Setting Up a React.js Environment Using Npm, Babel 6 and Webpack

Published Dec 15, 2015 Last updated Jan 18, 2017

Facebook has really changed the way we think about front-end UI development with the introduction of React. One of the main advantages of this component based approach is, it is easy to reason about as the view is just a function of props and state.

Though the learning curve of React is shallower when compared to that of its counterparts, one intimidating aspect for the beginners is the tools (Babel, Webpack) and libraries around it.

In fact, these tools are not required to use React and but in order to get the most out of the features of ES6, JSX and bundling, we need them. In this blog post, we are going to see how to setup a React development environment without being sidetracked by the tools.

A Disclaimer: The approach that I am going to share is just for beginners to understand how to get started with React, as going by this lean way has helped a lot when I started learning React.

Let's start from scratch

Create a new folder 'react-hello-world' and initialize it with npm.

```
mkdir react-hello-world
cd react-hello-world
npm init
```

Accept the default for all the prompts

Installing and Configuring Webpack

Webpack is a module bundler which takes modules with dependencies and generates static assets by bundling them together based on some configuration.

The support of loaders in Webpack makes it a perfect fit for using it along with React and we will discuss it later in this post with more details.

Let's start with installing webpack using npm

```
npm i webpack -S
```

Webpack requires some configuration settings to carry out its work and the best practice is doing it via a config file called *webpack.config.js*.

```
touch webpack.config.js
```

Update the config file as follows

```
var webpack = require('webpack');
var path = require('path');

var BUILD_DIR = path.resolve(__dirname, 'src/client/public');
var APP_DIR = path.resolve(__dirname, 'src/client/app');

var config = {
  entry: APP_DIR + '/index.jsx',
  output: {
    path: BUILD_DIR,
    filename: 'bundle.js'
  }
};

module.exports = config;
```

The minimalist requirement of a Webpack config file is the presence of entry and output properties.

The `APP_DIR` holds the directory path of the React application's codebase and the `BUILD_DIR` represents the directory path of the bundle file output.

As the name suggests, *entry* specifies the entry file using which the bundling process starts. If you are coming from C# or Java, it's similar to the class that contains *main* method. Webpack supports multiple entry points too. Here the *index.jsx* in the *src/client/app* directory is the starting point of the application

The *output* instructs Webpack what to do after the bundling process has been completed. Here, we are instructing it to use the *src/client/public* directory to output the bundled file with the name *bundle.js*

Let's create the *index.jsx* file in the *./src/client/app* and add the following code to verify this configuration.

```
console.log('Hello World!');
```

Now in the terminal run the following command

```
./node_modules/.bin/webpack -d
```

The above command runs the webpack in the development mode and generates the *bundle.js* file and its associated map file *bundle.js.map* in the *src/client/public* directory.

To make it more interactive, create an *index.html* file in the *src/client* directory and modify it to use this *bundle.js* file

```
<html>
  <head>
    <meta charset="utf-8">
    <title>React.js using NPM, Babel6 and Webpack</title>
  </head>
  <body>
    <div id="app" />
    <script src="public/bundle.js" type="text/javascript"></script>
  </body>
</html>
```

Now if you open the browser, you can see the *Hello World!* in the console log.

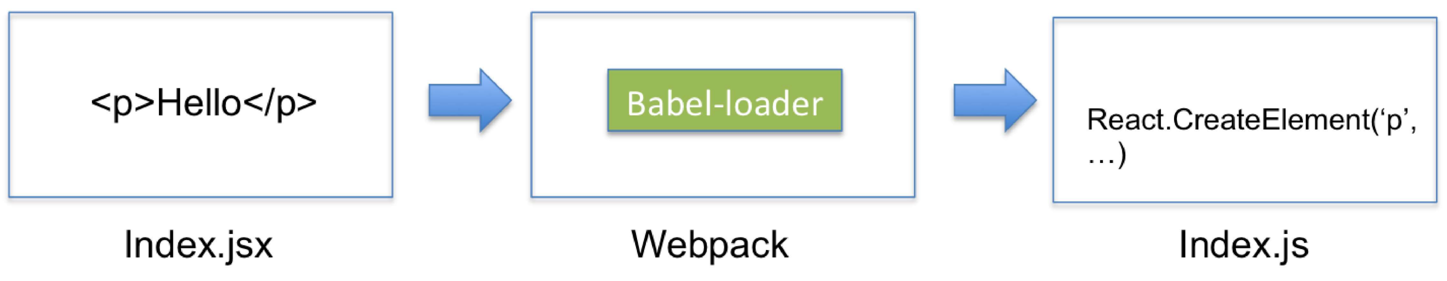
Note: There is a webpack loader called html-loader which automatically creates this html file with the correct location of *bundle.js*.

Setting Up Babel-Loader

As we have seen in the beginning, by using JSX and ES6 we can be more productive while working with React. But the JSX syntax and ES6, are not supported in all the browsers.

Hence, if we are using them in the React code, we need to use a tool which translates them to the format that has been supported by the browsers. It's where babel comes into the picture.

While installing Webpack, we touched a little on loaders. Webpack uses loaders to translate the file before bundling them



To setup, install the following npm packages

```
npm i babel-loader babel-preset-es2015 babel-preset-react -S
```

The *babel-preset-es2015* and *babel-preset-react* are plugins being used by the *babel-loader* to translate ES6 and JSX syntax respectively.

As we did for Webpack, *babel-loader* also requires some configuration. Here we need to tell it to use the ES6 and JSX plugins.

Create a `.babelrc` file and update it as below

```
touch .babelrc
```

```
{
  "presets" : ["es2015", "react"]
}
```

The next step is telling Webpack to use the babel-loader while bundling the files

open *webpack.config.js* file and update it as below

```
// Existing Code ....
var config = {
  // Existing Code ....
  module : {
    loaders : [
      {
        test : /\.jsx?/,
        include : APP_DIR,
        loader : 'babel'
      }
    ]
  }
}
```

The *loaders* property takes an array of loaders, here we are just using *babel-loader*. Each *loader* property should specify what are the file extension it has to process via the *test* property. Here we have configured it to process both *.js* and *.jsx* files using the regular expression. The *include* property specifies what is the directory to be used to look for these file extensions. The *loader* property represents the name of the loader.

Now we are done with all the setup. Let's write some code in React.

Hello React

Use npm to install react and react-dom

```
npm i react react-dom -S
```

Replace the existing `console.log` statement in the *index.jsx* with the following content

```
import React from 'react';
import {render} from 'react-dom';

class App extends React.Component {
  render () {
    return <p> Hello React!</p>;
  }
}

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

Then run the following command to update the bundle file with the new changes

```
./node_modules/.bin/webpack -d
```

Now, if you open the *index.html* in the browser, you can see *Hello React*

Adding Some Complexity

Making Webpack Watch Changes

Running the webpack command every time when you change the file is not a productive workflow. We can easily change this behavior by using the following command

```
./node_modules/.bin/webpack -d --watch
```

Now Webpack is running in the watch mode, which will automatically bundle the file whenever there is a change detected. To test it, change *Hello React* to something else and refresh the *index.html* in the browser. You can see your new changes.

If you don't like refreshing the browser to see the changes, you can use react-hot-loader!

Using npm as a tool runner

The command `./node_modules/.bin/webpack` can be made even simpler by leveraging npm.

Update the `packages.json` as below

```
{
  // ...
  "scripts": {
    "dev": "webpack -d --watch",
    "build" : "webpack -p"
  },
  // ...
}
```

Now the command `npm run build` runs Webpack in production mode, which minimizes the bundle file automatically, and
the command `npm run dev` runs the Webpack in the watch mode.

Adding some files

In the sample, we have seen only one Component called *App*. Let's add some more to test the bundling setup.

Create a new file *AwesomeComponent.jsx* and update it as below

```
import React from 'react';

class AwesomeComponent extends React.Component {

  constructor(props) {
    super(props);
    this.state = {likesCount : 0};
    this.onLike = this.onLike.bind(this);
  }

  onLike () {
    let newLikesCount = this.state.likesCount + 1;
    this.setState({likesCount: newLikesCount});
  }

  render() {
    return (
      <div>
        Likes : <span>{this.state.likesCount}</span>
        <div><button onClick={this.onLike}>Like Me</button></div>
      </div>
    );
  }

}

export default AwesomeComponent;
```

Then include it in the *index.jsx* file

```
// ...
import AwesomeComponent from './AwesomeComponent.jsx';
// ...
class App extends React.Component {
  render () {
    return (
      <div>
        <p> Hello React!</p>
        <AwesomeComponent />
      </div>
    );
  }
}

// ...
```

If your Webpack is already running in watch mode, then refresh the browser to see the AwesomeComponent in action!

Summary

In this blog post, we have seen a lean approach for setting up a development environment to work with React. In the next blog post, we will be extending this example to implement the flux architecture using Alt. You can get the source code associated with this blog post can be found in my github repository.