React Setup, Part III: Webpack

Install and Configure Webpack on your Local Computer

BACKGROUND

You've installed Babel, but you haven't "plugged it in" to your React app yet. You need to set up a system in which your React app will automatically run through Babel and compile your JSX, before reaching the browser.

Also, JSX to JavaScript is just one of many transformations that will need to happen to your React code. You need to set up a "transformation manager" that will take your code and run it through all of the transformations that you need, in the right order. How do you make that happen?

There are a lot of different software packages that can make this happen. The most popular as of this writing is a program called `webpack`.

INSTALL WEBPACK

`webpack` is a module that can be installed with `npm`, just like `react` and `react-dom`. You'll also be installing two webpack-related modules named `webpack-dev-server` and `html-webpack-plugin`, respectively. We'll explain these a little more soon.

`webpack` should be saved in development mode, just like `babel`.

Install `webpack`, `webpack-dev-server`, and `html-webpack-plugin` with one of these two terminal commands:

```
Dave ~
$ npm install --save-dev webpack webpack-dev-server html-webpack-plugin
```

```
Dave ~
$ npm i -D webpack webpack-dev-server html-webpack-plugin
```

WEBPACK.CONFIG.JS

Alright! Webpack has been installed!

Webpack's job is to run your React code through various *transformations*. Webpack needs to know exactly what transformations it should use!

You can set that information by making a special webpack configuration file. This file must be located in the outermost layer of your root directory, and must be named **webpack.config.js**. It is where you will put all of the details required to make webpack operate.

In your root directory, create a new file named **webpack.config.js**.

CONFIGURE WEBPACK

Webpack is going to take your JavaScript, run it through some transformations, and create a new, transformed JavaScript file. *This* file will be the ones that the browser actually reads.

In order to do this, Webpack needs to know three things:

1. What JavaScript file it should transform.
2. Which transformations it should use on that file.
3. Where the new, transformed file should go.

Let's walk through those three steps!

First, in **webpack.config.js**, write:

```
module.exports = {};
```

All of webpack's configuration will go inside of that object literal!.

WHAT JAVASCRIPT FILE SHOULD WEBPACK TRANSFORM?

The first thing that webpack needs to know is an *entry point*. The *entry point* is the file that Webpack will transform.

Your entry point should be the outermost component class of your React project. It might look something like this:

```
var React = require('react');
var ReactDOM = require('react-dom');
var App = require('./components/App');

ReactDOM.render(
  <App />,
  document.getElementById('app')
);
```

In this example, webpack will transform the result of `<App />`. If `<App />`'s render function contains components from other files, then those components will be transformed as well. If you make your *entry point* the outermost component class of your app, then webpack will transform your entire app!

To specify an *entry point*, give `module.exports` a property named `entry`. `entry`'s *value* can be a filepath, or an array of filepaths if you would like to have more than one entry point. For this project, you will only need one.

In **webpack.config.js**, update `module.exports` to look like this:

```
module.exports = {
  entry: __dirname + '/app/index.js'
};
```

In `Node.js`, `__dirname` refers to the currently executing file. `__dirname + /app/index.js` will create a filepath pointing to the currently executing file, down into a folder named `app`, and landing on a file named `index.js`.

WHAT TRANSFORMATIONS SHOULD WEBPACK PERFORM?

Webpack can now successfully grab your outermost component class file, and therefore grab your entire React app. Now that webpack can grab all of this code, you need to explain what webpack should *do* with it once it's been grabbed.

You can tell webpack what to do with the code that it's grabbed by adding a second property to `module.exports`. This property should have a *name* of `module` and a *value* of an object literal containing a `loaders` array:

```
module.exports = {
  entry: __dirname + '/app/index.js',
  module: {
    loaders: []
  }
};
```

Each "loader" that you add to the *loaders* array will represent a transformation that your code will go through before reaching the browser.

WRITE A LOADER

Each *loader* transformation should be written as an object literal. Here's your first loader, empty for now:

```
module.exports = {
  entry: __dirname + '/app/index.js',
  module: {
    loaders: [
      {}
    ]
  }
};
```

Each loader object needs a property called `test`. The `test` property specifies which files will be affected by the loader:

```
module.exports = {
  entry: __dirname + '/app/index.js',
  module: {
    loaders: [
      {
        test: /\.js$/
      }
    ]
  }
};
```

The regular expression `/\.js$/` represents all strings that end with the pattern, ".js". That means that this *loader* will perform a transformation on all ".js" files.

In addition to "test", each *loader* transformation can have a property named `include` or `exclude`. You can use "exclude" to specify files that match the "test" criteria, that you *don't* want to be transformed. Similarly you can use "include" to specify files that *don't* fit the "test" criteria, that you *do* want to be transformed:

```
module.exports = {
  entry: __dirname + '/app/index.js',
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /node_modules/
      }
    ]
  }
};
```

The `node_modules` folder contains lots of JavaScript files that will be caught by your `/\.js$/` test. However, you *don't* want anything in the `node_modules` folder to be transformed. `node_modules` holds the code for React itself, along with the other modules that you've downloaded. You don't want to transform that!

The final property of each *loader* is what transformation that loader should perform! You specify a particular transformation with a property named `loader`:

```
module.exports = {
  entry: __dirname + '/app/index.js',
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      }
    ]
  }
};
```

In this example, you have a *loader* with three properties: `test`, `exclude`, and `loader`. Your *loader* will search for all files ending in ".js", excluding files in the `node_modules` folder. Whatever files it finds, it will run through the 'babel-loader' transformation.

Where does the string `'babel-loader'` come from? When you ran the command `npm install --save-dev babel-core babel-loader babel-preset-react`, you installed babel-loader into your `node_modules` folder. Your `loader` property will automatically be able to find it there. The magic of `npm`!

Add babel-loader to **webpack.config.js**.

WHAT SHOULD WEBPACK DO WITH THE TRANSFORMED JAVASCRIPT?

Alright! Now you have told webpack which files to grab, and how to transform those files. Webpack will grab your React app and run it through babel-loader, translating all of your JSX into JavaScript.

The final question is, where should the transformed JavaScript go?

Answer this by adding another property to `module.exports`. This property should have a name of `output`, and a value of an object:

```
module.exports = {
  entry: __dirname + '/app/index.js',
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      }
    ]
  },
  output: {}
};
```

The `output` object should have two properties: `filename` and `path`. `filename` will be the name of the new, transformed JavaScript file. `path` will be the filepath to where that transformed JavaScript file ends up:

```
module.exports = {
  entry: __dirname + '/app/index.js',
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      }
    ]
  },
  output: {
    filename: 'transformed.js',
    path: __dirname + '/build'
  }
};
```

This will save your transformed JavaScript into a new file named ***build/transformed.js***.

Whenever you're ready, continue to our next article!