tuts+

CODE > JAVASCRIPT

Introduction to Webpack: Part 2

by Stuart Memo 11 May 2016

Difficulty: Intermediate Length: Short Languages: English

JavaScript CSS Web Development

This post is part of a series called Introduction to Webpack.

Introduction to Webpack: Part 1

In the previous tutorial we learned how to set up a Webpack project and how to use loaders to process our JavaScript. Where Webpack really shines, though, is in its ability to bundle other types of static assets such as CSS and images, and include them in our project only when they're required. Let's start by adding some styles to our page.

Style Loaders

First, create a normal CSS file in a styles directory. Call in `main.css` and add a style rule for the heading element.

```css
h2 {
    background: blue;
    color: yellow;
}
```

So how do we get this stylesheet into our page? Well, like most things with Webpack, we'll need another loader. Two in fact: css-loader and style-loader. The first reads all the styles from our CSS files, whilst the other injects said styles into our HTML page. Install them like so:

```
npm install style-loader css-loader
```

Next, we tell Webpack how to use them. In `webpack.config.js`, we need to add another object to the loaders array. In it we want to add a test to match only CSS files as well as specify which loaders to use.

```
{
    test: /\.css$/,
    exclude: /node_modules/,
    loader: 'style!css'
}
```

The interesting part of this code snippet is the `'style!css'` line. Loaders are read from right to left, so this tells Webpack to first read the styles of any file ending in `.css`, and then inject those styles into our page.

Because we've updated our configuration file, we'll need to restart the development server for our changes to be picked up. Use `ctrl+c` to stop the server and `webpack-dev-server` to start it again.

All we need to do now is require our stylesheet from within our `main.js` file. We do this in the same way as we would any other JavaScript module:

```
const sayHello = require('./say-hello');

require('./styles/main.css');

sayHello('Guybrush', document.querySelector('h2'));
```

Note how we haven't even touched `index.html`. Open up your browser to see the page with styled `h2`. Change the colour of the heading in your stylesheet to see it instantly update without a refresh. Lovely.

You've Got to Sass It

"But nobody uses CSS these days, Grandad! It's all about Sass". Of course it is. Luckily Webpack has a loader to do just the thing. Install it along with the node version of Sass using:

```
npm install sass-loader node-sass
```

Then update `webpack.config.js`:

```
{
    test: /\.scss$/,
    exclude: /node_modules/,
    loader: 'style!css!sass'
}
```

This is now saying that for any file ending with `.scss`, convert the Sass to plain CSS, read the styles from the CSS, and then insert the styles into the page. Remember to rename `main.css` to `main.scss`, and require the newly named file in instead. First some Sass:

```
$background: blue;

h2 {
    background: $background;
    color: yellow;
}
```

Then main.js:

```
require('./styles/main.scss');
```

Super. It's as easy as that. No converting and saving files, or even watching folders. We just require in our Sass styles directly.

Images

"So images, loaders too I bet?" Of course! With images, we want to use the url-loader. This loader takes the relative URL of your image and updates the path so that it's correctly included in your file bundle. As per usual:

```
npm install url-loader
```

Now, let's try something different in our `webpack.config.js`. Add another entry to the loaders array in the usual manner, but this time we want the regular expression to match images with different file extensions:

```
{
    test: /\.(jpg|png|gif)$/,
    include: /images/,
    loader: 'url'
}
```

Note the other difference here. We're not using the `exclude` key. Instead we're using `include`. This is more efficient as it is telling webpack to ignore everything that doesn't match a folder called "images".

Usually you'll be using some sort of templating system to create your HTML views, but we're going to keep it basic and create an image tag in JavaScript the old-fashioned way. First create an image element, set the required image to the src attribute, and then add the element to the page.

```
var imgElement = document.createElement('img');

imgElement.src = require('./images/my-image.jpg');

document.body.appendChild(imgElement);
```

Head back to your browser to see your image appear before your very eyes!

Preloaders

Another task commonly carried out during development is linting. Linting is a way of looking out for potential errors in your code along with checking that you've followed certain coding conventions. Things you may want to look for are "Have I used a variable without declaring it first?" or "Have I forgotten a semicolon at the end of a line?" By enforcing these rules, we can weed out silly bugs early on.

A popular tool for linting is JSHint. This looks at our code and highlights potential errors we've made. JSHint can be run manually at the command line, but that quickly becomes a chore during development. Ideally we'd like it to run automatically every time we save a file. Our Webpack server is already watching out for changes, so yes, you guessed it—another loader.

Install the jshint-loader in the usual way:

```
npm install jshint-loader
```

Again we have to tell Webpack to use it by adding it to our `webpack.config.js`. However, this loader is slightly different. It's not actually transforming any code—it's just having a look. We also don't want all our heavier code-modifying loaders to run and fail just because we've forgotten a semicolon. This is where preloaders come in. A preloader is any loader we specify to run before our main tasks. They're added to our `webpack.conf.js` in a similar way to loaders.

```
module: {
    preLoaders: [
        {
            test: /\.js$/,
            exclude: /node_modules/,
            loader: 'jshint'
        }
    ],
    loaders: [
        ...
    ]
}
```

Now our linting process runs and fails immediately if there's a problem detected. Before we restart our web server, we need to tell JSHint that we're using ES6, otherwise it will fail when it sees the `const` keyword we're using.

After the module key in our config, add another entry called "jshint" and a line to specify the version of JavaScript.

```
module: {
    preLoaders: [
        ...
    ],
    loaders: [
        ...
    ]
},
jshint: {
    esversion: 6
}
```

Save the file and restart `webpack-dev-server`. Running ok? Great. This means your code contains no errors. Let's introduce one by removing a semicolon from this line:

```
var imgElement = document.createElement('img')
```

Again, save the file and look at the terminal. Now we get this:

```
WARNING in ./main.js
jshint results in errors
  Missing semicolon. @ line 7 char 47
```

Thanks, JSHint!

Getting Ready for Production

Now that we're happy our code is in shape and it does everything we want it to, we need to get it ready for the real world. One of the most common things to do when putting your code live is to minify it, concatenating all your files into one and then compressing that into the smallest file possible. Before we continue, take a look at your current `bundle.js`. It's readable, has lots of whitespace, and is 32kb in size.

"Wait! Don't tell me. Another loader, right?" Nope! On this rare occasion, we don't need a loader. Webpack has minification built right in. Once you're happy with your code, simply run this command:

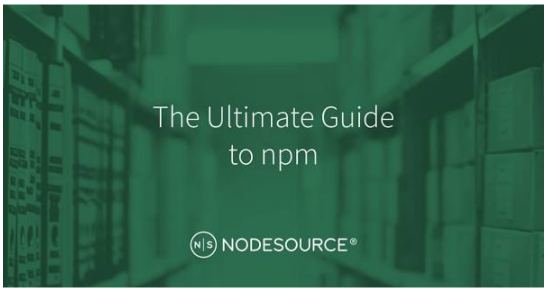
```
webpack -p
```

The `-p` flag tells Webpack to get our code ready for production. As it generates the bundle, it optimises as much as it can. After running this command, open `bundle.js` and you'll see it's all been squashed together, and that even with such a small amount of code we've saved 10kb.

Summary

I hope that this two-part tutorial has given you enough confidence to use Webpack in your own projects. Remember, if there's something you want to do in your build process then it's very likely Webpack has a loader for it. All loaders are installed via npm, so have a look there to see if someone's already made what you need.

Have fun!



Stuart Memo

Stuart Memo is a programmer and musician who believes that JavaScript is the new Punk Rock. He lives and works in Glasgow, Scotland where he spends much of his time making projects with silly names.

stuartmemo

Weekly email summary

Subscribe below and we'll send you a weekly email summary of all new Code tutorials. Never miss out on learning about the next big thing.

Email Address

Update me weekly

Translations

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post

Powered by native

7 Comments **Tuts+ Hub** leanne

Recommend 4 Share Sort by Best

Join the discussion…

DonM55 • 9 months ago

Update: you know need to include the "-loader" suffix in the line: loader: 'style-loader!css-loader' or you'll get errors

3 ^ | ⌄ • Reply • Share ›

Marco • a year ago

Hi there.
I believe you also need to install the jshint package along with the jshint-loader, otherwise your webpack config won't find the jshint module, probably you could update your tutorial. Cheers.

1 ^ | ⌄ • Reply • Share ›

Mark Hurwitz • 2 years ago

In the 'preloaders' section, I think you only installed the loader and forgot to also 'npm install jshint'.

Also, when adding to webpack.config.js 'jshint: { eversion: 6 }', I got an error:

```
jshint results in errors
Incompatible values for the 'esversion' and 'esnext' linting options. (0% scanned). @ line 0 char 0
undefined
```

So I changed to 'jshint: { esnext: 6 }' and the error went away. Not sure what that's about.

Nice tut! Very clear and useful.

1 ^ | ⌄ • Reply • Share ›

Porrapat Petchdamrongskul • a year ago

There are a lot of error found. Please update your tutorial.

Reply • Share ›

Sheikh Sohaib • a year ago

Awesome Tutorial , You've made my life easy, Thank you so much.

Reply • Share ›

Elad Nava • a year ago

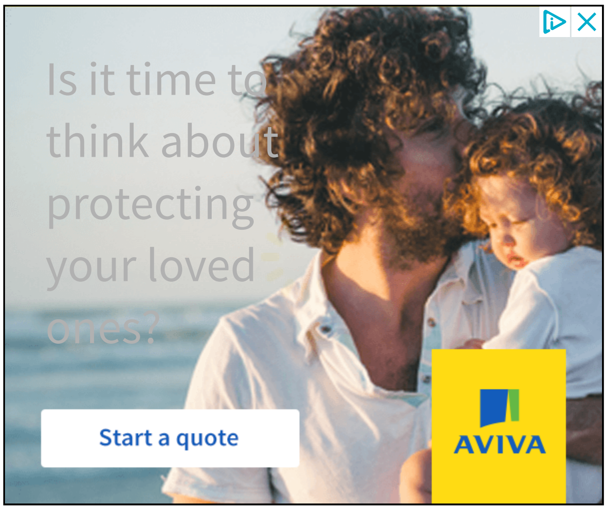
Excellent tutorial, thanks so much!

Reply • Share ›

Steve Popoola • a year ago

Thanks for this tutorial on webpack. One of the best I have come across yet for a webpack novice like me.

Reply • Share ›



QUICK LINKS - Explore popular categories

ENVATO TUTS+ +

JOIN OUR COMMUNITY +

HELP +

tuts+

25,530	1,106	21,031
Tutorials	Courses	Translations