Webpack From Zero to Hero

Chapter 3: Everything is a Module

Robert Fiolteks-Sanchez Constantino · Follow
May 23 · 5 min read

Fauzi Mahomi — Picture under the Creative Commons Licence (Attribution)

This article is part of the **Webpack from Zero to Hero series**, for more background or for the index you can check the Chapter 0: History.

Previous - Chapter 2: Tidying Up Webpack

Next - **Chapter 4: Dynamic Imports and Code Splitting (coming soon)**

Introduction

Right now we achieved a really simple application using Webpack and ES2015+ transpilation, but real apps are more than that, they have CSS, Images, custom Fonts and all sorts of different assets. The good thing is, in your code Webpack will treat them **equally as ES Modules**, so keep how the proper loaders for them.

Webpack & Buzz and Woody is, you, thinking on the config

But calm down, we started with **keeping Webpack config simple**, and we're going to stick to that! We're going to learn how to merge loaders, combine and pass options to them.

Importing Memes… or just any Images

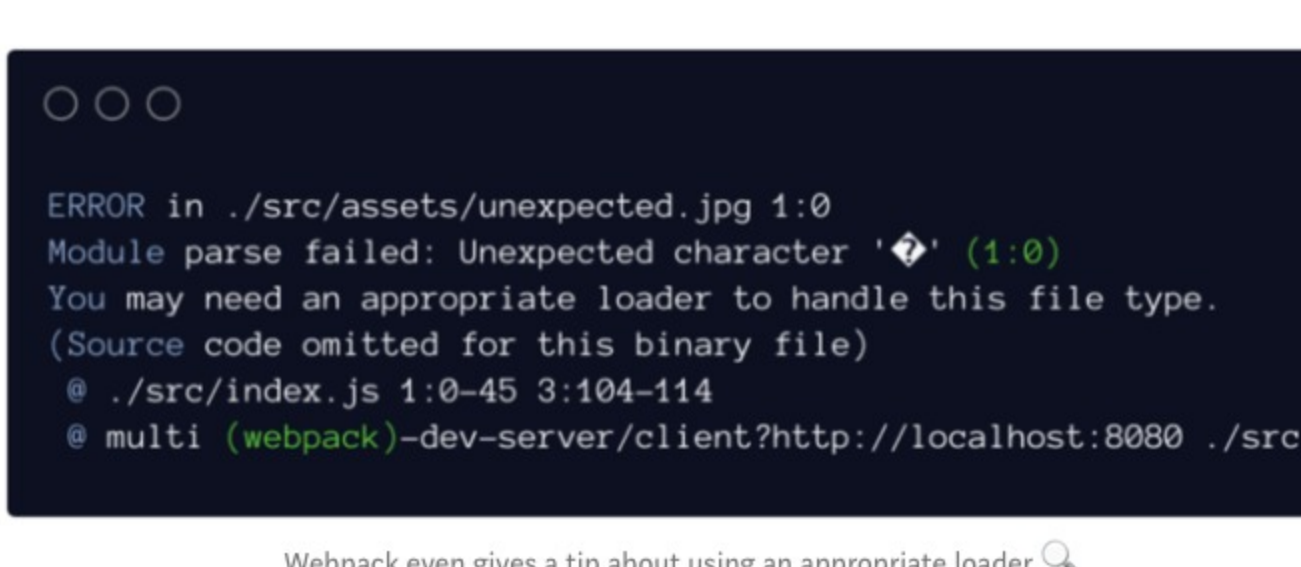
Now is the time to load assets. So, **create a directory called assets** in the root of the project and save any image you like there, and then you can follow the examples.

File loader

The **file-loader** treats your image as any asset and gives it a url path when importing it:

```
[illegible]
```

If you run this without installing the file loader first though, Webpack will complain:



Webpack just gives a tip about using an appropriate loader

Let's install it then and add to our configuration file:

```
yarn add file-loader --dev
```

Append this snippet to your `webpack.config.js` in the module rules section:

```
[illegible]
```

Translating the **regex**: we are searching for any file ending with **.gif**, **.png**, **.jpg / .jpeg** or **.svg**, all being case insensitive (the `/i` flag in the end).

Now if you run it again, webpack will successfully execute the build.

Image Webpack Loader

Using **image-webpack-loader** + file-loader will provide us with some nice optimisation to our image files. It's also easy to add:

```
yarn add image-webpack-loader --dev
```

And add the rule after the `file-loader` one:

```
[illegible]
```

It has some image compressors already enabled by default:

- mozjpeg — Compress JPEG images
- optipng — Compress PNG images
- pngquant — Compress PNG images
- svgo — Compress SVG images
- gifsicle — Compress GIF images

Media (Audio and Video)

Similar to images, we can set a rule for video and audio to be processed by the **file-loader**.

Just add the rule to the `webpack.config.js` in the config.rules section:

```
[illegible]
```

🤔 — "Wait there's already a rule for file loader, how this is going to work?"

That's true, but we can add many config rules for different kinds of files. We're just are adding the media files together with the rule that matches images because, on the last section, we changed it with the webpack image-loader and it will probably break when reading an audio/video file.

Now you can import audio files, same as you do with images (**file-loader** will behave the same, providing the source url for you):

```
import ambientNoise from './assets/ocean-noise.mp3';
```

Let's play with that and add the following to the end of the `index.js` file:

```
[illegible]
```

You can use any audio file, I'm using this one. Just beware, the audio is quite loud 🔊. Don't play it at work!

Adding Some Style with Sass

"Styles in Sweets," now let's go pass for real

Let's add Sass support to our app! Same as before, we need the proper loaders and the **sass parser**:

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

🤔 — "Whawhawhaaaaat? Why so many dependencies?"

Each one of this dependencies has a role to play:

- *sass-loader + node-sass*: receive your Sass files and output CSS
- *css-loader*: turns it into a JS module
- *style-loader*: gets the CSS module and inserts it into the page inside a `<style>` tag.

Adding that to the config:

```
[illegible]
```

Now let's play around. Create a `styles.scss` file and paste this code into it:

```
[illegible]
```

Now add an import in `index.js` file:

```
[illegible]
```

Production Styles

Each import you do will create a new `<style></style>` entry in the header of the file. This can quickly go out of control, so you may want just to join all the files together, minify and provide as a single optimized file. For this we have a solution, which is the mini-css-extract-plugin.

To install a simple:

```
yarn add mini-css-extract-plugin --dev
```

Let's add it to our production builds, in the `webpack.config.js`, on top of it add the require call:

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
```

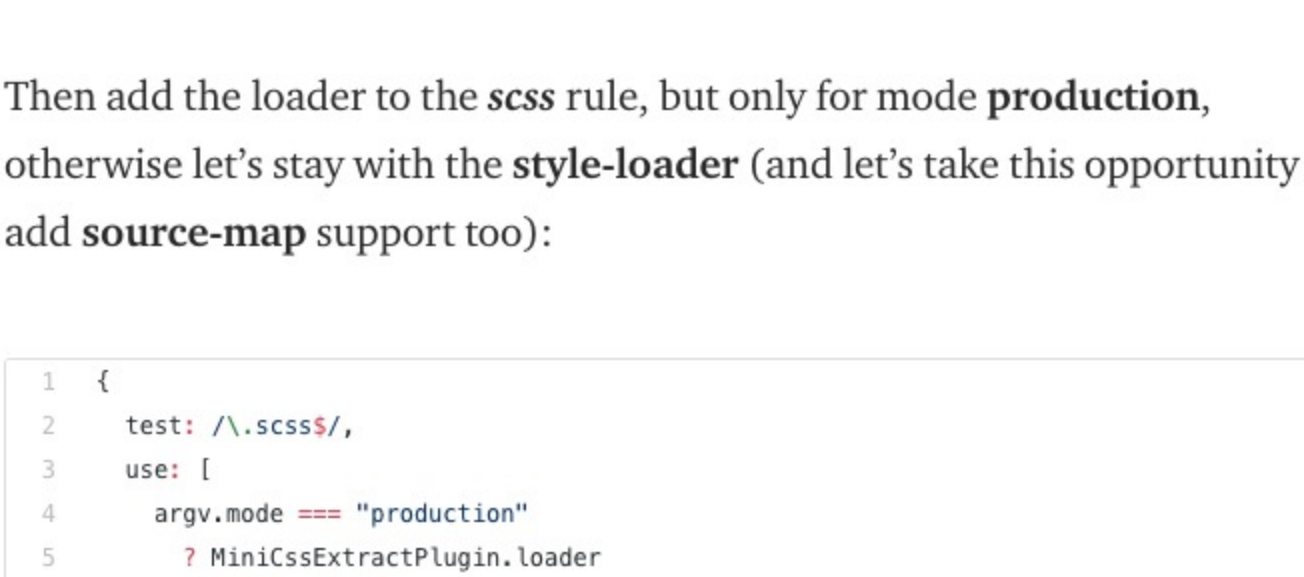
Then add the loader to the **scss** rule, but only for **production**, otherwise let's stay with the **style-loader** (and let's take this opportunity to add **source-map** support too):

```
[illegible]
```

And finally, we do the same for the **plugins** section:

```
[illegible]
```

Running the build, we'll see the style bundle being output:



The last one is generated with the build that includes styles

Keeping the Config Clean

As you see, the config already started to grow. But the advantage is, it's a Node.js file and we can split it into functions!

```
[illegible]
```

And require it in `webpack.config.js`:

```
[illegible]
```

Now you can see that is way cleaner, and you know where the module rules are because the filename follows the same name of the config options.

Remember what we learned in Chapter 2, **a clean config sparks joy** ✨

Basically Webpack loaders and plugins mostly follow the format we saw in this chapter. There are many loaders out there, some from the Webpack core team, some created by the open source community ❤️ and even one that's developed by me, webpack-chrome-extension-reloader (check it out 😉).

Let's roll here and go deeper in the next chapter. We will learn not only how to setup dynamic imports on our application, but play around with all Webpack code splitting strategies. See you in the next chapter!

JavaScript · Webpack · Technology · Nodejs · Frontend