***Learn Webpack with me***

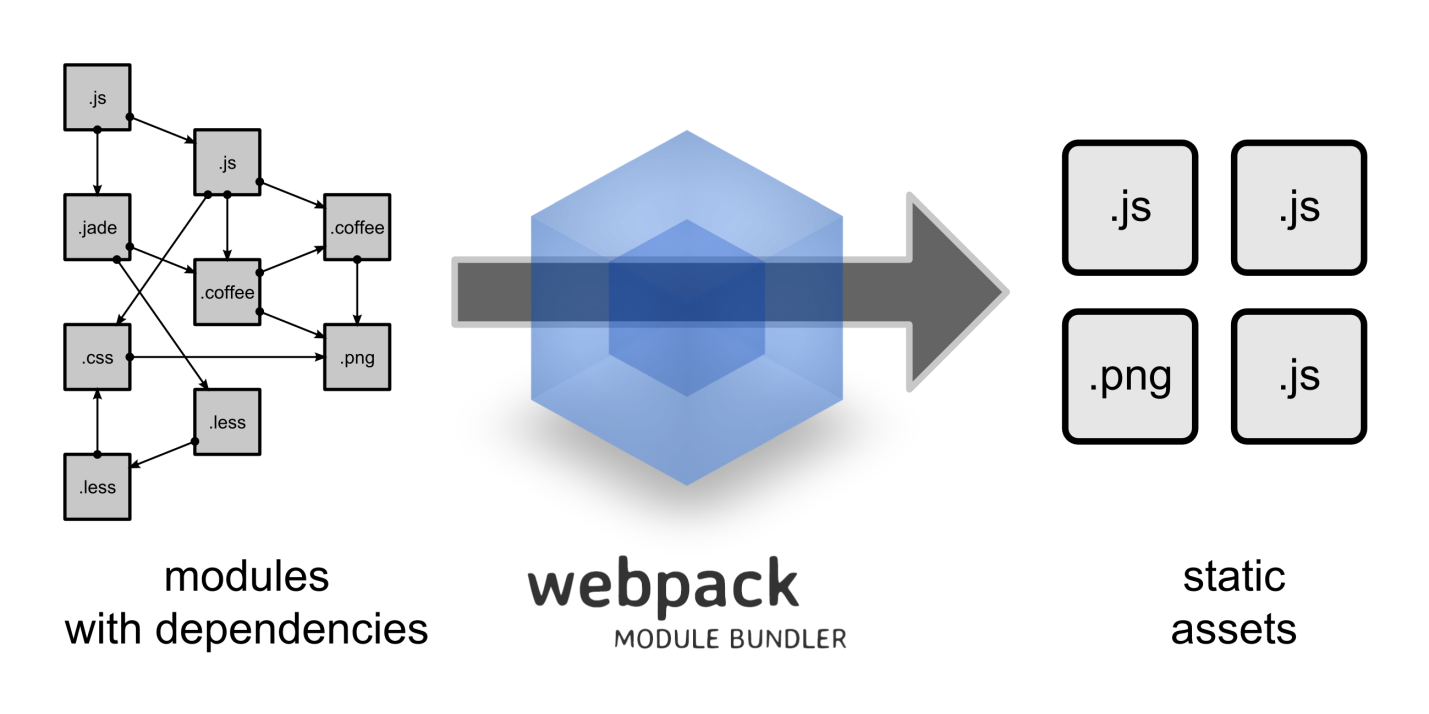
Webpack is formally referred to as a module bundler. The way that it works is that you specify a single file as your entry point. This file will be the root of your tree. Then every time you require a file from another file it's added to the tree. When you run webpack, all these files/modules are bundled into a single file [output file].

Webpack understands that CSS and images are also dependencies and treats them as such.

**Webpackfeatures:**

* Bundle your js files into a single file
* Write ES6/ES7 JavaScript (with help from babel)
* Minify/Optimize code
* Turn LESS/SCSS into CSS
* Use HMR (Hot Module Replacement)
* Include any type of file into your JavaScript

### Bundling



Here's a simple example:

[Dependency Tree](https://camo.githubusercontent.com/8399895a3372963cd4dee5d989aaacadd4134c57/687474703a2f2f692e696d6775722e636f6d2f6453676877774c2e706e67)

Given this picture you could have the directory:

MyDirectory

|- index.js

|- UIStuff.js

|- APIStuff.js

|- styles.css

|- extraFile.js

and this could be the content of your files

// index.js

require('./styles.css')

require('./UIStuff.js')

require('./APIStuff.js')

// UIStuff.js

var React =require('React')

React.createClass({

// stuff

})

// APIStuff.js

var fetch =require('fetch') // fetch polyfill

fetch('https://google.com')

/\* styles.css \*/

body {

background-color: rgb(200, 56, 97);

}

When you run webpack, you'll get a bundle with the contents of this tree, but extraFile.js, which was in the same directory, will not be part of the bundle because it was never required.

bundle.js will look like:

// contents of styles.css

// contents of UIStuff.js + React

// contents of APIStuff.js + fetch

The things that are bundled are only the things that you explicitly required across your files.

### The Command Line

Webpack install locally as dev dependency

npm install –save -dev webpack

npm i webpack –S

runtimedep:

npm install webpack –-save

To run webpack: install webpack globally

npm i webpack –g -> install globally webpack

If you want webpack to build every time you change a file:

webpack --watch

If you want to use a config file with webpack with a custom name:

webpack --config myconfig.js

### Loaders

As you probably noticed, I did something strange in the above example. I required a css file in a JavaScript file.

The really cool, and interesting thing about webpack is that you can require more than just JavaScript files.

There is this thing in webpack called a loader. Using these loaders, you can require anything from .css and .png ,.htmlfiles.

For example in the diagram above I had

// index.js

require('./styles.css')

If I include [the style-loader](https://github.com/webpack/style-loader) and the [thecss-loader](https://github.com/webpack/css-loader) in my webpackconfig, this is not only perfectly valid, but also will actually apply the CSS to my page.

This is just a single example of the many loaders you can use with webpack.

### Plugins

Plugins, like the name suggests, add extra functionality to webpack. One frequently used plugin is the UglifyJsPlugin, which lets you minify your JavaScript code.

## Your Config File –Webpack.config.js

Webpack does not work out of the box so you need to tailor it to your needs. In order to do this you need to create a file called

webpack.config.js

as this is the name that webpack recognizes by default. If you choose to use a different name you would have to use the --config flag to specify the file's name.

### A Minimal Example

[Example 2](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1/example2)

Say your directory structure looks like this:

MyDirectory

|- dist

|- src

|- index.js

|- webpack.config.js

Then a very minimal webpackconfig you can make is this

// webpack.config.js

var path =require('path')

module.exports= {

entry: ['./src/index'], // file extension after index is optional for .js files

output: {

path:path.join(\_\_dirname, 'dist'),

filename:'bundle.js'

}

}

Going over the new properties one by one:

* [entry](https://webpack.github.io/docs/configuration.html#entry) - The entrypoint of your bundle, which we discussed in the [bundling](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1#bundling) section. It's an array because webpack allows multiple entry points if you want to generate multiple bundles.
* [output](https://webpack.github.io/docs/configuration.html#output) - Dictates the form of the output by webpack
  + [path](https://webpack.github.io/docs/configuration.html#output-path) - where to put the bundle
  + [filename](https://webpack.github.io/docs/configuration.html#output-filename) - what to call the bundle

When you run webpack, this will create a file called bundle.js in the dist folder.

### Introducing Plugins

[Example 3](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1/example3)

Imagine that you've used webpack to bundle all your files together, and now you've realized that all together it's 900KB. This is a problem that can be ameliorated by minifying your bundle. To do this you need to use a plugin I mentioned earlier called the [UglifyJsPlugin](https://webpack.github.io/docs/list-of-plugins.html#uglifyjsplugin).

Moreover you will need to have webpack installed locally to actually be able to use the plugin.

npm install --save-dev webpack

Now you can require webpack and minify your code.

// webpack.config.js

var path =require('path')

var webpack=require('webpack')

module.exports= {

entry: ['./src/index'],

output: {

path:path.join(\_\_dirname, 'dist'),

filename:'bundle.js'

},

plugins: [

new webpack.optimize.UglifyJsPlugin({

compressor: {

warnings:false,

},

})

]

}

Going over the new properties one by one:

* plugins - An array that holds your plugins.
  + [webpack.optimize.UglifyJsPlugin](https://webpack.github.io/docs/list-of-plugins.html#uglifyjsplugin) - Minify your code, and suppress warning messages.

This time, when you run webpack, now that you have the UglifyJsPlugin this could reduce your imaginary 900KB file to 200KB through processes such as removing all the whitespace.

You can also add the [OccurrenceOrderPlugin](https://webpack.github.io/docs/list-of-plugins.html#occurrenceorderplugin)

Assign the module and chunk ids by occurrence count. Ids that are used often get lower (shorter) ids. This makes ids predictable, reduces to total file size and is recommended.

To be honest I'm not sure how the underlying mechanisms work, but in the current [webpack2 beta it's included by default](https://gist.github.com/sokra/27b24881210b56bbaff7) so I include it as well.

// webpack.config.js

var path =require('path')

var webpack=require('webpack')

module.exports= {

entry: ['./src/index'],

output: {

path:path.join(\_\_dirname, 'dist'),

filename:'bundle.js'

},

plugins: [

newwebpack.optimize.UglifyJsPlugin({

compressor: {

warnings:false,

},

}),

newwebpack.optimize.OccurrenceOrderPlugin()

]

}

So now we have written a config that allows us to minify and bundle our JavaScript. This bundle could be copied and pasted into another project's directory, and thrown into a <script> tag there.

## A More Complete Example

In the following section, we are going to create a very simple website using webpack. If you wish to follow along with the example, create a directory with the structure

MyDirectory

|- dist

|- src

|- index.js

|- index.html

|- styles.css

|- package.json

|- webpack.config.js

#### Contents

1. [Introducing Loaders](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1#introducing-loaders) - We will add loaders, which allow us to add CSS to our bundle.
2. [Adding More Plugins](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1#adding-more-plugins) - We will add a plugin that'll help us create/use an HTML file.
3. [The Development Server](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1#the-development-server) - We'll split our webpackconfig into separate development and production files. Then use the webpack-dev-server to view our website and enable HMR.
4. [Start Coding](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1#start-coding) - We will actually write some JavaScript.

#### Introducing Loaders

[Example 4](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1/example4)

Earlier in the tutorial I mentioned [loaders](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1#loaders). These will help us require non-js files in our code. In this case, we will need the style loader and the css loader. First we need to install the loaders:

npm install --save-dev style-loader css-loader

Now that it's installed we can tweak our config to include the css loader:

// webpack.config.js

var path =require('path')

varwebpack=require('webpack')

module.exports= {

entry: ['./src/index'],

output: {

path:path.join(\_\_dirname, 'dist'),

filename:'bundle.js'

},

plugins: [

newwebpack.optimize.UglifyJsPlugin({

compressor: {

warnings:false,

},

}),

newwebpack.optimize.OccurrenceOrderPlugin()

],

module: {

loaders: [{

test:/**\.**css$/,

loaders: ['style', 'css']

}]

}

}

Going over the new properties one by one:

* [module](http://webpack.github.io/docs/configuration.html#module) - Options affecting your files
  + [loaders](http://webpack.github.io/docs/configuration.html#module-loaders) - An array of loaders that we specify for our application
    - test - A regular expression to match the loader with a file
    - loaders - Which loaders to use for files that match the test

This time when you run webpack, if you require a file that ends in .css, then we will apply the style and css loaders to it, which adds the CSS to the bundle.

If we didn't have the loaders, then we would get an error like this:

ERROR in ./test.css

Module parse failed: /Users/Developer/workspace/tutorials/webpack/part1/example1/test.css

Line 1: Unexpected token {

You may need an appropriate loader to handle this file type.

**Optional**

If you want to use SCSS instead of CSS you would need to run:

npm install --save-dev sass-loader node-sass webpack

and instead your loader would be written as

{

test:/**\.**scss$/,

loaders: ["style", "css", "sass"]

}

The process is similar for LESS.

An important aspect to recognize is that there is an order to which these loaders need to be specified. In the above example, the sass loader is first applied to your .scss files, then the css loader, and finally the style loader. As you can see, the pattern is that these loaders are applied from right to left.

#### Adding More Plugins

[Example 5](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1/example5)

Now that we have the infrastructure for styling our website we need an actual page to style. We'll be doing this through the[html-webpack-plugin](https://github.com/ampedandwired/html-webpack-plugin), which lets us generate an HTML page or use an existing one. We'll use an existing one index.html.

First we install the plugin:

npm install --save-dev html-webpack-plugin@2

Then we can add it to our config

// webpack.config.js

var path =require('path')

varwebpack=require('webpack')

varHtmlWebpackPlugin=require('html-webpack-plugin')

module.exports= {

entry: ['./src/index'],

output: {

path:path.join(\_\_dirname, 'dist'),

filename:'bundle.js'

},

plugins: [

newwebpack.optimize.UglifyJsPlugin({

compressor: {

warnings:false,

},

}),

newwebpack.optimize.OccurrenceOrderPlugin(),

newHtmlWebpackPlugin({

template:'./src/index.html'

})

],

module: {

loaders: [{

test:/**\.**css$/,

loaders: ['style', 'css']

}]

}

}

This time, when you run webpack, because we specified an HtmlWebpackPlugin with a template of ./src/index.html, it will generate a file called index.html in our dist folder with the contents of ./src/index.html

There's no point in using index.html as a template if it's empty. Now would be a good time to actually populate it.

<html>

<head>

<title>Webpack Tutorial</title>

</head>

<body>

<h1>Very Website</h1>

<sectionid="color"></section>

<buttonid="button">Such Button</button>

</body>

</html>

Note that we aren't putting a <script> tag into our HTML for bundle.js. The plugin will actually automatically do that for you. If you do put in the script tag, you'll end up loading your same code twice.

and while we're at it let's add some basic styling in styles.css

h1 {

color: rgb(114, 191, 190);

text-align: center;

}

#color {

width: 300px;

height: 300px;

margin: 0auto;

}

button {

cursor: pointer;

display: block;

width: 100px;

outline: 0;

border: 0;

margin: 20pxauto;

}

#### The Development Server

[Example 6](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1/example6)

Now we want to actually see our website in the browser, which requires a web server to serve our code. Conveniently, webpack comes with the webpack-dev-server, which you need to install both locally and globally

npm install -g webpack-dev-server

npm install --save-devwebpack-dev-server

The dev server is an extremely useful resource for seeing what your website looks like in the browser, and more rapid development. By default you can visit it at http://localhost:8080. Unfortunately, features such as hot reloading don't work out of the box, and require some more configuration.

This is a good point to split up our webpackconfig into one meant for development and one meant for production. Since we're keeping it simple in this tutorial, it won't be a huge difference, but it's an introduction to the extreme configurability of webpack. We'll call them webpack.config.dev.js and webpack.config.prod.js.

// webpack.config.dev.js

var path =require('path')

varwebpack=require('webpack')

varHtmlWebpackPlugin=require('html-webpack-plugin')

module.exports= {

devtool:'cheap-eval-source-map',

entry: [

'webpack-dev-server/client?http://localhost:8080',

'webpack/hot/dev-server',

'./src/index'

],

output: {

path:path.join(\_\_dirname, 'dist'),

filename:'bundle.js'

},

plugins: [

newwebpack.HotModuleReplacementPlugin(),

newHtmlWebpackPlugin({

template:'./src/index.html'

})

],

module: {

loaders: [{

test:/**\.**css$/,

loaders: ['style', 'css']

}]

},

devServer: {

contentBase:'./dist',

hot:true

}

}

**Changes**

1. The devconfig omits the optimizations as they are unnecessary overhead when you are constantly rebuilding. So no webpack.optimize plugins.
2. The devconfig has the necessary configuration for the dev server, which you can read more about [here](https://webpack.github.io/docs/webpack-dev-server.html).

Summarized:

* entry: The two new entry points connect the server to the browser to allow for HMR.
* devServer
  + contentBase: Where to serve files from
  + hot: enable HMR
  + port: to specify the user wished local port number.

The prod config doesn't change much

// webpack.config.prod.js

var path =require('path')

varwebpack=require('webpack')

varHtmlWebpackPlugin=require('html-webpack-plugin')

module.exports= {

devtool:'source-map',

entry: ['./src/index'],

output: {

path:path.join(\_\_dirname, 'dist'),

filename:'bundle.js'

},

plugins: [

newwebpack.optimize.UglifyJsPlugin({

compressor: {

warnings:false,

},

}),

newwebpack.optimize.OccurrenceOrderPlugin(),

newHtmlWebpackPlugin({

template:'./src/index.html'

})

],

module: {

loaders: [{

test:/**\.**css$/,

loaders: ['style', 'css']

}]

}

}

I've also added a brand new property to both the devconfig and the prod config:

* [devtool](https://webpack.github.io/docs/configuration.html#devtool) - This is a debugging aid. Basically, when you get a error, it'll help you see where you made the mistake something like the chrome developer console. As for the difference between source-map and cheap-eval-source-map it's a little hard to glean from the docs. What I can say definitively is that source-map is meant for production and has a lot of overhead, and that cheap-eval-source-map has less overhead and is meant for developing only.

To run the dev server we have to run

webpack-dev-server --config webpack.config.dev.js

and to build the production code we have to run

webpack --config webpack.config.prod.js

To make our lives a little easier we are now going to use package.json as a simple task runner so that we don't need to keep typing out either command.

We add the scripts property to the config

// package.json

{

//...

"scripts": {

"build":"webpack --config webpack.config.prod.js",

"dev":"webpack-dev-server --config webpack.config.dev.js"

}

//...

}

We can run these commands with

npm run build

npm run dev

You can now view your beautiful website by running npm run dev, and navigating to http://localhost:8080.

**Side Note:** while I was testing this portion I realized that the server would not hot reload when I modified the index.html file. The solution to this problem is over at [html-reload](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1/html-reload). It's useful information that covers some more configuration options of webpack, which I recommend looking at, but I left it separate because I feel like it lengthens the tutorial for too trivial of a reason.

#### Start Coding

[Example 7](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1/example7)

The reason most people seem to be flustered by webpack is the fact that they need to go through all of this to get to the point where they finally write JavaScript; however, now we have arrived at the climax of our tutorial.

Just in case you haven't already: do npm run dev, and navigate to http://localhost:8080. Setting up that dev server with hot reloading wasn't for show. Every single time you save while editing any part of your project, the browser will reload to show your changes.

We are also going to require anpm package just to demonstrate how you can use them in your frontend now.

npm install --save pleasejs

PleaseJS is a random color generator, which we're going to hook up to our button to change the color of our div.

// index.js

// Accept hot module reloading

if (module.hot) {

module.hot.accept()

}

require('./styles.css') // The page is now styled

var Please =require('pleasejs')

var div =document.getElementById('color')

var button =document.getElementById('button')

functionchangeColor() {

div.style.backgroundColor=Please.make\_color()

}

button.addEventListener('click', changeColor)

Interestingly, [in order for Hot Module Replacement to work](https://webpack.github.io/docs/hot-module-replacement-with-webpack.html#what-is-needed-to-use-it) You need to include the code:

if (module.hot) {

module.hot.accept()

}

in a module or a parent of that module.

and we're done!

**Side Note:** You might have noticed a delay before your css was applied, or maybe you hate the fact that your css is in your javascript file. I've put aside an example, [css-extract](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1/css-extract), that describes how to put your CSS in a different file.

# Beginner Webpack Tutorial Part 2 - Using Webpack with Babel

Now that we learned the basics of using webpack, we're going to learn to leverage babel 6 in order to write ES6, the new specification of javascript.

If you've ever written ES6, it's hard to want to go back to writing ES5. If you haven't had the chance to write ES6 yet, a big reason is probably because setting up a dev environment, understanding all the configuration options, and deciding which ones to use can be discouraging.

I hope that this tutorial can make that process of getting started much easier.

## Requirements

1. If you haven't already please look at [part 1](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1)
2. For an overview of ES6, [this](https://github.com/DrkSephy/es6-cheatsheet) is a good resource.

## Contributing

I will gladly accept any and all contributions/corrections. If you have any questions, feel free to leave them as issues. If I made mistakes, please point them out. Finally, if you feel that I left anything out or could have explained something better, make sure to leave an issue or make a pull request.

## Table Of Contents

* [Babel](https://github.com/AriaFallah/WebpackTutorial/tree/master/part2#babel)
  + [What Does Babel Do?](https://github.com/AriaFallah/WebpackTutorial/tree/master/part2#what-does-babel-do)
  + [Configuring Babel](https://github.com/AriaFallah/WebpackTutorial/tree/master/part2#configuring-babel)
* [Webpack](https://github.com/AriaFallah/WebpackTutorial/tree/master/part2#webpack)
  + [A New Loader](https://github.com/AriaFallah/WebpackTutorial/tree/master/part2#a-new-loader)
* [We are Done?](https://github.com/AriaFallah/WebpackTutorial/tree/master/part2#we-are-done)
  + [Requiring With ES6 Modules](https://github.com/AriaFallah/WebpackTutorial/tree/master/part2#requiring-with-es6-modules)
* [Extra Credit](https://github.com/AriaFallah/WebpackTutorial/tree/master/part2#extra-credit)
  + [Production Environment Variables With Webpack and Babel](https://github.com/AriaFallah/WebpackTutorial/tree/master/part2#production-environment-variables-with-webpack-and-babel)
  + [Adding Linting](https://github.com/AriaFallah/WebpackTutorial/tree/master/part2#adding-linting)
* [Conclusion](https://github.com/AriaFallah/WebpackTutorial/tree/master/part2#conclusion)

## Babel

If you want a more in-depth explanation, and finer grained control over babel then please look at [their handbook](https://github.com/thejameskyle/babel-handbook/blob/master/translations/en/user-handbook.md). I'm paraphrasing the basics here.

### What Does Babel Do?

Simply stated, babel lets you take advantage of a much more fully featured specification of javascript, which isn't supported by most browsers and environments, and turns it into ES5, which has much more widespread support.

With babel, this code, which is just getting support now in the latest browsers

constsquare=n=> n \* n;

is transformed into something like

varsquare=functionsquare(n) {

return n \* n;

};

which you could run anywhere that supports javascript.

### Configuring Babel

Another tool, another config file. This time around we'll have a file called

.babelrc

Thankfully, the .babelrc file will only be a single line long.

{

"presets": ["es2015", "stage-2"]

}

The only option you need to specify is presets, which are described in the excerpt below:

JavaScript also has some proposals that are making their way into the standard through the TC39's (the technical committee behind the ECMAScript standard) process.

This process is broken through a 5 stage (0-4) process. As proposals gain more traction and are more likely to be accepted into the standard they proceed through the various stages, finally being accepted into the standard at stage 4.

Note that there is no stage-4 preset as it is simply the es2015 preset above.

To sum it up, presets are bundles of plugins that add features to the code you're writing. es2015 adds features that definitely are going to be in the official release of ES6, and the presets that are stages 0-3 are proposals for future specifications of Javascript that are still being drafted. The lower you go, the higher the risk that the features you're using are going to have support dropped.

From my experience, the lowest I've needed to go is stage-2 to be able to use something called [object spread](https://github.com/sebmarkbage/ecmascript-rest-spread). You can see the rest of the proposals [here](https://github.com/tc39/ecma262), and decide how low you want to go.

Anyways to use these presets, we need to install them

npm install --save-dev babel-preset-es2015 babel-preset-stage-2

and that's actually all you need to do.

## Webpack

We're going to use the same exact configs from [example 7 from part 1](https://github.com/AriaFallah/WebpackTutorial/tree/master/part1/example7), but add the functionality needed to use ES6.

Current configs:

// webpack.config.dev.js

var path =require('path')

varwebpack=require('webpack')

varHtmlWebpackPlugin=require('html-webpack-plugin')

module.exports= {

devtool:'cheap-eval-source-map',

entry: [

'webpack-dev-server/client?http://localhost:8080',

'webpack/hot/dev-server',

'./src/index'

],

output: {

path:path.join(\_\_dirname, 'dist'),

filename:'bundle.js'

},

plugins: [

newwebpack.HotModuleReplacementPlugin(),

newHtmlWebpackPlugin({

template:'./src/index.html'

})

],

module: {

loaders: [{

test:/**\.**css$/,

loaders: ['style', 'css']

}]

},

devServer: {

contentBase:'./dist',

hot:true

}

}

and

// webpack.config.prod.js

var path =require('path')

var webpack=require('webpack')

var HtmlWebpackPlugin=require('html-webpack-plugin')

module.exports= {

devtool:'source-map',

entry: ['./src/index'],

output: {

path:path.join(\_\_dirname, 'dist'),

filename:'bundle.js'

},

plugins: [

new webpack.optimize.UglifyJsPlugin({

compressor: {

warnings:false,

},

}),

new webpack.optimize.OccurrenceOrderPlugin(),

new HtmlWebpackPlugin({

template:'./src/index.html'

})

],

module: {

loaders: [{

test:/**\.**css$/,

loaders: ['style', 'css']

}]

}

}

### A New Loader

To transpile our code to ES5 we need to run it through a new loader called babel-loader, which has a dependency of babel-core. This loader will use our .babelrc config to understand and transform our code into its new form.

npm install --save-dev babel-loader babel-core

We add that to both our dev and prod configs:

// To save space I'll just show the "loaders" part

// Both webpack.config.dev.js and webpack.config.prod.js

module: {

loaders: [{

test:/**\.**css$/,

loaders: ['style', 'css']

}, {

test:/**\.**js$/,

loaders: ['babel'],

include:path.join(\_\_dirname, 'src')

}]

}

An **extremely important** thing to note is the usage of the include property. When we run webpack, because we have set our test to /.js$/, webpack will try to run the babel loader on every single js file in your dependency tree.

Can you spot the problem with this? What if I require('bluebird'), or any other large npm package? It'll try to run **node\_modules** through your babel-loader, which will extend your build process by an extreme amount.

include prevents this by specifying that this loader only applies to .js files in your src directory.

Alternatively you could change include: path.join(\_\_dirname, 'src') to exclude: /node\_modules/ which will then include everything but the node\_modules folder. More information can be found [here](https://webpack.github.io/docs/configuration.html#module-loaders).

## We are Done?

Honestly I thought this tutorial would be longer, but it seems I forgot that adding babel is actually pretty trivial. We can now update our earlier code in index.js to use the ES6 syntax:

// index.js

// Accept hot module reloading

if (module.hot) {

module.hot.accept()

}

require('./styles.css') // The page is now styled

constPlease=require('pleasejs')

constdiv=document.getElementById('color')

constbutton=document.getElementById('button')

constchangeColor= () =>div.style.backgroundColor=Please.make\_color()

button.addEventListener('click', changeColor)

### Requiring With ES6 Modules

Another thing to note is that now we can use the es6 module system. For example, instead of

constPlease=require('pleasejs')

we can now do

importPleasefrom'pleasejs'

## Extra Credit

Since that actually didn't take so long I'm going to cover two more topics that are pretty important and useful.

### Production Environment Variables WithWebpack and Babel

#### Webpack

If we don't want to execute a portion of code in production we can use the handy dandy [DefinePlugin](https://github.com/webpack/docs/wiki/list-of-plugins#defineplugin).

The plugin lets us create a global constant for our entire bundle, which we could name anything, such as DONT\_USE\_IN\_PRODUCTION: true, but more practically, it's a much better choice to use process.env.NODE\_ENV: JSON.stringify('production'). This is because many programs recognize and use process.env.NODE\_ENV for additional features and optimization of your code.

Why JSON.stringify? Because according to the docs:

If the value is a string it will be used as a code fragment.

This means a value of 'production' would just be an error. If you think JSON.stringify is weird, a valid alternative is '"production"'.

Your plugins array should now look like

plugins: [

new webpack.optimize.UglifyJsPlugin({

compressor: {

warnings:false,

},

}),

new webpack.optimize.OccurrenceOrderPlugin(),

new HtmlWebpackPlugin({

template:'./src/index.html'

}),

new webpack.DefinePlugin({

'process.env.NODE\_ENV':JSON.stringify('production')

})

]

Now, if we don't want to execute some code in production, we can put it in an if statement:

if (process.env.NODE\_ENV!=='production') {

// not for production

}

In our current project we could say to exclude the hot reloading if it's in production:

// Accept hot module reloading during development

if (process.env.NODE\_ENV!=='production') {

if (module.hot) {

module.hot.accept()

}

}

#### Babel

Defining our production variable as process.env.NODE\_ENV has another added benefit.

[From the handbook](https://github.com/thejameskyle/babel-handbook/blob/master/translations/en/user-handbook.md)

The current environment will use process.env.BABEL\_ENV. When BABEL\_ENV is not available, it willfallback to NODE\_ENV, and if that is not available it will default to "development".

This means that the babel environment will match our webpack environment.

We can take advantage of this, tweaking our .babelrc to have development only config by adding env:

{

"presets": ["es2015", "stage-2"],

"env": {

// only happens if NODE\_ENV is undefined or set to 'development'

"development": {

// ignored when NODE\_ENV is production!

}

}

We'll be using this in part 3 with react when we introduce the [React Transform HMR](https://github.com/gaearon/react-transform-hmr)

### Adding Linting

If you've looked at any project seeds/starters for Webpack/React, you've probably seen a file called .eslintrc. If you aren't using an IDE, but instead are using a text editor like Atom, Sublime, Emacs, Vim, etc., eslint provides syntax and style checks, pointing out your mistakes. Moreover, even if you're using an IDE, it can provide more features, and ensure uniformity in coding style for all contributors to the project.

Do take note that if you want it integrated into your editor, you need to install a package. For example, I use [linter-eslint](https://github.com/AtomLinter/linter-eslint) for Atom.

To reduce the amount of our config we write manually, we're going to take advantage of the fact that eslint lets you inherit from other people's configs. I always like to use a config based upon [airbnb's style guide](https://github.com/airbnb/javascript).

To get started, we need to install eslint as well as airbnb'sconfig

npm install eslint

npm install -g eslint-cli

npm install --save-dev eslinteslint-config-airbnb-base

Our starting config will look like:

// .eslintrc

{

"extends":"airbnb-base"// 'airbnb-base' because 'airbnb' assumes usage of react

}

However because linting is **highly opinionated**, I like to tweak this a bit. If you want to know what all these rules mean, or tweak them to fit your preferences look [here](http://eslint.org/docs/rules/):

// .eslintrc

{

"extends":"airbnb-base",

"rules": {

"comma-dangle":0,

"no-console":0,

"semi": [2, "never"],

"func-names":0,

"space-before-function-paren":0,

"no-multi-spaces":0

}

}

Additionally, out of the box, eslint does not support/recognize babel syntax so we be installing two new packages:

npm install --save-dev babel-eslinteslint-plugin-babel

and tweaking our config once more to add [babel specific rules](https://github.com/babel/eslint-plugin-babel):

// .eslintrc

{

"extends":"airbnb-base",

"parser":"babel-eslint",

"rules": {

"comma-dangle":0,

"no-console":0,

"semi": [2, "never"],

"func-names":0,

"space-before-function-paren":0,

"no-multi-spaces":0,

"babel/generator-star-spacing":1,

"babel/new-cap":1,

"babel/object-shorthand":1,

"babel/arrow-parens":1,

"babel/no-await-in-loop":1

},

"plugins": [

"babel"

]

}

Finally it's also a good idea to add an npm script for linting to our existing package.json file

// package.json

"scripts": {

"build":"webpack --config webpack.config.prod.js",

"dev":"webpack-dev-server --config webpack.config.dev.js",

"lint":"eslintsrc"

}

which you can run with npm run lint just to make sure none of your code violates the rules you specified.

## Conclusion

I've put the end result of all of this into [example 1](https://github.com/AriaFallah/WebpackTutorial/tree/master/part2/example1) just in case I wasn't clear on anything. If you still have trouble understanding, feel free to leave a question in issues.

So now we can easily write ES6 code, and additionally, understand the config that enables us to write it !

However, just because you can write it from scratch, it doesn't mean you have to. For your convenience, [I have made a separate repository you can clone](https://github.com/AriaFallah/minimal-babel-starter) to get started, which is minimal and based on this tutorial series.