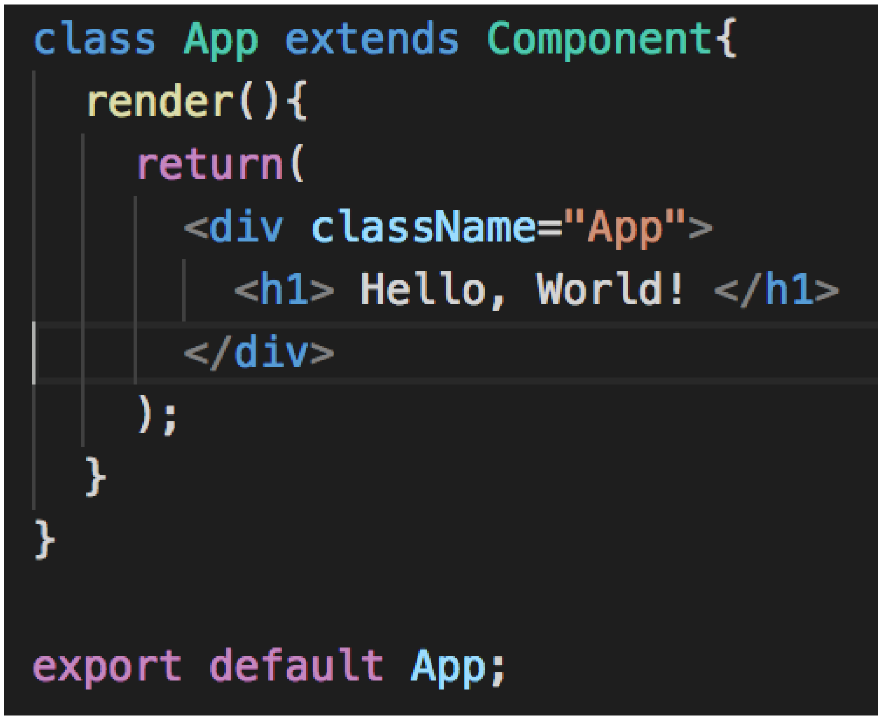
Jedai Saboteur Follow
Software Engineer / MERN Enthusiast / Layabout Scoundrel
Apr 20 · 8 min read

Creating a React App… From Scratch.



```
class App extends Component{
  render(){
    return(
      <div className="App">
        <h1> Hello, World! </h1>
      </div>
    );
  }
}

export default App;
```

React doesn't 'just work' out of the box. It uses keywords and syntax that node (as of v. 9.3.0, which we're using for this tutorial) hasn't quite caught up to just yet. It requires a fair bit of setup that can be cumbersome to figure out, and Facebook has provided an option that makes starting a React app easy. Why bother, right?

The thing is, create-react-app abstracts a lot of what makes a React app work away from you—at least without ejecting it and having to tweak all of the options by hand. There are a number of reasons you may want to make your own implementation, or at least have some idea of what it's doing under the hood.

As I mentioned, there are a couple of hurdles to starting a React app. The first is that node can't process all of the syntax (such as import/export and jsx). The second is that you'll either need to build your files or serve them somehow during development for your app to work—This is especially important in the latter situation.

Luckily, we can handle these issues with Babel and Webpack.

Setup

To get started, create a new directory for your React app. Then, initialize your project with `npm init` and open it in an editor of your choice. It's also a pretty good time to `git init` as well. In your new project folder, create the following structure:

```
.
+-- public
+-- src
```

Thinking ahead a little bit, we'll eventually want to build our app and we'll probably want to exclude the built version and our node modules from commits, so let's go ahead and add a `.gitignore` file excluding (at least) the directories `node_modules` and `dist` .

Our `public` directory will handle any static assets, and most importantly houses our `index.html` file, which react will utilize to render your app. The following code was sourced from the react documentation with some very slight modifications. Feel free to copy the following HTML markup into a new file `index.html` inside of the `public` directory.

```
1   <!-- sourced from https://raw.githubusercontent.com/r
2   <!DOCTYPE html>
3   <html>
4
5   <head>
6     <meta charset="UTF-8" />
7     <meta name="viewport" content="width=device-width,
8     <title>React Starter</title>
9   </head>
10
11  <body>
12    <div id="root"></div>
13    <noscript>
14      You need to enable JavaScript to run this app.
```

The most important lines to pay attention to are 10, which is the root our React app will hook into, and line 14, which references our (soon to be) built react app. You can name your built script whatever you like, but I'll be using `bundle.js` for this tutorial.

Now that we've got our HTML page set up, we can start getting serious. We're going to need to set up a few more things. First, we need to make sure the code we write can be compiled, so we'll need Babel.

Babel

Go ahead and run `npm install --save-dev @babel/core@7.1.0 @babel/cli@7.1.0 @babel/preset-env@7.1.0 @babel/preset-react@7.0.0`.

`babel-core` is the main babel package—We need this for babel to do any transformations on our code. `babel-cli` allows you to compile files from the command line. `preset-react` and `preset-env` are both presets that transform specific flavors of code—in this case, the `env` preset allows us to transform ES6+ into more traditional javascript and the `react` preset does the same, but with JSX instead.

In the project root, create a file called `.babelrc`. Here, we're telling babel that we're going to use the `env` and `react` presets.

```
{
  "presets": ["@babel/env", "@babel/preset-react"]
}
```

Babel also has a ton of plugins available that can be used if you only need to transform specific features or some feature you need isn't covered by `env` . We won't worry about those for now, but you can check them out here.

Webpack

Now we need to acquire and configure Webpack. We'll need a few more packages, and you'll want to save these as dev dependencies: `npm install --save-dev webpack@4.19.1 webpack-cli@3.1.1 webpack-dev-server@3.1.8 style-loader@0.23.0 css-loader@1.0.0 babel-loader@8.0.2`.

Webpack uses loaders to process different types of files for bundling. It also works easily alongside the development server that we're going to use to serve our React project in development and reload browser pages on (saved) changes to our React components. In order to utilize any of this though, we'll need to configure Webpack to use our loaders and prepare the dev server.

Create a new file at the root of the project called `webpack.config.js` . This file exports an object with webpack's configuration.

```
 1  const path = require("path");
 2  const webpack = require("webpack");
 3
 4  module.exports = {
 5    entry: "./src/index.js",
 6    mode: "development",
 7    module: {
 8      rules: [
 9        {
10          test: /\.(js|jsx)$/,
11          exclude: /(node_modules|bower_components)/,
12          loader: "babel-loader",
13          options: { presets: ["@babel/env"] }
14        },
15        {
16          test: /\.css$/,
17          use: ["style-loader", "css-loader"]
18        }
19      ]
20    },
21    resolve: { extensions: ["*", ".js", ".jsx"] },
22    output: {
```

Let's take a quick walk through this: `entry` (line 5) tells Webpack where our application starts and where to start bundling our files. The following line lets webpack know we're working in development mode —This saves us from having to add a mode flag when we run the development server.

The `module` object helps define how your exported javascript modules are transformed and which ones are included according to the given array of `rules` .

Our first rule is all about transforming our ES6 and JSX syntax. The `test` and `exclude` properties are conditions to match file against. In this case, it'll match anything outside of the `node_modules` and `bower_components` directories. Since we'll be *transforming* our `.js` and `.jsx` files as well, we'll need to direct Webpack to use Babel. Finally, we specify that we want to use the `env` preset in options.

The next rule is for processing CSS. Since we're not pre-or-post-processing our CSS, we just need to make sure to add `style-loader` and `css-loader` to the `use` property. `css-loader` requires `style-loader` in order to work. `loader` is a shorthand for the `use` property, when only one loader is being utilized.

The `resolve` property allows us to specify which extensions Webpack will resolve—this allows us to import modules without needing to add their extensions.

The `output` property tells Webpack where to put our bundled code. The `publicPath` property specifies what directory the bundle should go in, and also tells `webpack-dev-server` where to serve files from.

The `publicPath` property is a special property that helps us with our dev-server. It specifies the public URL of the the directory—at least as far as `webpack-dev-server` will know or care. If this is set incorrectly, you'll get 404's as the server won't be serving your files from the correct location!

We set up `webpack-dev-server` in the `devServer` property. This doesn't require much for our needs—just the location we're serving static files from (such as our `index.html`) and the port we want to run the server on. Note that `devServer` also has a `publicPath` property. This `publicPath` tells the server where our bundled code actually is.

That last bit might have been a little confusing—Pay really close attention here: `output.publicPath` and `devServer.publicPath` are different. Read both entries. Twice.

Finally, since we want to use Hot Module Replacement so we don't have to constantly refresh to see our changes. All we do for that in terms of this file is instantiate a new instance of the plugin in the `plugins` property and make sure that we set `hotOnly` to `true` in `devServer`. We still need to set up one more thing in React before HMR works, though.

We're done with the heavy setup. Now let's get React working!

React

First, we'll need to get two more packages: `react@16.5.2` and `react-dom@16.5.2`. Go ahead and save those as regular dependencies.

We'll need to tell our React app where to hook into the DOM (in our `index.html`). Create a file called `index.js` in your `src` directory. This is a very small file that does a lot in terms of your React app. Check it out.

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App.js";
ReactDOM.render(<App />, document.getElementById("root
```

`ReactDOM.render` is the function that tells React what to render and where to render it—In this case, we're rendering a component called `App` (which we'll create soon), and it's being rendered at the DOM element with the ID root (line 10 of `index.html`).

Now, create another file in `src` called `App.js`. If you've worked with React using `create-react-app` this part should be extremely familiar. This file is just a React component.

```
import React, { Component} from "react";
import "./App.css";

class App extends Component{
  render(){
    return(
      <div className="App">
        <h1> Hello, World! </h1>
      </div>
    );
```

While we're still here, I *did* mention that webpack also processes CSS (and we *are* requiring it our component). Let's add a really simple stylsheet to the `src` directory.

```css
.App {
  margin: 1rem;
  font-family: Arial, Helvetica, sans-serif;
}
```

Your final project structure should look like the following, unless you changed some names along the way:

```
.
+-- public
| +-- index.html
+-- src
| +-- App.css
| +-- App.js
| +-- index.js
+-- .babelrc
+-- .gitignore
+-- package-lock.json
+-- package.json
+-- webpack.config.js
```

We now have a functioning react app! We can start our dev server by executing `webpack-dev-server --mode development` in the terminal. I would advise putting it in your `start` script in `package.json` to save yourself nine whole keystrokes.

Finishing HMR

If you run the server now, you'll notice none of your changes actually cause anything to happen in the client. What gives?

Well, HMR needs to know what to actually replace and currently we haven't given it anything. For that, we're going to use a package one of the folks on the react team have provided us: `react-hot-loader@4.3.11`.

You can install this as regular dependency—as per the documentation

> *Note: You can safely install react-hot-loader as a regular dependency instead of a dev dependency as it automatically ensures it is not executed in production and the footprint is minimal.*

Now, import react-hot-loader in App.js and mark the exported object as hot-reloaded by modifying to code as follows.

```
import React, { Component} from "react";
import {hot} from "react-hot-loader";
import "./App.css";

class App extends Component{
  render(){
    return(
      <div className="App">
        <h1> Hello, World! </h1>
      </div>
    );
```

When you run your app now changes to the code should update the client immediately after saving.

Last Details

You might notice something interesting (or maybe startling) about starting your project: Built files never show up in your `dist` directory. See, `webpack-dev-server` is actually serving the bundled files from memory—once the server stops, they're gone. To actually build your files, we're going to utilize `webpack` proper—add a script called `build` in your `package.json` with the following command: `webpack --mode development`. You can replace `development` with `production`, but if you completely omit `--mode`, it will fall back to the latter and give you a warning.

That just about covers everything you need to be able to render a basic React app, without having to touch `create-react-app`. There's still more to add to the implementation to make it more *complete* though—For instance, images aren't set up to be processed by Webpack, but there's a loader for that. I'll leave that implementation to you. After all, if you don't need or want to serve files, well, *that's just bloat, right?*

I hope this article helped shine some light on what it takes to get a React app working, and how the basics work under the hood. I didn't dive too deeply into details about Babel and Webpack, but please explore any of the myriad links peppered throughout the article or right into their documentation. They're awesome tools that seem more intimidating at first glance than they really are and they can help take your development to the next level.

If you're still unclear on any of that or want another reference, here's the repository on github . You can also check out an older implementation (it goes a bit more in depth) or say hi on twitter.

This article was edited on April 24, 2018 to include package versions, May 13, 2018 to reflect a bugfix regarding webpack-dev-server, June 16, 2018 to reflect updates to React and other packages used throughout this guide, and September 23, 2018 for the same reason.

• • •