Using Webpack with Create React App

In most of our earlier projects, we loaded React with `script` tags in our apps' `index.html` files:

```
<script src='vendor/react.js'></script>
<script src='vendor/react-dom.js'></script>
```

Because we've been using ES6, we've also been loading the Babel library with `script` tags:

```
<script src='vendor/babel-standalone.js'></script>
```

With this setup we've been able to load in any ES6 JavaScript file we wanted in `index.html`, specifying that its type is `text/babel`:

```
<script type='text/babel' src='./client.js'></script>
```

Babel would handle the loading of the file, transpiling our ES6 JavaScript to browser-ready ES5 JavaScript.

> If you need a refresher on our setup strategy so far, we detail it in Chapter 1.

We began with this setup strategy because it's the simplest. You can begin writing React components in ES6 with little setup.

However, this approach has limitations. For our purposes, the most pressing limitation is the lack of support for **JavaScript modules**.

JavaScript modules

We saw modules in earlier apps. For instance, the time tracking app had a Client module. That module's file defined a few functions, like `getTimers()`. It then set `window.client` to an object that "exposed" each function as a property. That object looked like this:

```
// `window.client` was set to this object
// Each property is a function
{
  getTimers,
  createTimer,
  updateTimer,
  startTimer,
  stopTimer,
  deleteTimer,
};
```

This Client module only exposed these functions. These are the Client module's **public methods**. The file `public/js/client.js` also contained other function definitions, like `checkStatus()`, which verifies that the server returned a 2xx response code. While each of the public methods uses `checkStatus()` internally, `checkStatus()` is kept **private**. It is only accessible from within the module.

That's the idea behind a module in software. You have some self-contained component of a software system that is responsible for some discrete functionality. The module exposes a limited interface to the rest of the system, ideally the minimum viable interface the rest of the system needs to effectively use the module.

In React, **we can think of each of our individual components as their own modules**. Each component is responsible for some discrete part of our interface. React components might contain their own state or perform complex operations, but the interface for all of them is the same: they accept inputs (props) and output their DOM representation (`render`). Users of a React component need not know any of the internal details.

In order for our React components to be truly modular, we'd ideally have them live in their own files. In the upper scope of that file, the component might define a styles object or helper functions that only the component uses. But we want our component-module to only expose the component itself.

Until ES6, modules were not natively supported in JavaScript. Developers would use a variety of different techniques to make modular JavaScript. Some solutions only work in the browser, relying on the browser environment (like the presence of `window`). Others only work in Node.js.

Browsers don't yet support ES6 modules. But ES6 modules are the future. The syntax is intuitive, we avoid bizarre tactics employed in ES5, and they work both in and outside of the browser. Because of this, the React community has quickly adopted ES6 modules.

> If you look at `time_tracking_app/public/js/client.js` , you'll get an idea of how strange the techniques for creating ES5 JavaScript modules are.

However, due to the complexity of module systems, we can't simply use ES6's import/export syntax and expect it to "just work" in the browser, even with Babel. More tooling is needed.

For this reason and more, the JavaScript community has widely adopted **JavaScript bundlers**. As we'll see, JavaScript bundlers allow us to write modular ES6 JavaScript that works seamlessly in the browser. But that's not all. Bundlers pack numerous advantages. Bundlers provide a strategy for both organizing and distributing web apps. They have powerful toolchains for both iterating in development and producing production-optimized builds.

While there are several options for JavaScript bundlers, the React community's favorite is **Webpack**.

However, bundlers like Webpack come with a significant trade-off: They add complexity to the setup of your web application. Initial configuration can be difficult and you ultimately end up with an app that has more moving pieces.

In response to setup and configuration woes, the community has created loads of boilerplates and libraries developers can use to get started with more advanced React apps. But the React core team recognized that as long as there wasn't a core team sanctioned solution, the community was likely to remain splintered. The first steps for a bundler-powered React setup can be confusing for novice and experienced developers alike.

The React core team responded by producing the **Create React App** project.

Create React App

The create-react-app (https://github.com/facebookincubator/create-react-app) library provides a command you can use to initiate a new **Webpack-powered React app**:

```
$ create-react-app my-app-name
```

The library will configure a "black box" Webpack setup for you. It provides you with the benefits of a Webpack setup while abstracting away the configuration details.

Create React App is a great way to get started with a Webpack-React app using standard conventions. Therefore, we'll use it in all of our forthcoming Webpack-React apps.

In this chapter, we'll:

- See what a React component looks like when represented as an ES6 module
- Examine the setup of an app managed by Create React App
- Take a close look at how Webpack works
- Explore some of the numerous advantages that Webpack provides for both development and production use
- Peek under the hood of Create React App
- Figure out how to get a Webpack-React app to work alongside an API

> The idea of a "black box" controlling the inner-workings of your app might be scary. This is a valid concern. Later in the chapter, we'll explore a feature of Create React App, `eject`, which should hopefully assuage some of this fear.

Exploring Create React App

Let's install Create React App and then use it to initialize a Webpack-React app. We can install it globally from the command line using the `-g` flag. You can run this command anywhere on your system:

```
$ npm i -g create-react-app@1.4.1
```

> The `@1.4.1` above is used to specify a version number. We recommend using this version as it is the same version we tested the code in this book with.

Now, anywhere on your system, you can run the `create-react-app` command to initiate the setup for a new Webpack-powered React app.

Let's create a new app. We'll do this inside of the code download that came with the book. From the root of the code folder, change into the directory for this chapter:

```
$ cd webpack
```

That directory already has three folders:

```
$ ls
es6-modules/
food-lookup/
heart-webpack-complete/
```

The completed version of the code for this next section is available in `heart-webpack-complete`.

Run the following command to initiate a new React app in a folder called `heart-webpack`:

```
$ create-react-app heart-webpack --scripts-version=1.0.14
```

This will create the boilerplate for the new app and install the app's dependencies. This might take a while.

The `--scripts-version` flag above is important. We want to ensure your `react-scripts` version is the same as the one we're using here in the book. We'll see what the `react-scripts` package is in a moment.

When Create React App is finished, `cd` into the new directory:

```
$ cd heart-webpack
```

```
$ ls
README.md
node_modules/
package.json
public/
src/
```

Inside `src/` is a sample React app that Create React App has provided for demonstration purposes. Inside of `public/` is an `index.html`, which we'll look at first.

`public/index.html`

Opening `public/index.html` in a text editor:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
    <!--
    ... comment omitted ...
    -->
    <title>React App</title>
</head>
    <body>
        <div id="root"></div>
        <!--
        ... comment omitted ...
        -->
    </body>
</html>
```

The stark difference from the `index.html` we've used in previous apps: there are no `script` tags here. That means this file is not loading any external JavaScript files. We'll see why this is soon.

`package.json`

Looking inside of the project's `package.json`, we see a few dependencies and some script definitions:

```
{
  "name": "heart-webpack",
  "version": "0.1.0",
  "private": true,
  "devDependencies": {
    "react-scripts": "1.1.1"
  },
  "dependencies": {
    "react": "16.3.0",
    "react-dom": "16.3.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

Let's break it down.

`react-scripts`

`package.json` specifies a single development dependency, `react-scripts`:

```
  "devDependencies": {
    "react-scripts": "1.1.1"
  },
```

Create React App is just a boilerplate generator. That command produced the folder structure of our new React app, inserted a sample app, and specified our `package.json`. It's actually the `react-scripts` package that makes everything work.

`react-scripts` specifies all of our app's development dependencies, like Webpack and Babel. Furthermore, it contains scripts that "glue" all of these dependencies together in a conventional manner.

> Create React App is just a boilerplate generator. The `react-scripts` package, specified in `package.json`, is the engine that will make everything work.
>
> Even though `react-scripts` is the engine, throughout the chapter we'll continue to refer to the overall *project* as Create React App.

`react` and `react-dom`

Under `dependencies`, we see `react` and `react-dom` listed:

```
"dependencies": {
  "react": "16.3.0",
  "react-dom": "16.3.0"
},
```

In our first two projects, we loaded in `react` and `react-dom` via script tags in `index.html`. As we saw, those libraries were not specified in this project's `index.html`.

Webpack gives us the ability to use npm packages in the browser. We can specify external libraries that we'd like to use in `package.json`. This is incredibly helpful. Not only do we now have easy access to a vast library of packages. We also get to use npm to manage **all** the libraries that our app uses. We'll see in a bit how this all works.

Scripts

`package.json` specifies four commands under `scripts`. Each executes a command with `react-scripts`. Over this chapter and the next we'll cover each of these commands in depth, but at a high-level:

- `start`: Boots the Webpack development HTTP server. This server will handle requests from our web browser.
- `build`: For use in production, this command creates an optimized, static bundle of all our assets.
- `test`: Executes the app's test suite, if present.
- `eject`: Moves the innards of `react-scripts` into your project's directory. This enables you to abandon the configuration that `react-scripts` provides, tweaking the configuration to your liking.

For those weary of the black box that `react-scripts` provides, that last command is comforting. You have an escape hatch should your project "outgrow" `react-scripts` or should you need some special configuration.

> In a `package.json`, you can specify which packages are necessary in which environment. Note that `react-scripts` is specified under `devDependencies`.
>
> When you run `npm i`, npm will check the environment variable `NODE_ENV` to see if it's installing packages in a production environment. In production, npm only installs packages listed under `dependencies` (in our case, `react` and `react-dom`). In development, npm installs all packages. This speeds the process in production, foregoing the installation of unneeded packages like linters or testing libraries.
>
> Given this, you might wonder: Why is `react-scripts` listed as a development dependency? How will the app work in a production environment without it? We'll see why this is after taking a look at how Webpack prepares production builds.

`src/`

Inside of `src/` , we see some JavaScript files:

```
$ ls src
App.css
App.js
App.test.js
index.css
index.js
logo.svg
```

Create React App has created a boilerplate React app to demonstrate how files can be organized. This app has a single component, `App` , which lives inside `App.js` .

`App.js`

Looking inside `src/App.js` :

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <div className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h2>Welcome to React</h2>
        </div>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

There are a few noteworthy features here.

The import statements

We import `React` and `Component` at the top of the file:

```
import React, { Component } from 'react';
```

This is the ES6 module import syntax. Webpack will infer that by `'react'` we are referring to the npm package specified in our `package.json`.

> If ES6 modules are new to you, check out the entry in "Appendix B."

The next two imports may have surprised you:

```
import logo from './logo.svg';
import './App.css';
```

We're using `import` on files that aren't JavaScript! Webpack has you specify *all* your dependencies using this syntax. We'll see later how this comes into play. Because the paths are relative (they are preceded with `./`), Webpack knows we're referring to local files and not npm packages.

`App` is an ES6 module

The `App` component itself is simple and does not employ state or props. Its return method is just markup, which we'll see rendered in a moment.

What's special about the `App` component is that it's an ES6 module. Our `App` component lives inside its own dedicated `App.js`. At the top of this file, it specifies its dependencies and at the bottom it specifies its export:

```
export default App;
```