tuts+

CODE > JAVASCRIPT

Introduction to Webpack: Part 1

by Stuart Memo 5 May 2016

Difficulty: Intermediate Length: Short Languages: English

JavaScript HTML CSS Web Development

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

⏩ Introduction to Webpack: Part 2

It's fairly standard practice these days when building a website to have some sort of build process in place to help carry out development tasks and prepare your files for a live environment.

You may use Grunt or Gulp for this, constructing a chain of transformations that allow you to throw your code in one end and get some minified CSS and JavaScript out at the other.

These tools are extremely popular and very useful. There is, however, another way of doing things, and that's to use Webpack.

What Is Webpack?

Webpack is what is known as a "module bundler". It takes JavaScript modules, understands their dependencies, and then concatenates them together in the most efficient way possible, spitting out a single JS file at the end. Nothing special, right? Things like RequireJS have been doing this for years.

Well, here's the twist. With Webpack, modules aren't restricted to JavaScript files. By using Loaders, Webpack understands that a JavaScript module may require a CSS file, and that CSS file may require an image. The outputted assets will only contain exactly what is needed with minimum fuss. Let's get set up so we can see this in action.

Installation

As with most development tools, you'll need Node.js installed before you can continue. Assuming you have this correctly set up, all you need to do to install Webpack is simply type the following at the command line.

```
npm install webpack -g
```

This will install Webpack and allow you to run it from anywhere on your system. Next, make a new directory and inside create a basic HTML file like so:

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Webpack fun</title>
    </head>
    <body>
        <h2></h2>
        <script src="bundle.js"></script>
    </body>
</html>
```

The important part here is the reference to `bundle.js`, which is what Webpack will be making for us. Also note the H2 element—we'll be using that later.

Next, create two files in the same directory as your HTML file. Name the first `main.js`, which as you can imagine is the main entry point for our script. Then name the second `say-hello.js`. This is going to be a simple module that takes a person's name and DOM element, and inserts a welcome message into said element.

```
// say-hello.js

module.exports = function (name, element) {
    element.textContent = 'Hello ' + name + '!';
};
```

Now that we have a simple module, we can require this in and call it from `main.js`. This is as easy as doing:

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

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

Now if we were to open our HTML file then this message would obviously not be shown as we've not included `main.js` nor compiled the dependencies for the browser. What we need to do is get Webpack to look at `main.js` and see if it has any dependencies. If it does, it should compile them together and create a bundle.js file we can use in the browser.

Head back to the terminal to run Webpack. Simply type:

```
webpack main.js bundle.js
```

The first file specified is the entry file we want Webpack to start looking for dependencies in. It will work out if any required files require any other files and will keep doing this until it's worked out all the necessary dependencies. Once done, it outputs the dependencies as a single concatenated file to `bundle.js`. If you press return, you should see something like this:

```
Hash: 3d7d7339a68244b03c68
Version: webpack 1.12.12
Time: 55ms
    Asset     Size  Chunks             Chunk Names
bundle.js  1.65 kB       0  [emitted]  main
   [0] ./main.js 90 bytes {0} [built]
   [1] ./say-hello.js 94 bytes {0} [built]
```

Now open `index.html` in your browser to see your page saying hello.

Configuration

It isn't much fun specifying our input and output files each time we run Webpack. Thankfully, Webpack allows us to use a config file to save us the trouble. Create a file called `webpack.config.js` in the root of your project with the following contents:

```
module.exports = {
    entry: './main.js',
    output: {
        filename: 'bundle.js'
    }
};
```

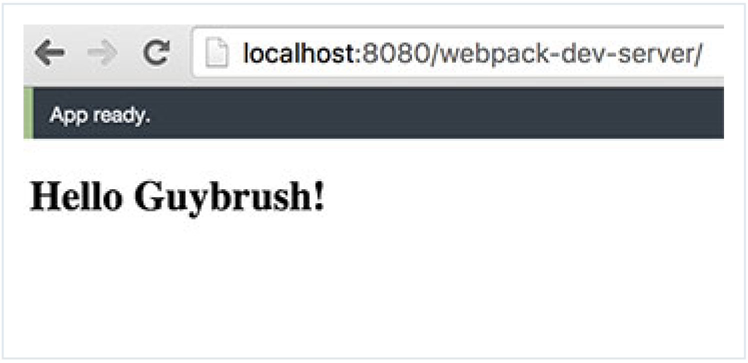
Now we can just type `webpack` and nothing else to achieve the same results.

Development Server

What's that? You can't even be bothered to type `webpack` every time you change a file? I don't know, today's generation etc, etc. Ok, let's set up a little development server to make things even more efficient. At the terminal, write:

```
npm install webpack-dev-server -g
```

Then run the command `webpack-dev-server`. This will start a simple web server running, using the current directory as the place to serve files from. Open a new browser window and visit http://localhost:8080/webpack-dev-server/. If all is well, you'll see something along the lines of this:



App ready.

Hello Guybrush!

Now, not only do we have a nice little web server here, we have one that watches our code for changes. If it sees we've altered a file, it will automatically run the webpack command to bundle our code and refresh the page without us doing a single thing. All with zero configuration.

Try it out, change the name passed to the `sayHello` function, and switch back to the browser to see your change applied instantly.

Loaders

One of the most important features of Webpack is Loaders. Loaders are analogous to "tasks" in Grunt and Gulp. They essentially take files and transform them in some way before they are included in our bundled code.

Say we wanted to use some of the niceties of ES2015 in our code. ES2015 is a new version of JavaScript that isn't supported in all browsers, so we need to use a loader to transform our ES2015 code into plain old ES5 code that *is* supported. To do this, we use the popular Babel Loader which, according to the instructions, we install like this:

```
npm install babel-loader babel-core babel-preset-es2015 --save-dev
```

This installs not only the loader itself but its dependencies and an ES2015 preset as Babel needs to know what type of JavaScript it is converting.

Now that the loader is installed, we just need to tell Babel to use it. Update `webpack.config.js` so it looks like this:

```
module.exports = {
    entry: './main.js',
    output: {
        filename: 'bundle.js'
    },
    module: {
        loaders: [
            {
                test: /\.js$/,
                exclude: /node_modules/,
                loader: 'babel',
                query: {
                    presets: ['es2015']
                }
            }
        ],
    }
};
```

There are a few important things to note here. The first is the line `test: /\.js$/`, which is a regular expression telling us to apply this loader to all files with a `.js` extension. Similarly `exclude: /node_modules/` tells Webpack to ignore the `node_modules` directory. `loader` and `query` are fairly self-explanatory—use the Babel loader with the ES2015 preset.

Restart your web server by pressing `ctrl+c` and running `webpack-dev-server` again. All we need to do now is use some ES6 code in order to test the transform. How about we change our `sayHello` variable to be a constant?

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

After saving, Webpack should have automatically recompiled your code and refreshed your browser. Hopefully you'll see no change whatsoever. Take a peek in `bundle.js` and see if you can find the `const` keyword. If Webpack and Babel have done their jobs, you won't see it anywhere—just plain old JavaScript.

On to Part 2

In Part 2 of this tutorial, we'll look at using Webpack to add CSS and images to your page, as well as getting your site ready for deployment.

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

THE ROYAL BRITISH
LEGION
Show your support with a regular gift
Support us

5 Comments **Tuts+ Hub** leanne

Recommend 6 Share Sort by Best

Join the discussion…

Sazal Kanti • 2 years ago

Very pointed, clean and simple explanation. Though webpack has bad reputation of having poor documentation but still its great. Don't know about the current documentation status. But article like this will surely help who are getting started with webpack. Beginners will like it specially with reactjs.

2 • Reply • Share ›

Dirk Helbert • 2 years ago

Good and simple explanation. Waiting for part 2

1 • Reply • Share ›

Jitendra Vyas • 2 years ago

Waiting for part 2. Followed your instructions of this article and everything is running fine.

1 • Reply • Share ›

Rinkesh Gupta • 8 months ago

Great Tutorial. Thanks a lot..

• Reply • Share ›

Porrapat Petchdamrongskul • a year ago

You shouldn't forget to do something like this.

npm init

Init package.json before install babel-loader

Then change

loader: 'babel',

to

loader: 'babel-loader',

If you found error.

Then it just fine.

• Reply • Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Privacy

Advertisement

QUICK LINKS - Explore popular categories

ENVATO TUTS+ ✚

JOIN OUR COMMUNITY ✚

HELP ✚

tuts+

25,508 Tutorials   1,106 Courses   20,734 Translations

Envato.com   Our products   Careers

Follow Envato Tuts+