This project was bootstrapped with [Create React App](https://github.com/facebookincubator/create-react-app).

Below you will find some information on how to perform common tasks.  
You can find the most recent version of this guide [here](https://github.com/facebookincubator/create-react-app/blob/master/packages/react-scripts/template/README.md).

**Table of Contents**

* [Updating to New Releases](https://github.com/baluragala/redux-saga-examples#updating-to-new-releases)
* [Sending Feedback](https://github.com/baluragala/redux-saga-examples#sending-feedback)
* [Folder Structure](https://github.com/baluragala/redux-saga-examples#folder-structure)
* [Available Scripts](https://github.com/baluragala/redux-saga-examples#available-scripts)
  + [npm start](https://github.com/baluragala/redux-saga-examples#npm-start)
  + [npm test](https://github.com/baluragala/redux-saga-examples#npm-test)
  + [npm run build](https://github.com/baluragala/redux-saga-examples#npm-run-build)
  + [npm run eject](https://github.com/baluragala/redux-saga-examples#npm-run-eject)
* [Supported Language Features and Polyfills](https://github.com/baluragala/redux-saga-examples#supported-language-features-and-polyfills)
* [Syntax Highlighting in the Editor](https://github.com/baluragala/redux-saga-examples#syntax-highlighting-in-the-editor)
* [Displaying Lint Output in the Editor](https://github.com/baluragala/redux-saga-examples#displaying-lint-output-in-the-editor)
* [Debugging in the Editor](https://github.com/baluragala/redux-saga-examples#debugging-in-the-editor)
* [Changing the Page <title>](https://github.com/baluragala/redux-saga-examples#changing-the-page-title)
* [Installing a Dependency](https://github.com/baluragala/redux-saga-examples#installing-a-dependency)
* [Importing a Component](https://github.com/baluragala/redux-saga-examples#importing-a-component)
* [Adding a Stylesheet](https://github.com/baluragala/redux-saga-examples#adding-a-stylesheet)
* [Post-Processing CSS](https://github.com/baluragala/redux-saga-examples#post-processing-css)
* [Adding a CSS Preprocessor (Sass, Less etc.)](https://github.com/baluragala/redux-saga-examples#adding-a-css-preprocessor-sass-less-etc)
* [Adding Images and Fonts](https://github.com/baluragala/redux-saga-examples#adding-images-and-fonts)
* [Using the public Folder](https://github.com/baluragala/redux-saga-examples#using-the-public-folder)
  + [Changing the HTML](https://github.com/baluragala/redux-saga-examples#changing-the-html)
  + [Adding Assets Outside of the Module System](https://github.com/baluragala/redux-saga-examples#adding-assets-outside-of-the-module-system)
  + [When to Use the public Folder](https://github.com/baluragala/redux-saga-examples#when-to-use-the-public-folder)
* [Using Global Variables](https://github.com/baluragala/redux-saga-examples#using-global-variables)
* [Adding Bootstrap](https://github.com/baluragala/redux-saga-examples#adding-bootstrap)
  + [Using a Custom Theme](https://github.com/baluragala/redux-saga-examples#using-a-custom-theme)
* [Adding Flow](https://github.com/baluragala/redux-saga-examples#adding-flow)
* [Adding Custom Environment Variables](https://github.com/baluragala/redux-saga-examples#adding-custom-environment-variables)
  + [Referencing Environment Variables in the HTML](https://github.com/baluragala/redux-saga-examples#referencing-environment-variables-in-the-html)
  + [Adding Temporary Environment Variables In Your Shell](https://github.com/baluragala/redux-saga-examples#adding-temporary-environment-variables-in-your-shell)
  + [Adding Development Environment Variables In .env](https://github.com/baluragala/redux-saga-examples#adding-development-environment-variables-in-env)
* [Can I Use Decorators?](https://github.com/baluragala/redux-saga-examples#can-i-use-decorators)
* [Integrating with an API Backend](https://github.com/baluragala/redux-saga-examples#integrating-with-an-api-backend)
  + [Node](https://github.com/baluragala/redux-saga-examples#node)
  + [Ruby on Rails](https://github.com/baluragala/redux-saga-examples#ruby-on-rails)
* [Proxying API Requests in Development](https://github.com/baluragala/redux-saga-examples#proxying-api-requests-in-development)
* [Using HTTPS in Development](https://github.com/baluragala/redux-saga-examples#using-https-in-development)
* [Generating Dynamic <meta> Tags on the Server](https://github.com/baluragala/redux-saga-examples#generating-dynamic-meta-tags-on-the-server)
* [Pre-Rendering into Static HTML Files](https://github.com/baluragala/redux-saga-examples#pre-rendering-into-static-html-files)
* [Injecting Data from the Server into the Page](https://github.com/baluragala/redux-saga-examples#injecting-data-from-the-server-into-the-page)
* [Running Tests](https://github.com/baluragala/redux-saga-examples#running-tests)
  + [Filename Conventions](https://github.com/baluragala/redux-saga-examples#filename-conventions)
  + [Command Line Interface](https://github.com/baluragala/redux-saga-examples#command-line-interface)
  + [Version Control Integration](https://github.com/baluragala/redux-saga-examples#version-control-integration)
  + [Writing Tests](https://github.com/baluragala/redux-saga-examples#writing-tests)
  + [Testing Components](https://github.com/baluragala/redux-saga-examples#testing-components)
  + [Using Third Party Assertion Libraries](https://github.com/baluragala/redux-saga-examples#using-third-party-assertion-libraries)
  + [Initializing Test Environment](https://github.com/baluragala/redux-saga-examples#initializing-test-environment)
  + [Focusing and Excluding Tests](https://github.com/baluragala/redux-saga-examples#focusing-and-excluding-tests)
  + [Coverage Reporting](https://github.com/baluragala/redux-saga-examples#coverage-reporting)
  + [Continuous Integration](https://github.com/baluragala/redux-saga-examples#continuous-integration)
  + [Disabling jsdom](https://github.com/baluragala/redux-saga-examples#disabling-jsdom)
  + [Snapshot Testing](https://github.com/baluragala/redux-saga-examples#snapshot-testing)
  + [Editor Integration](https://github.com/baluragala/redux-saga-examples#editor-integration)
* [Developing Components in Isolation](https://github.com/baluragala/redux-saga-examples#developing-components-in-isolation)
* [Making a Progressive Web App](https://github.com/baluragala/redux-saga-examples#making-a-progressive-web-app)
* [Deployment](https://github.com/baluragala/redux-saga-examples#deployment)
  + [Static Server](https://github.com/baluragala/redux-saga-examples#static-server)
  + [Other Solutions](https://github.com/baluragala/redux-saga-examples#other-solutions)
  + [Serving Apps with Client-Side Routing](https://github.com/baluragala/redux-saga-examples#serving-apps-with-client-side-routing)
  + [Building for Relative Paths](https://github.com/baluragala/redux-saga-examples#building-for-relative-paths)
  + [Azure](https://github.com/baluragala/redux-saga-examples#azure)
  + [Firebase](https://github.com/baluragala/redux-saga-examples#firebase)
  + [GitHub Pages](https://github.com/baluragala/redux-saga-examples#github-pages)
  + [Heroku](https://github.com/baluragala/redux-saga-examples#heroku)
  + [Modulus](https://github.com/baluragala/redux-saga-examples#modulus)
  + [Netlify](https://github.com/baluragala/redux-saga-examples#netlify)
  + [Now](https://github.com/baluragala/redux-saga-examples#now)
  + [S3 and CloudFront](https://github.com/baluragala/redux-saga-examples#s3-and-cloudfront)
  + [Surge](https://github.com/baluragala/redux-saga-examples#surge)
* [Advanced Configuration](https://github.com/baluragala/redux-saga-examples#advanced-configuration)
* [Troubleshooting](https://github.com/baluragala/redux-saga-examples#troubleshooting)
  + [npm start doesn’t detect changes](https://github.com/baluragala/redux-saga-examples#npm-start-doesnt-detect-changes)
  + [npm test hangs on macOS Sierra](https://github.com/baluragala/redux-saga-examples#npm-test-hangs-on-macos-sierra)
  + [npm run build silently fails](https://github.com/baluragala/redux-saga-examples#npm-run-build-silently-fails)
  + [npm run build fails on Heroku](https://github.com/baluragala/redux-saga-examples#npm-run-build-fails-on-heroku)
* [Something Missing?](https://github.com/baluragala/redux-saga-examples#something-missing)

**Updating to New Releases**

Create React App is divided into two packages:

* create-react-app is a global command-line utility that you use to create new projects.
* react-scripts is a development dependency in the generated projects (including this one).

You almost never need to update create-react-app itself: it delegates all the setup to react-scripts.

When you run create-react-app, it always creates the project with the latest version of react-scripts so you’ll get all the new features and improvements in newly created apps automatically.

To update an existing project to a new version of react-scripts, [open the changelog](https://github.com/facebookincubator/create-react-app/blob/master/CHANGELOG.md), find the version you’re currently on (check package.json in this folder if you’re not sure), and apply the migration instructions for the newer versions.

In most cases bumping the react-scripts version in package.json and running npm install in this folder should be enough, but it’s good to consult the [changelog](https://github.com/facebookincubator/create-react-app/blob/master/CHANGELOG.md) for potential breaking changes.

We commit to keeping the breaking changes minimal so you can upgrade react-scripts painlessly.

**Sending Feedback**

We are always open to [your feedback](https://github.com/facebookincubator/create-react-app/issues).

**Folder Structure**

After creation, your project should look like this:

my-app/

README.md

node\_modules/

package.json

public/

index.html

favicon.ico

src/

App.css

App.js

App.test.js

index.css

index.js

logo.svg

For the project to build, **these files must exist with exact filenames**:

* public/index.html is the page template;
* src/index.js is the JavaScript entry point.

You can delete or rename the other files.

You may create subdirectories inside src. For faster rebuilds, only files inside src are processed by Webpack.  
You need to **put any JS and CSS files inside src**, or Webpack won’t see them.

Only files inside public can be used from public/index.html.  
Read instructions below for using assets from JavaScript and HTML.

You can, however, create more top-level directories.  
They will not be included in the production build so you can use them for things like documentation.

**Available Scripts**

In the project directory, you can run:

**npm start**

Runs the app in the development mode.  
Open [http://localhost:3000](http://localhost:3000/) to view it in the browser.

The page will reload if you make edits.  
You will also see any lint errors in the console.

**npm test**

Launches the test runner in the interactive watch mode.  
See the section about [running tests](https://github.com/baluragala/redux-saga-examples#running-tests) for more information.

**npm run build**

Builds the app for production to the build folder.  
It correctly bundles React in production mode and optimizes the build for the best performance.

The build is minified and the filenames include the hashes.  
Your app is ready to be deployed!

See the section about [deployment](https://github.com/baluragala/redux-saga-examples#deployment) for more information.

**npm run eject**

**Note: this is a one-way operation. Once you eject, you can’t go back!**

If you aren’t satisfied with the build tool and configuration choices, you can eject at any time. This command will remove the single build dependency from your project.

Instead, it will copy all the configuration files and the transitive dependencies (Webpack, Babel, ESLint, etc) right into your project so you have full control over them. All of the commands except eject will still work, but they will point to the copied scripts so you can tweak them. At this point you’re on your own.

You don’t have to ever use eject. The curated feature set is suitable for small and middle deployments, and you shouldn’t feel obligated to use this feature. However we understand that this tool wouldn’t be useful if you couldn’t customize it when you are ready for it.

**Supported Language Features and Polyfills**

This project supports a superset of the latest JavaScript standard.  
In addition to [ES6](https://github.com/lukehoban/es6features) syntax features, it also supports:

* [Exponentiation Operator](https://github.com/rwaldron/exponentiation-operator) (ES2016).
* [Async/await](https://github.com/tc39/ecmascript-asyncawait) (ES2017).
* [Object Rest/Spread Properties](https://github.com/sebmarkbage/ecmascript-rest-spread) (stage 3 proposal).
* [Class Fields and Static Properties](https://github.com/tc39/proposal-class-public-fields) (stage 2 proposal).
* [JSX](https://facebook.github.io/react/docs/introducing-jsx.html) and [Flow](https://flowtype.org/) syntax.

Learn more about [different proposal stages](https://babeljs.io/docs/plugins/#presets-stage-x-experimental-presets-).

While we recommend to use experimental proposals with some caution, Facebook heavily uses these features in the product code, so we intend to provide [codemods](https://medium.com/@cpojer/effective-javascript-codemods-5a6686bb46fb) if any of these proposals change in the future.

Note that **the project only includes a few ES6**[**polyfills**](https://en.wikipedia.org/wiki/Polyfill):

* [Object.assign()](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) via [object-assign](https://github.com/sindresorhus/object-assign).
* [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) via [promise](https://github.com/then/promise).
* [fetch()](https://developer.mozilla.org/en/docs/Web/API/Fetch_API) via [whatwg-fetch](https://github.com/github/fetch).

If you use any other ES6+ features that need **runtime support** (such as Array.from() or Symbol), make sure you are including the appropriate polyfills manually, or that the browsers you are targeting already support them.

**Syntax Highlighting in the Editor**

To configure the syntax highlighting in your favorite text editor, head to the [relevant Babel documentation page](https://babeljs.io/docs/editors) and follow the instructions. Some of the most popular editors are covered.

**Displaying Lint Output in the Editor**

Note: this feature is available with react-scripts@0.2.0 and higher.

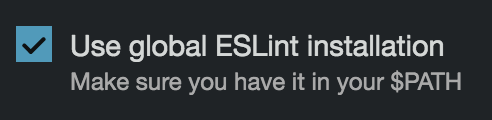
Some editors, including Sublime Text, Atom, and Visual Studio Code, provide plugins for ESLint.

They are not required for linting. You should see the linter output right in your terminal as well as the browser console. However, if you prefer the lint results to appear right in your editor, there are some extra steps you can do.

You would need to install an ESLint plugin for your editor first.

**A note for Atom linter-eslint users**

If you are using the Atom linter-eslint plugin, make sure that **Use global ESLint installation** option is checked:

[](https://camo.githubusercontent.com/b3963a54424d2790b7bdb2b0e85a9d8875eb9a14abac028987d121c88c8b4eb4/687474703a2f2f692e696d6775722e636f6d2f79564e4e484a4d2e706e67)

**For Visual Studio Code users**

VS Code ESLint plugin automatically detects Create React App's configuration file. So you do not need to create eslintrc.json at the root directory, except when you want to add your own rules. In that case, you should include CRA's config by adding this line:

{ // ... "extends": "react-app" }

Then add this block to the `package.json` file of your project:

```js

{

// ...

"eslintConfig": {

"extends": "react-app"

}

}

Finally, you will need to install some packages *globally*:

npm install -g eslint-config-react-app@0.3.0 eslint@3.8.1 babel-eslint@7.0.0 eslint-plugin-react@6.4.1 eslint-plugin-import@2.0.1 eslint-plugin-jsx-a11y@4.0.0 eslint-plugin-flowtype@2.21.0

We recognize that this is suboptimal, but it is currently required due to the way we hide the ESLint dependency. The ESLint team is already [working on a solution to this](https://github.com/eslint/eslint/issues/3458) so this may become unnecessary in a couple of months.

**Debugging in the Editor**

**This feature is currently only supported by**[**Visual Studio Code**](https://code.visualstudio.com/)**editor.**

Visual Studio Code supports live-editing and debugging out of the box with Create React App. This enables you as a developer to write and debug your React code without leaving the editor, and most importantly it enables you to have a continuous development workflow, where context switching is minimal, as you don’t have to switch between tools.

You would need to have the latest version of [VS Code](https://code.visualstudio.com/) and VS Code [Chrome Debugger Extension](https://marketplace.visualstudio.com/items?itemName=msjsdiag.debugger-for-chrome) installed.

Then add the block below to your launch.json file and put it inside the .vscode folder in your app’s root directory.

{

"version": "0.2.0",

"configurations": [{

"name": "Chrome",

"type": "chrome",

"request": "launch",

"url": "http://localhost:3000",

"webRoot": "${workspaceRoot}/src",

"userDataDir": "${workspaceRoot}/.vscode/chrome",

"sourceMapPathOverrides": {

"webpack:///src/\*": "${webRoot}/\*"

}

}]

}

Start your app by running npm start, and start debugging in VS Code by pressing F5 or by clicking the green debug icon. You can now write code, set breakpoints, make changes to the code, and debug your newly modified code—all from your editor.

**Changing the Page <title>**

You can find the source HTML file in the public folder of the generated project. You may edit the <title> tag in it to change the title from “React App” to anything else.

Note that normally you wouldn’t edit files in the public folder very often. For example, [adding a stylesheet](https://github.com/baluragala/redux-saga-examples#adding-a-stylesheet) is done without touching the HTML.

If you need to dynamically update the page title based on the content, you can use the browser [document.title](https://developer.mozilla.org/en-US/docs/Web/API/Document/title) API. For more complex scenarios when you want to change the title from React components, you can use [React Helmet](https://github.com/nfl/react-helmet), a third party library.

If you use a custom server for your app in production and want to modify the title before it gets sent to the browser, you can follow advice in [this section](https://github.com/baluragala/redux-saga-examples#generating-dynamic-meta-tags-on-the-server). Alternatively, you can pre-build each page as a static HTML file which then loads the JavaScript bundle, which is covered [here](https://github.com/baluragala/redux-saga-examples#pre-rendering-into-static-html-files).

**Installing a Dependency**

The generated project includes React and ReactDOM as dependencies. It also includes a set of scripts used by Create React App as a development dependency. You may install other dependencies (for example, React Router) with npm:

npm install --save <library-name>

**Importing a Component**

This project setup supports ES6 modules thanks to Babel.  
While you can still use require() and module.exports, we encourage you to use [import and export](http://exploringjs.com/es6/ch_modules.html) instead.

For example:

**Button.js**

import React, { Component } from 'react';

class Button extends Component {

render() {

// ...

}

}

export default Button; // Don’t forget to use export default!

**DangerButton.js**

import React, { Component } from 'react';

import Button from './Button'; // Import a component from another file

class DangerButton extends Component {

render() {

return <Button color="red" />;

}

}

export default DangerButton;

Be aware of the [difference between default and named exports](http://stackoverflow.com/questions/36795819/react-native-es-6-when-should-i-use-curly-braces-for-import/36796281#36796281). It is a common source of mistakes.

We suggest that you stick to using default imports and exports when a module only exports a single thing (for example, a component). That’s what you get when you use export default Button and import Button from './Button'.

Named exports are useful for utility modules that export several functions. A module may have at most one default export and as many named exports as you like.

Learn more about ES6 modules:

* [When to use the curly braces?](http://stackoverflow.com/questions/36795819/react-native-es-6-when-should-i-use-curly-braces-for-import/36796281#36796281)
* [Exploring ES6: Modules](http://exploringjs.com/es6/ch_modules.html)
* [Understanding ES6: Modules](https://leanpub.com/understandinges6/read#leanpub-auto-encapsulating-code-with-modules)

**Adding a Stylesheet**

This project setup uses [Webpack](https://webpack.github.io/) for handling all assets. Webpack offers a custom way of “extending” the concept of import beyond JavaScript. To express that a JavaScript file depends on a CSS file, you need to **import the CSS from the JavaScript file**:

**Button.css**

.Button {

padding: 20px;

}

**Button.js**

import React, { Component } from 'react';

import './Button.css'; // Tell Webpack that Button.js uses these styles

class Button extends Component {

render() {

// You can use them as regular CSS styles

return <div className="Button" />;

}

}

**This is not required for React** but many people find this feature convenient. You can read about the benefits of this approach [here](https://medium.com/seek-ui-engineering/block-element-modifying-your-javascript-components-d7f99fcab52b). However you should be aware that this makes your code less portable to other build tools and environments than Webpack.

In development, expressing dependencies this way allows your styles to be reloaded on the fly as you edit them. In production, all CSS files will be concatenated into a single minified .css file in the build output.

If you are concerned about using Webpack-specific semantics, you can put all your CSS right into src/index.css. It would still be imported from src/sagaHelpers.js, but you could always remove that import if you later migrate to a different build tool.

**Post-Processing CSS**

This project setup minifies your CSS and adds vendor prefixes to it automatically through [Autoprefixer](https://github.com/postcss/autoprefixer) so you don’t need to worry about it.

For example, this:

.App {

display: flex;

flex-direction: row;

align-items: center;

}

becomes this:

.App {

display: -webkit-box;

display: -ms-flexbox;

display: flex;

-webkit-box-orient: horizontal;

-webkit-box-direction: normal;

-ms-flex-direction: row;

flex-direction: row;

-webkit-box-align: center;

-ms-flex-align: center;

align-items: center;

}

If you need to disable autoprefixing for some reason, [follow this section](https://github.com/postcss/autoprefixer#disabling).

**Adding a CSS Preprocessor (Sass, Less etc.)**

Generally, we recommend that you don’t reuse the same CSS classes across different components. For example, instead of using a .Button CSS class in <AcceptButton> and <RejectButton> components, we recommend creating a <Button> component with its own .Button styles, that both <AcceptButton> and <RejectButton> can render (but [not inherit](https://facebook.github.io/react/docs/composition-vs-inheritance.html)).

Following this rule often makes CSS preprocessors less useful, as features like mixins and nesting are replaced by component composition. You can, however, integrate a CSS preprocessor if you find it valuable. In this walkthrough, we will be using Sass, but you can also use Less, or another alternative.

First, let’s install the command-line interface for Sass:

npm install node-sass --save-dev

Then in package.json, add the following lines to scripts:

"scripts": {

+ "build-css": "node-sass src/ -o src/",

+ "watch-css": "npm run build-css && node-sass src/ -o src/ --watch --recursive",

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test --env=jsdom",

Note: To use a different preprocessor, replace build-css and watch-css commands according to your preprocessor’s documentation.

Now you can rename src/App.css to src/App.scss and run npm run watch-css. The watcher will find every Sass file in src subdirectories, and create a corresponding CSS file next to it, in our case overwriting src/App.css. Since src/App.js still imports src/App.css, the styles become a part of your application. You can now edit src/App.scss, and src/App.css will be regenerated.

To share variables between Sass files, you can use Sass imports. For example, src/App.scss and other component style files could include @import "./shared.scss"; with variable definitions.

At this point you might want to remove all CSS files from the source control, and add src/\*\*/\*.css to your .gitignore file. It is generally a good practice to keep the build products outside of the source control.

As a final step, you may find it convenient to run watch-css automatically with npm start, and run build-css as a part of npm run build. You can use the && operator to execute two scripts sequentially. However, there is no cross-platform way to run two scripts in parallel, so we will install a package for this:

npm install --save-dev npm-run-all

Then we can change start and build scripts to include the CSS preprocessor commands:

"scripts": {

"build-css": "node-sass src/ -o src/",

"watch-css": "npm run build-css && node-sass src/ -o src/ --watch --recursive",

- "start": "react-scripts start",

- "build": "react-scripts build",

+ "start-js": "react-scripts start",

+ "start": "npm-run-all -p watch-css start-js",

+ "build": "npm run build-css && react-scripts build",

"test": "react-scripts test --env=jsdom",

"eject": "react-scripts eject"

}

Now running npm start and npm run build also builds Sass files. Note that node-sass seems to have an [issue recognizing newly created files on some systems](https://github.com/sass/node-sass/issues/1891) so you might need to restart the watcher when you create a file until it’s resolved.

**Adding Images and Fonts**

With Webpack, using static assets like images and fonts works similarly to CSS.

You can **import an image right in a JavaScript module**. This tells Webpack to include that image in the bundle. Unlike CSS imports, importing an image or a font gives you a string value. This value is the final image path you can reference in your code.

Here is an example:

import React from 'react';

import logo from './logo.png'; // Tell Webpack this JS file uses this image

console.log(logo); // /logo.84287d09.png

function Header() {

// Import result is the URL of your image

return <img src={logo} alt="Logo" />;

}

export default Header;

This ensures that when the project is built, Webpack will correctly move the images into the build folder, and provide us with correct paths.

This works in CSS too:

.Logo {

background-image: url(./logo.png);

}

Webpack finds all relative module references in CSS (they start with ./) and replaces them with the final paths from the compiled bundle. If you make a typo or accidentally delete an important file, you will see a compilation error, just like when you import a non-existent JavaScript module. The final filenames in the compiled bundle are generated by Webpack from content hashes. If the file content changes in the future, Webpack will give it a different name in production so you don’t need to worry about long-term caching of assets.

Please be advised that this is also a custom feature of Webpack.

**It is not required for React** but many people enjoy it (and React Native uses a similar mechanism for images).  
An alternative way of handling static assets is described in the next section.

**Using the public Folder**

Note: this feature is available with react-scripts@0.5.0 and higher.

**Changing the HTML**

The public folder contains the HTML file so you can tweak it, for example, to [set the page title](https://github.com/baluragala/redux-saga-examples#changing-the-page-title). The <script> tag with the compiled code will be added to it automatically during the build process.

**Adding Assets Outside of the Module System**

You can also add other assets to the public folder.

Note that we normally encourage you to import assets in JavaScript files instead. For example, see the sections on [adding a stylesheet](https://github.com/baluragala/redux-saga-examples#adding-a-stylesheet) and [adding images and fonts](https://github.com/baluragala/redux-saga-examples#adding-images-and-fonts). This mechanism provides a number of benefits:

* Scripts and stylesheets get minified and bundled together to avoid extra network requests.
* Missing files cause compilation errors instead of 404 errors for your users.
* Result filenames include content hashes so you don’t need to worry about browsers caching their old versions.

However there is an **escape hatch** that you can use to add an asset outside of the module system.

If you put a file into the public folder, it will **not** be processed by Webpack. Instead it will be copied into the build folder untouched. To reference assets in the public folder, you need to use a special variable called PUBLIC\_URL.

Inside index.html, you can use it like this:

<link rel="shortcut icon" href="%PUBLIC\_URL%/favicon.ico">

Only files inside the public folder will be accessible by %PUBLIC\_URL% prefix. If you need to use a file from src or node\_modules, you’ll have to copy it there to explicitly specify your intention to make this file a part of the build.

When you run npm run build, Create React App will substitute %PUBLIC\_URL% with a correct absolute path so your project works even if you use client-side routing or host it at a non-root URL.

In JavaScript code, you can use process.env.PUBLIC\_URL for similar purposes:

render() {

// Note: this is an escape hatch and should be used sparingly!

// Normally we recommend using `import` for getting asset URLs

// as described in “Adding Images and Fonts” above this section.

return <img src={process.env.PUBLIC\_URL + '/img/logo.png'} />;

}

Keep in mind the downsides of this approach:

* None of the files in public folder get post-processed or minified.
* Missing files will not be called at compilation time, and will cause 404 errors for your users.
* Result filenames won’t include content hashes so you’ll need to add query arguments or rename them every time they change.

**When to Use the public Folder**

Normally we recommend importing [stylesheets](https://github.com/baluragala/redux-saga-examples#adding-a-stylesheet), [images, and fonts](https://github.com/baluragala/redux-saga-examples#adding-images-and-fonts) from JavaScript. The public folder is useful as a workaround for a number of less common cases:

* You need a file with a specific name in the build output, such as [manifest.webmanifest](https://developer.mozilla.org/en-US/docs/Web/Manifest).
* You have thousands of images and need to dynamically reference their paths.
* You want to include a small script like [pace.js](http://github.hubspot.com/pace/docs/welcome/) outside of the bundled code.
* Some library may be incompatible with Webpack and you have no other option but to include it as a <script> tag.

Note that if you add a <script> that declares global variables, you also need to read the next section on using them.

**Using Global Variables**

When you include a script in the HTML file that defines global variables and try to use one of these variables in the code, the linter will complain because it cannot see the definition of the variable.

You can avoid this by reading the global variable explicitly from the window object, for example:

const $ = window.$;

This makes it obvious you are using a global variable intentionally rather than because of a typo.

Alternatively, you can force the linter to ignore any line by adding // eslint-disable-line after it.

**Adding Bootstrap**

You don’t have to use [React Bootstrap](https://react-bootstrap.github.io/) together with React but it is a popular library for integrating Bootstrap with React apps. If you need it, you can integrate it with Create React App by following these steps:

Install React Bootstrap and Bootstrap from npm. React Bootstrap does not include Bootstrap CSS so this needs to be installed as well:

npm install react-bootstrap --save

npm install bootstrap@3 --save

Import Bootstrap CSS and optionally Bootstrap theme CSS in the beginning of your src/sagaHelpers.js file:

import 'bootstrap/dist/css/bootstrap.css';

import 'bootstrap/dist/css/bootstrap-theme.css';

// Put any other imports below so that CSS from your

// components takes precedence over default styles.

Import required React Bootstrap components within src/App.js file or your custom component files:

import { Navbar, Jumbotron, Button } from 'react-bootstrap';

Now you are ready to use the imported React Bootstrap components within your component hierarchy defined in the render method. Here is an example [App.js](https://gist.githubusercontent.com/gaearon/85d8c067f6af1e56277c82d19fd4da7b/raw/6158dd991b67284e9fc8d70b9d973efe87659d72/App.js) redone using React Bootstrap.

**Using a Custom Theme**

Sometimes you might need to tweak the visual styles of Bootstrap (or equivalent package).  
We suggest the following approach:

* Create a new package that depends on the package you wish to customize, e.g. Bootstrap.
* Add the necessary build steps to tweak the theme, and publish your package on npm.
* Install your own theme npm package as a dependency of your app.

Here is an example of adding a [customized Bootstrap](https://medium.com/@tacomanator/customizing-create-react-app-aa9ffb88165) that follows these steps.

**Adding Flow**

Flow is a static type checker that helps you write code with fewer bugs. Check out this [introduction to using static types in JavaScript](https://medium.com/@preethikasireddy/why-use-static-types-in-javascript-part-1-8382da1e0adb) if you are new to this concept.

Recent versions of [Flow](http://flowtype.org/) work with Create React App projects out of the box.

To add Flow to a Create React App project, follow these steps:

1. Run npm install --save-dev flow-bin (or yarn add --dev flow-bin).
2. Add "flow": "flow" to the scripts section of your package.json.
3. Run npm run flow -- init (or yarn flow -- init) to create a [.flowconfig file](https://flowtype.org/docs/advanced-configuration.html) in the root directory.
4. Add // @flow to any files you want to type check (for example, to src/App.js).

Now you can run npm run flow (or yarn flow) to check the files for type errors. You can optionally use an IDE like [Nuclide](https://nuclide.io/docs/languages/flow/) for a better integrated experience. In the future we plan to integrate it into Create React App even more closely.

To learn more about Flow, check out [its documentation](https://flowtype.org/).

**Adding Custom Environment Variables**

Note: this feature is available with react-scripts@0.2.3 and higher.

Your project can consume variables declared in your environment as if they were declared locally in your JS files. By default you will have NODE\_ENV defined for you, and any other environment variables starting with REACT\_APP\_.

**The environment variables are embedded during the build time**. Since Create React App produces a static HTML/CSS/JS bundle, it can’t possibly read them at runtime. To read them at runtime, you would need to load HTML into memory on the server and replace placeholders in runtime, just like [described here](https://github.com/baluragala/redux-saga-examples#injecting-data-from-the-server-into-the-page). Alternatively you can rebuild the app on the server anytime you change them.

Note: You must create custom environment variables beginning with REACT\_APP\_. Any other variables except NODE\_ENV will be ignored to avoid accidentally [exposing a private key on the machine that could have the same name](https://github.com/facebookincubator/create-react-app/issues/865#issuecomment-252199527). Changing any environment variables will require you to restart the development server if it is running.

These environment variables will be defined for you on process.env. For example, having an environment variable named REACT\_APP\_SECRET\_CODE will be exposed in your JS as process.env.REACT\_APP\_SECRET\_CODE.

There is also a special built-in environment variable called NODE\_ENV. You can read it from process.env.NODE\_ENV. When you run npm start, it is always equal to 'development', when you run npm test it is always equal to 'test', and when you run npm run build to make a production bundle, it is always equal to 'production'. **You cannot override NODE\_ENV manually.** This prevents developers from accidentally deploying a slow development build to production.

These environment variables can be useful for displaying information conditionally based on where the project is deployed or consuming sensitive data that lives outside of version control.

First, you need to have environment variables defined. For example, let’s say you wanted to consume a secret defined in the environment inside a <form>:

render() {

return (

<div>

<small>You are running this application in <b>{process.env.NODE\_ENV}</b> mode.</small>

<form>

<input type="hidden" defaultValue={process.env.REACT\_APP\_SECRET\_CODE} />

</form>

</div>

);

}

During the build, process.env.REACT\_APP\_SECRET\_CODE will be replaced with the current value of the REACT\_APP\_SECRET\_CODE environment variable. Remember that the NODE\_ENV variable will be set for you automatically.

When you load the app in the browser and inspect the <input>, you will see its value set to abcdef, and the bold text will show the environment provided when using npm start:

<div>

<small>You are running this application in <b>development</b> mode.</small>

<form>

<input type="hidden" value="abcdef" />

</form>

</div>

The above form is looking for a variable called REACT\_APP\_SECRET\_CODE from the environment. In order to consume this value, we need to have it defined in the environment. This can be done using two ways: either in your shell or in a .env file. Both of these ways are described in the next few sections.

Having access to the NODE\_ENV is also useful for performing actions conditionally:

if (process.env.NODE\_ENV !== 'production') {

analytics.disable();

}

When you compile the app with npm run build, the minification step will strip out this condition, and the resulting bundle will be smaller.

**Referencing Environment Variables in the HTML**

Note: this feature is available with react-scripts@0.9.0 and higher.

You can also access the environment variables starting with REACT\_APP\_ in the public/index.html. For example:

<title>%REACT\_APP\_WEBSITE\_NAME%</title>

Note that the caveats from the above section apply:

* Apart from a few built-in variables (NODE\_ENV and PUBLIC\_URL), variable names must start with REACT\_APP\_ to work.
* The environment variables are injected at build time. If you need to inject them at runtime, [follow this approach instead](https://github.com/baluragala/redux-saga-examples#generating-dynamic-meta-tags-on-the-server).

**Adding Temporary Environment Variables In Your Shell**

Defining environment variables can vary between OSes. It’s also important to know that this manner is temporary for the life of the shell session.

**Windows (cmd.exe)**

set REACT\_APP\_SECRET\_CODE=abcdef&&npm start

(Note: the lack of whitespace is intentional.)

**Linux, macOS (Bash)**

REACT\_APP\_SECRET\_CODE=abcdef npm start

**Adding Development Environment Variables In .env**

Note: this feature is available with react-scripts@0.5.0 and higher.

To define permanent environment variables, create a file called .env in the root of your project:

REACT\_APP\_SECRET\_CODE=abcdef

These variables will act as the defaults if the machine does not explicitly set them.  
Please refer to the [dotenv documentation](https://github.com/motdotla/dotenv) for more details.

Note: If you are defining environment variables for development, your CI and/or hosting platform will most likely need these defined as well. Consult their documentation how to do this. For example, see the documentation for [Travis CI](https://docs.travis-ci.com/user/environment-variables/) or [Heroku](https://devcenter.heroku.com/articles/config-vars).

**Can I Use Decorators?**

Many popular libraries use [decorators](https://medium.com/google-developers/exploring-es7-decorators-76ecb65fb841) in their documentation.  
Create React App doesn’t support decorator syntax at the moment because:

* It is an experimental proposal and is subject to change.
* The current specification version is not officially supported by Babel.
* If the specification changes, we won’t be able to write a codemod because we don’t use them internally at Facebook.

However in many cases you can rewrite decorator-based code without decorators just as fine.  
Please refer to these two threads for reference:

* [#214](https://github.com/facebookincubator/create-react-app/issues/214)
* [#411](https://github.com/facebookincubator/create-react-app/issues/411)

Create React App will add decorator support when the specification advances to a stable stage.

**Integrating with an API Backend**

These tutorials will help you to integrate your app with an API backend running on another port, using fetch() to access it.

**Node**

Check out [this tutorial](https://www.fullstackreact.com/articles/using-create-react-app-with-a-server/). You can find the companion GitHub repository [here](https://github.com/fullstackreact/food-lookup-demo).

**Ruby on Rails**

Check out [this tutorial](https://www.fullstackreact.com/articles/how-to-get-create-react-app-to-work-with-your-rails-api/). You can find the companion GitHub repository [here](https://github.com/fullstackreact/food-lookup-demo-rails).

**Proxying API Requests in Development**

Note: this feature is available with react-scripts@0.2.3 and higher.

People often serve the front-end React app from the same host and port as their backend implementation.  
For example, a production setup might look like this after the app is deployed:

/ - static server returns index.html with React app

/todos - static server returns index.html with React app

/api/todos - server handles any /api/\* requests using the backend implementation

Such setup is **not** required. However, if you **do** have a setup like this, it is convenient to write requests like fetch('/api/todos') without worrying about redirecting them to another host or port during development.

To tell the development server to proxy any unknown requests to your API server in development, add a proxy field to your package.json, for example:

"proxy": "http://localhost:4000",

This way, when you fetch('/api/todos') in development, the development server will recognize that it’s not a static asset, and will proxy your request to http://localhost:4000/api/todos as a fallback. The development server will only attempt to send requests without a text/html accept header to the proxy.

Conveniently, this avoids [CORS issues](http://stackoverflow.com/questions/21854516/understanding-ajax-cors-and-security-considerations) and error messages like this in development:

Fetch API cannot load http://localhost:4000/api/todos. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:3000' is therefore not allowed access. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

Keep in mind that proxy only has effect in development (with npm start), and it is up to you to ensure that URLs like /api/todos point to the right thing in production. You don’t have to use the /api prefix. Any unrecognized request without a text/html accept header will be redirected to the specified proxy.

The proxy option supports HTTP, HTTPS and WebSocket connections.  
If the proxy option is **not** flexible enough for you, alternatively you can:

* Enable CORS on your server ([here’s how to do it for Express](http://enable-cors.org/server_expressjs.html)).
* Use [environment variables](https://github.com/baluragala/redux-saga-examples#adding-custom-environment-variables) to inject the right server host and port into your app.

**Using HTTPS in Development**

Note: this feature is available with react-scripts@0.4.0 and higher.

You may require the dev server to serve pages over HTTPS. One particular case where this could be useful is when using [the "proxy" feature](https://github.com/baluragala/redux-saga-examples#proxying-api-requests-in-development) to proxy requests to an API server when that API server is itself serving HTTPS.

To do this, set the HTTPS environment variable to true, then start the dev server as usual with npm start:

**Windows (cmd.exe)**

set HTTPS=true&&npm start

(Note: the lack of whitespace is intentional.)

**Linux, macOS (Bash)**

HTTPS=true npm start

Note that the server will use a self-signed certificate, so your web browser will almost definitely display a warning upon accessing the page.

**Generating Dynamic <meta> Tags on the Server**

Since Create React App doesn’t support server rendering, you might be wondering how to make <meta> tags dynamic and reflect the current URL. To solve this, we recommend to add placeholders into the HTML, like this:

<!doctype html>

<html lang="en">

<head>

<meta property="og:title" content="\_\_OG\_TITLE\_\_">

<meta property="og:description" content="\_\_OG\_DESCRIPTION\_\_">

Then, on the server, regardless of the backend you use, you can read index.html into memory and replace \_\_OG\_TITLE\_\_, \_\_OG\_DESCRIPTION\_\_, and any other placeholders with values depending on the current URL. Just make sure to sanitize and escape the interpolated values so that they are safe to embed into HTML!

If you use a Node server, you can even share the route matching logic between the client and the server. However duplicating it also works fine in simple cases.

**Pre-Rendering into Static HTML Files**

If you’re hosting your build with a static hosting provider you can use [react-snapshot](https://www.npmjs.com/package/react-snapshot) to generate HTML pages for each route, or relative link, in your application. These pages will then seamlessly become active, or “hydrated”, when the JavaScript bundle has loaded.

There are also opportunities to use this outside of static hosting, to take the pressure off the server when generating and caching routes.

The primary benefit of pre-rendering is that you get the core content of each page *with* the HTML payload—regardless of whether or not your JavaScript bundle successfully downloads. It also increases the likelihood that each route of your application will be picked up by search engines.

You can read more about [zero-configuration pre-rendering (also called snapshotting) here](https://medium.com/superhighfives/an-almost-static-stack-6df0a2791319).

**Injecting Data from the Server into the Page**

Similarly to the previous section, you can leave some placeholders in the HTML that inject global variables, for example:

<!doctype html>

<html lang="en">

<head>

<script>

window.SERVER\_DATA = \_\_SERVER\_DATA\_\_;

</script>

Then, on the server, you can replace \_\_SERVER\_DATA\_\_ with a JSON of real data right before sending the response. The client code can then read window.SERVER\_DATA to use it. **Make sure to**[**sanitize the JSON before sending it to the client**](https://medium.com/node-security/the-most-common-xss-vulnerability-in-react-js-applications-2bdffbcc1fa0)**as it makes your app vulnerable to XSS attacks.**

**Running Tests**

Note: this feature is available with react-scripts@0.3.0 and higher.  
[Read the migration guide to learn how to enable it in older projects!](https://github.com/facebookincubator/create-react-app/blob/master/CHANGELOG.md#migrating-from-023-to-030)

Create React App uses [Jest](https://facebook.github.io/jest/) as its test runner. To prepare for this integration, we did a [major revamp](https://facebook.github.io/jest/blog/2016/09/01/jest-15.html) of Jest so if you heard bad things about it years ago, give it another try.

Jest is a Node-based runner. This means that the tests always run in a Node environment and not in a real browser. This lets us enable fast iteration speed and prevent flakiness.

While Jest provides browser globals such as window thanks to [jsdom](https://github.com/tmpvar/jsdom), they are only approximations of the real browser behavior. Jest is intended to be used for unit tests of your logic and your components rather than the DOM quirks.

We recommend that you use a separate tool for browser end-to-end tests if you need them. They are beyond the scope of Create React App.

**Filename Conventions**

Jest will look for test files with any of the following popular naming conventions:

* Files with .js suffix in \_\_tests\_\_ folders.
* Files with .test.js suffix.
* Files with .spec.js suffix.

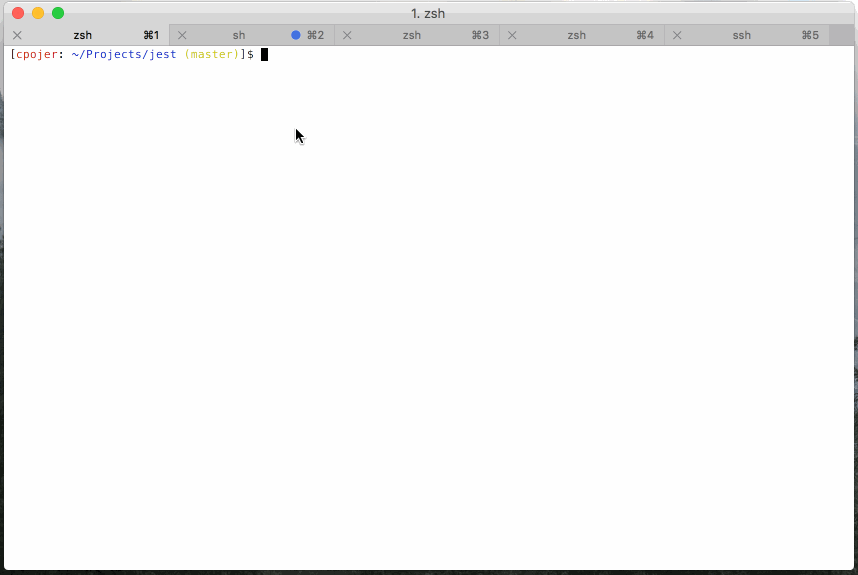
The .test.js / .spec.js files (or the \_\_tests\_\_ folders) can be located at any depth under the src top level folder.

We recommend to put the test files (or \_\_tests\_\_ folders) next to the code they are testing so that relative imports appear shorter. For example, if App.test.js and App.js are in the same folder, the test just needs to import App from './App' instead of a long relative path. Colocation also helps find tests more quickly in larger projects.

**Command Line Interface**

When you run npm test, Jest will launch in the watch mode. Every time you save a file, it will re-run the tests, just like npm start recompiles the code.

The watcher includes an interactive command-line interface with the ability to run all tests, or focus on a search pattern. It is designed this way so that you can keep it open and enjoy fast re-runs. You can learn the commands from the “Watch Usage” note that the watcher prints after every run:

[](https://camo.githubusercontent.com/8100466c15fea66872569734f24e0cb8b4b76fd02a2700973126929bab7cecc4/687474703a2f2f66616365626f6f6b2e6769746875622e696f2f6a6573742f696d672f626c6f672f31352d77617463682e676966)

**Version Control Integration**

By default, when you run npm test, Jest will only run the tests related to files changed since the last commit. This is an optimization designed to make your tests runs fast regardless of how many tests you have. However it assumes that you don’t often commit the code that doesn’t pass the tests.

Jest will always explicitly mention that it only ran tests related to the files changed since the last commit. You can also press a in the watch mode to force Jest to run all tests.

Jest will always run all tests on a [continuous integration](https://github.com/baluragala/redux-saga-examples#continuous-integration) server or if the project is not inside a Git or Mercurial repository.

**Writing Tests**

To create tests, add it() (or test()) blocks with the name of the test and its code. You may optionally wrap them in describe() blocks for logical grouping but this is neither required nor recommended.

Jest provides a built-in expect() global function for making assertions. A basic test could look like this:

import sum from './sum';

it('sums numbers', () => {

expect(sum(1, 2)).toEqual(3);

expect(sum(2, 2)).toEqual(4);

});

All expect() matchers supported by Jest are [extensively documented here](http://facebook.github.io/jest/docs/expect.html).  
You can also use [jest.fn() and expect(fn).toBeCalled()](http://facebook.github.io/jest/docs/expect.html#tohavebeencalled) to create “spies” or mock functions.

**Testing Components**

There is a broad spectrum of component testing techniques. They range from a “smoke test” verifying that a component renders without throwing, to shallow rendering and testing some of the output, to full rendering and testing component lifecycle and state changes.

Different projects choose different testing tradeoffs based on how often components change, and how much logic they contain. If you haven’t decided on a testing strategy yet, we recommend that you start with creating simple smoke tests for your components:

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

it('renders without crashing', () => {

const div = document.createElement('div');

ReactDOM.render(<App />, div);

});

This test mounts a component and makes sure that it didn’t throw during rendering. Tests like this provide a lot value with very little effort so they are great as a starting point, and this is the test you will find in src/App.test.js.

When you encounter bugs caused by changing components, you will gain a deeper insight into which parts of them are worth testing in your application. This might be a good time to introduce more specific tests asserting specific expected output or behavior.

If you’d like to test components in isolation from the child components they render, we recommend using [shallow() rendering API](http://airbnb.io/enzyme/docs/api/shallow.html) from [Enzyme](http://airbnb.io/enzyme/). You can write a smoke test with it too:

npm install --save-dev enzyme react-addons-test-utils

import React from 'react';

import { shallow } from 'enzyme';

import App from './App';

it('renders without crashing', () => {

shallow(<App />);

});

Unlike the previous smoke test using ReactDOM.render(), this test only renders <App> and doesn’t go deeper. For example, even if <App> itself renders a <Button> that throws, this test will pass. Shallow rendering is great for isolated unit tests, but you may still want to create some full rendering tests to ensure the components integrate correctly. Enzyme supports [full rendering with mount()](http://airbnb.io/enzyme/docs/api/mount.html), and you can also use it for testing state changes and component lifecycle.

You can read the [Enzyme documentation](http://airbnb.io/enzyme/) for more testing techniques. Enzyme documentation uses Chai and Sinon for assertions but you don’t have to use them because Jest provides built-in expect() and jest.fn() for spies.

Here is an example from Enzyme documentation that asserts specific output, rewritten to use Jest matchers:

import React from 'react';

import { shallow } from 'enzyme';

import App from './App';

it('renders welcome message', () => {

const wrapper = shallow(<App />);

const welcome = <h2>Welcome to React</h2>;

// expect(wrapper.contains(welcome)).to.equal(true);

expect(wrapper.contains(welcome)).toEqual(true);

});

All Jest matchers are [extensively documented here](http://facebook.github.io/jest/docs/expect.html).  
Nevertheless you can use a third-party assertion library like [Chai](http://chaijs.com/) if you want to, as described below.

Additionally, you might find [jest-enzyme](https://github.com/blainekasten/enzyme-matchers) helpful to simplify your tests with readable matchers. The above contains code can be written simpler with jest-enzyme.

expect(wrapper).toContainReact(welcome)

To setup jest-enzyme with Create React App, follow the instructions for [initializing your test environment](https://github.com/baluragala/redux-saga-examples#initializing-test-environment) to import jest-enzyme.

npm install --save-dev jest-enzyme

// src/setupTests.js

import 'jest-enzyme';

**Using Third Party Assertion Libraries**

We recommend that you use expect() for assertions and jest.fn() for spies. If you are having issues with them please [file those against Jest](https://github.com/facebook/jest/issues/new), and we’ll fix them. We intend to keep making them better for React, supporting, for example, [pretty-printing React elements as JSX](https://github.com/facebook/jest/pull/1566).

However, if you are used to other libraries, such as [Chai](http://chaijs.com/) and [Sinon](http://sinonjs.org/), or if you have existing code using them that you’d like to port over, you can import them normally like this:

import sinon from 'sinon';

import { expect } from 'chai';

and then use them in your tests like you normally do.

**Initializing Test Environment**

Note: this feature is available with react-scripts@0.4.0 and higher.

If your app uses a browser API that you need to mock in your tests or if you just need a global setup before running your tests, add a src/setupTests.js to your project. It will be automatically executed before running your tests.

For example:

**src/setupTests.js**

const localStorageMock = {

getItem: jest.fn(),

setItem: jest.fn(),

clear: jest.fn()

};

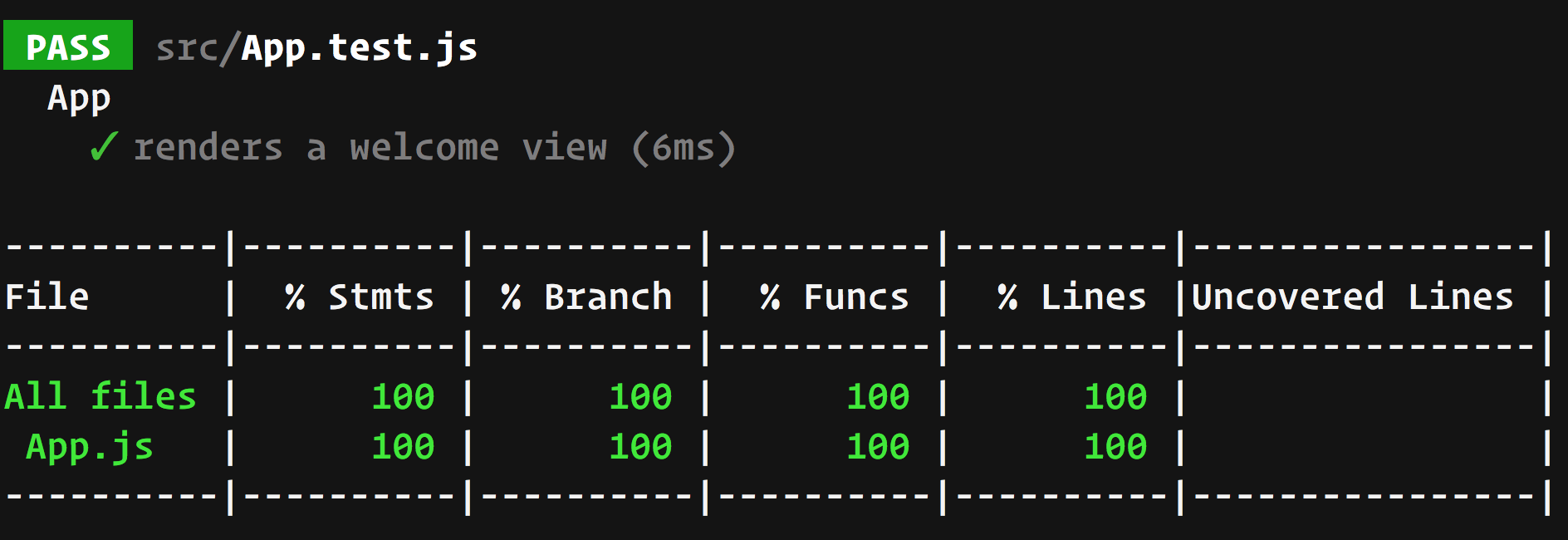
global.localStorage = localStorageMock

**Focusing and Excluding Tests**

You can replace it() with xit() to temporarily exclude a test from being executed.  
Similarly, fit() lets you focus on a specific test without running any other tests.

**Coverage Reporting**

Jest has an integrated coverage reporter that works well with ES6 and requires no configuration.  
Run npm test -- --coverage (note extra -- in the middle) to include a coverage report like this:

[](https://camo.githubusercontent.com/9517ff38949b3c3b2d06b3856aebaf77a1df0d587ed3776793b59e4736e61ad3/687474703a2f2f692e696d6775722e636f6d2f356246686e54532e706e67)

Note that tests run much slower with coverage so it is recommended to run it separately from your normal workflow.

**Continuous Integration**

By default npm test runs the watcher with interactive CLI. However, you can force it to run tests once and finish the process by setting an environment variable called CI.

When creating a build of your application with npm run build linter warnings are not checked by default. Like npm test, you can force the build to perform a linter warning check by setting the environment variable CI. If any warnings are encountered then the build fails.

Popular CI servers already set the environment variable CI by default but you can do this yourself too:

**On CI servers**

**Travis CI**

1. Following the [Travis Getting started](https://docs.travis-ci.com/user/getting-started/) guide for syncing your GitHub repository with Travis. You may need to initialize some settings manually in your [profile](https://travis-ci.org/profile) page.
2. Add a .travis.yml file to your git repository.

language: node\_js

node\_js:

- 4

- 6

cache:

directories:

- node\_modules

script:

- npm test

- npm run build

1. Trigger your first build with a git push.
2. [Customize your Travis CI Build](https://docs.travis-ci.com/user/customizing-the-build/) if needed.

**On your own environment**

**Windows (cmd.exe)**

set CI=true&&npm test

set CI=true&&npm run build

(Note: the lack of whitespace is intentional.)

**Linux, macOS (Bash)**

CI=true npm test

CI=true npm run build

The test command will force Jest to run tests once instead of launching the watcher.

If you find yourself doing this often in development, please [file an issue](https://github.com/facebookincubator/create-react-app/issues/new) to tell us about your use case because we want to make watcher the best experience and are open to changing how it works to accommodate more workflows.

The build command will check for linter warnings and fail if any are found.

**Disabling jsdom**

By default, the package.json of the generated project looks like this:

// ...

"scripts": {

// ...

"test": "react-scripts test --env=jsdom"

}

If you know that none of your tests depend on [jsdom](https://github.com/tmpvar/jsdom), you can safely remove --env=jsdom, and your tests will run faster.  
To help you make up your mind, here is a list of APIs that **need jsdom**:

* Any browser globals like window and document
* [ReactDOM.render()](https://facebook.github.io/react/docs/top-level-api.html#reactdom.render)
* [TestUtils.renderIntoDocument()](https://facebook.github.io/react/docs/test-utils.html#renderintodocument) ([a shortcut](https://github.com/facebook/react/blob/34761cf9a252964abfaab6faf74d473ad95d1f21/src/test/ReactTestUtils.js#L83-L91) for the above)
* [mount()](http://airbnb.io/enzyme/docs/api/mount.html) in [Enzyme](http://airbnb.io/enzyme/index.html)

In contrast, **jsdom is not needed** for the following APIs:

* [TestUtils.createRenderer()](https://facebook.github.io/react/docs/test-utils.html#shallow-rendering) (shallow rendering)
* [shallow()](http://airbnb.io/enzyme/docs/api/shallow.html) in [Enzyme](http://airbnb.io/enzyme/index.html)

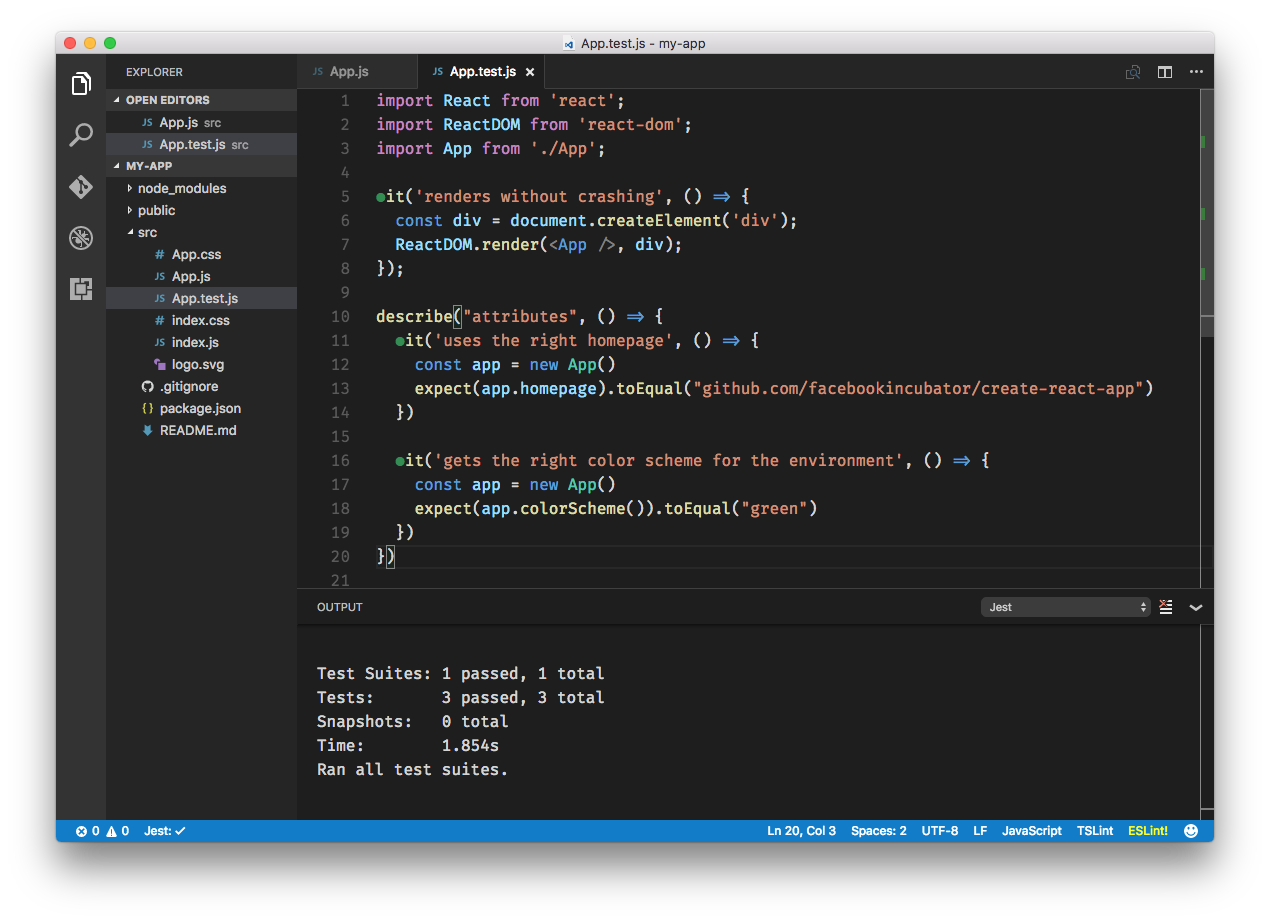
Finally, jsdom is also not needed for [snapshot testing](http://facebook.github.io/jest/blog/2016/07/27/jest-14.html).

**Snapshot Testing**

Snapshot testing is a feature of Jest that automatically generates text snapshots of your components and saves them on the disk so if the UI output changes, you get notified without manually writing any assertions on the component output. [Read more about snapshot testing.](http://facebook.github.io/jest/blog/2016/07/27/jest-14.html)

**Editor Integration**

If you use [Visual Studio Code](https://code.visualstudio.com/), there is a [Jest extension](https://github.com/orta/vscode-jest) which works with Create React App out of the box. This provides a lot of IDE-like features while using a text editor: showing the status of a test run with potential fail messages inline, starting and stopping the watcher automatically, and offering one-click snapshot updates.

[](https://cloud.githubusercontent.com/assets/49038/20795349/a032308a-b7c8-11e6-9b34-7eeac781003f.png)

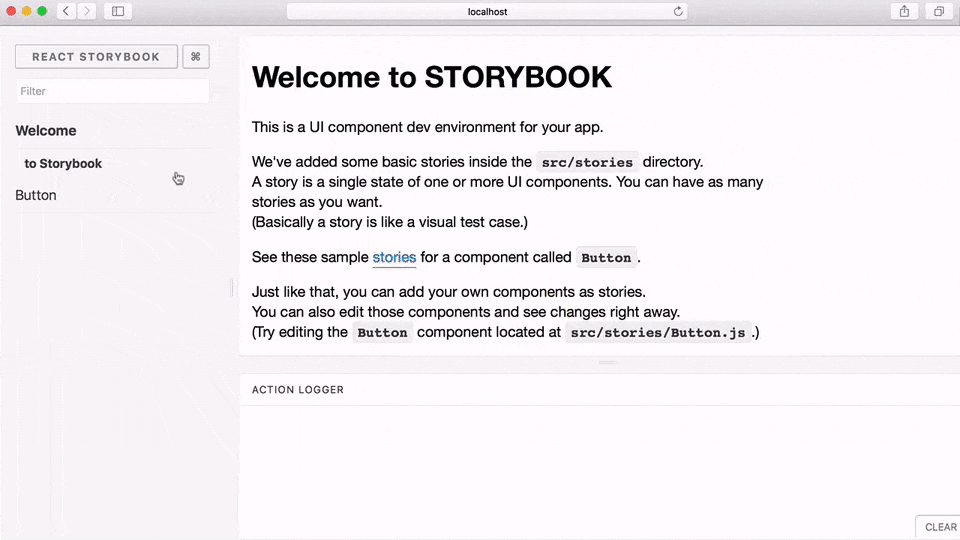
**Developing Components in Isolation**

Usually, in an app, you have a lot of UI components, and each of them has many different states. For an example, a simple button component could have following states:

* With a text label.
* With an emoji.
* In the disabled mode.

Usually, it’s hard to see these states without running a sample app or some examples.

Create React App doesn’t include any tools for this by default, but you can easily add [React Storybook](https://github.com/kadirahq/react-storybook) to your project. **It is a third-party tool that lets you develop components and see all their states in isolation from your app**.

[](https://camo.githubusercontent.com/c8122da0c79ee1c1abd4095238110dd4af7d67efb8314281baeec981e6e0c5b4/687474703a2f2f692e696d6775722e636f6d2f374349415770422e676966)

You can also deploy your Storybook as a static app. This way, everyone in your team can view and review different states of UI components without starting a backend server or creating an account in your app.

**Here’s how to setup your app with Storybook:**

First, install the following npm package globally:

npm install -g getstorybook

Then, run the following command inside your app’s directory:

getstorybook

After that, follow the instructions on the screen.

Learn more about React Storybook:

* Screencast: [Getting Started with React Storybook](https://egghead.io/lessons/react-getting-started-with-react-storybook)
* [GitHub Repo](https://github.com/kadirahq/react-storybook)
* [Documentation](https://getstorybook.io/docs)
* [Snapshot Testing](https://github.com/kadirahq/storyshots) with React Storybook

**Making a Progressive Web App**

You can turn your React app into a [Progressive Web App](https://developers.google.com/web/progressive-web-apps/) by following the steps in [this repository](https://github.com/jeffposnick/create-react-pwa).

**Deployment**

npm run build creates a build directory with a production build of your app. Set up your favourite HTTP server so that a visitor to your site is served index.html, and requests to static paths like /static/js/main.<hash>.js are served with the contents of the /static/js/main.<hash>.js file.

**Static Server**

For environments using [Node](https://nodejs.org/), the easiest way to handle this would be to install [serve](https://github.com/zeit/serve) and let it handle the rest:

npm install -g serve

serve -s build

The last command shown above will serve your static site on the port **5000**. Like many of [serve](https://github.com/zeit/serve)’s internal settings, the port can be adjusted using the -p or --port flags.

Run this command to get a full list of the options available:

serve -h

**Other Solutions**

You don’t necessarily need a static server in order to run a Create React App project in production. It works just as fine integrated into an existing dynamic one.

Here’s a programmatic example using [Node](https://nodejs.org/) and [Express](http://expressjs.com/):

const express = require('express');

const path = require('path');

const app = express();

app.use(express.static('./build'));

app.get('/', function (req, res) {

res.sendFile(path.join(\_\_dirname, './build', 'index.html'));

});

app.listen(9000);

The choice of your server software isn’t important either. Since Create React App is completely platform-agnostic, there’s no need to explicitly use Node.

The build folder with static assets is the only output produced by Create React App.

However this is not quite enough if you use client-side routing. Read the next section if you want to support URLs like /todos/42 in your single-page app.

**Serving Apps with Client-Side Routing**

If you use routers that use the HTML5 [pushState history API](https://developer.mozilla.org/en-US/docs/Web/API/History_API#Adding_and_modifying_history_entries) under the hood (for example, [React Router](https://github.com/ReactTraining/react-router) with browserHistory), many static file servers will fail. For example, if you used React Router with a route for /todos/42, the development server will respond to localhost:3000/todos/42 properly, but an Express serving a production build as above will not.

This is because when there is a fresh page load for a /todos/42, the server looks for the file build/todos/42 and does not find it. The server needs to be configured to respond to a request to /todos/42 by serving index.html. For example, we can amend our Express example above to serve index.html for any unknown paths:

app.use(express.static('./build'));

-app.get('/', function (req, res) {

+app.get('/\*', function (req, res) {

res.sendFile(path.join(\_\_dirname, './build', 'index.html'));

});

If you’re using [Apache](https://httpd.apache.org/), you need to create a .htaccess file in the public folder that looks like this:

Options -MultiViews

RewriteEngine On

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteRule ^ index.html [QSA,L]

It will get copied to the build folder when you run npm run build.

Now requests to /todos/42 will be handled correctly both in development and in production.

**Building for Relative Paths**

By default, Create React App produces a build assuming your app is hosted at the server root.  
To override this, specify the homepage in your package.json, for example:

"homepage": "http://mywebsite.com/relativepath",

This will let Create React App correctly infer the root path to use in the generated HTML file.

**Serving the Same Build from Different Paths**

Note: this feature is available with react-scripts@0.9.0 and higher.

If you are not using the HTML5 pushState history API or not using client-side routing at all, it is unnecessary to specify the URL from which your app will be served. Instead, you can put this in your package.json:

"homepage": ".",

This will make sure that all the asset paths are relative to index.html. You will then be able to move your app from http://mywebsite.com to http://mywebsite.com/relativepath or even http://mywebsite.com/relative/path without having to rebuild it.

**Azure**

See [this](https://medium.com/@to_pe/deploying-create-react-app-on-microsoft-azure-c0f6686a4321) blog post on how to deploy your React app to [Microsoft Azure](https://azure.microsoft.com/).

**Firebase**

Install the Firebase CLI if you haven’t already by running npm install -g firebase-tools. Sign up for a [Firebase account](https://console.firebase.google.com/) and create a new project. Run firebase login and login with your previous created Firebase account.

Then run the firebase init command from your project’s root. You need to choose the **Hosting: Configure and deploy Firebase Hosting sites** and choose the Firebase project you created in the previous step. You will need to agree with database.rules.json being created, choose build as the public directory, and also agree to **Configure as a single-page app** by replying with y.

=== Project Setup

First, let's associate this project directory with a Firebase project.

You can create multiple project aliases by running firebase use --add,

but for now we'll just set up a default project.

? What Firebase project do you want to associate as default? Example app (example-app-fd690)

=== Database Setup

Firebase Realtime Database Rules allow you to define how your data should be

structured and when your data can be read from and written to.

? What file should be used for Database Rules? database.rules.json

✔ Database Rules for example-app-fd690 have been downloaded to database.rules.json.

Future modifications to database.rules.json will update Database Rules when you run

firebase deploy.

=== Hosting Setup

Your public directory is the folder (relative to your project directory) that

will contain Hosting assets to uploaded with firebase deploy. If you

have a build process for your assets, use your build's output directory.

? What do you want to use as your public directory? build

? Configure as a single-page app (rewrite all urls to /index.html)? Yes

✔ Wrote build/index.html

i Writing configuration info to firebase.json...

i Writing project information to .firebaserc...

✔ Firebase initialization complete!

Now, after you create a production build with npm run build, you can deploy it by running firebase deploy.

=== Deploying to 'example-app-fd690'...

i deploying database, hosting

✔ database: rules ready to deploy.

i hosting: preparing build directory for upload...

Uploading: [============================== ] 75%✔ hosting: build folder uploaded successfully

✔ hosting: 8 files uploaded successfully

i starting release process (may take several minutes)...

✔ Deploy complete!

Project Console: https://console.firebase.google.com/project/example-app-fd690/overview

Hosting URL: https://example-app-fd690.firebaseapp.com

For more information see [Add Firebase to your JavaScript Project](https://firebase.google.com/docs/web/setup).

**GitHub Pages**

Note: this feature is available with react-scripts@0.2.0 and higher.

**Step 1: Add homepage to package.json**

**The step below is important!**  
**If you skip it, your app will not deploy correctly.**

Open your package.json and add a homepage field:

"homepage": "https://myusername.github.io/my-app",

Create React App uses the homepage field to determine the root URL in the built HTML file.

**Step 2: Install gh-pages and add deploy to scripts in package.json**

Now, whenever you run npm run build, you will see a cheat sheet with instructions on how to deploy to GitHub Pages.

To publish it at <https://myusername.github.io/my-app>, run:

npm install --save-dev gh-pages

Add the following scripts in your package.json:

// ...

"scripts": {

// ...

"predeploy": "npm run build",

"deploy": "gh-pages -d build"

}

The predeploy script will run automatically before deploy is run.

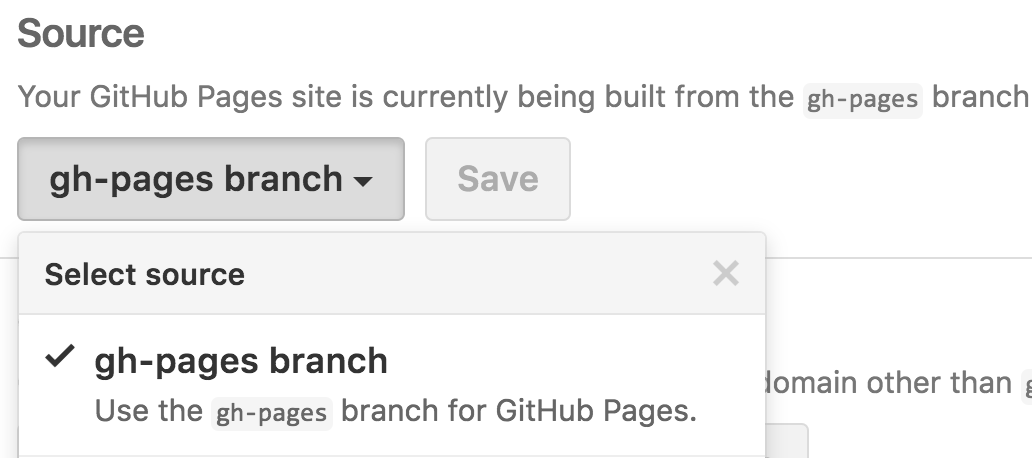
**Step 3: Deploy the site by running npm run deploy**

Then run:

npm run deploy

**Step 4: Ensure your project’s settings use gh-pages**

Finally, make sure **GitHub Pages** option in your GitHub project settings is set to use the gh-pages branch:

[](https://camo.githubusercontent.com/ae23eb7faebc46a07ba12c50e15edda89b3c2153387b8eff6c3aea94eaf6b31c/687474703a2f2f692e696d6775722e636f6d2f48556a4572396c2e706e67)

**Step 5: Optionally, configure the domain**

You can configure a custom domain with GitHub Pages by adding a CNAME file to the public/ folder.

**Notes on client-side routing**

GitHub Pages doesn’t support routers that use the HTML5 pushState history API under the hood (for example, React Router using browserHistory). This is because when there is a fresh page load for a url like http://user.github.io/todomvc/todos/42, where /todos/42 is a frontend route, the GitHub Pages server returns 404 because it knows nothing of /todos/42. If you want to add a router to a project hosted on GitHub Pages, here are a couple of solutions:

* You could switch from using HTML5 history API to routing with hashes. If you use React Router, you can switch to hashHistory for this effect, but the URL will be longer and more verbose (for example, http://user.github.io/todomvc/#/todos/42?\_k=yknaj). [Read more](https://github.com/reactjs/react-router/blob/master/docs/guides/Histories.md#histories) about different history implementations in React Router.
* Alternatively, you can use a trick to teach GitHub Pages to handle 404 by redirecting to your index.html page with a special redirect parameter. You would need to add a 404.html file with the redirection code to the build folder before deploying your project, and you’ll need to add code handling the redirect parameter to index.html. You can find a detailed explanation of this technique [in this guide](https://github.com/rafrex/spa-github-pages).

**Heroku**

Use the [Heroku Buildpack for Create React App](https://github.com/mars/create-react-app-buildpack).  
You can find instructions in [Deploying React with Zero Configuration](https://blog.heroku.com/deploying-react-with-zero-configuration).

**Resolving Heroku Deployment Errors**

Sometimes npm run build works locally but fails during deploy via Heroku. Following are the most common cases.

**"Module not found: Error: Cannot resolve 'file' or 'directory'"**

If you get something like this:

remote: Failed to create a production build. Reason:

remote: Module not found: Error: Cannot resolve 'file' or 'directory'

MyDirectory in /tmp/build\_1234/src

It means you need to ensure that the lettercase of the file or directory you import matches the one you see on your filesystem or on GitHub.

This is important because Linux (the operating system used by Heroku) is case sensitive. So MyDirectory and mydirectory are two distinct directories and thus, even though the project builds locally, the difference in case breaks the import statements on Heroku remotes.

**"Could not find a required file."**

If you exclude or ignore necessary files from the package you will see a error similar this one:

remote: Could not find a required file.

remote: Name: `index.html`

remote: Searched in: /tmp/build\_a2875fc163b209225122d68916f1d4df/public

remote:

remote: npm ERR! Linux 3.13.0-105-generic

remote: npm ERR! argv "/tmp/build\_a2875fc163b209225122d68916f1d4df/.heroku/node/bin/node" "/tmp/build\_a2875fc163b209225122d68916f1d4df/.heroku/node/bin/npm" "run" "build"

In this case, ensure that the file is there with the proper lettercase and that’s not ignored on your local .gitignore or ~/.gitignore\_global.

**Modulus**

See the [Modulus blog post](http://blog.modulus.io/deploying-react-apps-on-modulus) on how to deploy your react app to Modulus.

**Netlify**

**To do a manual deploy to Netlify’s CDN:**

npm install netlify-cli

netlify deploy

Choose build as the path to deploy.

**To setup continuous delivery:**

With this setup Netlify will build and deploy when you push to git or open a pull request:

1. [Start a new netlify project](https://app.netlify.com/signup)
2. Pick your Git hosting service and select your repository
3. Click Build your site

**Support for client-side routing:**

To support pushState, make sure to create a public/\_redirects file with the following rewrite rules:

/\* /index.html 200

When you build the project, Create React App will place the public folder contents into the build output.

**Now**

[now](https://zeit.co/now) offers a zero-configuration single-command deployment.

1. Install the now command-line tool either via the recommended [desktop tool](https://zeit.co/download) or via node with npm install -g now.
2. Install serve by running npm install --save serve.
3. Add this line to scripts in package.json:
4. "now-start": "serve build/",
5. Run now from your project directory. You will see a **now.sh** URL in your output like this:
6. > Ready! https://your-project-dirname-tpspyhtdtk.now.sh (copied to clipboard)

Paste that URL into your browser when the build is complete, and you will see your deployed app.

Details are available in [this article.](https://zeit.co/blog/now-static)

**S3 and CloudFront**

See this [blog post](https://medium.com/@omgwtfmarc/deploying-create-react-app-to-s3-or-cloudfront-48dae4ce0af) on how to deploy your React app to Amazon Web Services [S3](https://aws.amazon.com/s3) and [CloudFront](https://aws.amazon.com/cloudfront/).

**Surge**

Install the Surge CLI if you haven’t already by running npm install -g surge. Run the surge command and log in you or create a new account. You just need to specify the *build* folder and your custom domain, and you are done.

email: email@domain.com

password: \*\*\*\*\*\*\*\*

project path: /path/to/project/build

size: 7 files, 1.8 MB

domain: create-react-app.surge.sh

upload: [====================] 100%, eta: 0.0s

propagate on CDN: [====================] 100%

plan: Free

users: email@domain.com

IP Address: X.X.X.X

Success! Project is published and running at create-react-app.surge.sh

Note that in order to support routers that use HTML5 pushState API, you may want to rename the index.html in your build folder to 200.html before deploying to Surge. This [ensures that every URL falls back to that file](https://surge.sh/help/adding-a-200-page-for-client-side-routing).

**Advanced Configuration**

You can adjust various development and production settings by setting environment variables in your shell or with [.env](https://github.com/baluragala/redux-saga-examples#adding-development-environment-variables-in-env).

| **Variable** | **Development** | **Production** | **Usage** |
| --- | --- | --- | --- |
| BROWSER | ✅ | ❌ | By default, Create React App will open the default system browser, favoring Chrome on macOS. Specify a [browser](https://github.com/sindresorhus/opn#app) to override this behavior, or set it to none to disable it completely. |
| HOST | ✅ | ❌ | By default, the development web server binds to localhost. You may use this variable to specify a different host. |
| PORT | ✅ | ❌ | By default, the development web server will attempt to listen on port 3000 or prompt you to attempt the next available port. You may use this variable to specify a different port. |
| HTTPS | ✅ | ❌ | When set to true, Create React App will run the development server in https mode. |
| PUBLIC\_URL | ❌ | ✅ | Create React App assumes your application is hosted at the serving web server's root or a subpath as specified in [package.json (homepage)](https://github.com/baluragala/redux-saga-examples#building-for-relative-paths). Normally, Create React App ignores the hostname. You may use this variable to force assets to be referenced verbatim to the url you provide (hostname included). This may be particularly useful when using a CDN to host your application. |
| CI | 🔶 | ✅ | When set to true, Create React App treats warnings as failures in the build. It also makes the test runner non-watching. Most CIs set this flag by default. |

**Troubleshooting**

**npm start doesn’t detect changes**

When you save a file while npm start is running, the browser should refresh with the updated code.  
If this doesn’t happen, try one of the following workarounds:

* If your project is in a Dropbox folder, try moving it out.
* If the watcher doesn’t see a file called sagaHelpers.js and you’re referencing it by the folder name, you [need to restart the watcher](https://github.com/facebookincubator/create-react-app/issues/1164) due to a Webpack bug.
* Some editors like Vim and IntelliJ have a “safe write” feature that currently breaks the watcher. You will need to disable it. Follow the instructions in [“Working with editors supporting safe write”](https://webpack.github.io/docs/webpack-dev-server.html#working-with-editors-ides-supporting-safe-write).
* If your project path contains parentheses, try moving the project to a path without them. This is caused by a [Webpack watcher bug](https://github.com/webpack/watchpack/issues/42).
* On Linux and macOS, you might need to [tweak system settings](https://webpack.github.io/docs/troubleshooting.html#not-enough-watchers) to allow more watchers.
* If the project runs inside a virtual machine such as (a Vagrant provisioned) VirtualBox, create an .env file in your project directory if it doesn’t exist, and add CHOKIDAR\_USEPOLLING=true to it. This ensures that the next time you run npm start, the watcher uses the polling mode, as necessary inside a VM.

If none of these solutions help please leave a comment [in this thread](https://github.com/facebookincubator/create-react-app/issues/659).

**npm test hangs on macOS Sierra**

If you run npm test and the console gets stuck after printing react-scripts test --env=jsdom to the console there might be a problem with your [Watchman](https://facebook.github.io/watchman/) installation as described in [facebookincubator/create-react-app#713](https://github.com/facebookincubator/create-react-app/issues/713).

We recommend deleting node\_modules in your project and running npm install (or yarn if you use it) first. If it doesn't help, you can try one of the numerous workarounds mentioned in these issues:

* [facebook/jest#1767](https://github.com/facebook/jest/issues/1767)
* [facebook/watchman#358](https://github.com/facebook/watchman/issues/358)
* [ember-cli/ember-cli#6259](https://github.com/ember-cli/ember-cli/issues/6259)

It is reported that installing Watchman 4.7.0 or newer fixes the issue. If you use [Homebrew](http://brew.sh/), you can run these commands to update it:

watchman shutdown-server

brew update

brew reinstall watchman

You can find [other installation methods](https://facebook.github.io/watchman/docs/install.html#build-install) on the Watchman documentation page.

If this still doesn’t help, try running launchctl unload -F ~/Library/LaunchAgents/com.github.facebook.watchman.plist.

There are also reports that *uninstalling* Watchman fixes the issue. So if nothing else helps, remove it from your system and try again.

**npm run build silently fails**

It is reported that npm run build can fail on machines with no swap space, which is common in cloud environments. If [the symptoms are matching](https://github.com/facebookincubator/create-react-app/issues/1133#issuecomment-264612171), consider adding some swap space to the machine you’re building on, or build the project locally.

**npm run build fails on Heroku**

This may be a problem with case sensitive filenames. Please refer to [this section](https://github.com/baluragala/redux-saga-examples#resolving-heroku-deployment-errors).