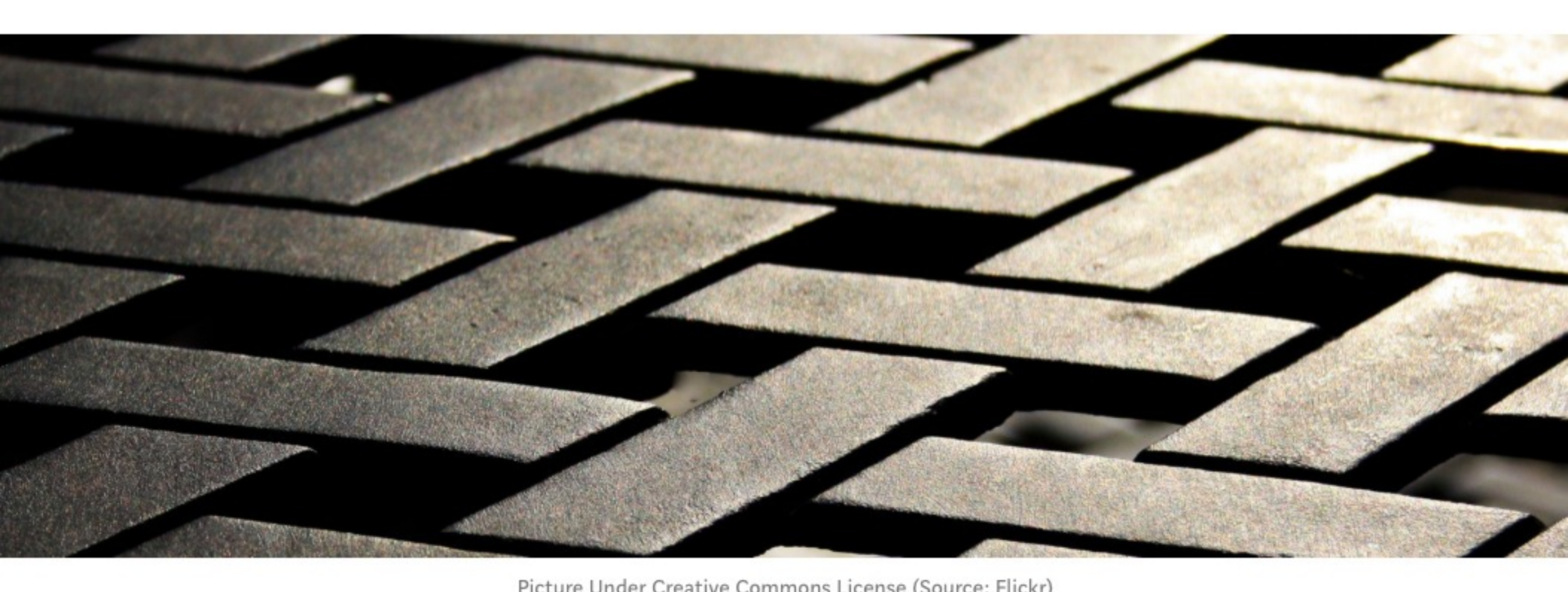
Webpack From Zero to Hero

Chapter 2: Tidying Up Webpack

Rubens Pinheiro Gonçalves Cavalcante Follow
May 6 · 5 min read



Picture Under Creative Commons License (Source: Flickr)

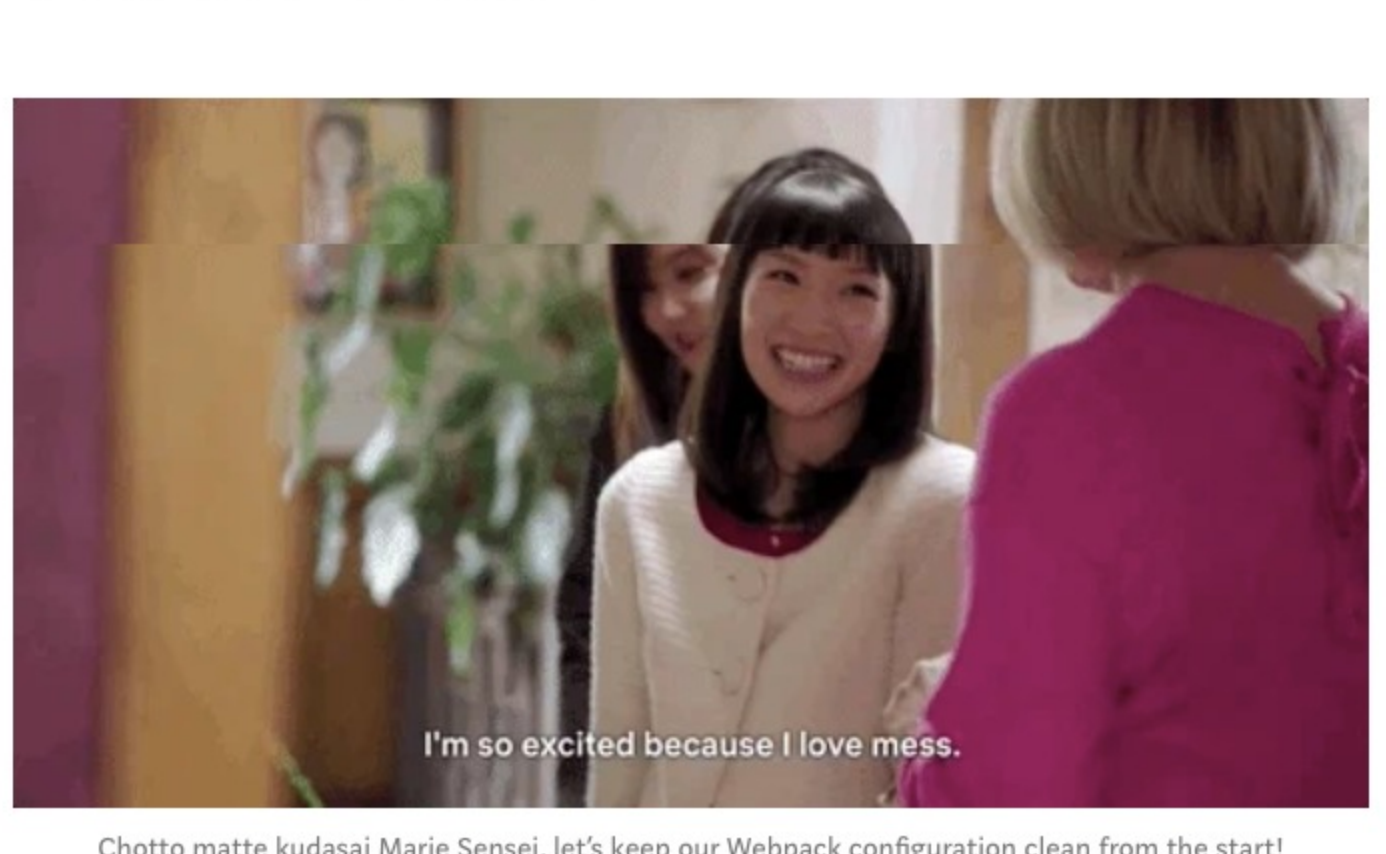
This article is part of the **Webpack from Zero to Hero series**, for more background or for the index, check the "Chapter 0: History".

Previous - Chapter 1: Getting Started with the Basics

Next - Chapter 3: Everything is a Module

Introduction

Last chapter we saw the need for BabelJS to transpile our code, and the way to make Webpack pass files to other parsers like BabelJS, which is through loaders. But until now, we were running Webpack with no **configuration** file. In this chapter we're going to create our very first Webpack **configuration,** learn how to use a **loader** and set up our local **development server**. Let's start!



Chotto matte kudasai Marie Sensei, let's keep our Webpack configuration clean from the start!

Babel requirements

First we need to install the dependencies:

```
yarn add @babel/core @babel/preset-env babel-loader --dev
```

- Babel Core: it has all logic necessary for transformations and also some polyfills;
- Babel Preset Env: it is able to choose the right transformations/polyfills depending on the target browser list;
- Babel Loader: it will be responsible for receiving the input file from Webpack and passing it through BabelJS.

Configuration Files

Babel

First let's setup the Babel to use the preset-env. Create a file called *.babelrc* with this content:

```
{
  "presets": ["@babel/preset-env"]
}
```

.babelrc hosted with ❤ by GitHub view raw

And set a **browser list range** on *package.json*:

```
"browserslist": [
  "last 2 versions",
  "not dead"
]
```

browserslist.json hosted with ❤ by GitHub view raw

> *Note: I'm creating a pretty generic query here. For production apps,* ***always*** *check analytics to properly choose your target browsers!*

Let's see how many browsers will be targeted with this query:

```
npx browserslist
```

As we don't want to install *browserslist* just for a single run, we will directly use it through **npx.** The output will be (*from the time of this article publication*):

```
and_chr 70
and_ff 63
and_qq 1.2
and_uc 11.8
android 67
android 4.4.3-4.4.4
baidu 7.12
chrome 71
chrome 70
edge 18
edge 17
firefox 64
firefox 63
ie 11
ie_mob 11
ios_saf 12.0-12.1
ios_saf 11.3-11.4
op_mini all
op_mob 46
opera 57
opera 56
safari 12
safari 11.1
samsung 7.2
samsung 6.2
```

So one of the baselines for transpiling/polyfilling will probably be Internet Explorer 11 (and its mobile version). As I said before, don't go for queries which are too generic, instead build the list based on usage data from your target audience.

Webpack

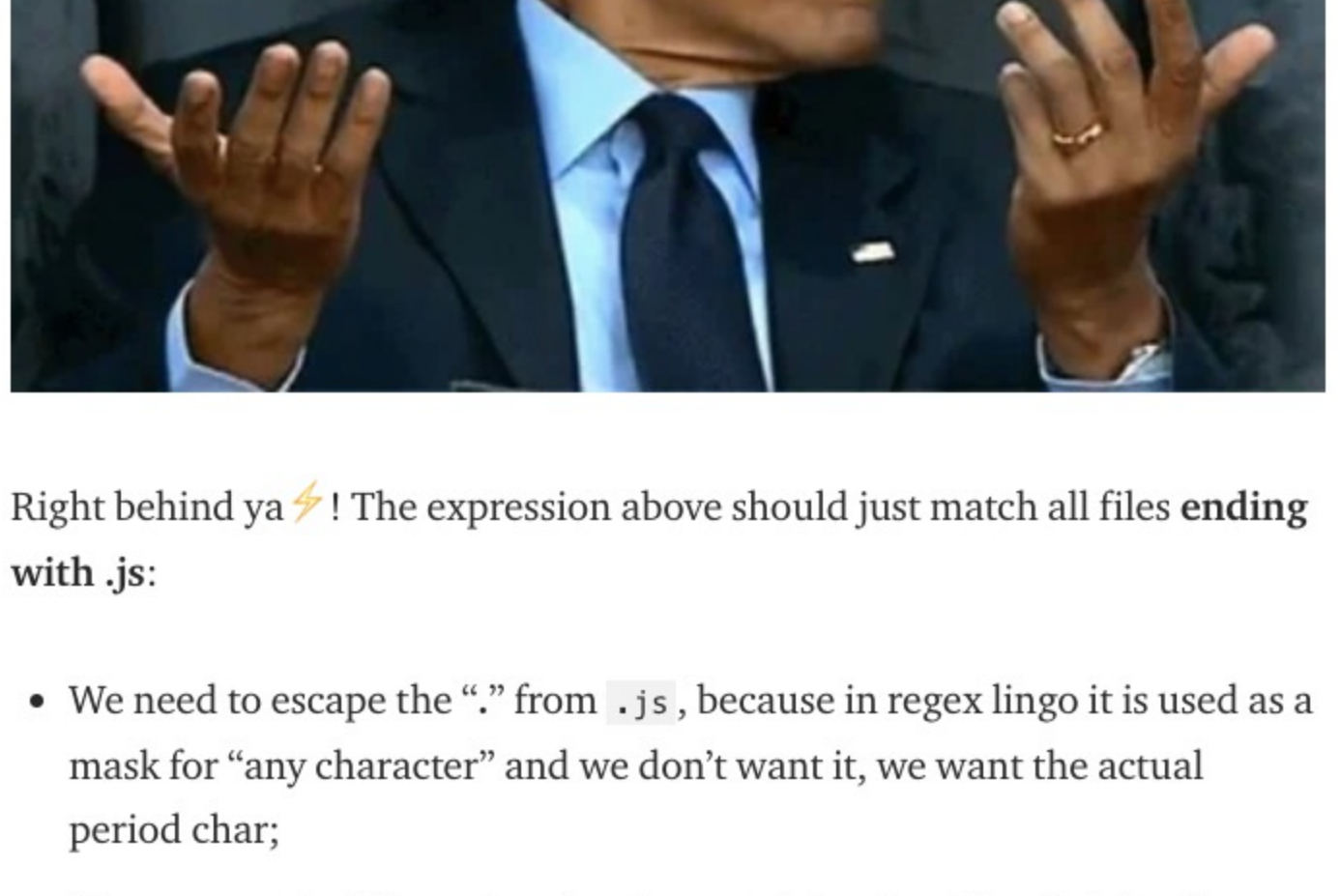
Now we just need to "**tell Webpack**" that all JS files should pass through Babel. Let's create a *webpack.config.js* file on the project root directory and add this code:

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        use: "babel-loader"
      }
    ]
  }
};
```

webpack.config.js hosted with ❤ by GitHub view raw

Webpack config is just a **NodeJS module,** exporting the configuration object.

> 😬 *"Hey, I don't understand regular expressions, can you explain that?"*



Right behind ya⚡! The expression above should just match all files **ending with .js**:

- We need to escape the "**.**" from `.js`, because in regex lingo it is used as a mask for "any character" and we don't want it, we want the actual period char;
- Then we set the "**$**", stating that the matching should end right after `.js`, so we don't mismatch things like `.json`.
- Be happy, you're a **regex** master now 💪!

A Clean Config Sparks Joy ✨

Some will say to **put all** babel and browserslist configuration inside the Webpack configuration, but in my point of view, both Babel and browserslist configuration tend to **stay the same (size-wise)**, while Webpack config tends **to grow**, so the key to keep it organized is to make it as modular as possible. Like with any normal source code (remember that the Webpack configuration is a Node.js file!), if you see it's growing, assuming too many responsibilities and repeating the code, you should break it up!

Development Environment

For any app/site development, we need to create a **dev** environment, where we can test and see the updates right away. And since we haven't seen our actual code running in the browser yet, I think is time already, let's go and do it!

Webpack Development Server

As you may know or have heard about, Webpack has a pretty nice tool called **webpack-dev-server**, where you can simulate an HTTP server on your machine integrated with a **hot module reloading** feature. It is pretty nice as the browser reloads every time a compilation is triggered, and you don't need to be reloading the page manually every time you do a change to your code.

Installation

We will install both the webpack dev server and the plugin to generate an `index.html` for us:

```
yarn add webpack-dev-server html-webpack-plugin --dev
```

Setup

On the webpack config we add the plugin to the "plugins" section:

```
const HtmlWebpackPlugin = require("html-webpack-plugin"); // first import ...

module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        use: "babel-loader"
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin() // ... then register it
  ]
};
```

webpack.cofig.js hosted with ❤ by GitHub view raw

Tip of the day: If you don't want to output an `index.html` on the **production** builds, we can skip it by checking the webpack *argv.mode*:

```
// To prevent argv being undefined, let's use a default value
module.exports = (env={}, argv={}) => ({
  // ...
  plugins: [
    // Any option given to Webpack client can be captured on the "argv"
    argv.mode === "development" ? new HtmlWebpackPlugin() : null
  ].filter(
    // To remove any possibility of "null" values inside the plugins array, we filter it
    plugin => !!plugin
  )
});
```

webpack.config.js hosted with ❤ by GitHub view raw

Some explanation for the code above:

- Webpack accepts both an *Object* or a *Function* as configuration. When you provide it as a function, it will inject the *env* and the *argv* as parameters:
- *env*: everything the client (webpack-cli) receives under the env param comes as an **env** object property, e.g.:

```
--env.test or --env.customValue="Hello there!"
```

- *argv*: all the arguments given to webpack config that are part of the configuration schema, e.g.:

```
--mode=production
```

As we're starting simple there's no need yet to create two configuration files, one for development and another for production, so let's stick to simplicity.

Now is time to run the server, which accepts the same arguments that the webpack client does (plus some additional ones). Let's remove the "build:dev" in **package.json** and change to:

```
"scripts": {
  "build": "webpack --mode=production",
  "start:dev": "webpack-dev-server --mode=development"
},
```

npm-scripts.json hosted with ❤ by GitHub view raw

Let's test it!

```
yarn start:dev
```

And you're going to see something like this:

```
ℹ ｢wds｣: Project is running at http://localhost:8080/
ℹ ｢wds｣: webpack output is served from /
ℹ ｢wdm｣: Hash: 8f593c3929cadaad9562
Version: webpack 4.29.0
Time: 1846ms
Built at: 2019-01-22 11:43:34
      Asset       Size  Chunks             Chunk Names
 index.html  180 bytes          [emitted]
    main.js    384 KiB    main  [emitted]  main
Entrypoint main = main.js
[0] multi (webpack)-dev-server/client?http://localhost:8080 ./src 40 bytes {main} [built]
[../../node_modules/ansi-html/index.js] /my-project/node_modules/ansi-html/index.js 4.26 KiB {main} [built]
[../../node_modules/ansi-regex/index.js] /my-project/node_modules/ansi-regex/index.js 136 bytes {main} [built]
[../../node_modules/events/events.js] /my-project/node_modules/events/events.js 12.7 KiB {main} [built]
[../../node_modules/html-entities/index.js] /my-project/node_modules/html-entities/index.js 230 bytes {main} [built]
[../../node_modules/loglevel/lib/loglevel.js] /my-project/node_modules/loglevel/lib/loglevel.js 6.84 KiB {main} [built]
[../../node_modules/strip-ansi/index.js] /my-project/node_modules/strip-ansi/index.js 162 bytes {main} [built]
[../../node_modules/url/url.js] /my-project/node_modules/url/url.js 22.2 KiB {main} [built]
[../../node_modules/webpack-dev-server/client/index.js?http://localhost:8080] (webpack)-dev-server/client?http://localhost:8080
7.79 KiB {main} [built]
[../../node_modules/webpack-dev-server/client/overlay.js] (webpack)-dev-server/client/overlay.js 3.58 KiB {main} [built]
[../../node_modules/webpack-dev-server/client/socket.js] (webpack)-dev-server/client/socket.js 1.05 KiB {main} [built]
[../../node_modules/webpack/hot sync ^\.\/log$] (webpack)/hot sync nonrecursive ^\.\/log$ 170 bytes {main} [built]
[../../node_modules/webpack/hot/emitter.js] (webpack)/hot/emitter.js 75 bytes {main} [built]
[./src/hello.js] 100 bytes {main} [built]
[./src/index.js] 51 bytes {main} [built]
    + 12 hidden modules
Child html-webpack-plugin for "index.html":
     1 asset
    Entrypoint undefined = index.html
    [../../node_modules/html-webpack-plugin/lib/loader.js!../../node_modules/html-webpack-plugin/default_index.ejs] /my-
project/node_modules/html-webpack-plugin/lib/loader.js!/my-project/node_modules/html-webpack-plugin/default_index.ejs 376 bytes
{0} [built]
    [../../node_modules/lodash/lodash.js] /my-project/node_modules/lodash/lodash.js 527 KiB {0} [built]
    [../../node_modules/webpack/buildin/global.js] (webpack)/buildin/global.js 475 bytes {0} [built]
    [../../node_modules/webpack/buildin/module.js] (webpack)/buildin/module.js 546 bytes {0} [built]
ℹ ｢wdm｣: Compiled successfully.
```

Now open the page at http://localhost:8080/, open the dev tools on the console tab, and you'll see this:

```
Hello OLX Dev!!
```

Source Maps

If you click on the link right after the *console.log* result on the dev tools console tab, you're going to be forwarded to the sources panel, and you're going to see something interesting:

```
__webpack_require__.r(__webpack_exports__);
/* harmony export (binding) */ __webpack_require__.d(__webpack_exports__, "hello", function() { return hello; });
var hello = function hello(subject) {
  return console.log("Hello ".concat(subject, "!"));
};
```

This is the **transpiled code** by Babel. But how can I check my **actual** code? Enter the *source maps!* ⚡

Source maps are something that will map your actual source to the final bundled source, letting you use breakpoints and see the actual code lines on stack traces in case of exceptions. To enable them, just add this to `webpack.config.js`:

```
devtool: "source-map",
```

Stop and run the dev server again and check the source on the console tab link, and this time you're going to see the actual source code!

· · ·

Now that we have everything up and running on our development server, we have paved a way to add more loaders and parse all kinds of files. But let's read about that in the **next chapter** - see you there!

JavaScript Webpack Nodejs Technology Frontend

335 claps