**How to Create a React app from scratch using Webpack 4**

<https://medium.freecodecamp.org/part-1-react-app-from-scratch-using-webpack-4-562b1d231e75>

**“” npm run start ‘’**

**‘’ npm run build ‘’**

For the past three weeks, I have been trying to create a React app from scratch to understand the set-up with Webpack. My aim was to set up a simple configuration which can then be grown upon. It’s been a struggle to understand Webpack. But thanks to this [tutorial](https://www.valentinog.com/blog/webpack-4-tutorial/) by [Valentino Gagliardi,](https://twitter.com/gagliardi_vale) I’m much enlightened.

What I’m planning to do is to make a search functionality with some fake JSON data (or real). In this blog post, I will go through the set up of my project. In the next one, I am planning to show how to set up testing. I would also like to add a server to this using Node.js, but not sure if the scope of my project would need that.

(**Note**: I have provided my Webpack setup at the end of this blog post)

Without further ado, let’s get on with the set up!

Make a **new project** and **cd** into it:

mkdir react\_search  
cd react\_search

Create a **package.json** file:

npm init

If you want to skip all the questions, add the -y flag:

npm init -y

We need to install **webpack** as a dev dependency and **webpack-cli**so that you can use webpack in the command line:

npm i webpack webpack-cli -D

* i: install
* -D: — save-dev

Create a **src folder**with **index.js** and place the following code as an example:

console.log("hello");

Now add the following scripts to your package.json (in bold):

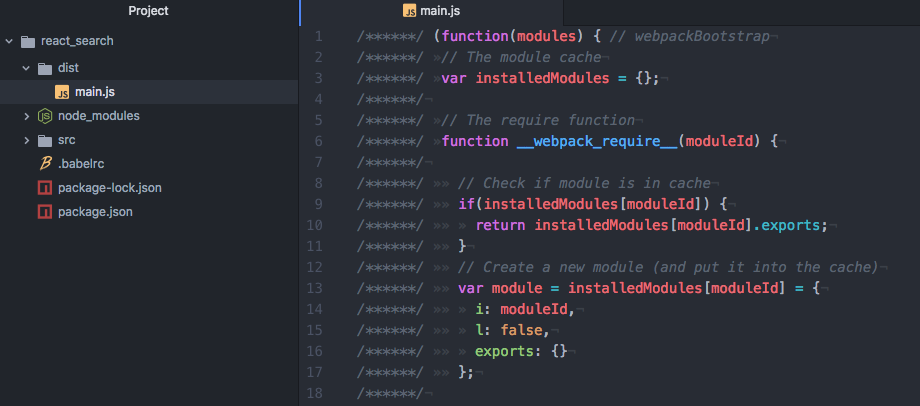
{  
 "name": "react\_search",  
 "version": "1.0.0",  
 "description": "Search app using React",  
 "main": "index.js",  
 **"scripts": {  
 "start": "webpack --mode development",  
 "build": "webpack --mode production"**  
 **},**  
 "keywords": [],  
 "author": "",  
 "license": "ISC",  
 "devDependencies": {  
 "webpack": "^4.0.1",  
 "webpack-cli": "^2.0.10"  
 }  
}

Webpack 4 now has two modes, **development**and**production** where code is minimised in the latter.

See it for yourself by running:

npm run start

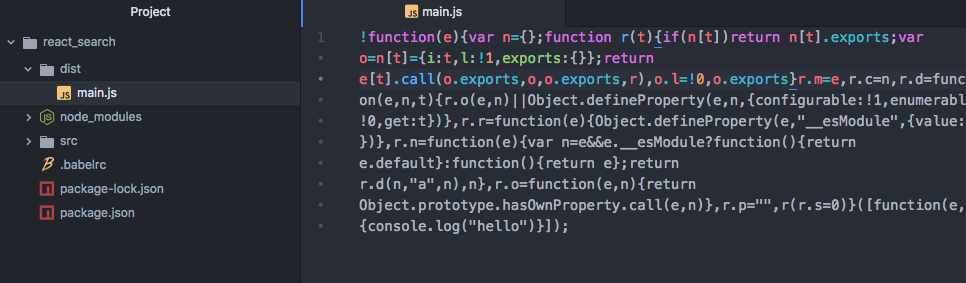
This will create a **dist**folder with **main.js** file inside (containing your src code).



If you now run:

npm run build

The following output is now like this:



### Setting Up React and Babel

To work with React, we need to install it along with Babel. This will transpile the code from ES6 to ES5, as not all browsers support ES6 yet (for example Internet Explorer).

Install **react** and **react-dom** as a dependency

npm i react react-dom -S

* -S: — save

Then install **babel-core**, **babel-loader**, **babel-preset-env** and **babel-preset-react** as a dev dependency:

npm i babel-core babel-loader babel-preset-env babel-preset-react -D

* **babel-core**: Transforms your ES6 code into ES5
* **babel-loader**: Webpack helper to transform your JavaScript dependencies (for example, when you import your components into other components) with Babel
* **babel-preset-env**: Determines which transformations/plugins to use and polyfills (provide modern functionality on older browsers that do not natively support it) based on the browser matrix you want to support
* **babel-preset-react**: Babel preset for all React plugins, for example turning JSX into functions

We need to create a **webpack.config.js** file to state the rules for our babel-loader.

We then need to make a separate file called **.babelrc** to provide the options for babel-loader. You can include it in the webpack.config.js file, but I have seen that most projects have this separated. This results in clearer readability, and it can be used by other tools unrelated to webpack. When you state that you’re using babel-loader in your webpack config, it will look for .babelrc file if there is one.

Next, change your **index.js** file to render a component:

We will also need to create an **index.html** file in the **src** folder where we can add our section element with id index. This is where we render our main react component:

Now we need to install **html-webpack-plugin** and use this in our webpack config file. This plugin generates an HTML file with <script> injected, writes this to **dist/index.html,** and minifies the file.

Install **html-webpack-plugin** as a dev dependency:

npm i html-webpack-plugin -D

Update the webpack config like so:

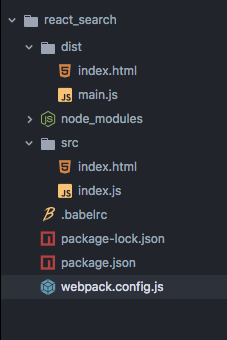
You can also input the plugin like this:

plugins: [  
 new HtmlWebPackPlugin({  
 template: "./src/index.html",  
 filename: "./index.html"  
 });  
]

But I prefer to extract this into a variable so I can see the list of plugins I am using.

The value I am giving the template key is where I am looking for my HTML file. The filename value is the name of the minified HTML that will be generated in the dist folder.

If you now run npm run start you should see **index.html** being generated in the dist folder.



Run open dist/index.html and you should see “Hello React” in your browser.

### Setting up webpack-dev-server

It is a bit tedious to keep running this command every time you want to see your changes in the browser. To have webpack “watch” our changes and thus refresh whenever we have made changes to any of our components, we can use **webpack-dev-server** module.

Go ahead and install this as a dev dependency

npm i webpack-dev-server -D

Then change your package.json start scripts like so (in bold):

{  
 "name": "react\_search",  
 "version": "1.0.0",  
 "description": "Search app using React",  
 "main": "index.js",  
 "scripts": {  
  **"start": "webpack-dev-server --mode development --open",**  
 "build": "webpack --mode production"  
 },  
 "keywords": [],  
 "author": "",  
 "license": "ISC",  
 "dependencies": {  
 "react": "^16.2.0",  
 "react-dom": "^16.2.0"  
 "devDependencies": {  
 "babel-core": "^6.26.0",  
 "babel-loader": "^7.1.4",  
 "babel-preset-env": "^1.6.1",  
 "babel-preset-react": "^6.24.1",  
 "html-webpack-plugin": "^3.0.6",  
 "webpack": "^4.1.1",  
 "webpack-cli": "^2.0.10",  
 "webpack-dev-server": "^3.1.0"  
 }  
}

If you now run npm run start you should see **localhost:8080** open up in your default browser — that’s what the —-open flag is for. Now everytime you make changes, it will refresh the page.

You can also add a --hot flag to your npm start script which will allow you to only reload the component that you’ve changed instead of doing a full page reload. This is [Hot Module Replacement](https://webpack.js.org/concepts/hot-module-replacement/#src/components/Sidebar/Sidebar.jsx).

### Setting up CSS

The last part involves setting up our CSS. As we will be importing CSS files into our React components, we need **css-loader** module to resolve them. Once that’s resolved, we also need a **style-loader** to inject this into our DOM — adding a <style> tag into the <head> element of our HTML.

Go ahead and install both of these modules as a dev dependency:

npm i css-loader style-loader -D

We then need to update our webpack.config.js file like so:

Note that the order of adding these loaders is important. First, we need to resolve the CSS files before adding them to the DOM with the style-loader. By default, webpack uses the loaders from the right (last element in the array) to the left (first element in the array).

#### Making CSS modular

We can also make CSS modular using webpack. This means class name will be scoped locally and specific to only the component in question.

To do this, we can provide some options to css-loader:

As we need to give options, each loader is now an object with a key-value pair. To enable CSS modules, we need to set **module** option for css-loader to be **true**. The **importLoaders** option configures how many loaders before css-loader should be applied. For example, sass-loader would have to come before css-loader.

The **localIdentName** allows you to configure the generated identification.

* **[name]** will take the name of your component
* **[local]** is the name of your class/id
* **[hash:base64]** is the randomly generated hash which will be unique in every component’s CSS

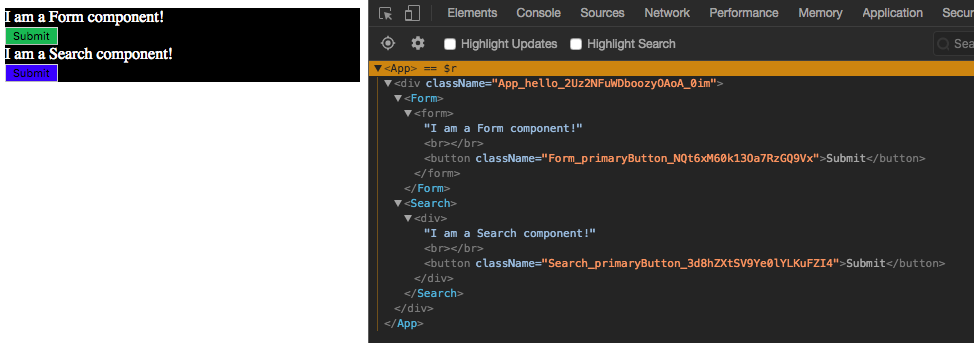
To make this a bit more visual, I’ll give you an example. Say I have a component named Form and I have a button with a CSS class primaryButton. I also have another component called Search and a button in it with a CSS class primaryButton. However, both of these classes have different CSS:

**Form button**  
.primaryButton {  
 background-color: green;  
}

**Search button**  
.primaryButton {  
 background-color: blue;  
}

When webpack bundles your application, depending on which CSS comes latest, both of your buttons could have the color green or blue instead of Form having green and Search having blue.

This is where the localIdentName comes into place. With this, once your application is bundled, your buttons will have a unique class name!



As you can see, the button class name in the Form component is different to the one in the Search component — their naming starts with the name of the component, class name, and unique hash code.

So with this, you won’t have to worry about whether you have given the same class name throughout your whole application — you only have to worry about whether you have used it in the same component.

This concludes the first part of setting a React app from scratch. In the next blog post, I aim to explain how to set up tests for TDD and how to write them.

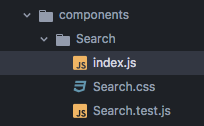
Please let me know if something is unclear and I’ll explain the best as I can. I value and welcome constructive feedback as this helps me to improve :)

Hope this helps!

### EDIT

#### Importing CSS

I’ve had a few comments asking me how they can render CSS which I didn’t touch on previously. What you need to do is import the CSS file in your React component. For example, say you have a Search component and this is your tree directory:



You will need to import your CSS file in your Search component like so:

import style from "./Search.css"

You can then apply different CSS class styles such as:

const Search = () => {  
 return <div className={style.**nameOfYourCSSClass**}>  
 Hello Search Component :)  
 </div>  
}

You don’t have to call it style but what I found is that most people have given it this name in their projects.