Chris Parker

File Structure for React Applications Created with create-react-app

Chris Parker

Aug 15, 2019 • 15 Min read • 30,239 Views

Aug 15, 2019 • 15 Min read • 30,239 Views

Web Development React

Introduction

69

Introduction

When we start working on any React application, it can be a big challenge to set up the overall project - setting up the dependencies, configuration, etc. This has to be handled even before we start writing our first line of code. With create-react-app, we can tackle all of the above issues within a few minutes and get started with our actual application code. The create-react-app tool was launched by Facebook and is a recommended way of setting up a new project. In this guide, we would look at a project created using this tool and also how the project gets structured.

Create a New React Project

To create a new app/project using this tool, all we need to do is run the command "create-react-app" followed by the app name.

javascript

```
create-react-app my-sample-app
```

After running the above command, a new folder called "my-sample-app" will get created and that would have all of our application code.

Project Layout

Below is how the project will be structured:

my-sample-app

```
├── README.md
├── node_modules
           ├── package.json
           ├── .gitignore
           ├── build
           ├── public
           │   ├── favicon.ico
           │   ├── index.html
           │   └── manifest.json
           └── src
               ├── App.css
               ├── App.js
               ├── App.test.js
               ├── index.css
               ├── index.js
               ├── logo.svg
               └── serviceWorker.js


```

Looking at the above, we can see that only the bare minimum structure is created without any complexities.

Details About the Structure

Let's go through each of these and understand the purpose of each of the generated file/folder.

build represents the path to our final production build. This folder would actually be created after we run the npm build.

We can see all the "dependencies" and "devDependencies" required by our React app in node_modules. These are as specified or seen in our package.json file. If we just run the ls -l command, we'll see almost 800 sub-directories. This directory gets added to .gitignore so it does not really get uploaded/published as such. Also, once we minimize or compress our code for production, our sample app should be less than 100 KB in size.

Our static files are located in the public directory. Files in this directory will retain the same name when deployed to production. Thus, they can be cached at the client-side and improve the overall download times.

All of the dynamic components will be located in the src. To ensure that, at the client side, only the most recent version is downloaded and not the cached copy, Webpack will generally have the updated files a unique file name in the final build. Thus, we can use simple file names e.g. header.png, instead of using header-2019-01-01.png. Webpack would take care of renaming header.png to header.dynamic-hash.png. This unique hash would get updated only when our header.png would change. We can also see files like App.js which is kind of our main JS component and the corresponding styles go in App.css. In case, we want to add any unit tests, we can use the App.test.js for that. Also, index.js is the entry point for our App and it triggers the registerServiceWorker.js. As a side-note, we mostly add a 'components' directory here to add new components and their associated files, as that improves the organization of our structure.

The overall configuration for the React project is outlined in the package.json. Below is what that looks like:

json

```json
{
    "name": "my-sample-app",
    "version": "0.0.1",
    "private": true,
    "dependencies": {
        "react": "^16.5.2",
        "react-dom": "^16.5.2",
    },
    "devDependencies": {
        "react-scripts": "1.0.7"
    },
    "scripts": {
        "start": "react-scripts start",
        "build": "react-scripts build",
        "test": "react-scripts test --env=jsdom",
        "eject": "react-scripts eject"
    }
}
```

We can see the following attributes:

- name - Represents the app name which was passed to create-react-app.
- version - Shows the current version.
- dependencies - List of all the required modules/versions for our app. By default, npm would install the most recent major version.
- devDependencies - Lists all the modules/versions for running the app in a development environment.
- scripts - List of all the aliases that can be used to access react-scripts commands in an efficient manner. For example, if we run npm build in the command line, it would run "react-scripts build" internally.

The dependencies which are shared by our application can go to the assets directory. These can include mixins, images, etc. Thus, they would represent a single location for files external to our main project itself.

We also need to have a utilities folder. This would contain a list of helper functions used globally across the app. We can add common logic to this utilities folder and import that wherever we want to use it. While the naming can vary slightly, the standard naming conventions are seen include helpers, utils, utilities, etc.

With that, our structure would now looks something like below:

```
my-sample-app
├── build
    ├── node_modules
    ├── public
    │   ├── favicon.ico
    │   ├── index.html
    │   └── manifest.json
    ├── src
    │   ├── assets
    │   │   └──images
    │   │      └── logo.svg
    │   ├── components
    │   │   └── app
    │   │       ├── App.css
    │   │       ├── App.js
    │   │       └── App.test.js
    │   ├── utilities
    │   ├── Index.css
    │   ├── Index.js
    │   └── service-worker.js
    ├── .gitignore
    ├── package.json
    └── README.md


```

manifest.json This file is used to describe our app e.g. On mobile phones, if a shortcut is added to the home screen. Below is how that would look like;

json

```json
{
      "short_name": "My Sample React App",
      "name": "My Create React App Sample",
      "icons": [
        {
          "src": "favicon.ico",
          "sizes": "64x64 32x32 24x24 16x16",
          "type": "image/x-icon"
        }
      ],
      "start_url": ".",
      "display": "standalone",
      "theme_color": "#efefef",
      "background_color": "#000000"
    }
```

When our web app is added to user's home screen, it is this metadata which determines the icon, theme colors, names, etc.

favicon.ico This is the icon image file used by our project. It is also linked inside index.html and manifest.json.

Component Directory

The component directory structure is the most important thing in any React app. While components can reside in src/components/my-component-name, it is recommended to have an index.js inside that directory. Thus, whenever someone imports the component using src/components/my-component-name, instead of importing the directory, this would actually import the index.js file.

Also component involves many files, including stateless and stateful containers, SASS files, utilities shared within that component, and even child components.

Thus, our component directory structure would look something like below:

```
my-sample-app
└── src
    └── components
        └── my-component-name
            ├── my-component-name.css
            ├── my-component-name.scss
            ├── my-component-name-container.js
            ├── my-component-name-redux.js
            ├── my-component-name-styles.js
            ├── my-component-name-view.js
            └── index.js
```

my-component-name.css represents the CSS file imported by our stateless view Component. my-component-name.scss is the SASS file imported by our stateless view Component. my-component-name-container.js would contain the business logic as well as state management. my-component-name-redux.js would include mapStateToProps, mapDispatchToProps and connect functionality provided by Redux. my-component-name-styles.js would represent our JSS (e.g. storing Material UI styles). my-component-name-view.js would mostly be a pure functional Component index.js is the entry point for importing our Component.

Unit Tests

For unit tests, we would follow the same principle of grouping all our related files. Thus, we can add them within the components directory we have as shown below;

```
my-sample-app
└── src
    └── components
        └── my-component-name
            ├── my-component-name-container.js
            ├── my-component-name-container.test.js
            ├── my-component-name-redux.js
```

```
                    ├── my-component-name-redux.test.js
                    ├── my-component-name-view.js
                    └── my-component-name-view.test.js
```

Index Page

Let's also have a look inside the index.js as well as the index.html page which gets generated.

Below is how our index.js file looks;

javascript

```javascript
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(<App />, document.getElementById('root'));
registerServiceWorker();
```

Below is the html page;

html

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1, shrink-to-fit=no"
    />
    <meta name="theme-color" content="#000000" />
    <!--
      manifest.json provides metadata used when your web app is installec
      user's mobile device or desktop. See https://developers.google.com/
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the `public` folder during the
      Only files inside the `public` folder can be referenced from the HT

      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico"
      work correctly both with client-side routing and a non-root public
      Learn how to configure a non-root public URL by running `npm run bu
    -->
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.

      You can add webfonts, meta tags, or analytics to this file.
      The build step will place the bundled scripts into the <body> tag.

      To begin the development, run `npm start` or `yarn start`.
      To create a production bundle, use `npm run build` or `yarn build`.
    -->
  </body>
</html>
```

As we can see, that it is a very basic HTML page with a few meta tags and some link elements. Also, we can see that there is an empty div element which is added with id "root". We can always update that to something else, like "content", as well as add any additional CSS or external JS libraries e.g. say we want to add Bootstrap library to our project. To do that, we can directly add a CDN reference to our index.html, as shown below:

html

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1, shrink-to-fit=no"
    />
    <meta name="theme-color" content="#000000" />
    <!--
      manifest.json provides metadata used when your web app is installe
      user's mobile device or desktop. See https://developers.google.com
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the `public` folder during the
      Only files inside the `public` folder can be referenced from the HT

      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico"
      work correctly both with client-side routing and a non-root public
      Learn how to configure a non-root public URL by running `npm run bu
    -->
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="content"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.

      You can add webfonts, meta tags, or analytics to this file.
      The build step will place the bundled scripts into the <body> tag.

      To begin the development, run `npm start` or `yarn start`.
      To create a production bundle, use `npm run build` or `yarn build`.
    -->
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquer
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/boots

    <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" crossc
    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.11.0/
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/js/
  </body>
</html>
```

Since we changed the default container element Id to "content", we also have to update the same in our index.js as shown below:

javascript

```javascript
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(<App />, document.getElementById('content'));
registerServiceWorker();
```

The other way of adding the bootstrap library (say we want to use it only within a specific component) is to use npm to install the library and then add the import as shown below:

javascript

```
npm install --save bootstrap
```

javascript

```javascript
import '../node_modules/bootstrap/dist/css/bootstrap.min.css';
```

OR

javascript

```javascript
import 'bootstrap/dist/css/bootstrap.min.css';
```

Ejecting

Let's say that, after you generated the project using create-react-app, you want to do some additional customization. Ejecting would allow you to do that.

Following command can be run to eject from create-react-app:

```javascript
npm eject
```

Ejecting would mean that all the configuration gets exposed to us and we would be responsible for maintaining all the configuration from that point onward.

Thus, it essentially allows us more control over the project. It is important to remember that this is a on-way command i.e. we cannot go back once we eject.

Conclusion

It is highly recommended to use create-react-app to setup your project structure and, especially, if you are not familiar with configuration and tools like Babel, Webpack, Browserify, etc. Using create-react-app helps the developer to focus on the core areas of the app and not have to worry about other things. However, if you later want to take things in your own hands, you always have the option of using eject to pull your app out of the create-react-app context and into the standard webpack configuration.

69