[ext]'
    })
// ... lines 58 - 71
;
// ... lines 73 - 74
```

Before we restart Encore, shouldn't we delete some of these old files... at least to get them out of the way and clean things up? Nope! Well, *yes*, but it's already happening. One *other* optional feature that we're using is called `cleanOutputBeforeBuild()`:

webpack.config.js

```
// ... lines 1 - 2
Encore
// ... lines 4 - 38
    .cleanupOutputBeforeBuild()
// ... lines 40 - 71
;
// ... lines 73 - 74
```

This is responsible for emptying the `build/` directory each time we build.

Ok, go restart Encore: `Ctrl`+`C`, then:

```
yarn watch
```

Let's go check it out! Beautiful! Everything *now* copies to `images/` *and* includes a hash.

Public Path to Versioned Copied Files: manifest.json

Oh, but... that's a problem. What path are we supposed to use for the `img` tag? Should we put `build/images/meteor-shower.5c77...jpg`? No, because if we ever updated that image, the hash would change and all our `img` tags would break. And because they aren't being processed by Webpack, that failure would be the *worst* kind: it would fail *silently*!

In the `build/` directory, there are *two* special JSON files generated by Encore. The first - `entrypoints.json` - is awesome because the Twig helpers can use it to generate all of the script and link tags for an entry. But there's *another* file: `manifest.json`.

This is a big, simple, beautiful map that contains *every* file that Encore outputs. It maps from the *original* filename to the *final* filename. For most files, because we haven't activated versioning globally yet, the paths are the same. But check out the images! It maps from `build/images/meteor-shower.jpg` to the *real*, versioned path! If we could read this file, we could automagically get the correct hash!

When we installed WebpackEncoreBundle, the recipe added a `config/packages/assets.yaml` file. Inside, oh! It has `json_manifest_path` set to the path to `manifest.json`:

config/packages/assets.yaml

```yaml
framework:
    assets:
        json_manifest_path: '%kernel.project_dir%/public/build/manifest.json'
```

The *significance* of this line is that *anytime* we use the `asset()` function in Twig, it will take that path and *look* for it inside of `manifest.json`. If it finds it, it will use the final, versioned path.

This means that if we want to point to `meteor-shower.jpg`, all *we* need to do is use the `build/images/meteor-shower.jpg` path. Copy that, go to the homepage template, and paste it here:

templates/article/homepage.html.twig

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

            <!-- Article List -->

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

                <!-- H1 Article -->
                <a class="main-article-link" href="#">
                    <div class="main-article mb-5 pb-3">
                        <img src="{{ asset('build/images/meteor-shower.jpg') }}" alt="meteor shower">
// ... line 15
                    </div>
                </a>
// ... lines 18 - 40
            </div>
// ... lines 42 - 61
        </div>
    </div>
{% endblock %}
```

There are a few other images tags in this file. Search for `<img`. This is pointing to an uploaded file, not a static file - so, that's good. Ah, but this one needs to change: `build/images/alien-profile.png`:

templates/article/homepage.html.twig

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

            <!-- Article List -->

            <div class="col-sm-12 col-md-8">
// ... lines 10 - 20
                {% for article in articles %}
                <div class="article-container my-1">
                    <a href="{{ path('article_show', {slug: article.slug}) }}">
// ... line 24
                        <div class="article-title d-inline-block pl-3 align-middle">
// ... lines 26 - 34
                            <span class="align-left article-details"><img class="article-author-img rounded-circle" src="{{ asset('build/images/alien-profile.png') }}"> {{ article.author }} </span>
// ... line 36
                        </div>
                    </a>
                </div>
                {% endfor %}
            </div>
// ... lines 42 - 61
        </div>
    </div>
{% endblock %}
```

And one more, add `build/` before `space-ice.png`:

templates/article/homepage.html.twig

```
// ... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">
// ... lines 6 - 45
            <div class="col-sm-12 col-md-4 text-center">
                <div class="ad-space mx-auto mt-1 pb-2 pt-2">
                    <img class="advertisement-img" src="{{
asset('build/images/space-ice.png') }}">
// ... lines 49 - 50
                </div>
// ... lines 52 - 60
            </div>
        </div>
    </div>
{% endblock %}
```

Let's try it! Move over, refresh and... we got it! Inspect element: it's the final, versioned filename. Let's update the *last* `img` tags - they're in `show.html.twig`. Search for `img` tags again, then... `build/`, `build/` and `build/`:

templates/article/show.html.twig

```
// ... lines 1 - 4
{% block content_body %}
    <div class="row">
        <div class="col-sm-12">
// ... line 8
            <div class="show-article-title-container d-inline-block pl-3 align-middle">
// ... lines 10 - 11
                <span class="align-left article-details"><img class="article-author-img rounded-circle" src="{{ asset('build/images/alien-profile.png') }}"> {{ article.author }} </span>
// ... lines 13 - 24
            </div>
        </div>
    </div>
// ... lines 28 - 39
    <div class="row">
        <div class="col-sm-12">
// ... lines 42 - 44
            <div class="row mb-5">
                <div class="col-sm-12">
                    <img class="comment-img rounded-circle" src="{{ asset('build/images/astronaut-profile.png') }}">
// ... lines 48 - 54
                </div>
            </div>

            {% for comment in article.nonDeletedComments %}
            <div class="row">
                <div class="col-sm-12">
                    <img class="comment-img rounded-circle" src="{{ asset('build/images/alien-profile.png') }}">
// ... lines 62 - 71
                </div>
            </div>
            {% endfor %}

        </div>
    </div>

{% endblock %}
// ... lines 80 - 86
```

Click to go view one of the articles. These comment avatars are *now* using the system.

`copyFiles()` is nice because it lets you keep *all* your frontend files in the same directory... even if some need to be copied to the build directory. But to sweeten the deal, you're rewarded with free asset versioning.

By the way, this function was added by @Lyrkan, one of the core devs for Encore and... even though it's pretty simple, it's an absolutely brilliant implementation that I haven't seen used anywhere else. So, if you like it, give him a thanks on Symfony Slack or Twitter.

Next, let's create *multiple* entry points to support page-specific CSS and JavaScript.

Chapter 11: Page-Specific JS: Multiple Entries

On the article show page, if you check the console... it's an error!

> *"$ is undefined"*

Coming from `article_show.js`. This shouldn't be surprising. And not *just* because I seem to make a lot of mistakes. Open that template and go to the bottom. Ah, this brings in a `js/article_show.js` file:

templates/article/show.html.twig

```
// ... lines 1 - 80
{% block javascripts %}
    {{ parent() }}

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

Go find that: in `public/`, I'll close `build/` and... there it is:

public/js/article_show.js

```
$(document).ready(function() {
    $('.js-like-article').on('click', function(e) {
        e.preventDefault();

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

        $.ajax({
            method: 'POST',
            url: $link.attr('href')
        }).done(function(data) {
            $('.js-like-article-count').html(data.hearts);
        })
    });
});
```

This contains some traditional JavaScript from a previous tutorial. The problem is that the global `$` variable doesn't exist anymore. If you look closely on this page, you'll see that, at the bottom,

we include the `app.js` file first and *then* `article_show.js`. And, of course, the `app.js` file *does* import jQuery:

assets/js/app.js

```
// ... lines 1 - 10
import $ from 'jquery';
// ... lines 12 - 26
```

But as we learned, this does not create a *global* variable and *local* variables in Webpack don't "leak" beyond the file they're defined in.

So... this file is broken. And that's *fine* because I want to refactor it anyways to go through Encore so that we can *properly* import the variable on top.

Before we do that, let's organize *one* tiny thing. In `assets/js`, create a new `components/` directory. Move `get_nice_messages.js` into that... and because that breaks our build... update the import statement in `app.js` to point here:

assets/js/app.js

```
// ... lines 1 - 14
import getNiceMessage from './components/get_nice_message';
// ... lines 16 - 26
```

Creating the Second Entry

Ok: I *originally* put this code into a separate file because it's only needed on the article show page. We *could* copy all of this, put it into `app.js`... and that would work! But sometimes, instead of having one *big* JavaScript file, you might want to split page-specific CSS and JavaScript into their *own* files.

To do that, we'll create a second Webpack "entry". Move `article_show.js` into `assets/js/`. Next, go into `webpack.config.js` and, up here, call `addEntry()` *again*. Name it `article_show` and point it at `./assets/js/article_show.js`:

webpack.config.js

```
// ... lines 1 - 2
Encore
// ... lines 4 - 19
    .addEntry('app', './assets/js/app.js')
    .addEntry('article_show', './assets/js/article_show.js')
// ... lines 22 - 72
;
// ... lines 74 - 75
```

Now when we build Webpack, it will *still* load `app.js`, follow all the imports, and create `app.js` and `app.css` files. But now it will *also* load `article_show.js`, follow all of *its* imports and output new `article_show.js` and `article_show.css` files.

Each "entry", or "entry point" is like a standalone application that contains *everything* it needs.

And now that we have this new `article_show` entry, inside `show.html.twig`, instead of our *manual* `<script>` tag, use `{{ encore_entry_script_tags('article_show') }}`:

templates/article/show.html.twig

```
// ... lines 1 - 80
{% block javascripts %}
    {{ parent() }}

    {{ encore_entry_script_tags('article_show') }}
{% endblock %}
```

I don't have a `link` tag anywhere... nope - it's not hiding on top either. That's ok, because, so far, `article_show.js` isn't importing *any* CSS. And so, Webpack is smart enough to *not* output an empty `article_show.css` file. But you *could* still plan ahead if you wanted: `encore_entry_link_tags()` will print *nothing* if there's no CSS file. So, no harm.

Ok: because we made a change to our `webpack.config.js` file, stop and restart Encore:

```
yarn watch
```

And... cool! The `app` entry caused these three files to be created... thanks to the split chunks stuff, and `article_show` just made `article_show.js`.

If you find your browser and refresh now... oh, same error... because we *still* haven't imported that. Back in `article_show.js`, `import $ from 'jquery'`:

assets/js/article_show.js

```
1  import $ from 'jquery';
2
3  $(document).ready(function() {
↕  // ... lines 4 - 16
17 });
```

Refresh again and... boom! Error is gone. We can click the fancy JavaScript-powered heart icon.

Importing CSS

Because we *haven't* imported any CSS yet from `article_show.js`, we already saw that Webpack was smart enough to not output a CSS file. But! Open up `_articles.scss`. *Part* of this file is CSS for the article show page... which doesn't *really* need to be included on *every* page:

assets/css/components/_articles.scss

```
// ... lines 1 - 68
/* ARTICLE SHOW PAGE */

.show-article-container {
    width: 100%;
    background-color: #fff;
}

.show-article-container.show-article-container-border-green {
    border-top: 3px solid green;
    border-radius: 3px;
}

.show-article-img {
    width: 250px;
    height: auto;
    border-radius: 5px;
}

.show-article-title {
    font-size: 2em;
}

.like-article, .like-article:hover {
    color: red;
    text-decoration: none;
}

@media (max-width: 991px) {
    .show-article-title {
        font-size: 1.5em;
    }

    .show-article-title-container {
        max-width: 220px;
    }
}

.article-text {
    margin-top: 20px;
}

.share-icons i {
    font-size: 1.5em;
}

```

```
.comment-container {
    max-width: 600px;
}

.comment-img {
    width: 50px;
    height: auto;
    border: 1px solid darkgray;
}

.commenter-name {
    font-weight: bold;
}

.comment-form {
    min-width: 500px;
}

@media (max-width: 767px) {
    .comment-form {
        min-width: 260px;
    }
    .comment-container {
        max-width: 280px;
    }
}
```

Let's copy *all* of this code, remove it, and, at the root of the `css/` directory, create a new file called `article_show.scss` and... paste!

assets/css/article_show.scss

```
/* ARTICLE SHOW PAGE */

.show-article-container {
    width: 100%;
    background-color: #fff;
}

.show-article-container.show-article-container-border-green {
    border-top: 3px solid green;
    border-radius: 3px;
}

.show-article-img {
    width: 250px;
    height: auto;
    border-radius: 5px;
}

.show-article-title {
    font-size: 2em;
}

.like-article, .like-article:hover {
    color: red;
    text-decoration: none;
}

@media (max-width: 991px) {
    .show-article-title {
        font-size: 1.5em;
    }

    .show-article-title-container {
        max-width: 220px;
    }
}

.article-text {
    margin-top: 20px;
}

.share-icons i {
    font-size: 1.5em;
}

.comment-container {
```

```
    max-width: 600px;
}

.comment-img {
    width: 50px;
    height: auto;
    border: 1px solid darkgray;
}

.commenter-name {
    font-weight: bold;
}

.comment-form {
    min-width: 500px;
}

@media (max-width: 767px) {
    .comment-form {
        min-width: 260px;
    }
    .comment-container {
        max-width: 280px;
    }
}
```

Both `app.js` and `article_show.js` are *meant* to import *everything* that's needed for the layout and for the article show page. `app.scss` and `article_show.scss` are kinda the same thing: they should import all the *CSS* that's needed for each spot.

At the top of `article_show.scss`, we don't *strictly* need to do this, but let's `@import 'helper/variables'` to drive home the point that this is a standalone file that *imports* anything it needs:

assets/css/article_show.scss

```
@import './helper/variables';

/* ARTICLE SHOW PAGE */
// ... lines 4 - 74
```

Finally, back in `article_show.js` add `import '../css/article_show.scss'`:

assets/js/article_show.js

```
import '../css/article_show.scss';
import $ from 'jquery';
// ... lines 3 - 19
```

Ok, check your terminal! Suddenly, gasp! Webpack is outputting an `article_show.css` file! And wow! You can *really* see code splitting in action! That `vendors~app~article_show.js` probably contains jQuery, because Webpack saw that it's used by *both* entries and so isolated it into its own file so it could be re-used.

Anyways, back in `show.html.twig` copy the `javascripts` block, paste, rename it to `stylesheets` and then change to `encore_entry_link_tags()`:

templates/article/show.html.twig

```
// ... lines 1 - 80
{% block javascripts %}
    {{ parent() }}

    {{ encore_entry_script_tags('article_show') }}
{% endblock %}

{% block stylesheets %}
    {{ parent() }}

    {{ encore_entry_link_tags('article_show') }}
{% endblock %}
```

That should do it! Move over, refresh and... cool! The page still looks good and the heart still works. If you inspect element on this page, in the `head`, we have *two* CSS files: `app.css` to power the layout and `article_show.css` to power this page.

At the bottom, we have *4* JavaScript files to power the two entrypoints. By the way, WebpackEncoreBundle is smart enough to *not* duplicate the `vendors~app~article_show.js` script tag just because *both* entries need it. Smart!

Next: we are *close* to having our whole app in Encore. Let's refactor a *bunch* more un-Webpack-ified code.

Chapter 12: Entry Refactoring

Here's our mission: to get rid of *all* the JavaScript and CSS stuff from our `public/` directory. Our next target is `admin_article_form.js`:

public/js/admin_article_form.js

```
Dropzone.autoDiscover = false;

$(document).ready(function() {
// ... lines 4 - 33
});

// todo - use Webpack Encore so ES6 syntax is transpiled to ES5
class ReferenceList
{
// ... lines 39 - 123
}

/**
 * @param {ReferenceList} referenceList
 */
function initializeDropzone(referenceList) {
// ... lines 130 - 148
}
```

This probably won't come as a *huge* shock, but this is used in the admin section. Go to `/admin/article`. If you need to log in, use `admin1@thespacebar.com`, password `engage`. Then click to edit any of the articles.

This page has JavaScript to handle the Dropzone upload and a few other things. Open the template: `templates/article_admin/edit.html.twig` and scroll down. Ok: we have a traditional `<script>` tag for `admin_article_form.js` as well as two external JavaScript files that we'll handle in a minute:

templates/article_admin/edit.html.twig

```
// ... lines 1 - 35
{% block javascripts %}
    {{ parent() }}

    <script src="https://cdnjs.cloudflare.com/ajax/libs/dropzone/5.5.1/min/dropzone.min.js" integrity="sha256-cs4thShDfjkqFGk5s2Lxj35sgSRr4MRcyccmi0WKqCM=" crossorigin="anonymous"></script>
    <script src="https://cdn.jsdelivr.net/npm/sortablejs@1.8.3/Sortable.min.js" integrity="sha256-uNITVqEk9HNQeW6mAAm2PJwFX2gN45l8a4yocqsFI6I=" crossorigin="anonymous"></script>
    <script src="{{ asset('js/admin_article_form.js') }}"></script>
{% endblock %}
```

The Repeatable Process of Refactoring to an Entry

This is *super* similar to what we just did. First, move `admin_article_form.js` into `assets/js`. This will be our *third* entry. So, in `webpack.config.js` copy `addEntry()`, call this one `admin_article_form` and point it to `admin_article_form.js`:

webpack.config.js

```
// ... lines 1 - 2
Encore
// ... lines 4 - 21
    .addEntry('admin_article_form', './assets/js/admin_article_form.js')
// ... lines 23 - 73
;
// ... lines 75 - 76
```

Finally, inside `edit.html.twig`, change this to use `{{ encore_entry_script_tags('admin_article_form') }}`:

templates/article_admin/edit.html.twig

```
// ... lines 1 - 35
{% block javascripts %}
// ... lines 37 - 40
    {{ encore_entry_script_tags('admin_article_form') }}
{% endblock %}
```

Now, stop and restart Encore:

```
yarn watch
```

Perfect! 3 entries and a lot of good code splitting. But we shouldn't be *too* surprised that when we refresh, we get our *favorite* JavaScript error:

> *"$ is not defined"*

Let's implement phase 2 of refactoring. In `admin_article_form.js`, `import $ from 'jquery'`:

assets/js/admin_article_form.js

```
1  import $ from 'jquery';
↕  // ... lines 2 - 152
```

And... we're good to go!

Refactoring the External script Tags

In addition to moving things out of `public/`, I *also* want to remove all of these external script tags:

templates/article_admin/edit.html.twig

```
↕  // ... lines 1 - 35
36 {% block javascripts %}
↕  // ... lines 37 - 38
39     <script src="https://cdnjs.cloudflare.com/ajax/libs/dropzone/5.5.1/min/dropzone.min.js" integrity="sha256-cs4thShDfjkqFGk5s2Lxj35sgSRr4MRcyccmi0WKqCM=" crossorigin="anonymous"></script>
40     <script src="https://cdn.jsdelivr.net/npm/sortablejs@1.8.3/Sortable.min.js" integrity="sha256-uNITVqEk9HNQeW6mAAm2PJwFX2gN45l8a4yocqsFI6I=" crossorigin="anonymous"></script>
↕  // ... line 41
42 {% endblock %}
```

Actually, there's nothing wrong with including external scripts - and you can *definitely* argue that including some things - like jQuery - could be good for performance. If you *do* want to keep a few script tags for external stuff, check out Webpack's "externals" feature to make it work nicely.

The reason I don't like them is that, in the *new* way of writing JavaScript, you never want undefined variables. If we need a `$` variable, we need to import `$`! But check it out: we're referencing `Dropzone`:

assets/js/admin_article_form.js

```
↕ // ... lines 1 - 2
3 Dropzone.autoDiscover = false;
↕ // ... lines 4 - 152
```

Where the heck does that come from? Answer: it's a global variable created by this Dropzone script tag!

templates/article_admin/edit.html.twig

```
↕ // ... lines 1 - 35
36 {% block javascripts %}
↕ // ... lines 37 - 38
39     <script src="https://cdnjs.cloudflare.com/ajax/libs/dropzone/5.5.1/min/dropzone.min.js" integrity="sha256-cs4thShDfjkqFGk5s2Lxj35sgSRr4MRcyccmi0WKqCM=" crossorigin="anonymous"></script>
40     <script src="https://cdn.jsdelivr.net/npm/sortablejs@1.8.3/Sortable.min.js" integrity="sha256-uNITVqEk9HNQeW6mAAm2PJwFX2gN45l8a4yocqsFI6I=" crossorigin="anonymous"></script>
↕ // ... line 41
42 {% endblock %}
```

The same is true for `Sortable` further down. I *don't* want to rely on global variables anymore.

Trash both of these script tags. Then, find your terminal, go to your open tab and run:

```
yarn add dropzone sortablejs --dev
```

I already looked up those exact package names to make sure they're right. Next, inside `admin_article_form.js`, these variables will truly be undefined now. Try it: refresh. A most *excellent* error!

> *"Dropzone is undefined"*

It sure is! Fix that with `import Dropzone from 'dropzone'` and also `import Sortable from 'sortablejs'`:

assets/js/admin_article_form.js

```
import $ from 'jquery';
import Dropzone from 'dropzone';
import Sortable from 'sortablejs';
// ... lines 4 - 154
```

Now it works.

Importing the CSS

But there's *one* more thing hiding in our edit template: we have a CDN link to the Dropzone CSS!

templates/article_admin/edit.html.twig

```
// ... lines 1 - 29
{% block stylesheets %}
    {{ parent() }}

    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/dropzone/5.5.1/min/dropzone.min.css" integrity="sha256-e47xOkXs1JXFbjjpoRr1/LhVcqSzRmGmPqsrUQeVs+g=" crossorigin="anonymous" />
{% endblock %}
// ... lines 35 - 41
```

We don't need that either. Instead, in `admin_article_form.js`, we can import the CSS from the Dropzone package directly. Hold `Command` or `Control` and click to open Dropzone. I'll double-click the `dropzone` directory to take us there.

Inside `dist`... there it is: `dropzone.css`. *That's* the path we want to import. How? With `import 'dropzone/dist/dropzone.css'`:

assets/js/admin_article_form.js

```
// ... line 1
import Dropzone from 'dropzone';
import 'dropzone/dist/dropzone.css'
// ... lines 4 - 155
```

Most of the time, we're lazy and we say `import` then the package name. But it's totally legal to import the package name / a specific file path.

As *soon* as we do that, go check out the Encore watch tab. Wow! The code splitting is getting crazy! Hiding inside there is *one* CSS file: `vendors~admin_article_form.css`.

Flip back to the edit template and add `{{ encore_entry_link_tags('admin_article_form') }}`:

templates/article_admin/edit.html.twig

```
// ... lines 1 - 29
{% block stylesheets %}
    {{ parent() }}

    {{ encore_entry_link_tags('admin_article_form') }}
{% endblock %}
// ... lines 35 - 41
```

Try it! Find your browser and refresh! Ok, it looks like the Dropzone CSS is still working. I think we're good!

Including script & link on the New Page

This *same* JavaScript & CSS code is needed on one other page. Go back to `/admin/article` and click create. Oof, we still have some problems here. I'll close up `node_modules/` and open `templates/article_admin/new.html.twig`:

templates/article_admin/new.html.twig

```
// ... lines 1 - 2
{% block javascripts %}
    {{ parent() }}

    <script src="https://cdn.jsdelivr.net/autocomplete.js/0/autocomplete.jquery.min.js"></script>
    <script src="{{ asset('js/algolia-autocomplete.js') }}"></script>
    <script src="{{ asset('js/admin_article_form.js') }}"></script>
{% endblock %}
// ... lines 10 - 24
```

Ah, cool. Replace the `admin_article_form.js` script with our helper Twig function:

templates/article_admin/new.html.twig

```
// ... lines 1 - 2
{% block javascripts %}
// ... lines 4 - 7
    {{ encore_entry_script_tags('admin_article_form') }}
{% endblock %}
// ... lines 10 - 25
```

Under stylesheets, the new page doesn't use Dropzone, so it didn't have that same link tag here. Add `{{ encore_entry_link_tags('admin_article_form') }}` anyways so that this page has *all* the JS and CSS it needs:

templates/article_admin/new.html.twig

```
// ... lines 1 - 10
{% block stylesheets %}
    {{ parent() }}

    {{ encore_entry_link_tags('admin_article_form') }}
// ... line 15
{% endblock %}
// ... lines 17 - 25
```

But this *does* highlight one... let's say... "not ideal" thing. Some of the JavaScript on the edit page - like the Dropzone & Sortable stuff - isn't needed here... but it's part of `admin_article_form.js` anyways. And actually, the reverse is true! That autocomplete stuff? That's needed on the "new" page, but not the edit page. At the end of the tutorial, we'll talk about async imports, which is one really nice way to help avoid packaging code all the time that is only needed *some* of the time.

Anyways, if we refresh now... the page is still *totally* broken! Apparently this "autocomplete" library we're importing is trying to reference jQuery. Let's fix that next... which will involve a... sort of "magical" feature of Webpack and Encore.

Chapter 13: Refactoring Autocomplete JS & CSS

We still have work to do to get the `new.html.twig` template working:

templates/article_admin/new.html.twig

```
// ... lines 1 - 2
{% block javascripts %}
// ... lines 4 - 5
    <script src="https://cdn.jsdelivr.net/autocomplete.js/0/autocomplete.jquery.min.js"></script>
    <script src="{{ asset('js/algolia-autocomplete.js') }}"></script>
// ... line 8
{% endblock %}
// ... lines 10 - 25
```

we have a script tag for this external autocomplete library and one for our own `public/js/algolia-autocomplete.js` file... which is our *last* JavaScript file in the `public/` directory! Woo!

public/js/algolia-autocomplete.js

```
$(document).ready(function() {
    $('.js-user-autocomplete').each(function() {
        var autocompleteUrl = $(this).data('autocomplete-url');

        $(this).autocomplete({hint: false}, [
            {
                source: function(query, cb) {
                    $.ajax({
                        url: autocompleteUrl+'?query='+query
                    }).then(function(data) {
                        cb(data.users);
                    });
                },
                displayKey: 'email',
                debounce: 500 // only request every 1/2 second
            }
        ])
    });
});
```

This holds code that adds auto-completion... on this author box... which, yes, is *totally* broken.

Installing the Autocomplete Library

To start, remove the CDN link to this autocomplete library:

templates/article_admin/new.html.twig

```
// ... lines 1 - 2
{% block javascripts %}
// ... lines 4 - 5
    <script src="https://cdn.jsdelivr.net/autocomplete.js/0/autocomplete.jquery.min.js"></script>
// ... lines 7 - 8
{% endblock %}
// ... lines 10 - 25
```

And, at your terminal, install it properly!

```
yarn add autocomplete.js --dev
```

Organizing our Autocomplete into a Component

Next, you know the drill, take the `algolia-autocomplete.js` file and move it into the `assets/js/` directory. But I'm *not* going to make this a new entry point. We *could* do that, but really, we *already* have an entry file that's included on this page: `admin_article_form`:

templates/article_admin/new.html.twig

```
// ... lines 1 - 2
{% block javascripts %}
// ... lines 4 - 7
    {{ encore_entry_script_tags('admin_article_form') }}
{% endblock %}
// ... lines 10 - 25
```

So really, `admin_article_form.js` should *probably* just *use* the code from `algolia-autocomplete.js`.

So, move that file into the *components/* directory... which is kind of meant for reusable modules. And... well, this isn't really *written* like a re-usable module yet because it just executes code instead or returning something, like a function. But, we'll work on that later.

Let's also take the `algolia-autocomplete.css` file and move that all the way up here into `assets/css/`. And just because we can, I'll make it an SCSS file!

Okay! Back in `admin_article_form.js`, let's bring in this code: `import './components/algolia-autocomplete'`:

assets/js/admin_article_form.js

```
// ... lines 1 - 4
import './components/algolia-autocomplete';
// ... lines 6 - 157
```

We don't need an `import from` yet... because that file doesn't actually *export* anything. For the CSS: `import '../css/algolia-autocomplete.scss'`:

assets/js/admin_article_form.js

```
// ... lines 1 - 4
import './components/algolia-autocomplete';
import '../css/algolia-autocomplete.scss';
// ... lines 7 - 157
```

Back in `new.html.twig`, the *great* thing is, we don't need to import this CSS file anymore or any of these script files. This is *really* how we want our templates to look: a single a call to `{{ encore_entry_script_tags() }}` and a single call to `{{ encore_entry_link_tags() }}`:

templates/article_admin/new.html.twig

```
// ... lines 1 - 2
{% block javascripts %}
    {{ parent() }}

    {{ encore_entry_script_tags('admin_article_form') }}
{% endblock %}

{% block stylesheets %}
    {{ parent() }}

    {{ encore_entry_link_tags('admin_article_form') }}
{% endblock %}
// ... lines 14 - 22
```

So if we refresh right now, not surprisingly, it *still* won't work! And it's our *favorite* error!

> *"$ is undefined"*

from `algolia-autocomplete.js`. Yes, this *is* the error I see when I close my eyes at night.

Using the autocomplete.js Library

Let's get to work. Of course, *we* are referencing `$`. So, `import $ from 'jquery'`:

assets/js/components/algolia-autocomplete.js

```
1  import $ from 'jquery';
↕  // ... lines 2 - 23
```

We're *also* using the autocomplete library in here. No problem: `import autocomplete from 'autocomplete.js'`:

assets/js/components/algolia-autocomplete.js

```
1  import $ from 'jquery';
2  import autocomplete from 'autocomplete.js';
↕  // ... lines 3 - 23
```

Wait... that's not quite right. This `autocomplete.js` library is a standalone JavaScript library that can be used with anything - jQuery, React, whatever. But... our *existing* code isn't using the "standalone" version of the library. It's using a jQuery plugin - this `.autocomplete()` function - that comes with that package:

assets/js/components/algolia-autocomplete.js

```
↕  // ... lines 1 - 3
4  $(document).ready(function() {
5      $('.js-user-autocomplete').each(function() {
↕  // ... lines 6 - 7
8          $(this).autocomplete({hint: false}, [
↕  // ... lines 9 - 19
20         ])
21     });
22 });
```

So, we *could* refactor our code down here to use the, kind of, *official* way of using this library - independent of jQuery. But... that's the *easy* way out! Let's see if we can get this to work as a jQuery plugin.

Finding and Using the jQuery Plugin

I'll hold `Command` or `Control` and click into `autocomplete.js`. Then double-click the directory to *zoom* us there. The "main" file is this `index.js` at the root of the directory. But if you look in `dist/`, hey! `autocomplete.jquery.js`! That's what we were including *before* via the `<script>` tag!

So instead of importing the main file, let's import `autocomplete.js/dist/autocomplete.jquery`:

assets/js/components/algolia-autocomplete.js

```
1  import $ from 'jquery';
2  import 'autocomplete.js/dist/autocomplete.jquery';
↕  // ... lines 3 - 23
```

And remember, we don't use import *from* with jQuery plugins... because they don't return anything: they *modify* the jQuery object.

Ok, I think we're *great* and I think we're ready. Move over, refresh and... huh:

> *"jQuery is not defined"*

Notice it doesn't say "$ is not defined": it says "jQuery is not defined"... and it's coming from `autocomplete.jquery.js`! It's coming from the *third* party package!

This... is tricky. Plain and simple, that file is written *incorrectly*. Yea, it *only* works if jQuery is a *global* variable! And in Webpack... it's not! Let's talk more about this *and* fix it with some black magic, next.

Chapter 14: Auto-Provide jQuery for Mischievous Packages

Everything *should* be working... but nope! We've got this

> *"jQuery is not defined"*

error... but it's not from *our* code! It's coming from inside of `autocomplete.jquery.js` - that third party package we installed!

Poorly-Behaved jQuery Packages

This is the *second* jQuery plugin that we've used. The first was bootstrap... and that worked brilliantly! Look inside `app.js`:

assets/js/app.js

```
↕ // ... lines 1 - 10
11 import $ from 'jquery';
12 import 'bootstrap'; // adds functions to jQuery
↕ // ... lines 13 - 26
```

We imported bootstrap and, yea... that was it. Bootstrap is a well-written jQuery plugin, which means that inside, it *imports* `jquery` - just like we do - and then modifies it.

But this Algolia `autocomplete.js` plugin? Yea, it's *not* so well-written. Instead of detecting that we're inside Webpack and *importing* `jQuery`, it just says... `jQuery`! And expects it to be available as a global variable. *This* is why jQuery plugins are a special monster: they've been around for *so* long, that they don't always play nicely in the modern way of doing things.

So... are we stuck? I mean, this 3rd-party package is *literally* written incorrectly! What can we do?

autoProvidejQuery()

Well... it's Webpack to the rescue! Open up `webpack.config.js` and find some commented-out code: `autoProvidejQuery()`. Uncomment that:

webpack.config.js

```
↕ // ... lines 1 - 2
3 Encore
↕ // ... lines 4 - 67
68     // uncomment if you're having problems with a jQuery plugin
69     .autoProvidejQuery()
↕ // ... lines 70 - 73
74 ;
↕ // ... lines 75 - 76
```

Then, go restart Encore:

```
yarn watch
```

When it finishes, move back over and... refresh! No errors! And if I start typing in the autocomplete box... it works! What black magic is this?!

The `.autoProvidejQuery()` method... yea... it sorta *is* black magic. Webpack is already scanning all of our code. When you enable this feature, *each* time it finds a `jQuery` or `$` variable- *anywhere* in *any* of the code that we use - that is *uninitialized*, it replaces it with `require('jquery')`. It basically *rewrites* the broken code to be correct.

Including CSS from the Algolia JS

While we're here, there's an organizational improvement I want to make. Look inside `admin_article_form.js`. Hmm, we include both the JavaScript file *and* the CSS file for Algolia autocomplete:

assets/js/admin_article_form.js

```
↕ // ... lines 1 - 4
5 import './components/algolia-autocomplete';
6 import '../css/algolia-autocomplete.scss';
↕ // ... lines 7 - 157
```

But if you think about it, this CSS file is *meant* to support the `algolia-autocomplete.js` file. To say it differently: the CSS file is a *dependency* of `algolia-autocomplete.js`: if that file was

ever used *without* this CSS file, things wouldn't look right.

Take out the `import` and move it into `algolia-autocomplete.js`. Make sure to update the path:

assets/js/components/algolia-autocomplete.js

```
1  import $ from 'jquery';
2  import 'autocomplete.js/dist/autocomplete.jquery';
3  import '../../css/algolia-autocomplete.scss';
↕  // ... lines 4 - 24
```

That's nice! If we want to use this autocomplete logic somewhere else, we *only* need to import the JavaScript file: *it* takes care of importing everything else. The result is the same, but cleaner.

Making algolia-autocomplete.js a Proper Module

Well, this file still isn't *as* clean as I want it. We're importing the `algolia-autocomplete.js` file... but it's not *really* a "module". It doesn't export some reusable function or class: it just *runs* code. I *really* want to start thinking of *all* of our JavaScript files - except for the entry files themselves - as *reusable* components.

Check it out: instead of just "doing" stuff, let's `export` a new function that can initialize the autocomplete logic. Replace `$(document).ready()` with `export default function()` with three arguments: the jQuery `$elements` that we want to attach the autocomplete behavior to, the `dataKey`, which will be used down here as a way of a defining *where* to get the data from on the Ajax response, and `displayKey` - another config option used at the bottom, which is the key on each result that should be displayed in the box:

assets/js/components/algolia-autocomplete.js

```
↕  // ... lines 1 - 4
5  export default function($elements, dataKey, displayKey) {
↕  // ... lines 6 - 25
26 };
```

Basically, we're taking out all the specific parts and replacing them with generic variables.

Now we can say `$elements.each()`:

assets/js/components/algolia-autocomplete.js

```
// ... lines 1 - 4
export default function($elements, dataKey, displayKey) {
    $elements.each(function() {
// ... lines 7 - 24
    });
};
```

And for `dataKey`, we can put a bit of logic: `if (dataKey)`, then `data = data[dataKey]`, and finally just `cb(data)`:

assets/js/components/algolia-autocomplete.js

```
// ... lines 1 - 4
export default function($elements, dataKey, displayKey) {
    $elements.each(function() {
        var autocompleteUrl = $(this).data('autocomplete-url');

        $(this).autocomplete({hint: false}, [
            {
                source: function(query, cb) {
                    $.ajax({
                        url: autocompleteUrl+'?query='+query
                    }).then(function(data) {
                        if (dataKey) {
                            data = data[dataKey];
                        }
                        cb(data);
                    });
                },
// ... lines 21 - 22
            }
        ])
    });
};
```

Some of this is specific to exactly how the Autocomplete library itself works - we set that up in an earlier tutorial. Down at the bottom, set `displayKey` to `displayKey`:

assets/js/components/algolia-autocomplete.js

```
// ... lines 1 - 4
export default function($elements, dataKey, displayKey) {
    $elements.each(function() {
        var autocompleteUrl = $(this).data('autocomplete-url');

        $(this).autocomplete({hint: false}, [
            {
                source: function(query, cb) {
                    $.ajax({
                        url: autocompleteUrl+'?query='+query
                    }).then(function(data) {
                        if (dataKey) {
                            data = data[dataKey];
                        }
                        cb(data);
                    });
                },
                displayKey: displayKey,
                debounce: 500 // only request every 1/2 second
            }
        ])
    });
};
```

Beautiful! Instead of *doing* something, this file returns a reusable function. That should feel familiar if you come from the Symfony world: we organize code by creating files that contain reusable *classes*, instead of files that contain procedural code that instantly *does* something.

Ok! Back in `admin_article_form.js`, let's `import autocomplete from './components/algolia-autocomplete'`:

assets/js/admin_article_form.js

```
// ... lines 1 - 4
import autocomplete from './components/algolia-autocomplete';
// ... lines 6 - 161
```

Oooo. And then, `const $autoComplete = $('.js-user-autocomplete')` - to find the same element we were using before:

assets/js/admin_article_form.js

```
↕ // ... lines 1 - 8
9 $(document).ready(function() {
10     const $autoComplete = $('.js-user-autocomplete');
↕ // ... lines 11 - 44
45 });
↕ // ... lines 46 - 161
```

Then, if *not* `$autoComplete.is(':disabled')`, call `autocomplete()` - because that's the variable we imported - and pass it `$autoComplete`, `users` for `dataKey` and `email` for `displayKey`:

assets/js/admin_article_form.js

```
↕ // ... lines 1 - 8
9 $(document).ready(function() {
10     const $autoComplete = $('.js-user-autocomplete');
11     if (!$autoComplete.is(':disabled')) {
12         autocomplete($autoComplete, 'users', 'email');
13     }
↕ // ... lines 14 - 44
45 });
↕ // ... lines 46 - 161
```

I love it! By the way, the reason I'm added this `:disabled` logic is that we originally set up our forms so that the `author` field that we're adding this autocomplete to is *disabled* on the edit form. So, there's no reason to try to add the autocomplete stuff in that case.

Ok, refresh... then type `admi`... it works! Double-check that we didn't break the edit page: go back to `/admin/article`, edit any article and, yea! Looks good! The field is disabled, but nothing is breaking.

Hey! We have *no* more JavaScript files in our `public/` directory. Woo! *But*, we *do* still have 2 CSS files. Let's handle those next.

Chapter 15: addStyleEntry(): CSS-Only Entrypoint

There are only two files left in the `public/` directory, and they're both CSS files! Celebrate by crushing your `js/` directory.

We have *two* page-specific CSS files left. Open `account/index.html.twig`:

templates/account/index.html.twig

```
↕ // ... lines 1 - 4
5 {% block stylesheets %}
↕ // ... lines 6 - 7
8     <link rel="stylesheet" href="{{ asset('css/account.css') }}">
9 {% endblock %}
↕ // ... lines 10 - 51
```

Yep, this has a link tag to the first... and in `security/login.html.twig`, here's the other:

templates/security/login.html.twig

```
↕ // ... lines 1 - 4
5 {% block stylesheets %}
↕ // ... lines 6 - 7
8     <link rel="stylesheet" href="{{ asset('css/login.css') }}">
9 {% endblock %}
↕ // ... lines 10 - 37
```

Oh, and we also include `login.css` from `register.html.twig`:

templates/security/register.html.twig

```
↕ // ... lines 1 - 28
29 {% block stylesheets %}
↕ // ... lines 30 - 31
32     <link rel="stylesheet" href="{{ asset('css/login.css') }}">
33 {% endblock %}
↕ // ... lines 34 - 78
```

This is kind of a tricky situation.... because what Webpack *wants* you to do is *always* start with a *JavaScript* entry file. And of course, if you *happen* to import some CSS, it'll nicely dump a CSS file. This comes from the single-page application mindset: if *everything* in your app is built by JavaScript, then of *course* you have a JavaScript file!

So... hmm. I mean, we *could* leave those files in `public/` - we don't *need* them to go through Webpack. Though... I *would* like to use Sass. We could *also* create `account.js` and `login.js` files... and then just import each CSS file from inside. That would work... but then Webpack would output empty `account.js` and `login.js` files... which isn't *horrible*, but not ideal... and kinda weird.

In the Encore world, just like with Webpack, we really *do* want you to *try* to do it the "proper" way: create a JavaScript entry file and "import" any CSS that it needs. But, we *also* recognize that this is a legitimate situation. So, Encore has a little extra magic.

First, move both of the files up into our `assets/css/` directory. And just because we can, make both of them `scss` files.

Next, in `webpack.config.js` add a special thing called `addStyleEntry()`. We'll have one called `account` pointing to `./assets/css/account.scss` and another one called `login` pointing to `login.scss`:

webpack.config.js

```
// ... lines 1 - 2
Encore
// ... lines 4 - 22
    .addStyleEntry('account', './assets/css/account.scss')
    .addStyleEntry('login', './assets/css/login.scss')
// ... lines 25 - 75
;
// ... lines 77 - 78
```

Easy enough! Find your Encore build, press `Control` + `C`, and restart it:

```
yarn watch
```

Awesome! We can see that the `account` and `login` entries both only dump CSS files.

And *this* means that, back in `index.html.twig`, we can replace the link tag with `{{ encore_entry_link_tags('account') }}`:

templates/account/index.html.twig

```
// ... lines 1 - 4
{% block stylesheets %}
    {{ parent() }}

    {{ encore_entry_link_tags('account') }}
{% endblock %}
// ... lines 10 - 51
```

Copy that and do the same thing in `login.html.twig` for the `login` entry:

templates/security/login.html.twig

```
// ... lines 1 - 4
{% block stylesheets %}
    {{ parent() }}

    {{ encore_entry_link_tags('login') }}
{% endblock %}
// ... lines 10 - 37
```

And then in `register.html.twig`, one more time for `login`:

templates/security/register.html.twig

```
// ... lines 1 - 28
{% block stylesheets %}
    {{ parent() }}

    {{ encore_entry_link_tags('login') }}
{% endblock %}
// ... lines 34 - 78
```

Ok! Let's double-check that the site doesn't explode. Go to the `/account` profile page. Everything looks fine.

So... yea, `addStyleEntry()` is available for this. But... to pull it off, Encore does some hacking internally. Really, `addStyleEntry()` is the *same* as `addEntry()`, which means that Webpack *does* try to output an empty JavaScript file. Encore basically just deletes that file so that we don't have to look at it.

Next, oh, we get to talk about one of my *favorite* things about Webpack and Encore: how to *automatically* convert your CSS - and JavaScript - so that it's understood by older browsers. *And* how to control *exactly* which browsers your site needs to support.

Chapter 16: Support any Browser with PostCSS & Babel

Go back to `/admin/article` and click to edit one of the articles. View the source and search for `.js`. Okay, we have several JavaScript files, because Webpack is splitting them. Click to look at `build/admin_article_form.js`, which will probably contain all the non-vendor code from that entry point.

The top of the file contains some Webpack booostrap stuff, then our code is below, still mixed in with some things that makes Webpack work.

Now, check this out: in the *original* `admin_article_form.js` file, we created a class called `ReferenceList`:

assets/js/admin_article_form.js

```
↕ // ... lines 1 - 8
9 $(document).ready(function() {
10     const $autoComplete = $('.js-user-autocomplete');
↕ // ... lines 11 - 44
45 });
46
47 // todo - use Webpack Encore so ES6 syntax is transpiled to ES5
48 class ReferenceList
49 {
↕ // ... lines 50 - 134
135 }
↕ // ... lines 136 - 161
```

And we also use the `const` keyword for `const $autoComplete`. Back in the compiled file, search for `$autoComplete`. Woh! It's not `const $autoComplete`, it's `var $autoComplete`! And if you search for `ReferenceList`... and get down to the class... there's no class syntax! It's wrapped in some sort of a "pure" function thingy.

Surprise! Something is *rewriting* our code! But, who? And, why?

Hello Babel

The *who* is Babel: an amazing library that has the superpower of reading your JavaScript and *rewriting* it to *older* JavaScript that's compatible with older browsers. And this is *seriously* important! Because if JavaScript comes out with a new feature, we do *not* want to wait 10 years for all of the browsers to support it! Babel solves this: you can use *brand* new language features and it compiles it to boring, traditional code.

But... wait. How is Babel deciding which browsers our site needs to support? Different sites need to support different browsers... and so, in theory, Babel should be able to rewrite the code *differently* for different sites. For example, if you need to support *super* old browsers, you probably need to rewrite `const` to `var`. But if all of your users are *awesome*... like our SymfonyCasts users... and all use *new* browsers, then you *don't* need to rewrite this. In general, converting new code to *old* code makes your JavaScript *larger*, so avoiding unnecessary changes is a good thing.

Rewriting CSS for Older Browsers?

Let's answer the question of "how" we can control Babel by talking about something *completely* different: CSS. Babel does *not* rewrite CSS. But, if you think about it, it *would* sorta make sense.

For example, if you're using a `border-radius` and need to support older browsers, you need to add some vendor prefixes, like `-webkit-border-radius`. You can see one we added manually down here: we have `box-shadow`, but we *also* have `-webkit-box-shadow` to make it work in some older browsers... which we might not even need, depending on what browsers we decide we need to support:

assets/css/account.scss

```
↕ // ... lines 1 - 13
14 div.user-menu-container {
↕ // ... lines 15 - 20
21   -webkit-box-shadow: 0 1px 6px rgba(0, 0, 0, 0.175);
22   box-shadow: 0 1px 6px rgba(0, 0, 0, 0.175);
23 }
↕ // ... lines 24 - 90
```

Anyways, forgetting about Webpack and Babel for a minute, in the CSS world, you do *not* need to add these vendor prefixes by hand. Nope! There's a wonderful library that can do it for you called `autoprefixer`. *You* write code correctly - like using `box-shadow` - tell it *which* browsers you need to support, and it adds the vendor prefixes for you.

Enabling PostCSS

Because that sounds *amazing*... let's add it! In `webpack.config.js`, anywhere, but how about below `.enableSassLoader()`, add `.enablePostCssLoader()`:

webpack.config.js

```
// ... lines 1 - 2
Encore
// ... lines 4 - 56
    .enablePostCssLoader()
// ... lines 58 - 76
;
// ... lines 78 - 79
```

PostCSS is a library that allows you to run things at the "end" of your CSS being processed. And it's the easiest way to integrate `autoprefixer`.

Next, because we just changed our `webpack.config.js` file, go restart Encore:

```
yarn watch
```

Hey! This is familiar! Just like when we enabled Sass, this requires us to install a few things. Copy the command, go to your open terminal and run that!

```
yarn add postcss-loader@^3.0.0 --dev
```

Ok, let's try Encore again:

```
yarn watch
```

Hmm, *another* error! This is kinda cool: to use PostCSS, you *need* to create a `postcss.config.js` file. Encore walks you through that process and sets it up to use `autoprefixer` to start. Copy that, go to the root of your project, create the `postcss.config.js` file and paste:

postcss.config.js

```
module.exports = {
  plugins: {
    'autoprefixer': {},
  }
}
```

Ok, hit `Control` + `C` and try that again:

```
yarn watch
```

Sheesh! One last error. PostCSS is probably *the* most involved thing to get running. This error isn't as obvious:

> *"loading PostCSS plugin failed: Cannot find module autoprefixer"*

We know what that word "module" means! It's trying to find that library. We told PostCSS to use `autoprefixer`, but that doesn't exist in our project yet. Run:

```
yarn add autoprefixer --dev
```

And *now* try Encore.

Tip

If you get an error like `true is not a PostCSS plugin`, either downgrade autoprefixer to version 9 or upgrade PostCSS to version 8. Basically, autoprefixer 10 doesn't play nicely with PostCSS 7 and lower.

```
yarn watch
```

No errors! So... it's *probably* working? Let's see it in action next *and* learn how we can tell PostCSS *and* Babel *exactly* which browsers we need to support.

Chapter 17: browserslist: What Browsers do you need to Support?

PostCSS is running! Let's see what it does! Go back to your browser. We haven't reloaded the page yet. I'll search for `app.css` and click to open that. Search for one of the vendor prefixes: `-webkit`. Ok, so *before* adding PostCSS, we have 77 occurrences - coming from our code and Bootstrap.

In *theory*, if we told PostCSS that we need to support *really* old browsers, this number should get *way* higher! How can we do that? Some config in `postcss.config.js`? Actually, no. It's *way* cooler than that.

Hello browserslist

In the JavaScript world, there is a *wonderful* library called `browserslist`. It's a pretty simple idea: `browserslist` allows you to *describe* which browsers your site needs to support, in a *bunch* of useful ways. Then, *any* tool that needs this information can read it from a central spot.

Check it out: open up your `package.json` file. Yes, *this* is where we'll configure what browsers we need to support. Add a new key: `browserslist` set to an array:

package.json

```
1  {
↕  // ... lines 2 - 25
26     "browserslist": [
↕  // ... line 27
28     ]
29 }
```

You can do a *ton* of things in here - like say that you want to support the last "2" versions of every browser *or* any browser that is used by more than 1% of the web *or* some *specific* browser that you know is used a lot on your site. Yea, `browserslist` uses *real-world* usage data to figure out *which* browsers you should support!

Let's use a simple example: `> .05%`:

package.json

```
{
// ... lines 2 - 25
    "browserslist": [
        "> .05%"
    ]
}
```

This is actually a pretty *unrealistic* setting. This says: I want to support all browsers that have *at least* .05% of the global browser usage. So this will include some *really* old browsers that, maybe only .06% of the world uses!

Stop and restart Webpack to force a rebuild and make sure PostCSS reads the new setting:

```
yarn watch
```

Now, go back, refresh `app.css`, search again for `-webkit` and woh! 992 results! That's amazing! By the way, there is *also* a tool called BrowserList-GA that reads from your Google Analytics account and *dumps* a data file with *your* real-world usage data. You can then use that in your `browserslist` config, by saying something like: `> 0.5% in my stats`, which literally means: support any browsers that is responsible for more than .5% of traffic from *my* site's real-world data. Cool.

Configuring Babel

So what about our JavaScript? Does Babel read this same `browserslist` config? Totally! Search for `.js` and click to open the compiled `admin_article_form.js` file. Inside, search for `$autocomplete`. Yep! We saw earlier that Babel is outputting `var $autoComplete`, even though this was *originally* `const $autoComplete`. That makes sense: we said that we want to support *really* old browsers.

So... what if we change the `browserslist` config to `> 5%`?

package.json

```
{
// ... lines 2 - 25
    "browserslist": [
        "> 5%"
    ]
}
```

That's probably still a *bit* unrealistic: this will only support the *most* popular browsers and versions: pretty much *no* old stuff. Stop and re-run Encore:

```
yarn watch
```

Then move back over to `admin_article_form.js` and refresh. I'll do a force refresh to be sure... then search for `$autoComplete`. And... huh? It's *still* var? Hmm, that *might* be right... but `const` was added in 2015 - it *should* be fully supported by all modern browsers by now.

It turns out... it *is*, and we're not seeing the changes due to a small bug in Babel. Behind the scenes, Babel uses some smart caching so that it doesn't need to reparse and recompile *every* JavaScript file *every* time Webpack builds. But, at the time of recording, Babel's cache *isn't* smart enough to know that it needs invalidate itself when the `browserslist` config changes.

Once you know this, it's no big deal: *anytime* you change the `browserslist` config, you need to manually clear Babel's cache. In my terminal, I'll run:

```
rm -rf node_modules/.cache/babel-loader/
```

Now restart Encore:

```
yarn watch
```

Let's check it out! Refresh and search for `$autoComplete`. There it is: `const $autoComplete`. Look also for `class ReferenceList`. Now that we're only supporting new browsers, that code doesn't need to be rewritten either.

Oh, but there *is* one type of thing that Babel can't simply rewrite into code that's compatible with olders browsers. When you use a totally new *feature* of JavaScript - like the `fetch()` function for AJAX calls, you need to include a *polyfill* library so that old browsers have this. But... even for this, Babel has a trick up its sleeve. That's next.

Chapter 18: Polyfills & Babel

Babel is pretty amazing. But, it's even doing something *else* automatically that we haven't realized yet! Back in `admin_article_form.js`, and it doesn't matter where, but down in `ReferenceList`, I'm going to add `var stuff = new WeakSet([]);`:

assets/js/admin_article_form.js

```
↕ // ... lines 1 - 47
48 class ReferenceList
49 {
50     constructor($element) {
51         var stuff = new WeakSet([]);
↕ // ... lines 52 - 81
82     }
↕ // ... lines 83 - 136
137 }
↕ // ... lines 138 - 163
```

`WeakSet` is an object that was introduced to JavaScript, um, ECMAScript in 2015. Because the Encore watch script is running, go over and refresh the built file. Here it is: `var stuff = new WeakSet([]);`.

New Features & Polyfills

That's not surprising, right? I mean, we're telling Babel that we *only* need to support *really* new browsers, so there's no need to rewrite this to some old, compatible code... right? Well... it's more complicated than that. `WeakSet` is not a new *syntax* that Babel can simply change to some old syntax: it's an entirely new feature! There are a bunch of these and some are *really* important, like the `Promise` object and the `fetch()` function for AJAX calls.

To support totally new features, you need something called a *polyfill*. A polyfill is a normal JavaScript library that *adds* a feature if it's missing. For example, there's a polyfill *just* for `WeakSet`, which you can import if you want to make sure that `WeakSet` will work in *any* browser.

But, keeping track of whether or not you imported a polyfill... and whether or not you even *need* a polyfill - maybe the feature is *already* available in the browsers you need to support - is a pain!

So... Encore pre-configures Babel to... just do it for us.

Check it out. Go back to `package.json` and change this to support older browsers:

package.json

```
{
// ... lines 2 - 25
    "browserslist": [
        "> .05%"
    ]
}
```

Then, just like before, go to your terminal and manually clear the Babel cache:

```
rm -rf node_modules/.cache/babel-loader/
```

And restart Encore:

```
yarn watch
```

Ok, let's go back to the browser, refresh the built JavaScript file and search for `WeakSet`. It *still* looks *exactly* like our original code. But *now*, just search for "weak". Woh. This is a bit hard to read, but it's importing something called `core-js/modules/es.weak-set`.

This `core-js` package is a library *full* of polyfills. Babel *realized* that we're trying to use `WeakSet` and so it *automatically* added an import statement for the polyfill! This is *identical* to us *manually* going to the top of the file and adding `import 'core-js/modules/es.weak-set'`. How cool is that?!

A Polyfill from the Past!

And... this is *not* the first time Babel has automatically added a polyfill! Open up `build/app.js`. Back in the editor, the `get_nice_message` module used a String method called `repeat()`:

assets/js/components/get_nice_message.js

```
1  export default function(exclamationCount) {
2      return 'Hello Webpack Encore! Edit me in
   assets/js/app.js'+'!'.repeat(exclamationCount);
3  };
```

Whelp, it turns out that `repeat()` is a *fairly* new feature!

Search for "repeat" in the built file. There it is: it's importing `core-js/modules/es.string.repeat`. When I used this function, I wasn't even *thinking* about whether or not that feature was new and if it was available in the browsers we need to support! But because Encore has our back, it wasn't a problem. That's a powerful idea.

By the way, this is all configured in `webpack.config.js`: it's this `.configureBabel()` method:

webpack.config.js

```
↕  // ... lines 1 - 2
3  Encore
↕  // ... lines 4 - 48
49     // enables @babel/preset-env polyfills
50     .configureBabel(() => {}, {
51         useBuiltIns: 'usage',
52         corejs: 3
53     })
↕  // ... lines 54 - 76
77 ;
↕  // ... lines 78 - 79
```

Generally-speaking, this is how you can configure Babel. The `useBuiltIns: 'usage'` and `corejs: 3` are the key parts. Together, these say:

> *"Please, automatically import polyfills when you see that I'm using a new feature and I've already installed version 3 of `corejs`."*

That package was pre-installed in the original `package.json` we got from the recipe.

Next: let's demystify a feature that we disabled *way* back at the beginning of this tutorial: the single runtime chunk.

Chapter 19: The Single Runtime Chunk

Head back to the homepage and click any of the articles. In an earlier tutorial, we added this heart icon that, when you click it, makes an AJAX request and increases the counter. Well, part of this is faked on the backend, but you get the idea.

To make this more clear, let's add a Bootstrap tooltip: when the user hovers over the heart, we can say something like "Click to like". No problem: open up the template: `article/show.html.twig`. And I'll remind you that this page has its own entry: `article_show.js`:

templates/article/show.html.twig

```
// ... lines 1 - 80
{% block javascripts %}
    {{ parent() }}

    {{ encore_entry_script_tags('article_show') }}
{% endblock %}
// ... lines 86 - 92
```

Go open that: `assets/js/article_show.js`.

Ok, let's find the anchor tag in the template... there it is... and use multiple lines for sanity. Now add `title="Click to Like"`:

templates/article/show.html.twig

```
// ... lines 1 - 4
{% block content_body %}
    <div class="row">
        <div class="col-sm-12">
// ... line 8
            <div class="show-article-title-container d-inline-block pl-3 align-middle">
// ... lines 10 - 15
                <span class="pl-2 article-details">
// ... line 17
                    <a href="{{ path('article_toggle_heart', {slug: article.slug}) }}" class="fa fa-heart-o like-article js-like-article" title="Click to Like!"></a>
                </span>
// ... lines 20 - 24
            </div>
        </div>
    </div>
// ... lines 28 - 78
{% endblock %}
// ... lines 80 - 92
```

To make this work, all *we* need to do is copy the `js-like-article` class, go back to `article_show.js` and add `$('.js-like-article').tooltip()`, which is a function added by Bootstrap:

assets/js/article_show.js

```
// ... lines 1 - 3
$(document).ready(function() {
    $('.js-like-article').tooltip();
// ... lines 6 - 19
});
```

Coolio! Let's try it. Refresh and... of course. It doesn't work:

> *"...tooltip is not a function"*

This may or may *not* surprise you. Think about it: at the bottom of the page, the `app.js` `<script>` tags are loaded first. And, if you remember, inside of `app.js`, we import `jquery` and then `bootstrap`, which *adds* the `tooltip()` function to jQuery:

assets/js/app.js

```
↕ // ... lines 1 - 10
11 import $ from 'jquery';
12 import 'bootstrap'; // adds functions to jQuery
↕ // ... lines 13 - 26
```

Are Modules Shared across Entries?

So, it's *reasonable* to think that, inside `article_show.js`, when we import `jquery`, we will get the *same* jQuery object that's already been modified by `bootstrap`. And... that's *almost* true. When two different files import the same module, they *do* get the exact same object in memory.

However, by default, Webpack treats different entrypoints like totally separate applications. So if we import `jquery` from `app.js` and also from `get_nice_message.js`, which is part of the same entry:

assets/js/app.js

```
↕ // ... lines 1 - 10
11 import $ from 'jquery';
12 import 'bootstrap'; // adds functions to jQuery
↕ // ... lines 13 - 14
15 import getNiceMessage from './components/get_nice_message';
↕ // ... lines 16 - 26
```

They *will* get the *same* jQuery object. But when we import `jquery` from `article_show.js`, we get a *different* object in memory. Each entrypoint has an isolated environment. It doesn't mean that jQuery is downloaded twice, it just means that we are given two different instances.

So the fix is simple: `import 'bootstrap'`.

Refresh and... this time, it works.

enableSingleRuntimeChunk()

Understanding that modules are *not* shared across entries is good to know. But this *also* relates to a feature I want to talk about: the runtime chunk.

In `webpack.config.js`, at the *very* beginning of the tutorial, we commented out `enableSingleRuntimeChunk()` and replaced it with `disableSingleRuntimeChunk()`:

webpack.config.js

```
↕ // ... lines 1 - 2
3 Encore
↕ // ... lines 4 - 30
31     // will require an extra script tag for runtime.js
32     // but, you probably want this, unless you're building a single-page app
33     //.enableSingleRuntimeChunk()
34     .disableSingleRuntimeChunk()
↕ // ... lines 35 - 76
77 ;
↕ // ... lines 78 - 79
```

Now, let's reverse that:

webpack.config.js

```
↕ // ... lines 1 - 2
3 Encore
↕ // ... lines 4 - 30
31     // will require an extra script tag for runtime.js
32     // but, you probably want this, unless you're building a single-page app
33     .enableSingleRuntimeChunk()
34     //.disableSingleRuntimeChunk()
↕ // ... lines 35 - 76
77 ;
↕ // ... lines 78 - 79
```

Because we just modified the Webpack config, come back over, press `Control` + `C` and restart it:

```
yarn watch
```

If you watch closely, you'll see an immediate difference. Every single entry *now* includes a new file called `runtime.js`, which means that it's a *new* file that needs to be included as the *first* script tag before *any* entry. Of course, that's not a detail that *we* need to worry about because, when we refresh and view the page source, our Twig functions took care of rendering everything.

Ok, so... why? What did this change and why did we care? There are two things.

Single Runtime Chunk & Caching

First, `runtime.js` contains Webpack's "runtime" code: stuff it needs to get its job done. By enabling the single runtime chunk you're saying:

> *"Hey Webpack! Instead of adding this code at the beginning of `app.js` and at the beginning of `article_show.js` and all my other entry files, only add it once to `runtime.js`"*

The user *now* has to download an extra file, but all the entry files are a bit smaller. But, there's *more* to it than that. The `runtime.js` file contains something called the "manifest", which is a fancy name that Webpack gives to code that contains some internal IDs that Webpack uses to identify different parts of your code. The *key* this is that those IDs often *change* between builds. So, by isolating that code into `runtime.js`, it means that our *other* JavaScript files - the ones that contain our big code - will change less often: when those internal IDs change, it will *not* affect their content.

The tl;dr is that the smaller `runtime.js` will change more often, but our bigger JavaScript files will change less often. That's great for caching.

Shared Runtime/Modules

The *other* thing that `enableSingleRuntimeChunk()` changes may or may not be a good thing. Go back to `article_show.js` and comment out `import 'bootstrap'`. Now, move over and refresh.

Yea, it *works*! When you enable the single runtime chunk, it has a *side effect*: modules are shared *across* your entry points: they all work a bit more like one, single application. That's not necessarily a good or bad thing: just something to be aware of. I still *do* recommend treating each entry file like its own independent environment, even if there *is* some sharing.

Next: it's time to talk about async imports! Have some code that's only used in certain situations? Make your built files smaller by loading it... effectively, via AJAX.

Chapter 20: Async Imports

Head back to `/admin/article`. We have a... sort of... "performance" issue here. When you create a new article, we have an author field that uses a bunch of autocomplete JavaScript and CSS. The thing is, if you go back and edit an article, this is purposely *not* used here.

So, what's the problem? Open `admin_article_form.js`. We import `algolia-autocomplete`:

assets/js/admin_article_form.js

```
// ... lines 1 - 4
import autocomplete from './components/algolia-autocomplete';
// ... lines 6 - 163
```

And it imports a third-party library and some CSS:

assets/js/components/algolia-autocomplete.js

```
import $ from 'jquery';
import 'autocomplete.js/dist/autocomplete.jquery';
import '../../css/algolia-autocomplete.scss';
// ... lines 4 - 27
```

So, it's not a *tiny* amount of code to get this working. The `admin_article_form.js` entry file is included on both the new *and* edit pages. But really, a big chunk of that file is *totally* unused on the edit page. What a waste!

Conditionally Dependencies?

The problem is that you can't conditionally import things: you can't put an if statement around the import, because Webpack needs to know, at build time, whether or not it should include the content of that import into the final built `admin_article_form.js` file.

But, this *is* a real-world problem! For example, suppose that when a user clicks a specific link on your site, a dialog screen pops up that requires a lot of JavaScript and CSS. Cool. But what if *most* users *don't* ever click that link? Making *all* your users download the dialog box JavaScript and CSS when only a *few* of them will ever need it is a waste! You're slowing down *everyone's* experience.

We need to be able to lazily load dependencies. And here's how.

Hello Async/Dynamic import()

Copy the file path then delete the import:

assets/js/admin_article_form.js

```
↕ // ... lines 1 - 4
5 import autocomplete from './components/algolia-autocomplete';
↕ // ... lines 6 - 163
```

All imports are *normally* at the top of the file. But now... down inside the if statement, *this* is when we know that we need to use that library. Use `import()` like a *function* and pass it the path that we want to import.

This works almost exactly like an AJAX call. It's not instant, so it returns a *Promise*. Add `.then()` and, for the callback, Webpack will pass us the module that we're importing: `autocomplete`:

assets/js/admin_article_form.js

```
↕ // ... lines 1 - 7
8  $(document).ready(function() {
9      const $autoComplete = $('.js-user-autocomplete');
10     if (!$autoComplete.is(':disabled')) {
11         import('./components/algolia-autocomplete').then((autocomplete) => {
↕ // ... line 12
13         });
14     }
↕ // ... lines 15 - 45
46 });
↕ // ... lines 47 - 164
```

Finish the arrow function, then move the old code inside:

assets/js/admin_article_form.js

```
// ... lines 1 - 7
$(document).ready(function() {
    const $autoComplete = $('.js-user-autocomplete');
    if (!$autoComplete.is(':disabled')) {
        import('./components/algolia-autocomplete').then((autocomplete) => {
            autocomplete($autoComplete, 'users', 'email');
        });
    }
// ... lines 15 - 45
});
// ... lines 47 - 164
```

So, it will hit our `import` code, download the JavaScript - just like an AJAX call - and when it finishes, call our function. *And*, because the "traditional" import call is gone from the top of the file, the autocomplete stuff *won't* be included in `admin_article_form.js`. That entry file just got smaller. That's freakin' awesome!

By the way, if we were running the code, like, after a user *clicked* something, there would be a small delay while the JavaScript was being downloaded. To make the experience fluid, you could add a loading animation before the `import()` call and stop it inside the callback.

Ok, let's try this! Go back to `/admin/article/new`. And... oh!

> *"autocomplete is not a function"*

Using module_name.default

in `article_form.js`. So... this is a little bit of a gotcha. If your module uses the newer, trendier, `export default` syntax:

assets/js/components/algolia-autocomplete.js

```
// ... lines 1 - 4
export default function($elements, dataKey, displayKey) {
// ... lines 6 - 25
};
```

When you use "async" or "dynamic" imports, you need to say `autocomplete.default()` in the callback:

assets/js/admin_article_form.js

```
// ... lines 1 - 7
$(document).ready(function() {
    const $autoComplete = $('.js-user-autocomplete');
    if (!$autoComplete.is(':disabled')) {
        import('./components/algolia-autocomplete').then((autocomplete) => {
            autocomplete.default($autoComplete, 'users', 'email');
        });
    }
// ... lines 15 - 45
});
// ... lines 47 - 164
```

Move back over and refresh again. No errors! And it works! But also, look at the Network tab - filter for "scripts". It downloaded `1.js` and `0.js`. The `1.js` file contains the *autocomplete* vendor library and `0.js` contains *our* JavaScript. It loaded this lazily *and* it's even "code splitting" our lazy JavaScript into two files... which is kinda crazy. The `0.js` *also* contains the CSS... well, it *says* it does... but it's not really there. *Because*, in the CSS tab, it's loaded via its own `0.css` file.

If you look at the DOM, you can even see how Webpack hacked the `script` and `link` tags into the `head` of our page: these were *not* there on page-load.

So... dynamic imports... just work! And you can imagine how powerful this could be in a single page application where you can asynchronously load the components for a page when the user *goes* to that page... instead of having one *gigantic* JavaScript file for your whole site.

By the way, the dynamic import syntax can be even simpler if you use the `await` keyword and some fancy destructuring. You'll also need to install a library called `regenerator-runtime`. Check out the code on this page for an example.

```
// and run: yarn add regenerator-runtime --dev

async function initializeAutocomplete($autoComplete) {
    const { default: autocomplete } = await import('./components/algolia-autocomplete')

    autocomplete($autoComplete, 'users', 'email');
}

$(document).ready(function() {
    const $autoComplete = $('.js-user-autocomplete');
    if (!$autoComplete.is(':disabled')) {
```

```
        initializeAutocomplete($autoComplete);
    }

    // ...
}
```

Next: there's just one more thing to talk about: how to build our assets for production, and some tips on deployment.

Chapter 21: Production Build & Deployment

Ok team: just one more thing to talk about: how the heck can we deploy all of this to production?

Well, *before* that, our files aren't even ready for production yet! Open the `public/build/` directory. If you open any of these files, you'll notice that they are *not* minified. And at the bottom, each has a bunch of extra stuff for "sourcemaps": a bit of config that makes debugging our code easier in the browser.

Building For Production

We get *all* of this because we've been creating a *development* build. *Now*, at your terminal, run:

```
yarn build
```

This is a shortcut for `yarn encore production`. When we installed Encore, we got a pre-started `package.json` file with... this `scripts` section:

package.json

```
{
// ... lines 2 - 19
    "scripts": {
        "dev-server": "encore dev-server",
        "dev": "encore dev",
        "watch": "encore dev --watch",
        "build": "encore production --progress"
    },
// ... lines 26 - 28
}
```

So, the *real* command to build for production is `encore production`, or, really:

```
./node_modules/.bin/encore production
```

Anyways, that's the key thing: Encore has two main modes: `dev` and `production`.

And... done! On a big project, this might take a bit longer - production builds can be much slower than dev builds.

Now we have a *very* different `build/` directory. First, all of the names are bit obfuscated. Before, we had names that included things like `app~vendor`, which kind of exposed the internal structure of what entry points we had and how they're sharing data. No *huge* deal, but that's gone: replaced by these numbered files.

Also, if you look inside any of these, they're now totally minified and won't have the sourcemap at the bottom. You *will* still see these license headers - that's there for legal reasons, though you *can* configure them to be removed. Those are the only comments that are left in these final files.

And *even* though all the filenames just changed, we instantly move over, refresh, and... it works: the Twig helpers are rendering the new filenames.

Free Versioning

In fact, you may have noticed something special about the new filenames: every single one now has a *hash* in it. Inside our `webpack.config.js` file, this is happening thanks to this line: `enableVersioning()`:

webpack.config.js

```
// ... lines 1 - 2
Encore
// ... lines 4 - 45
    // enables hashed filenames (e.g. app.abc123.css)
    .enableVersioning(Encore.isProduction())
// ... lines 48 - 76
;
// ... lines 78 - 79
```

And check it out, the first argument - which is a boolean of whether or not we want versioning - is using a helper called `Encore.isProduction()`. That disables versioning for our dev builds, just cause we don't need it, but *enables* it for production.

The *really* awesome thing is that *every* time the *contents* of this `article_show.css` file changes, it will automatically get a new hash: the hash is built from the *contents* of the file. Of course, we don't need to change anything in our code, because the Twig helpers will

automatically render the new filename in the `script` or `link` tag. Basically... we get free file versioning, or browser cache busting.

This *also* means that you should *totally* take advantage of something called long-term caching. This is where you configure your web server - like `Nginx` - to set an `Expires` header on *every* file it serves from the `/build` directory with some super-distant value, like 1 year from now:

```
server {
    # ...

    location ~ ^\/build\/ {
        expires 365d;
        add_header Cache-Control "public";
    }
}
```

The result is that, once a user has downloaded these files, they will *never* ask our server for them again: they'll just use their browser cache. But, as soon as we *update* a file, it'll have a new filename and the user's browser will ask for it again. It's just free performance. And if you got a step further and put something like CloudFlare in front of your site, your server will receive even *less* requests for your assets.

Deployment

Now that we have these, optimized, versioned files, how can we deploy them up to production? Well... it depends. It depends on how sophisticated your deployment is.

If you have a really *simple* deployment, where you basically, run `git pull` on production and then clear the Symfony cache, you're probably going to need to install node on your production server, run `yarn install`, and then run `yarn build` up on production, each time you deploy. That's not ideal, but if you have a simple deployment system, that *keeps* it simple.

> **Tip**
>
> We show this on practice in our Animated Deployment with Ansistrano course.

If you have a slightly more sophisticated system, you can do it better. The *key* thing to understand is that, once you've run `yarn build`, the *only* thing that needs to go to production

is the `public/build` directory. So you could literally run `yarn build` on a different server - or even locally - and then just make sure that this `build/` directory gets copied to production.

That's it! You don't need to have `node` installed on production and you don't need to run anything with `yarn`. If you followed our tutorial on Ansistrano, you would run `yarn` wherever you're executing Ansistrano, then use the `copy` module to copy the directory.

More Features

Ok, that's it! Actually, there are *more* features inside Encore - many more, like enabling TypeScript, React or Vue support. But getting those all going should be easy for you now. Go try them, and report back.

And, like always, if you have any questions, find us in the comments section.

All right friends, seeya next time.

With <3 from SymfonyCasts