Exercise 02 – Setting Up ReactJS

Objectives

To be able to set up a ReactJS project using npm (Node Package Manager), Babel and Webpack.

To compare the differences to the set up using the create-react-app package binary method.

Overview

This exercise will give you the commands and information needed to set up a ReactJS project using the command line (or terminal) interface and then check to see if the setup has been successful.

You will then use the create-react-app package binary to set up and project and compare the differences.

Exercise Instructions

Part 1 - Setting up a project from scratch

Dependency installation

1. On the command line or terminal, navigate to the folder: **Exercises/EG02_SettingUpReactJS/starter**
2. On the command line or terminal enter:

```
npm init -y
```

This creates a **package.json** file in the folder, allowing anyone viewing the project the chance to install the exact same dependencies. The **–y** extension automatically accepts the defaults for each key.

3. Install the react packages and a polyfill from babel by entering the following:

```
npm install --save react react-dom babel-polyfill
```

This will place a **node_modules** folder into the project. This folder includes the **react**, **react-dom** and **babel-polyfill** packages as dependencies that will be included as part of any build of the project.

4. Install **babel** and its required plugins, by entering the following:

```
npm install --save-dev babel-core babel-loader babel-preset-react
babel-preset-env
```

This process adds to the **node_modules** folder with all of the babel files as development dependencies. These are not required to be part of any build of the project.

5. Install the webpack module bundler, its command-line interface and development server by typing:

```
npm install --save-dev webpack webpack-dev-server webpack-cli
```

The server provides a development environment that will be configured to update automatically. Again, this process adds relevant files to the **node_modules** folder as development dependencies.

Creating the Base Files

6. In a suitable text editor, open **Exercises/EG02_SettingUpReactJS/Starter** as a project and then create index.html in its root. Input the following code:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>My First React App</title>
    </head>
    <body>
        <div id="app"></div>
        <script src="index.js"></script>
    </body>
</html>
```

7. Create **main.js** in the project root and enter:

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
    <h1>Hello World</h1>,
    document.querySelector('#app')
);
```

These two files essentially create a page to view (**index.html**) and then some content that will be placed into the div with the id of app (through the render method in **main.js**). We will discuss this further later in the course.

Configure Webpack

This is by far the most difficult part of the exercise and in general. Webpack requires a configuration object that is provided in a file called webpack.config.js. As you create this object, each section will be explained.

8. Create a file in the project root called **webpack.config.js.**
9. Open this file for editing and then type the following code:

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

Within this object we need to tell webpack about **entry**, **output**, **resolve**, **devServer**, **devtool** and **module**.

10. The **entry** key/value pair identifies the entry JavaScript file and ensure that **babel-polyfill** is used too. Enter this code into the **module.exports** object:

```
entry: ['babel-polyfill', './main.js'],
```

11. **output** is an object that defines where Webpack should put the bundled file. In this instance we ask it to put the bundled file into the same folder with a name of **index.js**. Add this under **entry** in the object:

```
output: {
    path: __dirname,
    filename: 'index.js'
},
```

12. **resolve** tells Weback to use the defined file extensions when working with file imports. Add this under **output** in the object:

```
resolve: {
    extensions: ['.js', '.jsx']
},
```

13. In this set up we are going to use Webpack's development server. This creates a temporary copy of the bundled file and serves this. It can be set up to auto-refresh the page as we make changes to our JSX files. We also define the port we wish to serve the application to on localhost. Add this under **resolve** in the object:

```
devServer: {
    inline: true,   // Auto-refresh page on the fly
    port: 8080      // Arbitrarily chosen for demo
},
```

14. The next item in the configuration object forces Webpack to provide source map files. This allows us to examine and debug our JSX code directly in the browser using the normal developer tools. Add this next:

```
devtool: 'source-map',
```

15. The final item in the configuration object tells Webpack how to deal with a **module** (i.e. individual JS and JSX files. We provide an array of **rules** which specify the file extensions that Webpack will look at and tell it not to look in the **node_modules** folder. Once a file is established to bundle, we tell Webpack to pass it through **Babel**, using its **env** preset and its **react** preset to ensure transpilation goes smoothly! Add this next:

```
    module: {
        rules: [
            {
                test: /\.jsx?$/,
                exclude: /node_modules/,
                use: {
                    loader: 'babel-loader',
                    options: {
                        presets: ['env', 'react']
                }
            }
        ]
    }
```

That's the Webpack configuration done. The next thing we need to do is create some commands for **npm** to use with our project.

npm scripts

16. Open the **package.json** file and replace the line "**test**" in "**scripts**" with:

```
"start": "webpack-dev-server --mode development",
```

17. On the next line, add another command to the scripts:

```
"build": "webpack -p"
```

This command will create the index.js file in the location specified in the output object of the config file.

18. Save the **package.json** file.
19. Build the application by running the following from the command line or terminal:

```
npm run build
```

This essentially compiles the application for first running. The **index.js** file that is created is the production version of your application for deployment. During development, this file exists as a virtual file on the webpack-dev-server. Have a look in the file, it should have a lot of unreadable JavaScript that has been bundled from your files and dependencies, transpiled into ES5 JS and then minified. You may also notice that an **index.map.js** file has been created. This is a result of the **devtool** line in the webpack config file.

20. To view the application, execute the following on the command line or terminal:

```
npm start
```

21. Open a browser and navigate to **http://localhost:8080.**

'Hello, World!' should be displayed in the browser window.

Checking the refresh works

22. Open the **main.js file** and replace the text "**Hello, World!**" with your own message.
23. Switch back to the browser and observe that the text has changed. If it hasn't and you can't work out why, call your instructor.

Part 2 - Setting up a project with create-react-app

1. Open a command line at the **starter** folder for **EG02**.
2. Run the following command to set up a React project with zero configuration needed

```
npx create-react-app my-app
```

At this point, I suggest you go and stretch your legs whilst it does its thing!

3. Once installation is complete, examine the project files and see if you can work out how it all goes together.
4. When you have had a good look at the files, see them in action by running the command:

```
npm start
```

Your browser should come alive with a React application that took you exactly **38** key-strokes!

If you have time, see if you can change the title of the application and some of the text that is displayed!