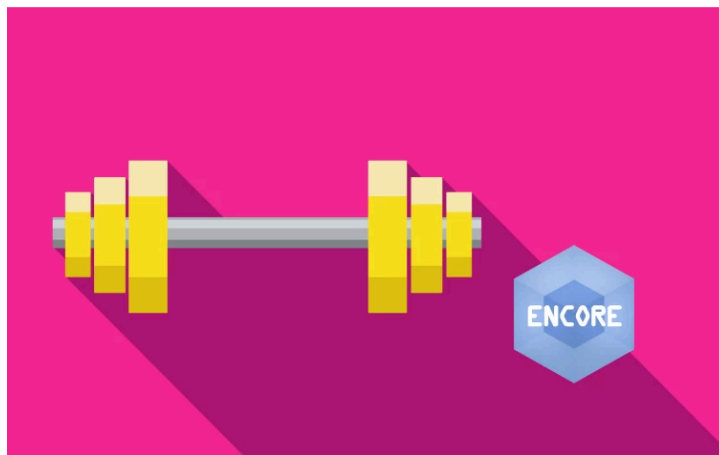# Webpack Encore: A Party for your Assets

Chapter 1: Installing Encore

Hiya guys! And welcome to our tutorial on Webpack Encore! I have to admit, this tutorial is near and dear to my heart, because I helped *write* Webpack Encore. But also because I think you're going to *love* working with it and I *know* that it's going to *drastically* improve the way you write JavaScript.

Basically, Encore is a wrapper around Webpack... because honestly, Webpack - while *amazing* - is a pain to setup. And what does Webpack do? We'll get to that.

Setting up the Project

And when we do.... you're *definitely* going to want to code along with me. Because, we're going to code JavaScript... dramatic pause... *correctly*!

Download the course code from this page. After you unzip it, you'll find a `start/` directory that has the same code you see here. Follow the `README.md` file to get setup details and, of course, a Haiku about Webpack.

The last step will be to find a terminal, move into the project, and run:

```
php bin/console server:run
```

to start the built-in web server. Find your most favorite browser and go to: http://localhost:8000.

Aw yea, it's Lift Stuff! Our startup for keeping track of all of the stuff... we lift! Login with username `ron_furgandy` password `pumpup`.

This is a two-page app: the login page and *this* incredible page: where we can record that - while programming today - we lifted our cat 10 times. I *love* exercise! Everything on this page works via AJAX and JavaScript... but the JavaScript is pretty traditional. If you watched our Webpack tutorial, we've actually reset this project back to *before* we introduced Webpack. Yep, in the `public/` directory, there are some normal CSS and JavaScript files. Nothing special.

Oh, and this is a Symfony *4* application... but that doesn't matter much. For you Symfony users out there, the only special setup I've done is to install the Asset component so that we can use the Twig `asset()` function:

```
templates/base.html.twig
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
    // ... lines 4 - 10
11      {% block stylesheets %}
    // ... lines 12 - 13
14          <link href="{{ asset('assets/css/main.css') }}" rel="stylesheet"
    />
15      {% endblock %}
    // ... lines 16 - 17
18  </head>
    // ... lines 19 - 108
109 </html>
```

On a fresh Symfony 4 project, run:

```
composer require asset
```

to get it.

The Magical require Statement

Ok... so what's all the fuss about with Webpack anyways? Well, the JavaScript file that runs the main page is called `RepLogApp.js`. Look inside: it holds *two* classes:

```
public/assets/js/RepLogApp.js
1   'use strict';
2
3   (function(window, $, Routing, swal) {
4
5       let HelperInstances = new WeakMap();
6
7       class RepLogApp {
⬍   // ... lines 8 - 192
193       }
194
195       /**
196        * A "private" object
197        */
198       class Helper {
⬍   // ... lines 199 - 226
227       }
228
229       const rowTemplate = (repLog) => `
230   <tr data-weight="${repLog.totalWeightLifted}">
231       <td>${repLog.itemLabel}</td>
232       <td>${repLog.reps}</td>
233       <td>${repLog.totalWeightLifted}</td>
234       <td>
235           <a href="#"
236               class="js-delete-rep-log"
237               data-url="${repLog.links._self}"
238           >
239               <span class="fa fa-trash"></span>
240           </a>
241       </td>
242   </tr>
243   `;
244
245       window.RepLogApp = RepLogApp;
246   })(window, jQuery, Routing, swal);
```

If you haven't see the `class` syntax in JavaScript, go back and watch episode 2 of our JavaScript series. It's cool stuff.

Anyways, we have a class `RepLogApp` and then.... way down below, we have `Helper`:

```
public/assets/js/RepLogApp.js
1    'use strict';
2
3    (function(window, $, Routing, swal) {
↕    // ... lines 4 - 6
7        class RepLogApp {
↕    // ... lines 8 - 192
193      }
194
195      /**
196       * A "private" object
197       */
198      class Helper {
↕    // ... lines 199 - 226
227      }
↕    // ... lines 228 - 245
246  })(window, jQuery, Routing, swal);
```

In PHP, we would *never* do this: we would organize each class into a different file. But in JavaScript, that's a pain! Because, if I move this `Helper` code to another file, then in my template, I need to remember to include a second `script` tag:

```
templates/lift/index.html.twig
↕    // ... lines 1 - 53
54   {% block javascripts %}
↕    // ... lines 55 - 57
58       <script src="{{ asset('assets/js/RepLogApp.js') }}"></script>
↕    // ... lines 59 - 65
66   {% endblock %}
```

This is why we can't have nice things.

But... what if we could *require* files in JavaScript... just like we can in PHP? Um... let's try it! Copy the Helper class, remove it, then - in the `js/` directory, add a new file: `RepLogHelper.js`. Paste the class here - I'll remove the comment on top:

```
public/assets/js/RepLogHelper.js
1  'use strict';
2
3  class Helper {
4      constructor(repLogs) {
5          this.repLogs = repLogs;
6      }
7
8      calculateTotalWeight() {
9          return Helper._calculateWeights(
10             this.repLogs
11         );
12     }
13
14     getTotalWeightString(maxWeight = 500) {
15         let weight = this.calculateTotalWeight();
16
17         if (weight > maxWeight) {
18             weight = maxWeight + '+';
19         }
20
21         return weight + ' lbs';
22     }
23
24     static _calculateWeights(repLogs) {
25         let totalWeight = 0;
26         for (let repLog of repLogs) {
27             totalWeight += repLog.totalWeightLifted;
28         }
29
30         return totalWeight;
31     }
32 }
   // ... lines 33 - 35
```

You see, in Node - which is server-side JavaScript, they have this idea of *modules*. Each file is a "module"... which doesn't mean much except that each file can export a *value* from itself. Then, other files, um, modules, can *require* that file to get that value.

In `RepLogHelper.js`, we really to make this `Helper` class available to other files. To *export* it, at the bottom, add `module.exports = Helper`:

```
public/assets/js/RepLogHelper.js
 1  'use strict';
 2
 3  class Helper {
 ↕  // ... lines 4 - 31
32  }
33
34  module.exports = Helper;
```

Now, in `RepLogApp`, at the top, add `const Helper = require('./RepLogHelper');`:

```
public/assets/js/RepLogApp.js
  1  'use strict';
  2
  3  const Helper = require('./RepLogHelper');
  4
  5  (function(window, $, Routing, swal) {
  ↕  // ... lines 6 - 213
214  })(window, jQuery, Routing, swal);
```

I want to highlight *two* things. First, you do *not* need the `.js` at the end of the filename. You *can* add it... but you don't need it - it's assumed. Second, the `./` *is* important: this tells the `require` function to look relative to this file. Later, we'll find out what it means to *not* start with `./`.

So here's the reality: if we ran this code on the *server* via Node... it would work! Yea! This `require()` thing is real! But... does it work in a browser?

Let's find out! Move over, open the debugging console and... refresh! Oh man!

> "*require* is not defined"

Booo! So... the `require()` function is *not* something that works in a browser... in *any* browser. And, the thing is, it *can't* work. PHP and Node are *server-side* languages, so Node can instantly read this file from the filesystem. But in a browser, in order to get this `RepLogHelper.js` file, it would need to make an AJAX request... and of course that's *far* from instant.

The point is: the `require()` function just doesn't make sense in a browser. And *this* is the problem that Webpack solves. Webpack is a command that can read this file, parse through *all* of the require calls and create one final JavaScript file packed with *all* the code we need.

But, we're not going to install Webpack directly. Google for "Webpack Encore" to find its [documentation on Symfony.com](#).

Installing Webpack Encore

Click into the Installation page and copy the `yarn add` line. And, some background: Webpack is a Node executable, so you'll need to make sure it's installed. And second... Node has *two* package managers: yarn and npm. You can use either - I'll use Yarn. So make sure you have that installed too.

Then, find your terminal, open a fresh new tab, lift your cat, and then run:

```
yarn add @symfony/webpack-encore --dev
```

> 💡 **Tip**
>
> Encore version 0.21.0 contains a few cool changes. Don't worry, we'll tell you in this tutorial where anything is now different.

If you're a Symfony user, there is also a composer line you can use. Actually, all it *really* does is install a Flex recipe that creates a few files for you to get you started faster. We'll do everything manually so that we can see how it works.

Move back and... it's done! If you're new to Yarn, this did two things. First, it created a `package.json` file:

```
package.json
1  {
2      "devDependencies": {
3          "@symfony/webpack-encore": "^0.19.0"
4      }
5  }
```

That's just like `composer.json` for Node - and also a `yarn.lock` file - that's like `composer.lock`. Second, it downloaded everything into a `node_modules/` directory: that's the `vendor/` directory for Node.

And *just* like in PHP, we do *not* want to commit all those vendor files. Open your `.gitignore` file and ignore `/node_modules/*`:

```.gitignore
1   /node_modules/*
    // ... lines 2 - 18
```

Brilliant! Encore is installed. Let's do some webpacking!

Chapter 2: Our First Encore

Create a new file: `webpack.config.js`. Here's how Webpack works: we create a config file that tells it *which* file to load, where to save the final file, and a few other things. Then... it does the rest!

But... Webpack's configuration is *complex*. A fully-featured setup will probably be a few *hundred* lines of complicate config! Heck, our Webpack tutorial was over 3 hours long! Very simply: Encore is a tool that helps generate that complex config.

Setting up webpack.config.js

Click on the documentation to find the "First Example". Hey! A `webpack.config.js` file to get us started! Copy that! Then, paste it in our file. But, I'm going to simplify and delete a few things: we'll add this stuff back later. *Just* keep `setOutputPath()`, `setPublicPath()` and `addEntry()`:

```
webpack.config.js
1   var Encore = require('@symfony/webpack-encore');
2
3   Encore
4       // the project directory where all compiled assets will be stored
5       .setOutputPath('public/build/')
6
7       // the public path used by the web server to access the previous
    directory
8       .setPublicPath('/build')
9
10      .addEntry('rep_log', './public/assets/js/RepLogApp.js')
11  ;
12
13  // export the final configuration
14  module.exports = Encore.getWebpackConfig();
```

And hey, check out that first line! Since this file will be executed by Node, we can require stuff! This imports the `Encore` object:

```
webpack.config.js
1  var Encore = require('@symfony/webpack-encore');
   // ... lines 2 - 15
```

Then, at the bottom, we ask Encore to give us the final config, and we *export* it:

```
webpack.config.js
   // ... lines 1 - 12
13  // export the final configuration
14  module.exports = Encore.getWebpackConfig();
```

There are only *three* things we need to tell Webpack: the directory where we want to save the final files - `public/build` - the public path to that directory - so `/build` since `public/` is the document root - and an "entry":

```
webpack.config.js
    // ... lines 1 - 2
3   Encore
4       // the project directory where all compiled assets will be stored
5       .setOutputPath('public/build/')
6
7       // the public path used by the web server to access the previous
    directory
8       .setPublicPath('/build')
9
10      .addEntry('rep_log', './public/assets/js/RepLogApp.js')
11  ;
    // ... lines 12 - 15
```

Point this to our JavaScript file: `./public/assets/js/RepLogApp.js`. Change the first argument to `rep_log`:

```
webpack.config.js
    // ... lines 1 - 2
3   Encore
    // ... lines 4 - 9
10      .addEntry('rep_log', './public/assets/js/RepLogApp.js')
11  ;
    // ... lines 12 - 15
```

This tells Webpack to work its magic on `RepLogApp.js`. The first argument will be the name of the final file, `.js` - so `rep_log.js`.

Running Encore

And... that's it! Find your terminal. Encore has its own executable. To use it, run:

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

The "dev" part tells Encore to create a "development" build. And... cool! Two files written to `public/build`. Let's check them out! Alright! There's `rep_log.js`. We'll talk about `manifest.json` later.

Cool! Let's point our script tag at the new file. Open `templates/lift/index.html.twig`. This is the template that runs our main page. At the bottom, change the path to `build/rep_log.js`:

```twig
templates/lift/index.html.twig
1   {% extends 'base.html.twig' %}
    // ... lines 2 - 53
54  {% block javascripts %}
    // ... lines 55 - 57
58      <script src="{{ asset('build/rep_log.js') }}"></script>
    // ... lines 59 - 65
66  {% endblock %}
```

If you're *not* a Symfony user, don't worry, the `asset()` function isn't doing anything special. Ok, let's try it! Find your browser and, refresh! Woo! It works! People, this is *amazing*! We can *finally* organize JavaScript into multiple files and not worry about "forgetting" to add all the script tags we need. The `require` function is a *game-changer*!

If you look at the compiled `rep_log.js` file, you can see a bunch of Webpack code at the top, which helps things work internally - and... below, our code! It's not minimized because this is a *development* build. We'll talk about production builds later.

Making PhpStorm Happy

If you're using PhpStorm like me, there are a few things we can do to make our life *much* more awesome. Open Preferences and search for ECMAScript. Under "Languages & Frameworks" -> "JavaScript", make sure that `ECMAScript 6` is selected.

Then, search for "Node" and find the "Node.js and NPM" section. Click to "Enable" the Node.js Core library.

And *finally*, if you're using Symfony, search for Symfony. If you don't see a Symfony section, you should *totally* download the Symfony plugin - we have some details about this in a <u>different screencast</u>. Make sure it's enabled, and - most importantly - change the web directory to `public`. This will give auto-completion on the `asset` function.

Watching for Changes

Back to Encore! There's one *big* bummer when introducing a "build" system for JavaScript like we just did: each time you change a file, you will need to re-run Encore! Lame! That's why Encore has a fancy "watch" option. Run:

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

This will build, but now it's watching for changes! Let's just add a space here and save. Yes! Encore *already* re-built the files. Stop this whenever you want with `Ctrl`+`C`.

Oh, and since this command is *long*, there's a shortcut:

```
yarn run encore dev
```

or, better... use the `--watch` flag:

```
yarn run encore dev --watch
```

Build Notifications!

Great! But... sometimes... we're going to make mistakes. Yes, I know, it's hard to imagine. Let's make a syntax error. Back at the terminal, woh! The build *failed*! But if you weren't watching the terminal closely, you might not realize this happened!

No problem! Let's enable a build notification system! In `webpack.config.js`, just add `enableBuildNotifications()`:

```
webpack.config.js
↕   // ... lines 1 - 2
 3  Encore
↕   // ... lines 4 - 11
12      .enableBuildNotifications()
13  ;
↕   // ... lines 14 - 17
```

The "watch" functionality has *one* weakness: whenever you update `webpack.config.js`, you'll need to restart Encore before it sees those changes. So... stop it and run Encore again:

```
yarn run encore dev --watch
```

Bah, error! Scroll up! Check this out, it says:

> *"Install `webpack-notifier` to use `enableBuildNotifications()`"*

And then it tells us to run a command. Cool! Encore has a *ton* of features... but to stay light, it doesn't *ship* with the all of the dependencies for these optional features. But, it's no problem: if you need to install something, Encore will tell you. Copy that command and run:

```
yarn add webpack-notifier --dev
```

Run encore again:

```
yarn run encore dev --watch
```

It works! And cool! A desktop notification. Now, make a mistake! Yes! An obvious build error. Fix it and... build successful!

Ok, we've got a pretty sweet system already. But Webpack is going to let us do *so* much more.

Chapter 3: Require Outside Libraries

When you use Webpack, the *hardest* thing is that you need to start thinking about your JavaScript differently. You need to *stop* thinking about global variables, and *start* thinking about how you can code *correctly*. It's not as easy as it sounds: we've been using global variables in JavaScript... well... forever!

For example, in `RepLogApp.js`, we created this self-executing function to give our code a little bit of isolation:

```
public/assets/js/RepLogApp.js
⇕  // ... lines 1 - 4
5  (function(window, $, Routing, swal) {
⇕  // ... lines 6 - 213
214 })(window, jQuery, Routing, swal);
```

That part isn't too important. But at the bottom, we are *relying* on there to be a *global* `jQuery` variable. It just *must* exist, or else everything will explode! On top, this becomes a `$` variable in the function.

Open the base layout file - `base.html.twig`. The *only* reason our code works is that, at the bottom, yep! We have a script tag for jQuery, which adds a global `jQuery` variable:

```
templates/base.html.twig
⇕   // ... lines 1 - 98
99  {% block javascripts %}
100     <script src="https://code.jquery.com/jquery-3.1.1.min.js"
    integrity="sha256-hVVnYaiADRTO2PzUGmuLJr8BLUSjGIZsDYGmIJLv2b8="
    crossorigin="anonymous"></script>
⇕   // ... lines 101 - 105
106 {% endblock %}
⇕   // ... lines 107 - 110
```

And this is the process we've used for *years*: add a script tag for a JS library, then reference its global variable everywhere else.

I *hate* this! In `RepLogApp.js`, I just have to *hope* that jQuery was included correctly. That's madness, and it needs to stop. So, from now on, we have a *new* philosophy: if we need a

variable in a file - like `$` - then we need to *require* it in the same way that we are requiring `Helper`.

The *only* difference is that jQuery is a third-party library. Well... in PHP, we would use Composer to install third-party libraries. And... yea! In JavaScript, we can use Yarn to do the same thing!

Installing jQuery via Yarn

Check this out: open a *third* terminal tab - we're getting greedy! Then run:

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

Yep! We can use yarn to download *front-end* libraries! Oh, and you can search for package names on `npmjs.com` or `npms.io`.

This downloads jquery into the `node_modules/` directory and adds it to `package.json`:

```
package.json
1  {
2      "devDependencies": {
   // ... line 3
4          "jquery": "^3.3.1",
   // ... line 5
6      }
7  }
```

Requiring jQuery

So... how do we require it? Oh, it's *awesome*: `const $ = require('jquery')`:

```
public/assets/js/RepLogApp.js
   // ... lines 1 - 2
3  const Helper = require('./RepLogHelper');
4  const $ = require('jquery');
   // ... lines 5 - 216
```

That's it! When a require path does *not* start with a `.`, Webpack knows to look for a *package* in `node_modules/` with that name.

And now that we are *properly* importing the `$` variable - yay us - remove `$` and `jQuery` from the self-executing function:

```
public/assets/js/RepLogApp.js
⥮  // ... lines 1 - 3
4   const $ = require('jquery');
5
6   (function(window, Routing, swal) {
⥮  // ... lines 7 - 214
215 })(window, Routing, swal);
```

Yep, when we use the `$` variable below, it is *no longer* dependent on any global jQuery variable! Responsible coding for the win!

But... does it work? Try it! Go back to our site and refresh! It does! That's because, back on the terminal, if you run:

```
ls -la public/build
```

... yep! Our `rep_log.js` file now has jQuery *inside* of it - you know because it's now 300kb! Don't worry, we'll talk about optimizations later.

But the point is this: all *we* need to do is require the libraries we need, and Webpack takes care of the rest!

Installing & Using SweetAlert2

Let's require one more outside package. Search for "swal". We're using a really cool library called SweetAlert to bring up the delete dialog. But... the *only* reason this works is that, in the template, we're including a script tag for it:

```
templates/lift/index.html.twig
     // ... lines 1 - 47
48   {% block stylesheets %}
     // ... lines 49 - 50
51       <link rel="stylesheet"
     href="https://cdn.jsdelivr.net/npm/sweetalert2@7.11.0/dist/sweetalert2.min.c
     />
52   {% endblock %}
     // ... line 53
54   {% block javascripts %}
     // ... lines 55 - 56
57       <script
     src="https://cdn.jsdelivr.net/npm/sweetalert2@7.11.0/dist/sweetalert2.all.mi
     </script>
     // ... lines 58 - 65
66   {% endblock %}
```

Boo! Let's refactor this to *require* that library properly.

If you search for this package, you'll find out that it's called `sweetalert2`. Let's install it:

```
yarn add sweetalert2@^7.11.0 --dev
```

This time, delete the script tag entirely. We can't remove the jQuery script tag yet because we're still using the global variable in a few places. But, we'll fix that soon.

Then, in `RepLogApp.js`, remove the argument from the self-executing function: that global variable doesn't even exist anymore!

```
public/assets/js/RepLogApp.js
     // ... lines 1 - 6
7    (function(window, Routing) {
     // ... lines 8 - 215
216  })(window, Routing);
```

To prove it, refresh! Awesome!

> "*swal* is not defined"

To get it back, add `const swal = require('sweetalert2');`:

```
public/assets/js/RepLogApp.js
     ↕  // ... lines 1 - 4
     5  const swal = require('sweetalert2');
     6
     7  (function(window, Routing) {
     ↕  // ... lines 8 - 215
   216  })(window, Routing);
```

As *soon* as we save this, Webpack recompiles, we refresh and... it works! Yes! We can use *any* outside library by running one command and adding one require line.

Let's use our new unstoppable skills to refactor our code into re-usable components.

Chapter 4: Component Organization

With our new-found super-power to require files, we can really start to clean things up! First, remove the self-executing function that's around everything:

```
public/assets/js/RepLogApp.js
// ... lines 1 - 6
7   let HelperInstances = new WeakMap();
8
9   class RepLogApp {
// ... lines 10 - 194
195 }
196
197 const rowTemplate = (repLog) => `
// ... lines 198 - 210
211 `;
212
213 window.RepLogApp = RepLogApp;
```

We *originally* added this because it gave our code a little bit of isolation. It helped us to, for example, avoid accidentally overriding global variables, but... now that `RepLogApp` is being processed by Webpack, it is *itself* a module! And Webpack automatically wraps it - behind the scenes - so that it's isolated. Basically, we don't need to worry about silly things like self-executing functions.

Creating a Skinny "entry" File

Next, look in the template: `index.html.twig`. We include the `rep_log.js` file... but we *also* have a *little* bit of JavaScript that is responsible for using that object and initializing it:

```
templates/lift/index.html.twig
↕   // ... lines 1 - 53
54  {% block javascripts %}
↕   // ... lines 55 - 56
57      <script src="{{ asset('build/rep_log.js') }}"></script>
58
59      <script>
60          $(document).ready(function() {
61              var $wrapper = $('.js-rep-log-table');
62              var repLogApp = new RepLogApp($wrapper);
63          });
64      </script>
65  {% endblock %}
```

This is.... kind of a bummer: it relies on the `RepLogApp` variable to be *global*... and that only works because, at the bottom, we're purposely creating a global variable with `window.RepLogApp = RepLogApp`:

```
public/assets/js/RepLogApp.js
↕    // ... lines 1 - 212
213  window.RepLogApp = RepLogApp;
```

Also, to *fully* Webpackify our app, we will eventually want to remove *all* JavaScript from our templates. Yep, you'll just include the one JS file and... that's it!

Skinny Entries

And this brings us to an important point about organization. Usually, the *entry* file - so the file that we list in `webpack.config.js`:

```
webpack.config.js
↕   // ... lines 1 - 2
3   Encore
↕   // ... lines 4 - 9
10      .addEntry('rep_log', './public/assets/js/RepLogApp.js')
↕   // ... lines 11 - 12
13  ;
↕   // ... lines 14 - 17
```

Should contain a small amount of logic that calls out to *other* modules. It's kind of like a controller in Symfony: it's *supposed* to have just a *few* lines of code that call out to *other* parts of our app.

Actually, the code in `index.html.twig` is a pretty good example of what I'd expect in an entry file:

```twig
templates/lift/index.html.twig
// ... lines 1 - 53
54  {% block javascripts %}
// ... lines 55 - 58
59      <script>
60          $(document).ready(function() {
61              var $wrapper = $('.js-rep-log-table');
62              var repLogApp = new RepLogApp($wrapper);
63          });
64      </script>
65  {% endblock %}
```

Let me show you what I mean: in the `js/` directory, create a new file: called `rep_log.js`.

Next, open `webpack.config.js`: let's use *this* as the entry file instead:

```js
webpack.config.js
// ... lines 1 - 2
3  Encore
// ... lines 4 - 9
10      .addEntry('rep_log', './public/assets/js/rep_log.js')
// ... lines 11 - 12
13  ;
// ... lines 14 - 17
```

And since I just made a change, find your terminal and restart Encore:

```
yarn run encore dev --watch
```

Copy the code from `index.html.twig`, remove it, and paste it here:

```js
public/assets/js/rep_log.js
// ... lines 1 - 5
6  $(document).ready(function() {
7      var $wrapper = $('.js-rep-log-table');
8      var repLogApp = new RepLogApp($wrapper);
9  });
```

Perfect!

And now that we are *responsible* JavaScript developers... finally... we need to require any dependencies. Oh, but first, add `'use strict';` on top - that's optional, but I like it:

```
public/assets/js/rep_log.js
1  'use strict';
↕  // ... lines 2 - 5
6  $(document).ready(function() {
7      var $wrapper = $('.js-rep-log-table');
8      var repLogApp = new RepLogApp($wrapper);
9  });
```

Now add `const $ = require('jquery')` and, to get `RepLogApp`, `const RepLogApp = require('./RepLogApp');`:

```
public/assets/js/rep_log.js
1  'use strict';
2
3  const $ = require('jquery');
4  const RepLogApp = require('./RepLogApp');
5
6  $(document).ready(function() {
7      var $wrapper = $('.js-rep-log-table');
8      var repLogApp = new RepLogApp($wrapper);
9  });
```

I love it! Does it work? Move of it and... refresh! Bah!

> "*RepLogApp* is not a constructor"

Ooof. This is a technical way of saying:

> "Hey! You're using RepLogApp like a class... but it's not!"

Open `RepLogApp.js`, and scroll to the bottom:

```
public/assets/js/RepLogApp.js
    ↕   // ... lines 1 - 8
    9   class RepLogApp {
    ↕   // ... lines 10 - 194
  195   }
  196
  197   const rowTemplate = (repLog) => `
    ↕   // ... lines 198 - 210
  211   `;
  212
  213   window.RepLogApp = RepLogApp;
```

Aha! We forgot to *export* a value from this module. Replace the global variable with
`module.exports = RepLogApp`:

```
public/assets/js/RepLogApp.js
    ↕   // ... lines 1 - 8
    9   class RepLogApp {
    ↕   // ... lines 10 - 194
  195   }
  196
  197   const rowTemplate = (repLog) => `
    ↕   // ... lines 198 - 210
  211   `;
  212
  213   module.exports = RepLogApp;
```

Try it again! It works!

You can start to see the pattern: create a small entry file and organize everything *else* into
reusable classes or functions.

Moving into a Components Directory

Let's take this a step further and organize into directories. Create a new directory in `js/` called
`Components/`. Let's move our re-usable stuff here: `RepLogApp` and `RepLogHelper`.

Build failure! Of course! In `rep_log.js`, update the path: `./Components/RepLogApp`:

```
public/assets/js/rep_log.js
    ↕   // ... lines 1 - 3
    4   const RepLogApp = require('./Components/RepLogApp');
    ↕   // ... lines 5 - 10
```

Build successful! Make sure it still works... it does!

Next! This is great! But can Encore handle apps that are *not* single-page apps? Like, what if I need a *different* JavaScript file for my login page? Encore has you covered.

Chapter 5: Multiple Pages / Entries

This is all *really* nice... but, so far... it *kinda* looks like Webpack only works for single-page apps! I mean, if this were the *only* page in our app, we could write all of our JavaScript in the one entry file, require what we need and... be done!

But even our small app has another page: `/login`. And this page has its *own* custom JavaScript.... which right now, is done in this boring, old, non-Webpack-ified `login.js` file:

```
public/assets/js/login.js
```

```
 1  'use strict';
 2
 3  (function(window, $) {
 4      $(document).ready(function() {
 5          $('.js-recommended-login').on('click', '.js-show-login',
    function(e) {
 6              e.preventDefault();
 7
 8              $('.js-recommended-login-details').toggle();
 9          });
10
11          $('.js-login-field-username').on('keydown', function(e) {
12              const $usernameInput = $(e.currentTarget);
13              // remove any existing warnings
14              $('.login-long-username-warning').remove();
15
16              if ($usernameInput.val().length >= 20) {
17                  const $warning = $('<div class="login-long-username-
    warning">This is a really long username - are you sure that is right?
    </div>');
18                  $usernameInput.before($warning);
19              }
20          });
21      });
22  })(window, jQuery);
```

So... how can we *also* process *this* file through Webpack? We *could* require it from `rep_log.js`, but that's wasteful! We *really* only need this code on the *login* page.

Multiple Entries

The answer is... so simple: add a *second* entry called `login` that will load the `assets/js/login.js` file:

```
webpack.config.js
```
```
// ... lines 1 - 2
3    Encore
// ... lines 4 - 9
10       .addEntry('rep_log', './public/assets/js/rep_log.js')
11       .addEntry('login', './public/assets/js/login.js')
// ... lines 12 - 13
14   ;
// ... lines 15 - 18
```

I want you to think of each entry as a separate *page* on your site. Or, you can think of each entry as a separate JavaScript application that runs on your site. Like, we have our main "rep log" application and also our "login" application.

Because we just changed the webpack config, go back and restart Encore:

```
yarn run encore dev --watch
```

Webpack-Sponsored Cleanup

And *now* we can improve things! First, remove that self-executing function:

```
public/assets/js/login.js
1   'use strict';
    // ... lines 2 - 4
5   $(document).ready(function() {
6       $('.js-recommended-login').on('click', '.js-show-login', function(e) {
7           e.preventDefault();
8
9           $('.js-recommended-login-details').toggle();
10      });
11
12      $('.js-login-field-username').on('keydown', function(e) {
13          const $usernameInput = $(e.currentTarget);
14          // remove any existing warnings
15          $('.login-long-username-warning').remove();
16
17          if ($usernameInput.val().length >= 20) {
18              const $warning = $('<div class="login-long-username-
    warning">This is a really long username - are you sure that is right?
    </div>');
19              $usernameInput.before($warning);
20          }
21      });
22  });
```

Then, more importantly, *require* the dependencies we need: in this case jQuery with
`const $ = require('jquery')`:

```
public/assets/js/login.js
 1  'use strict';
 2
 3  const $ = require('jquery');
 4
 5  $(document).ready(function() {
 6      $('.js-recommended-login').on('click', '.js-show-login', function(e) {
 7          e.preventDefault();
 8
 9          $('.js-recommended-login-details').toggle();
10      });
11
12      $('.js-login-field-username').on('keydown', function(e) {
13          const $usernameInput = $(e.currentTarget);
14          // remove any existing warnings
15          $('.login-long-username-warning').remove();
16
17          if ($usernameInput.val().length >= 20) {
18              const $warning = $('<div class="login-long-username-
    warning">This is a really long username - are you sure that is right?
    </div>');
19              $usernameInput.before($warning);
20          }
21      });
22  });
```

That's it! Go back and... refresh! Bah:

> "`require` is not defined"

Boo! My bad - I forgot to *use* the new built file. Open
`templates/bundles/FOSUserBundle/Security/login.html.twig`. Point the script
tag to `build/login.js`:

```
templates/bundles/FOSUserBundle/Security/login.html.twig
↕   // ... lines 1 - 10
11  {% block javascripts %}
↕   // ... lines 12 - 13
14      <script src="{{ asset('build/login.js') }}"></script>
15  {% endblock %}
↕   // ... lines 16 - 72
```

And *now*... it works! When I type a *really* long username, this message appears thanks to that
JavaScript.

The "layout" Entry

But... there's one last problem. Open the base layout file: `base.html.twig`. Yep, we *also* include one JavaScript file on *every* page:

```twig
templates/base.html.twig
// ... lines 1 - 98
99  {% block javascripts %}
// ... lines 100 - 104
105     <script src="{{ asset('assets/js/layout.js') }}"></script>
106  {% endblock %}
// ... lines 107 - 110
```

It doesn't do much... just adds a tooltip when you hover over your username:

```js
public/assets/js/layout.js
1  'use strict';
2
3  (function(window, $) {
4      $(document).ready(function() {
5          $('[data-toggle="tooltip"]').tooltip();
6      });
7  })(window, jQuery);
```

So... how do we handle this? How can we Webpackify *this* file? I mean, the layout is not its own *page*... so... can it be its own entry? The answer is... yes! Add another entry called `layout` and point it to `assets/js/layout.js`:

```js
webpack.config.js
// ... lines 1 - 2
3  Encore
// ... lines 4 - 11
12     .addEntry('layout', './public/assets/js/layout.js')
// ... lines 13 - 14
15  ;
// ... lines 16 - 19
```

Here's the deal: *usually* you will include exactly *one* script tag for a built JavaScript file on each page - like `rep_log.js` or `login.js`. But, if you have some JavaScript that should be included on *every* page, you can think of *that* JavaScript as its own, mini JS application. In that case, you'll have *two* built files per page: your layout JavaScript *and* your page-specific JavaScript... if you have any for that page.

Go back and restart Webpack so it reads the new config.

```
yarn run encore dev --watch
```

But... let's *not* refactor this file yet: we'll do that next. In `base.html.twig`, use the new file: `build/layout.js`:

```
templates/base.html.twig
↕   // ... lines 1 - 98
99  {% block javascripts %}
↕   // ... lines 100 - 104
105     <script src="{{ asset('build/layout.js') }}"></script>
106  {% endblock %}
↕   // ... lines 107 - 110
```

Boom! Try it! Refresh the page! Yes! It *still* works. Next, let's refactor `layout.js` to remove the self-executing function and require its dependencies. But this time... there's a surprise!

Chapter 6: jQuery Plugins / Bootstrap

Now that Webpack is handling `layout.js`, let's simplify it! Remove the self-executing function. And, of course, add `const $ = require('jquery')`:

```js
public/assets/js/layout.js
1  'use strict';
2
3  const $ = require('jquery');
4
5  $(document).ready(function() {
6      $('[data-toggle="tooltip"]').tooltip();
7  });
```

Perfect, right? Well... we're in for a surprise! Go back to the main page and... refresh! Bah!

> "`tooltip` is not a function"

Uh oh! The `tooltip` function comes from Bootstrap... and if you look in our base layout, yea! We *are* including jQuery and then Bootstrap:

```twig
templates/base.html.twig
1    <!DOCTYPE html>
2    <html lang="en">
     // ... lines 3 - 19
20   <body>
     // ... lines 21 - 98
99   {% block javascripts %}
100      <script src="https://code.jquery.com/jquery-3.1.1.min.js"
     integrity="sha256-hVVnYaiADRTO2PzUGmuLJr8BLUSjGIZsDYGmIJLv2b8="
     crossorigin="anonymous"></script>
101      <script
     src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
     integrity="sha384-
     Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
     crossorigin="anonymous"></script>
     // ... lines 102 - 105
106  {% endblock %}
107
108  </body>
109  </html>
```

Which *should* add this function to jQuery!

Trouble with jQuery Plugins

But be careful: this is where Webpack can get tricky! Internally, the Bootstrap JavaScript *expects* there to be a *global* `jQuery` variable that it can add the `tooltip()` function to. And there *is* a global jQuery variable! It's *this* jQuery that's included in the layout. So, Bootstrap adds `.tooltip()` to *that* jQuery object.

But, in `layout.js`, when we `require('jquery')`:

```
public/assets/js/layout.js
⤢  // ... lines 1 - 2
3  const $ = require('jquery');
⤢  // ... lines 4 - 8
```

This imports an entirely *different* jQuery object... and this one does *not* have the tooltip function!

To say this in a different way, if you look at *just* this file, we are *not* requiring bootstrap... so it should be no surprise that bootstrap hasn't been able to add its `tooltip()` function! What's the fix? Require Bootstrap!

Find your open terminal and run:

```
yarn add bootstrap@3 --dev
```

Bootstrap 4 just came out, but our app is built on Bootstrap 3. Now that it's installed, go back and add: `require('bootstrap')`:

```
public/assets/js/layout.js
⤢  // ... lines 1 - 2
3  const $ = require('jquery');
4  require('bootstrap');
⤢  // ... lines 5 - 9
```

And... that's it! Well, there *is* one strange thing... and it's really common for jQuery plugins: when you require bootstrap, it doesn't *return* anything. Nope, its whole job is to *modify* jQuery... not return something.

Now that it's fixed, go back and... refresh! What! The *same* error!!! *This* is where things get *really* interesting.

At this point, we're no longer using the global jQuery variable or Bootstrap JavaScript anywhere: *all* of our code *now* uses proper require statements. To celebrate, remove the two script tags from the base layout:

```twig
templates/base.html.twig
1   <!DOCTYPE html>
2   <html lang="en">
    // ... lines 3 - 19
20  <body>
    // ... lines 21 - 98
99  {% block javascripts %}
100     <script src="https://code.jquery.com/jquery-3.1.1.min.js"
        integrity="sha256-hVVnYaiADRTO2PzUGmuLJr8BLUSjGIZsDYGmIJLv2b8="
        crossorigin="anonymous"></script>
101     <script
        src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
        integrity="sha384-
        Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
        crossorigin="anonymous"></script>
    // ... lines 102 - 105
106 {% endblock %}
107
108 </body>
109 </html>
```

And now... refresh!

Fascinating!

> "`jQuery` is not defined"

And it's coming from inside of Bootstrap!

Ah, ha! When we require `bootstrap`, internally in that file, *it* looks for a *global* variable called `jQuery` and then modifies it. But when you *require* `jquery`, it does *not* create a global variable: it just returns a value. And now that there is *no* global `jQuery` variable available, it fails! This is a *really* common situation for jQuery plugins... and there's a great fix. Actually, there are *two* ways to fix it... but only one good one.

The *ugly* fix is to say `window.jQuery = $`:

```
public/assets/js/layout.js
    // ... lines 1 - 2
3   const $ = require('jquery');
4   window.jQuery = $;
5   require('bootstrap');
    // ... lines 6 - 10
```

Try it! Go back and refresh! All better. Yep, we just *made* a global variable... so that when we require `bootstrap`, it uses it. But... come on! We're trying to *remove* global variables from our code - not re-add them!

```
public/assets/js/layout.js
    // ... lines 1 - 2
3   const $ = require('jquery');
4   require('bootstrap');
    // ... lines 5 - 9
```

So here's the better solution: go to `webpack.config.js` and add `autoProvidejQuery()`:

```
webpack.config.js
    // ... lines 1 - 2
3   Encore
    // ... lines 4 - 14
15      // fixes modules that expect jQuery to be global
16      .autoProvidejQuery()
17  ;
    // ... lines 18 - 21
```

That's it. Find your terminal and restart Webpack:

```
yarn run encore dev --watch
```

And... refresh! Yes! It *works*! But... what the heck just happened? You've just experienced a *crazy* super power of Webpack. Thanks to `autoProvidejQuery()`, whenever Webpack finds a module that references an uninitialized global `jQuery` variable - yep, Webpack is *smart* enough to know this:

```
// node_modules/bootstrap/.../bootstrap.js


function ($) {
```

```
    // ...
} (jQuery)
```

It *rewrites* that code to `require('jquery')`:

```
// node_modules/bootstrap/.../bootstrap.js

function ($) {
    // ...
} (require('jquery'))
```

Yea... it basically *rewrites* the code so that it's written correctly! And so suddenly, Bootstrap requires the *same* `jquery` instance that *we're* using! This makes jQuery plugins work beautifully.

> 💡 **Tip**
>
> Not *all* jQuery plugins have this problem: some *do* behave properly out-fo-the-box.

Handling Legacy Template Code

Oh, but there's one other jQuery legacy situation I want to mention. If you're upgrading an existing app to Webpack, then you might not be able to move all of your JavaScript out of your templates at once. And that JavaScript *probably* needs jQuery. Here's my recommendation: remove jQuery from the base layout like we've already done. But then, in your `layout.js` file, require `jquery` and add: `global.$ = $`.

```
// ...
const $ = require('jquery');
global.$ = $;
require('bootstrap');
// ...
```

This `global` variable is special to Webpack - well... it's technically a Node thing, but that's not important. The point is, when you do this, it creates a *global* `$` variable, which means that any JavaScript in your templates will be able to use it - as long as you make sure your code is included *after* your `layout.js` script tag.

Later, you should *totally* remove this when your code is refactored. But, it's a nice helper for upgrading.

Next, let's talk about how CSS fits into all of this!

Chapter 7: Require CSS!?

Oh, *before* we talk about CSS, I forgot to mention that these `public/build` files do *not* need to be committed to your repository: we can rebuild them whenever we want from the source files. So inside `.gitignore`, add `/public/build/*` to make sure we don't commit them:

```
.gitignore
// ... line 1
2   /public/build/*
// ... lines 3 - 19
```

Ok, onto something more important! Go to `/login`. Thanks to some JavaScript, if you type a really long username, a message pops up. The styling for this message comes from `login.css`. This is included in the template: `login.html.twig`:

```
templates/bundles/FOSUserBundle/Security/login.html.twig
// ... lines 1 - 4
5   {% block stylesheets %}
// ... lines 6 - 7
8       <link rel="stylesheet" href="{{ asset('assets/css/login.css') }}" />
9   {% endblock %}
10
11  {% block javascripts %}
// ... lines 12 - 13
14      <script src="{{ asset('build/login.js') }}"></script>
15  {% endblock %}
// ... lines 16 - 72
```

This makes sense: we have a script tag for `login.js` and also a link tag for `login.css`. But remember: I want you to start thinking about your JavaScript as an *application*... an application that can *require* its dependencies. And... isn't this CSS *really* a dependency of our app? I mean, if we forgot to include the CSS on this page, the application wouldn't *break* exactly... but it would look horrible! Honestly, it would look like I designed it.

What I'm saying is: wouldn't it be *cool* if we could *require* CSS from right inside our JavaScript?

Requiring CSS

Whelp... surprise! We can! Inside `login.js`, add `require('../css/login.css')`:

```
public/assets/js/login.js
// ... lines 1 - 2
3  const $ = require('jquery');
4  require('../css/login.css');
// ... lines 5 - 24
```

We don't need to assign this to any variable.

So... what the heck does that do? Does that somehow magically embed that CSS onto the page? Well, it's not *that* magic. Look inside the `build/` directory - you may need to right click and select "Synchronize" to update it. Woh! Suddenly, there is a new `login.css` file... which contains all of the stuff from our *source* `login.css`:

```css
public/assets/css/login.css
 1  .wrapper {
 2      margin-top: 80px;
 3      margin-bottom: 20px;
 4  }
 5
 6  .form-signin {
 7    max-width: 420px;
 8    padding: 30px 38px 66px;
 9    margin: 0 auto;
10    background-color: #eee;
11    border: 3px dotted rgba(0,0,0,0.1);
12    }
13
14  .form-signin-heading {
15    text-align:center;
16    margin-bottom: 30px;
17  }
18
19  .form-control {
20    position: relative;
21    font-size: 16px;
22    height: auto;
23    padding: 10px;
24  }
25
26  input[type="text"] {
27    margin-bottom: 0px;
28    border-bottom-left-radius: 0;
29    border-bottom-right-radius: 0;
30  }
31
32  input[type="password"] {
33    margin-bottom: 20px;
34    border-top-left-radius: 0;
35    border-top-right-radius: 0;
36  }
37
38  .colorgraph {
39    height: 7px;
40    border-top: 0;
41    background: #c4e17f;
42    border-radius: 5px;
43    background-image: -webkit-linear-gradient(left, #c4e17f, #c4e17f 12.5%,
    #f7fdca 12.5%, #f7fdca 25%, #fecf71 25%, #fecf71 37.5%, #f0776c 37.5%,
    #f0776c 50%, #db9dbe 50%, #db9dbe 62.5%, #c49cde 62.5%, #c49cde 75%,
    #669ae1 75%, #669ae1 87.5%, #62c2e4 87.5%, #62c2e4);
```

```css
44    background-image: -moz-linear-gradient(left, #c4e17f, #c4e17f 12.5%,
      #f7fdca 12.5%, #f7fdca 25%, #fecf71 25%, #fecf71 37.5%, #f0776c 37.5%,
      #f0776c 50%, #db9dbe 50%, #db9dbe 62.5%, #c49cde 62.5%, #c49cde 75%,
      #669ae1 75%, #669ae1 87.5%, #62c2e4 87.5%, #62c2e4);
45    background-image: -o-linear-gradient(left, #c4e17f, #c4e17f 12.5%,
      #f7fdca 12.5%, #f7fdca 25%, #fecf71 25%, #fecf71 37.5%, #f0776c 37.5%,
      #f0776c 50%, #db9dbe 50%, #db9dbe 62.5%, #c49cde 62.5%, #c49cde 75%,
      #669ae1 75%, #669ae1 87.5%, #62c2e4 87.5%, #62c2e4);
46    background-image: linear-gradient(to right, #c4e17f, #c4e17f 12.5%,
      #f7fdca 12.5%, #f7fdca 25%, #fecf71 25%, #fecf71 37.5%, #f0776c 37.5%,
      #f0776c 50%, #db9dbe 50%, #db9dbe 62.5%, #c49cde 62.5%, #c49cde 75%,
      #669ae1 75%, #669ae1 87.5%, #62c2e4 87.5%, #62c2e4);
47  }
48
49  .login-long-username-warning {
50    color: #8a6d3b;
51    background-color: #fcf8e3;
52    padding: 15px;
53    margin-bottom: 10px;
54    border: 1px solid #faebcc;
55    border-radius:4px;
56  }
```

Here's what's happening: we point Webpack at our login entry file: `login.js`:

webpack.config.js
```
// ... lines 1 - 2
3  Encore
// ... lines 4 - 10
11      .addEntry('login', './public/assets/js/login.js')
// ... lines 12 - 16
17  ;
// ... lines 18 - 21
```

Then, it parses *all* of the require statements inside:

public/assets/js/login.js
```
// ... lines 1 - 2
3  const $ = require('jquery');
4  require('../css/login.css');
// ... lines 5 - 24
```

For any required JS files, it puts them in the final `login.js`. But it *also* parses the CSS files... and puts any CSS it finds into a final file - called `login.css`.

Actually, it's a bit confusing: the name `login.css` comes from the name of the *entry*: `login`:

```
webpack.config.js
⬍   // ... lines 1 - 2
3   Encore
⬍   // ... lines 4 - 10
11      .addEntry('login', './public/assets/js/login.js')
⬍   // ... lines 12 - 16
17  ;
⬍   // ... lines 18 - 21
```

Yep, each entry will cause one JavaScript file to be built *and* - if any of that JavaScript requires a CSS file - then it will *also* cause a CSS file to be created with the same name.

Of course, to use this in the template, we *still* need *one* link tag pointed to `build/login.css`:

```
templates/bundles/FOSUserBundle/Security/login.html.twig
⬍   // ... lines 1 - 4
5   {% block stylesheets %}
⬍   // ... lines 6 - 7
8      <link rel="stylesheet" href="{{ asset('build/login.css') }}" />
9   {% endblock %}
⬍   // ... lines 10 - 72
```

Let's try it - refresh! If you type a long name... it *works*! And... bonus time! When we talk about creating a *production* build later, this CSS file will automatically be minified.

Requiring the Layout CSS

So let's do this *everywhere*. Open `layout.js` and also the base layout: `base.html.twig`. Look at the top: we have a *few* css files, the first is `main.css`:

```twig
templates/base.html.twig
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
↕   // ... lines 4 - 10
11      {% block stylesheets %}
↕   // ... lines 12 - 13
14          <link href="{{ asset('assets/css/main.css') }}" rel="stylesheet"
    />
15      {% endblock %}
↕   // ... lines 16 - 17
18  </head>
↕   // ... lines 19 - 106
107 </html>
```

In `layout.js`, require this: `../css/main.css`:

```js
public/assets/js/layout.js
↕   // ... lines 1 - 3
4   require('bootstrap');
5   require('../css/main.css');
↕   // ... lines 6 - 10
```

As *soon* as we hit save, synchronize the build directory again... Yes! We have a *new* `layout.css` file! In `base.html.twig`, update the `link` tag to use this:

```twig
templates/base.html.twig
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
↕   // ... lines 4 - 10
11      {% block stylesheets %}
↕   // ... lines 12 - 13
14          <link href="{{ asset('build/layout.css') }}" rel="stylesheet" />
15      {% endblock %}
↕   // ... lines 16 - 17
18  </head>
↕   // ... lines 19 - 106
107 </html>
```

Yep... everything still looks *fine*.

Handling Images

But... wait! Something amazing just happened! Look inside `main.css`. Woh! We're referencing a background image: `../images/dumbell-mini.png`:

```
public/assets/css/main.css
     // ... lines 1 - 7
  8  .mini-dumbbell {
     // ... lines 9 - 10
 11      background: url('../images/dumbbell-mini.png') center center  no-
         repeat;
     // ... line 12
 13  }
     // ... lines 14 - 78
```

That's a problem! Why? Because the final file lives in a completely *different* directory, so that `../` path will *break*!

Actually... it's *not* a problem! Webpack is amazing! It parses our CSS looking for any background images or fonts. When it finds one, it moves *it* into a `build/images/` directory and rewrites the path inside the final CSS file to point there.

> 💡 **Tip**
>
> The `file-loader` has `esModule: true` by default since v5.0.0. If the generated URL looks like `[object Module]` - you will need to set `esModule` to `false`:
>
> ```
> // webpack.config.js
>
> Encore
>     // ...
>     .configureUrlLoader({
>         images: {
>             esModule: false
>         }
>     })
>     // ...
> ```

The point is: all *we* need to do is write our CSS files correctly and... well... Webpack takes care of the rest!

Requiring Bootstrap & FontAwesome CSS

We're on a roll! There are *two* CSS files left in `base.html.twig`: Bootstrap and FontAwesome:

templates/base.html.twig

```twig
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
    // ... lines 4 - 10
11      {% block stylesheets %}
12          <link
    href="https://netdna.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
    rel="stylesheet" />
13          <link href="https://maxcdn.bootstrapcdn.com/font-
    awesome/4.7.0/css/font-awesome.min.css" rel="stylesheet"
    integrity="sha384-
    wvfXpqpZZVQGK6TAh5PVlGOfQNHSoD2xbE+QkPxCAFlNEevoEH3Sl0sibVcOQVnN"
    crossorigin="anonymous" />
    // ... line 14
15      {% endblock %}
    // ... lines 16 - 17
18  </head>
    // ... lines 19 - 106
107 </html>
```

You know the drill: require this! Remove the Bootstrap link tag first:

templates/base.html.twig

```twig
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
    // ... lines 4 - 10
11      {% block stylesheets %}
12          <link
    href="https://netdna.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
    rel="stylesheet" />
    // ... lines 13 - 14
15      {% endblock %}
    // ... lines 16 - 17
18  </head>
    // ... lines 19 - 106
107 </html>
```

In `layout.js`, *above* `main.css`, so that our CSS overrides their stuff, add `require()`... um... require... what? If we just `require('bootstrap')`, that will require the *JavaScript* file!

So... how can we include CSS files? Look in the `node_modules/` directory... and scroll down to find `bootstrap/`. Ah, ok. Inside, there is a `dist/` directory, then `css/` and `bootstrap.css`.

A *little* bit of explanation: when you require the name of a module, Node reads a special key in that package's `package.json` file called `main` to figure out *which* file to *actually* require. But, if you want to require a specific file... just do it: `bootstrap/dist/css/bootstrap.css`:

```
public/assets/js/layout.js
     // ... lines 1 - 3
4    require('bootstrap');
5    require('bootstrap/dist/css/bootstrap.css');
6    require('../css/main.css');
     // ... lines 7 - 11
```

This time, we don't need to make *any* other changes to `base.html.twig`: we already have a `link` tag for `layout.css`, which has everything we need:

```
templates/base.html.twig
1    <!DOCTYPE html>
2    <html lang="en">
3    <head>
     // ... lines 4 - 10
11       {% block stylesheets %}
     // ... line 12
13           <link href="{{ asset('build/layout.css') }}" rel="stylesheet" />
14       {% endblock %}
     // ... lines 15 - 16
17   </head>
     // ... lines 18 - 105
106  </html>
```

To prove it, go back and refresh! It's still beautiful!

Yep, the built `layout.css` *now* has Bootstrap inside. And actually, Bootstrap itself references some fonts... and hey! There are now *fonts* in the `build/` directory too! Those are handled *just* like background images.

Ok: *one* more CSS file to remove: FontAwesome. It's getting easy now! Remove the `link` tag from the layout:

```twig
templates/base.html.twig
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
⇕   // ... lines 4 - 10
11      {% block stylesheets %}
12          <link href="https://maxcdn.bootstrapcdn.com/font-
        awesome/4.7.0/css/font-awesome.min.css" rel="stylesheet"
        integrity="sha384-
        wvfXpqpZZVQGK6TAh5PVlGOfQNHSoD2xbE+QkPxCAFlNEevoEH3Sl0sibVcOQVnN"
        crossorigin="anonymous" />
⇕   // ... line 13
14      {% endblock %}
⇕   // ... lines 15 - 16
17  </head>
⇕   // ... lines 18 - 105
106 </html>
```

Then, install that library:

```
yarn add font-awesome@4 --dev
```

I added `@4` to make sure we get the version compatible with *this* project. Oh, and how did I know to use `font-awesome` as the exact library name? I cheated: I already used npms.io before recording to find it.

Back in `layout.js`, require `font-awesome`. Oh, but we need to find the *exact* file... in `node_modules/font-awesome`... ah! It looks like `css/font-awesome.css` - add that to the require:

```js
public/assets/js/layout.js
⇕   // ... lines 1 - 3
4   require('bootstrap');
5   require('bootstrap/dist/css/bootstrap.css');
6   require('font-awesome/css/font-awesome.css');
7   require('../css/main.css');
⇕   // ... lines 8 - 12
```

And Webpack is happy! Try it! Find the site and refresh! We still have our Bootstrap CSS and... yes! Our little user icon from FontAwesome is there! And on the homepage... yep! Those trash icons are from FontAwesome too!

Now, our `base.html.twig` file looks great! We have one CSS file and one JS file:

```twig
templates/base.html.twig
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
    // ... lines 4 - 10
11      {% block stylesheets %}
12          <link href="{{ asset('build/layout.css') }}" rel="stylesheet" />
13      {% endblock %}
    // ... lines 14 - 15
16  </head>
17
18  <body>
    // ... lines 19 - 96
97  {% block javascripts %}
    // ... lines 98 - 100
101     <script src="{{ asset('build/layout.js') }}"></script>
102  {% endblock %}
103
104  </body>
105  </html>
```

And *all* our dependencies are being required internally.

Chapter 8: Handling Images with the CopyPlugin

Bonus! A *really* cool side-effect of using Webpack is that *none* of these files in the `assets/` directory need to be public anymore! I mean they *live* in the public directory currently... but the user *never* needs to access them directly: Webpack processes and moves them into `build/`.

To celebrate, let's move `assets/` *out* of the `public/` directory and into the *root* of our project. We don't *need* to do this... but if something doesn't need to be publicly accessible, why make it public?

This change breaks almost *nothing*. The *only* things we need to update are the paths in `webpack.config.js`:

```
webpack.config.js
↕   // ... lines 1 - 2
3   Encore
↕   // ... lines 4 - 9
10      .addEntry('rep_log', './assets/js/rep_log.js')
11      .addEntry('login', './assets/js/login.js')
12      .addEntry('layout', './assets/js/layout.js')
↕   // ... lines 13 - 16
17  ;
↕   // ... lines 18 - 21
```

After making that change, restart Encore!

```
yarn run encore dev --watch
```

And... refresh! Woohoo! Wait... there's a missing image! Bah! I was lying! There *is* one file that *still* needs to be publicly accessible!

Open `index.html.twig`... ah! We have a good, old-fashioned `img` tag that references one of the images in the `assets/` directory:

```
templates/lift/index.html.twig
      // ... lines 1 - 2
 3    {% block body %}
 4        <div class="row">
      // ... lines 5 - 34
35            <div class="col-md-5">
36                <div class="leaderboard">
37                    <h2 class="text-center">
38                        <img class="dumbbell" src="{{
      asset('assets/images/dumbbell.png') }}" />
      // ... line 39
40                    </h2>
      // ... lines 41 - 42
43                </div>
44            </div>
45        </div>
46    {% endblock %}
      // ... lines 47 - 59
```

And... whoops! It's not public anymore. My bad!

This is one of the *few* cases - maybe the *only* case - where we need to reference public images from *outside* a file that Webpack processes. The simple problem is that Webpack doesn't know that it needs to move this file!

Of course, there's an easy fix: we could just move this *one* file back into the `public/` directory. But... that sucks: I'd *rather* keep all of my assets in one place.

Installing copy-webpack-plugin

> **💡 Tip**
>
> Great news! The latest version of Encore has a `copyFiles()`! You can use this instead of installing this plugin.

To do this, we can take advantage of a Webpack *plugin* that can copy the file for us. Google for `copy-webpack-plugin` to find its GitHub page. Encore gives you *a lot* of features... but it doesn't give you *everything*. But... no worries! We're using Webpack under-the-hood. So if you find a Webpack plugin you want, you can totally use it!

Side note, Encore *will* have a `copy()` method soon. Then you'll be able to do this without a plugin. Yay! But, this is still a *great* example of how to extend Webpack beyond Encore.

Anyways, install the plugin first. Notice that they use `npm`. I'm going to use `yarn`. So copy the name of that plugin, find your terminal, and run:

```
yarn add copy-webpack-plugin --dev
```

Adding Custom Webpack Config

To use the plugin, we need to require it at the top of the Webpack config file. No problem:

```
webpack.config.js
1  var Encore = require('@symfony/webpack-encore');
2  const CopyWebpackPlugin = require('copy-webpack-plugin');
   // ... lines 3 - 27
```

And then below, um.... `config =`... and `plugins:`... what the heck does this mean?

Well... earlier, I told you that `webpack.config.js` *normally* returns a big configuration object. And Encore is just a tool to help *generate* that config. In fact, at the bottom, we can *see* what that config looks like if we want! Just `console.log(module.exports)`.

Then, restart Encore:

```
yarn run encore dev --watch
```

Woh! There's our config! Actually, it's not so scary: there are keys for `entry`, `output`, `module`, `plugins` and a few other things.

For example, see the `plugins` key? Back on their docs, *that* is what they're referring to: they want you to add *their* plugin to that config key.

Ok, so how can we do that? Well, you could always just add it manually: `module.exports.plugins.push()` and then the plugin. Yep: you could literally add

something to the plugins array!

But, fortunately, Encore gives you an easier way to modify the most common things. In this case, use `addPlugin()` and then `new CopyWebpackPlugin()`. Pass this an array - this will soon be the paths it should copy:

```
webpack.config.js
↕   // ... line 1
2   const CopyWebpackPlugin = require('copy-webpack-plugin');
3
4   Encore
↕   // ... lines 5 - 18
19      .addPlugin(new CopyWebpackPlugin([
↕   // ... lines 20 - 21
22      ]))
23   ;
↕   // ... lines 24 - 27
```

Copying Images into build/

But, before we fill that in... let's think about this. I don't need to copy *all* of my images to the `build/` directory... just *one* of them right now. So let's create a *new* directory called `static/` and move any files that need to be copied into *that* directory, like `dumbell.png`.

In the `CopyWebpackPlugin` config, set `from` to `./assets/static` and `to` to just `static`:

```
webpack.config.js
↕   // ... line 1
2   const CopyWebpackPlugin = require('copy-webpack-plugin');
3
4   Encore
↕   // ... lines 5 - 18
19      .addPlugin(new CopyWebpackPlugin([
20          // copies to {output}/static
21          { from: './assets/static', to: 'static' }
22      ]))
23   ;
↕   // ... lines 24 - 27
```

This will copy to the output directory `/static`.

Ok, go restart Encore!

```
yarn run encore dev --watch
```

Once the build finishes... inside `public/build`... yes! We have a new `static` directory. It's nothing fancy, but this is a nice way to move files so that we can reference them publicly in a template:

```twig
templates/lift/index.html.twig
// ... lines 1 - 2
3  {% block body %}
4      <div class="row">
// ... lines 5 - 34
35          <div class="col-md-5">
36              <div class="leaderboard">
37                  <h2 class="text-center">
38                      <img class="dumbbell" src="{{
   asset('build/static/dumbbell.png') }}" />
// ... line 39
40                  </h2>
// ... lines 41 - 42
43              </div>
44          </div>
45      </div>
46  {% endblock %}
// ... lines 47 - 59
```

There's one *more* reference in the login template: search for "bell" and... update this one too:

```twig
templates/bundles/FOSUserBundle/Security/login.html.twig
// ... lines 1 - 16
17  {% block fos_user_content %}
18      <div class="container">
19          <div class="wrapper">
20              <form action="{{ path("fos_user_security_check") }}"
   method="post" class="form-signin">
21                  <h3><img class="dumbbell" src="{{
   asset('build/static/dumbbell.png') }}">Login! Start Lifting!</h3>
// ... lines 22 - 67
68              </form>
69          </div>
70      </div>
71  {% endblock fos_user_content %}
```

Try it! Find your browser and refresh. There it is!

Next, let's make our CSS sassier... with... Sass of course!

Chapter 9: Sass & Sourcemaps

Our `layout.js` file requires `main.css`:

```
assets/js/layout.js
↕   // ... lines 1 - 6
7   require('../css/main.css');
↕   // ... lines 8 - 12
```

That's cool... I *guess*... if you like using boring old CSS. But I want to be more Hipster, so let's use Sass instead. Well, I could use Stylus to be *super* hipster... and Encore *does* support that, but let's use something a bit more familiar.

To start, rename the file to `main.scss`. Now, we can use a fancier syntax for these pseudo-selectors:

```
assets/css/main.scss
↕    // ... lines 1 - 16
17   .btn-login {
↕        // ... lines 18 - 26
27       &:hover,&:focus {
28           color: #fff;
29           background-color: #53A3CD;
30           border-color: #53A3CD;
31       }
32   }
↕    // ... lines 33 - 77
```

OooooOoOooOooo.

Obviously, the build is failing because, in `layout.js`, that file path is wrong! Change it to `main.scss`:

```
assets/js/layout.js
↕   // ... lines 1 - 6
7   require('../css/main.scss');
↕   // ... lines 8 - 12
```

So... does it already work?

Activating Optional Features

No! On the watch tab of our terminal, it *failed* when loading `main.scss`. Out-of-the-box, Encore *cannot* process Sass files. But, it tells you how to fix this! We just need to enable it inside our config & install some extra packages. Remember: Encore is *full* of features. But to stay light, it doesn't enable *everything* automatically. Instead, *you* are in control: enable what you need, and Encore will tell you what to do. It's kinda cool!

Go back to `webpack.config.js` and add `.enableSassLoader()`:

```
webpack.config.js
↕   // ... lines 1 - 3
4   Encore
↕   // ... lines 5 - 23
24      .enableSassLoader()
25  ;
↕   // ... lines 26 - 29
```

Then, back on your terminal, copy the `yarn add` line, stop Encore with `Ctrl`+`C`, and paste!

```
● ● ●

  yarn add sass-loader node-sass --dev
```

Let it do its thing, then... restart Encore!

```
● ● ●

  yarn run encore dev --watch
```

No errors! To prove it works, move over to your browser and... refresh! It still looks great! Well, *most* importantly, on the login page, when we hover of the button, it *does* have that styling.

Encore Versus Webpack Concepts

There's one thing I want you to notice: the *name* of this method: `enableSassLoader()`:

```
webpack.config.js
  ↕    // ... lines 1 - 3
  4    Encore
  ↕    // ... lines 5 - 23
 24        .enableSassLoader()
 25    ;
  ↕    // ... lines 26 - 29
```

"Loader" is a *Webpack* concept. Encore tries to make Webpack as *easy* as possible, but it *reuses* Webpack's language and terms whenever possible. And that's important! If you ever need to do something custom with Webpack, it's usually pretty easy to figure out how that fits into Encore.

Also, we're requiring `bootstrap.css` right now:

```
assets/js/layout.js
  ↕    // ... lines 1 - 4
  5    require('bootstrap/dist/css/bootstrap.css');
  ↕    // ... lines 6 - 12
```

But, with Sass support, you could instead import Bootstrap's Sass files *directly*. The advantage is that you can override Bootstrap's Sass variables and take control of colors, sizes and other stuff. To do that with Bootstrap 3, you'll need the `bootstrap-sass` package. For Bootstrap 4, the Sass files are included in the main package.

Sourcemaps!

Let's fix *one* more problem quickly: sourcemaps! If you click on a row, we have some `console.log()` debugging code. But, where does that code come from? Well, if you click on the `rep_log.js` link, apparently it's coming from line 197. But, that's a lie! Well, sort of. This is the *built* `rep_log.js` file, not the *source* file.

And *this* highlights a classic problem: when you build many files into one file, debugging gets harder because error messages and other info don't point to the *real* line number or the original filename.

Let's fix that! Back in `webpack.config.js`, add `.enableSourceMaps()` with an argument: `!Encore.isProduction()`:

```
webpack.config.js
  ↕  // ... lines 1 - 3
  4  Encore
  ↕  // ... lines 5 - 24
 25      .enableSourceMaps(!Encore.isProduction())
 26  ;
  ↕  // ... lines 27 - 30
```

This enables extra debugging info - called sourcemaps - whenever we are creating a
development build.

Because we just updated the Webpack config, restart Encore:

```
yarn run encore dev --watch
```

Thanks to this, all of our JavaScript *and* CSS files now have some extra content at the bottom
that hints to our browser where the source content came from. This time, when I click a row, in
the console, awesome! It's coming from `RepLogApp.js` line 104. That is the *real* spot.

Oh, by the way: if you *don't* enable sourcemaps, you may still see *some* sourcemap info at the
bottom of your CSS files during development. That's just an internal quirk - it won't be there on
production.

Chapter 10: Integrating FOSJsRoutingBundle

Open `Components/RepLogApp.js` and search for `Routing`:

```
assets/js/Components/RepLogApp.js
↕    // ... lines 1 - 8
9    class RepLogApp {
↕    // ... lines 10 - 43
44       loadRepLogs() {
45           $.ajax({
46               url: Routing.generate('rep_log_list'),
↕    // ... lines 47 - 50
51           })
52       }
↕    // ... lines 53 - 194
195  }
↕    // ... lines 196 - 214
```

Guess what? *This* `Routing` variable is a *global* variable. Boo! It's our *last* one. In `templates/`, open the base layout:

```
templates/base.html.twig
↕    // ... lines 1 - 96
97   {% block javascripts %}
98       <script src="{{ asset('bundles/fosjsrouting/js/router.js') }}">
     </script>
99       <script src="{{ path('fos_js_routing_js', { callback:
     'fos.Router.setData' }) }}"></script>
↕    // ... lines 100 - 101
102  {% endblock %}
↕    // ... lines 103 - 106
```

Other than a polyfill - which we won't talk about - there are only *two* script tags left. These give us the `Router` variable and they come from FOSJsRoutingBundle: a *really* cool bundle that allows you to generate URLs from Symfony routes in JavaScript.

Our goal is clear: refactor our code so that we can *require* the Router instead of relying on the global variable.

Requiring the router.js File

The *first* interesting thing is that this is *not* a Node package. Nope, it's a normal PHP package that *happens* to have a JavaScript file inside. But, that doesn't really make any difference... except that the *path* for it is ugly: it lives in
`vendor/friendsofsymfony/jsrouting-bundle/Resources/public/js/router.js`.
Wow! Ok then: `const Routing = require()`, then go up a few directories and follow the path:
`vendor/friendsofsymfony/jsrouting-bundle/Resources/public/js/router.js`:

```
assets/js/Components/RepLogApp.js
↕  // ... lines 1 - 5
6  const Routing = require('../../../vendor/friendsofsymfony/jsrouting-
   bundle/Resources/public/js/router.min.js');
↕  // ... lines 7 - 215
```

Simple enough! Let's try it! In your browser, refresh! Bah! Error!

> *"The route `rep_log_list` does not exist"*

Booo! This error comes from *inside* the Router. Here's what's going on: this JavaScript library is more complex than most. The *first* script tag gives us the Router variable. But the *second* executes a dynamic endpoint that fetches a JSON list of the route information and then *sets* that on the router.

When we simply require the router... we *do* get the Router object... but it has no routes! So the question is: how can we get the dynamic route info so that it can be set into the router?

Actually, this *is* possible! If you look at the Usage section of the bundle's docs, it talks about how to integrate with Webpack Encore. Basically, by running a `bin/console` command, you can *dump* your route information to a static JSON file. Then, you can *require* that JSON from Webpack and set it on the Router. Oh, and don't worry about this `import` syntax - it's basically the same as `require()`, and we'll talk about it next.

So this is really cool! It shows how you can even require JSON files from JavaScript! But... it has a downside: each time you add a new route, you need to re-run the command. That can be a pain during development. It's still a *great* option - and is a bit faster on production - but it *does* have that weakness.

Creating the Fake Router Module

And there *is* another option. It's not quite as fancy or awesome... but it's easier. Inside `assets/js/Components`, create a new file called `Routing.js`. Inside, um, just say, `module.exports = window.Routing`:

```
assets/js/Components/Routing.js
1  /**
2   * For now, we rely on the router.js script tag to be included
3   * in the layout. This is just a helper module to get that object.
4   */
5  module.exports = window.Routing;
```

Yep! We *are* going to continue using the global variable. But *now*, we can *at least* require *this* file from everywhere else so that our code *looks* more responsible: `const Routing = require('./Routing')`:

```
assets/js/Components/RepLogApp.js
↕  // ... lines 1 - 5
6  const Routing = require('./Routing');
↕  // ... lines 7 - 215
```

And now, when we refresh, it works. The *cool* thing about this hacky solution is that if you want to change to the better solution later, it's easy! Just put the correct code in `Router.js`, and everything will already be using it. Nice!

Chapter 11: ES6 Import & Export

If you watched <u>episode 2</u> of our JavaScript series, then you know that ECMAScript is the official name of the JavaScript language standard and that ECMAScript version 6 - or ES6 - introduced the idea of *modules*. Modules are what we've been taking advantage of this *entire* tutorial: exporting and requiring values from different files.

But... surprise! In ECMAScript, the `require()` function does *not* exist. Whaaaat?! The `require()` statement was basically invented by Node, back *before* ES6: Node needed a way to require files, so they invented their *own* way. Later, ECMAScript decided to make the *idea* of modules part of the standard language. But when they did, they used a *different* keyword than require! Yep, there are *two* valid syntaxes for working with modules! But... it's not a big deal: they work *exactly* the same, except that the official syntax has *one* small advantage.

Hello import & export

Let's use the *official* module syntax. Instead of saying `const Helper = require()`, say `import Helper from`:

```
assets/js/Components/RepLogApp.js
    // ... lines 1 - 2
 3  import Helper from './RepLogHelper';
    // ... lines 4 - 215
```

It's that simple! And it doesn't change *anything*. In `RepLogHelper`, we *also* need to change our *export* to use the new syntax. Instead of `module.exports = Helper`, use `export default Helper`:

```
assets/js/Components/RepLogHelper.js
    // ... lines 1 - 33
34  export default Helper;
```

We'll talk about what the `default` part means later. But for now, it's *always* export *default* and then what you want to export.

You *can* mix the two syntaxes - `require` and `import` - to a certain point, but you may run into some problems. Your best bet is to pick your favorite - mine is `import` and `export` - and use it *everywhere*. So let's update everything: `import $ from 'jquery'`, `import swal from 'sweetalert2'` and `import Routing from './Routing'`:

```
assets/js/Components/RepLogApp.js
⬍   // ... lines 1 - 2
3   import Helper from './RepLogHelper';
4   import $ from 'jquery';
5   import swal from 'sweetalert2';
6   import Routing from './Routing';
⬍   // ... lines 7 - 215
```

At the bottom, use `export default RepLogApp`:

```
assets/js/Components/RepLogApp.js
⬍   // ... lines 1 - 213
214  export default RepLogApp;
```

Cool! `RepLogHelper` is already ok, and in `Routing.js`, change this to: `export default window.Routing`:

```
assets/js/Components/Routing.js
⬍   // ... lines 1 - 4
5   export default window.Routing;
```

Keep going for the 3 entry files: `import $ from 'jquery'`:

```
assets/js/layout.js
⬍   // ... lines 1 - 2
3   import $ from 'jquery';
⬍   // ... lines 4 - 12
```

If you don't need a return value, it's even easier: just `import 'bootstrap'`. Repeat that for the CSS files:

```
assets/js/layout.js
⬍   // ... lines 1 - 2
3   import $ from 'jquery';
4   import 'bootstrap';
5   import 'bootstrap/dist/css/bootstrap.css';
6   import 'font-awesome/css/font-awesome.css';
7   import '../css/main.scss';
⬍   // ... lines 8 - 12
```

In `login.js`, import jQuery again, then import the CSS file:

```
assets/js/login.js
   // ... lines 1 - 2
3  import $ from 'jquery';
4  import '../css/login.css';
   // ... lines 5 - 24
```

And *one* more time in `rep_log.js`: import jQuery and `import RepLogApp`:

```
assets/js/rep_log.js
   // ... lines 1 - 2
3  import $ from 'jquery';
4  import RepLogApp from './Components/RepLogApp';
   // ... lines 5 - 10
```

And... assuming I didn't mess anything up, our build *should* still be happy! Check out the terminal: yes! No errors. Move over to your browser and check it! Looks great!

Importing Named Modules

And... yea! That's it! Just two nearly-identical syntaxes... because... more is better?! The *biggest* reason I want you to know about `import` and `export` is so that you know what it *means* when you see it in code or documentation.

But, there is *one* small advantage to `import` and `export`, and it relates to this `default` keyword:

```
assets/js/Components/RepLogApp.js
     // ... lines 1 - 213
214  export default RepLogApp;
```

Usually, you'll want to export just *one* value from a module. In that case, you say export `default` and then you receive this value when using `import`.

But... technically... you can export *multiple* things from a module, as long as you give each of them a *name*. For example, instead of `export default Helper`, we could export an object with a `Helper` key and a `foo` key:

```
import {Helper, foo} from './RepLogHelper';
```

Then, the import has a slightly different syntax where you say explicitly *which* of those keys you want to import.

I don't usually do this in my code, but there *is* one case where it can be helpful. Imagine you're using a huge external library - like `lodash` - which is really just a collection of independent functions. *If* that library exports its values correctly, you could import *just* the functions you need, instead of importing the *entire* exported value:

```
import isEqual from 'lodash.isequal';
```

Then, at least in theory, thanks to a feature called "tree shaking", Webpack would *realize* that you're only using a *few* parts of that library, and *only* include those in the final, compiled file. In reality, this *still* seems a bit buggy: the unused code doesn't always get removed. But, the point is this: `import` and `export` have a subtle advantage and are the ECMAScript standard. So, use them!

Ok, it's time to find out how we can create a crazy-fast production build!

Chapter 12: Building for Production

I *love* our new setup! So it's time to talk about optimizing our build files for production. Yep, it's time to get serious, and make sure our files are minified and optimized to kick some performance butt!

Because, right now, if you check out the size of the build directory:

```
ls -la public/build
```

... yea! These files are pretty huge - `rep_log.js` is over 1 megabyte and so is `layout.js`! If you looked inside, you would find the problem *immediately*:

```
assets/js/login.js
// ... lines 1 - 2
3  import $ from 'jquery';
// ... lines 4 - 24
```

jQuery is packaged individually inside *each* of these! That's super wasteful! Our users should *only* need to download jQuery *one* time.

The Shared Entry

> 💡 **Tip**
>
> The `createdSharedEntry()` feature still works great, but in the latest version of Encore, there is a new way to solve this problem called `splitChunks()`. Read about it here:
> https://symfony.com/doc/current/frontend/encore/split-chunks.html

No problem! Webpack has an *awesome* solution. Open `webpack.config.js`. Move the layout entry to the top - though, order doesn't matter. Now, change the method to `createSharedEntry()`:

```
webpack.config.js
⇕    // ... lines 1 - 3
4    Encore
⇕    // ... lines 5 - 10
11       .createSharedEntry('layout', './assets/js/layout.js')
⇕    // ... lines 12 - 25
26   ;
⇕    // ... lines 27 - 30
```

Before we talk about this, move back to your terminal and restart Encore:

```
yarn run encore dev --watch
```

Then, I'll open a *new* tab - I love tabs! - and, when it finishes, check the file sizes again:

```
ls -la public/build
```

Woh! `rep_log.js` is down from 1 megabyte to 300kb! `layout.js` is still big because it *does* still contain jQuery. But `login.js` - which was almost 800kb is now... 4!

What is this *magical* shared entry!? To *slightly* over-simplify it, each project should have exactly *one* shared entry. And its JS file and CSS file should be included on *every* page.

When you set `layout.js` as a shared entry, any modules included in `layout.js` are *not* repeated in other files. For example, when Webpack sees that `jquery` is required by `login.js`, it says:

> *"Hold on! `jquery` is already included in `layout.js` - the shared entry. So, I don't need to also put it in `login.js`."*

It's a *great* solution to the duplication problem: if you have a library that is commonly used, just make sure that you import it in `layout.js`, even if you don't need it there. You can experiment with the right balance.

The manifest.js File

As *soon* as you do this, if you refresh, it works! I'm kidding - you'll totally get an error:

> "`webpackJsonp` *is not defined*"

To fix that, in your base layout, *right* before `layout.js`, add one more script tag. Point it to a new `build/manifest.js` file:

```twig
templates/base.html.twig
// ... lines 1 - 96
97  {% block javascripts %}
// ... lines 98 - 100
101     <script src="{{ asset('build/manifest.js') }}"></script>
102     <script src="{{ asset('build/layout.js') }}"></script>
103  {% endblock %}
// ... lines 104 - 107
```

The *reason* we need to do this is... well.. a bit technical. But basically, this helps with long-term caching, because it allows your giant `layout.js` file to *change* less often between deploys.

Production Build

Ok, this is *great*, but the files are still *pretty* big because they're *not* being minified. How can we tell Encore to do that? In your terminal, run:

```
yarn run encore production
```

That's it! This will take a bit longer: there's more magic happening behind the scenes. When it finishes, go back to your first open tab and run:

```
ls -la public/build
```

Let's check out the file sizes! The development `rep_log.js` that *was* 310kb is down to 74! Layout went from about 1Mb to 125kb. The CSS files are also way smaller. Yep, building for production is just *one* command: Encore handles all the details.

Adding Shortcut scripts

Oh, and here's a trick to be even *lazier*. Open `package.json`. I'm going to paste a new `script` section:

```json
package.json
1  {
2      "devDependencies": {
   // ... lines 3 - 11
12     },
13     "scripts": {
14         "dev-server": "encore dev-server",
15         "dev": "encore dev",
16         "watch": "encore dev --watch",
17         "build": "encore production"
18     }
19 }
```

This gives you different shortcut commands for the different ways that you'll run Encore. Oh, we didn't talk about the `dev-server`, but it's *another* option for local development.

Anyways, *now*, in the terminal, we can just say:

```
yarn watch
```

Or any of the other script commands - like `yarn build` for production.

How to Deploy

Talking about production, there's *one* last *big* question we need to answer: how the heck do you *deploy* your assets to production? Do we need to install Node on the production server?

The answer is.... it depends. It depends on how sophisticated your deployment system is. Honestly, if you have a *very* simple deploy system - like a simple script, or maybe even some commands you run manually - then the easiest option is to install Node and yarn on your server and run `encore production` *on* your server after pulling down the latest files.

I know: this isn't a *great* solution: it's a bummer to install Node *just* for this reason. But, it *is* a valid option and *totally* simple.

A *better* solution is to run Encore on a *different* machine and then send the final, built files to your server. This highlights an important point: after you execute Encore, 100% of the files you need live in `public/build`. So, for example, after you execute:

```
yarn run encore production
```

you could send the `public/build` directory to your production machine and it would work perfectly. If you have a "build" server, that's a *great* place to run this command. Or, if you watched our <u>Ansistrano Tutorial</u>, you could run Encore locally, and use the `copy` module to deploy those files.

If you have any questions on your specific situation, you can ask us in the comments.

Chapter 13: Asset Versioning & Cache Busting

There is *one* last thing I want to talk about, and it's one of my *favorite* features in Encore. Here's the question: how can we *version* our assets? Or, even *more* simple, how can we bust browser cache? For example, right now, if I change something in `RepLogApp.js`, then of course Webpack will create an updated `rep_log.js` file. But, when an existing user comes back to our site, their browser might use the old, *cached* version! Lame!

Enabling Versioning

This is a *classic* problem. But with Encore, we can solve it beautifully and automatically! In `webpack.config.js`, first add `.cleanupOutputBeforeBuild()`:

```
webpack.config.js
⬍  // ... lines 1 - 3
4   Encore
⬍  // ... lines 5 - 25
26     .cleanupOutputBeforeBuild()
⬍  // ... line 27
28  ;
⬍  // ... lines 29 - 32
```

That's a nice little function that will empty the `public/build` directory whenever you run Encore. Then, here's the key: `.enableVersioning()`:

```
webpack.config.js
⬍  // ... lines 1 - 3
4   Encore
⬍  // ... lines 5 - 25
26     .cleanupOutputBeforeBuild()
27     .enableVersioning()
28  ;
⬍  // ... lines 29 - 32
```

That's it! Because we just changed our config, restart Encore:

```
yarn watch
```

Now look at the `build/` directory. Woh! Suddenly, all of our files have a *hash* in the filename! The hash is based on the file's *contents*: so whenever the file changes, it gets a new filename. This is *awesome*! *Now* when `rep_log.js` changes, it will have a new filename. And when we deploy to production, the user's browser will see the *new* filename and load it, instead of using the old, cached version.

Versioned Filenamed with manifest.json

Perfect! Except... we just broke *everything*. Find your browser and refresh. Yep! It's horrible! And this makes sense: in the base layout, our `script` tag simply points to `build/layout.js`:

```twig
templates/base.html.twig
// ... lines 1 - 96
97  {% block javascripts %}
// ... lines 98 - 101
102     <script src="{{ asset('build/layout.js') }}"></script>
103 {% endblock %}
// ... lines 104 - 107
```

But this is *not* the filename anymore - it's missing the hash part!

Of course, we *could* type the filename manually here. But, gross! Then, *every* time we updated a file, we would need to update its script tag.

Here's the *key* to fix this. Behind the scenes, as *soon* as we started using Encore, it generated a `manifest.json` file automatically. This is a map from the *source* filename to the current *hashed* filename! That's great! If we could somehow tell Symfony's `asset()` function to read this and make the transformation, then, well... everything would work perfectly!

And... yea! That feature exists! Open `config/packages/framework.yaml`. Anywhere, but I'll do it at the bottom, add `assets:` then `json_manifest_path` set to `%kernel.project_dir%/public/build/manifest.json`:

```
config/packages/framework.yaml
  1  framework:
  ↕  // ... lines 2 - 35
  36      assets:
  37          json_manifest_path:
      '%kernel.project_dir%/public/build/manifest.json'
```

This is a built-in feature that tells Symfony to look for a JSON file at this path, and to use it to lookup the *real* filename. In other words... just, refresh! Yea, everything is beautiful again! Check out the page source: it's using the *hashed* filename from the manifest file.

And if you change one of the files - like `layout.js`: add a `console.log()`... as *soon* as we do this, Webpack rebuilds. In the `build/` directory - you might need to synchronize it, but yes! It creates a *new* filename. When you refresh, the system automatically uses that inside the source.

Long-Lived Expires Headers

This is *free* asset versioning and cache busting! If you want to get *really* crazy, you can also now give your site a performance boost! To do that, you'll need to configure your web server to set long-lived `Expires` header on any files in the `/build` directory.

Basically, by setting an `Expires` header, your web server can instruct the browser of each user to cache any downloaded assets... forever! Then, when the user continues browsing your site, it will load faster because their browser *knows* it's safe to use these files from cache. And of course, when we *do* make a change to a file in the future, the browser *will* download it thanks to its new filename.

The exact config is different in Nginx versus Apache, but it's a common thing to add. Google for "Nginx expires header for directory".

OK guys, I hope, hope, hope you love Webpack Encore as much as I do! It has even more features that we didn't talk about, like `enableReactPreset()` to build React apps or `enableVueLoader()` for Vue.js. And we're adding new features all the time so that it's easier to use front-end frameworks and enjoy some of the really amazing things that are coming from the JavaScript world... without needing to read 100 blog posts every day.

So get out there and write amazing JavaScript! And I hope you'll stay with us for our next tutorial about React.js & Symfony!

All right guys, seeya next time!