This project was bootstrapped with 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.

Table of Contents

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, 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 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.

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`**, otherwise 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 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 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 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 Browsers

By default, the generated project uses the latest version of React.

You can refer to the React documentation for more information about supported browsers.

Supported Language Features and Polyfills

This project supports a superset of the latest JavaScript standard.
In addition to ES6 syntax features, it also supports:

- Exponentiation Operator (ES2016).
- Async/await (ES2017).
- Object Rest/Spread Properties (stage 3 proposal).
- Dynamic import() (stage 3 proposal)
- Class Fields and Static Properties (part of stage 3 proposal).
- JSX and Flow syntax.

Learn more about different proposal stages.

While we recommend using experimental proposals with some caution, Facebook heavily uses these features in the product code, so we intend to provide codemods if any of these proposals change in the future.

Note that **the project only includes a few ES6 polyfills**:

- `Object.assign()` via `object-assign`.
- `Promise` via `promise`.
- `fetch()` via `whatwg-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.

Also note that using some newer syntax features like `for...of` or `[...nonArrayValue]` causes Babel to emit code that depends on ES6 runtime features and might not work without a polyfill. When in doubt, use Babel REPL to see what any specific syntax compiles down to.

Syntax Highlighting in the Editor

To configure the syntax highlighting in your favorite text editor, head to the relevant Babel documentation page 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.
> It also only works with npm 3 or 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. Then, add a file called `.eslintrc` to the project root:

```
{
  "extends": "react-app"
}
```

Now your editor should report the linting warnings.

Note that even if you edit your `.eslintrc` file further, these changes will **only affect the editor integration**. They won't affect the terminal and in-browser lint output. This is because Create React App intentionally provides a minimal set of rules that find common mistakes.

If you want to enforce a coding style for your project, consider using Prettier instead of ESLint style rules.

Debugging in the Editor

This feature is currently only supported by Visual Studio Code and WebStorm.

Visual Studio Code and WebStorm support 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.

Visual Studio Code

You would need to have the latest version of VS Code and VS Code Chrome Debugger Extension 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": "${workspaceRoot}/src",
    "sourceMapPathOverrides": {
      "webpack:///src/*": "${webRoot}/*"
    }
  }]
}
```

> Note: the URL may be different if you've made adjustments via the HOST or PORT environment variables.

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.

Having problems with VS Code Debugging? Please see their troubleshooting guide.

WebStorm

You would need to have WebStorm and JetBrains IDE Support Chrome extension installed.

In the WebStorm menu `Run` select `Edit Configurations...`. Then click `+` and select `JavaScript Debug`. Paste `http://localhost:3000` into the URL field and save the configuration.

> Note: the URL may be different if you've made adjustments via the HOST or PORT environment variables.

Start your app by running `npm start`, then press `^D` on macOS or `F9` on Windows and Linux or click the green debug icon to start debugging in WebStorm.

The same way you can debug your application in IntelliJ IDEA Ultimate, PhpStorm, PyCharm Pro, and RubyMine.

Formatting Code Automatically

Prettier is an opinionated code formatter with support for JavaScript, CSS and JSON. With Prettier you can format the code you write automatically to ensure a code style within your project. See the Prettier's GitHub page for more information, and look at this page to see it in action.

To format our code whenever we make a commit in git, we need to install the following dependencies:

```
npm install --save husky lint-staged prettier
```

Alternatively you may use `yarn`:

```
yarn add husky lint-staged prettier
```

- husky makes it easy to use githooks as if they are npm scripts.
- lint-staged allows us to run scripts on staged files in git. See this blog post about lint-staged to learn more about it.
- prettier is the JavaScript formatter we will run before commits.

Now we can make sure every file is formatted correctly by adding a few lines to the package.json in the project root.

Add the following line to scripts section:

```
  "scripts": {
+   "precommit": "lint-staged",
    "start": "react-scripts start",
    "build": "react-scripts build",
```

Next we add a 'lint-staged' field to the package.json, for example:

```
  "dependencies": {
    // ...
  },
+ "lint-staged": {
+   "src/**/*.{js,jsx,json,css}": [
+     "prettier --single-quote --write",
+     "git add"
+   ]
+ },
  "scripts": {
```

Now, whenever you make a commit, Prettier will format the changed files automatically. You can also run `./node_modules/.bin/prettier --single-quote --write "src/**/*.{js,jsx,json,css}"` to format your entire project for the first time.

Next you might want to integrate Prettier in your favorite editor. Read the section on Editor Integration on the Prettier GitHub page.

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 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` API. For more complex scenarios when you want to change the title from React components, you can use 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. Alternatively, you can pre-build each page as a static HTML file which then loads the JavaScript bundle, which is covered here.

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 react-router
```

Alternatively you may use `yarn`:

```
yarn add react-router
```

This works for any library, not just `react-router`.

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` 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. 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?
- Exploring ES6: Modules
- Understanding ES6: Modules

Code Splitting

Instead of downloading the entire app before users can use it, code splitting allows you to split your code into small chunks which you can then load on demand.

This project setup supports code splitting via dynamic `import()`. Its proposal is in stage 3. The `import()` function-like form takes the module name as an argument and returns a `Promise` which always resolves to the namespace object of the module.

Here is an example:

`moduleA.js`

```
const moduleA = 'Hello';

export { moduleA };
```

`App.js`

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

class App extends Component {
  handleClick = () => {
    import('./moduleA')
      .then(({ moduleA }) => {
        // Use moduleA
      })
      .catch(err => {
        // Handle failure
      });
  };

  render() {
    return (
      <div>
        <button onClick={this.handleClick}>Load</button>
      </div>
    );
  }
}

export default App;
```

This will make `moduleA.js` and all its unique dependencies as a separate chunk that only loads after the user clicks the 'Load' button.

You can also use it with `async` / `await` syntax if you prefer it.

With React Router

If you are using React Router check out this tutorial on how to use code splitting with it. You can find the companion GitHub repository here.

Also check out the Code Splitting section in React documentation.

Adding a Stylesheet

This project setup uses Webpack 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. 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/index.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 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.

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).

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 --save node-sass-chokidar
```

Alternatively you may use `yarn`:

```
yarn add node-sass-chokidar
```

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

```
   "scripts": {
+    "build-css": "node-sass-chokidar src/ -o src/",
+    "watch-css": "npm run build-css && node-sass-chokidar 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.

To enable importing files without using relative paths, you can add the `--include-path` option to the command in `package.json`.

```
"build-css": "node-sass-chokidar --include-path ./src --include-path
./node_modules src/ -o src/",
"watch-css": "npm run build-css && node-sass-chokidar --include-path ./src --
include-path ./node_modules src/ -o src/ --watch --recursive",
```

This will allow you to do imports like

```
@import 'styles/_colors.scss'; // assuming a styles directory under src/
@import 'nprogress/nprogress'; // importing a css file from the nprogress node
module
```

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 npm-run-all
```

Alternatively you may use `yarn`:

```
yarn add npm-run-all
```

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

```
   "scripts": {
     "build-css": "node-sass-chokidar src/ -o src/",
     "watch-css": "npm run build-css && node-sass-chokidar 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-js": "react-scripts build",
+    "build": "npm-run-all build-css build-js",
     "test": "react-scripts test --env=jsdom",
     "eject": "react-scripts eject"
   }
```

Now running `npm start` and `npm run build` also builds Sass files.

Why `node-sass-chokidar`?

`node-sass` has been reported as having the following issues:

- `node-sass --watch` has been reported to have *performance issues* in certain conditions when used in a virtual machine or with docker.

- Infinite styles compiling #1939
- `node-sass` has been reported as having issues with detecting new files in a directory #1891

`node-sass-chokidar` is used here as it addresses these issues.

Adding Images, Fonts, and Files

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

You can **`import` a file right in a JavaScript module**. This tells Webpack to include that file in the bundle. Unlike CSS imports, importing a file gives you a string value. This value is the final path you can reference in your code, e.g. as the `src` attribute of an image or the `href` of a link to a PDF.

To reduce the number of requests to the server, importing images that are less than 10,000 bytes returns a data URI instead of a path. This applies to the following file extensions: bmp, gif, jpg, jpeg, and png. SVG files are excluded due to #1153.

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. 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 and 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, 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`.
- You have thousands of images and need to dynamically reference their paths.
- You want to include a small script like `pace.js` 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 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 --save react-bootstrap bootstrap@3
```

Alternatively you may use `yarn`:

```
yarn add react-bootstrap bootstrap@3
```

Import Bootstrap CSS and optionally Bootstrap theme CSS in the beginning of your `src/index.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` 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 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 if you are new to this concept.

Recent versions of Flow 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 flow-bin` (or `yarn add 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 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 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.

Adding a Router

Create React App doesn't prescribe a specific routing solution, but React Router is the most popular one.

To add it, run:

```
npm install --save react-router-dom
```

Alternatively you may use `yarn`:

```
yarn add react-router-dom
```

To try it, delete all the code in `src/App.js` and replace it with any of the examples on its website. The Basic Example is a good place to get started.

Note that you may need to configure your production server to support client-side routing before deploying your app.

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. 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. 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.

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: Quotes around the variable assignment are required to avoid a trailing whitespace.)

Windows (Powershell)

```
($env:REACT_APP_SECRET_CODE = "abcdef") -and (npm start)
```

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
```

> 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. Changing any environment variables will require you to restart the development server if it is running.

`.env` files **should be** checked into source control (with the exclusion of `.env*.local`).

What other `.env` files can be used?

> Note: this feature is **available with `react-scripts@1.0.0` and higher**.

- `.env`: Default.
- `.env.local`: Local overrides. **This file is loaded for all environments except test.**
- `.env.development`, `.env.test`, `.env.production`: Environment-specific settings.
- `.env.development.local`, `.env.test.local`, `.env.production.local`: Local overrides of environment-specific settings.

Files on the left have more priority than files on the right:

- `npm start`: `.env.development.local`, `.env.development`, `.env.local`, `.env`
- `npm run build`: `.env.production.local`, `.env.production`, `.env.local`, `.env`
- `npm test`: `.env.test.local`, `.env.test`, `.env` (note `.env.local` is missing)

These variables will act as the defaults if the machine does not explicitly set them.
Please refer to the dotenv documentation 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 or Heroku.

Expanding Environment Variables In `.env`

> Note: this feature is available with `react-scripts@1.1.0` and higher.

Expand variables already on your machine for use in your `.env` file (using dotenv-expand).

For example, to get the environment variable `npm_package_version`:

```
REACT_APP_VERSION=$npm_package_version
# also works:
# REACT_APP_VERSION=${npm_package_version}
```

Or expand variables local to the current `.env` file:

```
DOMAIN=www.example.com
REACT_APP_FOO=$DOMAIN/foo
REACT_APP_BAR=$DOMAIN/bar
```

Can I Use Decorators?

Many popular libraries use decorators 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
- #411

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

Fetching Data with AJAX Requests

React doesn't prescribe a specific approach to data fetching, but people commonly use either a library like axios or the `fetch()` API provided by the browser. Conveniently, Create React App includes a polyfill for `fetch()` so you can use it without worrying about the browser support.

The global `fetch` function allows to easily makes AJAX requests. It takes in a URL as an input and returns a `Promise` that resolves to a `Response` object. You can find more information about `fetch` here.

This project also includes a Promise polyfill which provides a full implementation of Promises/A+. A Promise represents the eventual result of an asynchronous operation, you can find more information about Promises here and here. Both axios and `fetch()` use Promises under the hood. You can also use the `async / await` syntax to reduce the callback nesting.

You can learn more about making AJAX requests from React components in the FAQ entry on the React website.

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. You can find the companion GitHub repository here.

Ruby on Rails

Check out this tutorial. You can find the companion GitHub repository here.

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 `text/html` in its `Accept` header to the proxy.

Conveniently, this avoids CORS issues 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:

- Configure the proxy yourself
- Enable CORS on your server (here's how to do it for Express).
- Use environment variables to inject the right server host and port into your app.

"Invalid Host Header" Errors After Configuring Proxy

When you enable the `proxy` option, you opt into a more strict set of host checks. This is necessary because leaving the backend open to remote hosts makes your computer vulnerable to DNS rebinding attacks. The issue is explained in this article and this issue.

This shouldn’t affect you when developing on `localhost`, but if you develop remotely like described here, you will see this error in the browser after enabling the `proxy` option:

> Invalid Host header

To work around it, you can specify your public development host in a file called `.env.development` in the root of your project:

```
HOST=mypublicdevhost.com
```

If you restart the development server now and load the app from the specified host, it should work.

If you are still having issues or if you’re using a more exotic environment like a cloud editor, you can bypass the host check completely by adding a line to `.env.development.local`. **Note that this is dangerous and exposes your machine to remote code execution from malicious websites:**

```
# NOTE: THIS IS DANGEROUS!
# It exposes your machine to attacks from the websites you visit.
DANGEROUSLY_DISABLE_HOST_CHECK=true
```

We don’t recommend this approach.

Configuring the Proxy Manually

> Note: this feature is available with `react-scripts@1.0.0` and higher.

If the `proxy` option is **not** flexible enough for you, you can specify an object in the following form (in `package.json`).
You may also specify any configuration value `http-proxy-middleware` or `http-proxy` supports.

```
{
  // ...
  "proxy": {
    "/api": {
      "target": "<url>",
      "ws": true
      // ...
    }
  }
  // ...
}
```

All requests matching this path will be proxies, no exceptions. This includes requests for `text/html`, which the standard `proxy` option does not proxy.

If you need to specify multiple proxies, you may do so by specifying additional entries. Matches are regular expressions, so that you can use a regexp to match multiple paths.

```
{
  // ...
  "proxy": {
    // Matches any request starting with /api
    "/api": {
      "target": "<url_1>",
      "ws": true
      // ...
    },
    // Matches any request starting with /foo
    "/foo": {
      "target": "<url_2>",
      "ssl": true,
      "pathRewrite": {
        "^/foo": "/foo/beta"
      }
      // ...
    },
    // Matches /bar/abc.html but not /bar/sub/def.html
    "/bar/[^/]*[.]html": {
      "target": "<url_3>",
      // ...
    },
    // Matches /baz/abc.html and /baz/sub/def.html
    "/baz/.*/.*[.]html": {
      "target": "<url_4>"
      // ...
    }
  }
  // ...
}
```

Configuring a WebSocket Proxy

When setting up a WebSocket proxy, there are a some extra considerations to be aware of.

If you’re using a WebSocket engine like Socket.io, you must have a Socket.io server running that you can use as the proxy target. Socket.io will not work with a standard WebSocket server. Specifically, don't expect Socket.io to work with the websocket.org echo test.

There’s some good documentation available for setting up a Socket.io server.

Standard WebSockets **will** work with a standard WebSocket server as well as the websocket.org echo test. You can use libraries like ws for the server, with native WebSockets in the browser.

Either way, you can proxy WebSocket requests manually in `package.json`:

```
{
  // ...
  "proxy": {
    "/socket": {
      // Your compatible WebSocket server
      "target": "ws://<socket_url>",
      // Tell http-proxy-middleware that this is a WebSocket proxy.
      // Also allows you to proxy WebSocket requests without an additional HTTP
request
      // https://github.com/chimurai/http-proxy-middleware#external-websocket-
upgrade
      "ws": true
      // ...
    }
  }
  // ...
}
```

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 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
```

Windows (Powershell)

```
($env:HTTPS = $true) -and (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 or react-snap 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.

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 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!

Create React App uses Jest as its test runner. To prepare for this integration, we did a major revamp 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, 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.

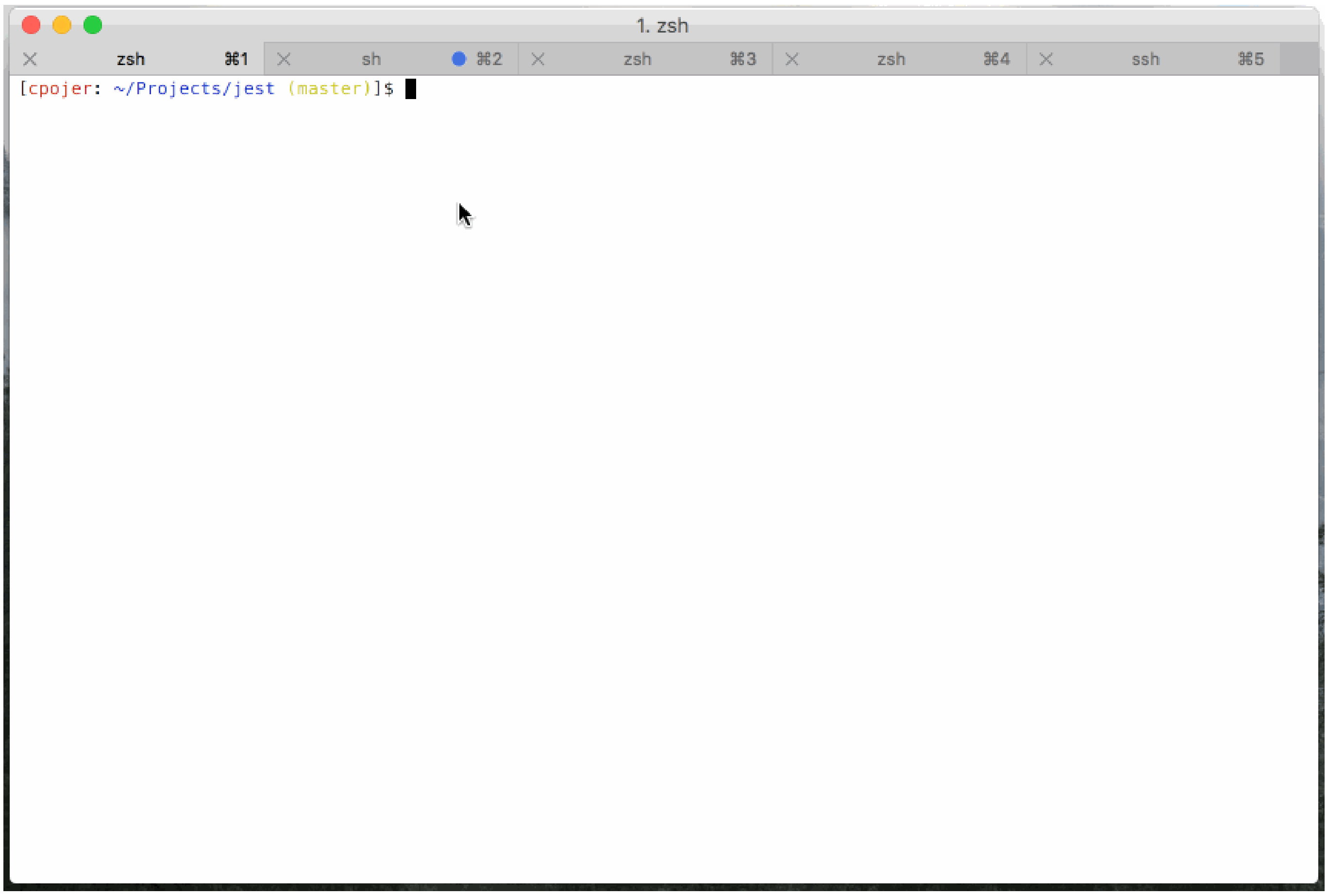
The `.test.js` / `.spec.js` files (or the `__tests__` folders) can be located at any depth under the `src` top level folder.

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:



```
[cpojer: ~/Projects/jest (master)]$
```

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 run 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 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.
You can also use `jest.fn()` and `expect(fn).toBeCalled()` 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 of 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 from Enzyme. To install it, run:

```
npm install --save enzyme enzyme-adapter-react-16 react-test-renderer
```

Alternatively you may use `yarn`:

```
yarn add enzyme enzyme-adapter-react-16 react-test-renderer
```

As of Enzyme 3, you will need to install Enzyme along with an Adapter corresponding to the version of React you are using. (The examples above use the adapter for React 16.)

The adapter will also need to be configured in your global setup file:

`src/setupTests.js`

```
import { configure } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

configure({ adapter: new Adapter() });
```

> Note: Keep in mind that if you decide to "eject" before creating `src/setupTests.js`, the resulting `package.json` file won't contain any reference to it. Read here to learn how to add this after ejecting.

Now you can write a smoke test with it:

```
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()`, and you can also use it for testing state changes and component lifecycle.

You can read the Enzyme documentation 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.
Nevertheless you can use a third-party assertion library like Chai if you want to, as described below.

Additionally, you might find jest-enzyme helpful to simplify your tests with readable matchers. The above `contains` code can be written more simply with jest-enzyme.

```
expect(wrapper).toContainReact(welcome)
```

To enable this, install `jest-enzyme`:

```
npm install --save jest-enzyme
```

Alternatively you may use `yarn`:

```
yarn add jest-enzyme
```

Import it in `src/setupTests.js` to make its matchers available in every test:

```
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, and we'll fix them. We intend to keep making them better for React, supporting, for example, pretty-printing React elements as JSX.

However, if you are used to other libraries, such as Chai and Sinon, 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`

```js
const localStorageMock = {
  getItem: jest.fn(),
  setItem: jest.fn(),
  clear: jest.fn()
};
global.localStorage = localStorageMock
```

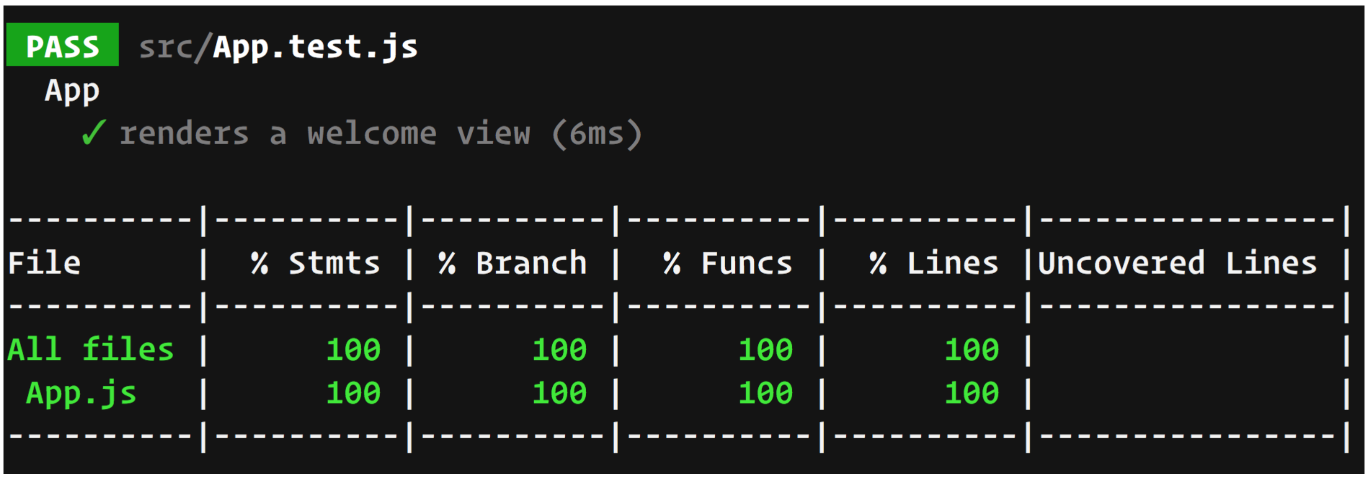
> Note: Keep in mind that if you decide to "eject" before creating `src/setupTests.js`, the resulting `package.json` file won't contain any reference to it, so you should manually create the property `setupTestFrameworkScriptFile` in the configuration for Jest, something like the following:
>
> ```js
> "jest": {
>   // ...
>   "setupTestFrameworkScriptFile": "<rootDir>/src/setupTests.js"
>  }
> ```

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:



```
 PASS  src/App.test.js
  App
    ✓ renders a welcome view (6ms)

----------|----------|----------|----------|----------|----------------|
File      |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------|----------|----------|----------|----------|----------------|
All files |      100 |      100 |      100 |      100 |                |
 App.js   |      100 |      100 |      100 |      100 |                |
----------|----------|----------|----------|----------|----------------|
```

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

Configuration

The default Jest coverage configuration can be overriden by adding any of the following supported keys to a Jest config in your package.json.

Supported overrides:

- `collectCoverageFrom`
- `coverageReporters`
- `coverageThreshold`
- `snapshotSerializers`

Example package.json:

```
{
  "name": "your-package",
  "jest": {
    "collectCoverageFrom" : [
      "src/**/*.{js,jsx}",
      "!<rootDir>/node_modules/",
      "!<rootDir>/path/to/dir/"
    ],
    "coverageThreshold": {
      "global": {
        "branches": 90,
        "functions": 90,
        "lines": 90,
        "statements": 90
      }
    },
    "coverageReporters": ["text"],
    "snapshotSerializers": ["my-serializer-module"]
  }
}
```

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 guide for syncing your GitHub repository with Travis. You may need to initialize some settings manually in your profile page.
2. Add a `.travis.yml` file to your git repository.

```
language: node_js
node_js:
  - 6
cache:
  directories:
    - node_modules
script:
  - npm run build
  - npm test
```

1. Trigger your first build with a git push.
2. Customize your Travis CI Build if needed.

CircleCI

Follow this article to set up CircleCI with a Create React App project.

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.)

Windows (Powershell)

```
($env:CI = $true) -and (npm test)
```

```
($env:CI = $true) -and (npm run build)
```

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 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": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom"
```

If you know that none of your tests depend on jsdom, you can safely remove `--env=jsdom`, and your tests will run faster:

```
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
-   "test": "react-scripts test --env=jsdom"
+   "test": "react-scripts test"
```

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()`
- `TestUtils.renderIntoDocument()` (a shortcut for the above)
- `mount()` in Enzyme

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

- `TestUtils.createRenderer()` (shallow rendering)
- `shallow()` in Enzyme

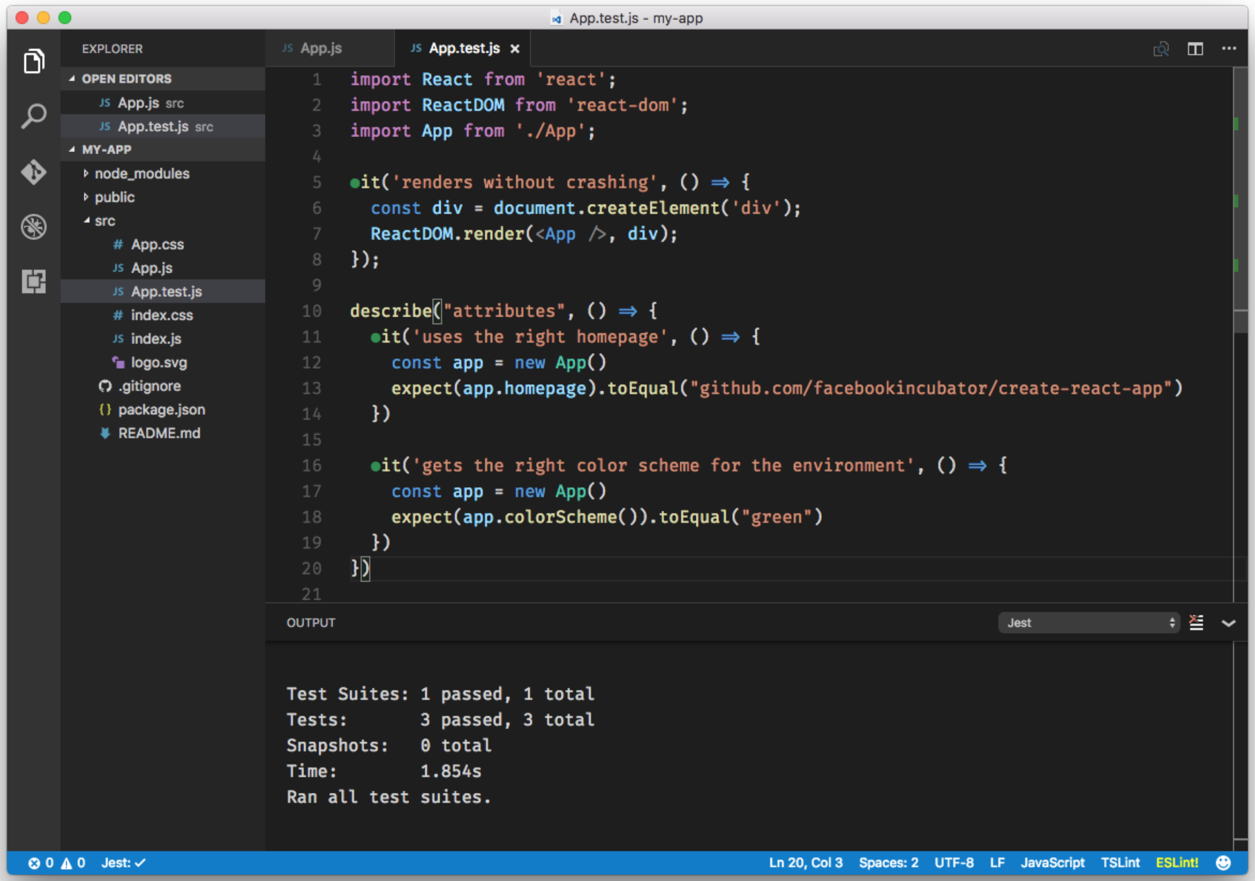
Finally, jsdom is also not needed for snapshot testing.

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.

Editor Integration

If you use Visual Studio Code, there is a Jest extension 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.



```
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);
});

describe("attributes", () => {
  it('uses the right homepage', () => {
    const app = new App()
    expect(app.homepage).toEqual("github.com/facebookincubator/create-react-app")
  })

  it('gets the right color scheme for the environment', () => {
    const app = new App()
    expect(app.colorScheme()).toEqual("green")
  })
})
```

```
Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.854s
Ran all test suites.
```

Debugging Tests

There are various ways to setup a debugger for your Jest tests. We cover debugging in Chrome and Visual Studio Code.

> Note: debugging tests requires Node 8 or higher.

Debugging Tests in Chrome

Add the following to the `scripts` section in your project's `package.json`

```
"scripts": {
    "test:debug": "react-scripts --inspect-brk test --runInBand --env=jsdom"
  }
```

Place `debugger;` statements in any test and run:

```
$ npm run test:debug
```

This will start running your Jest tests, but pause before executing to allow a debugger to attach to the process.

Open the following in Chrome

```
about:inspect
```

After opening that link, the Chrome Developer Tools will be displayed. Select `inspect` on your process and a breakpoint will be set at the first line of the react script (this is done simply to give you time to open the developer tools and to prevent Jest from executing before you have time to do so). Click the button that looks like a "play" button in the upper right hand side of the screen to continue execution. When Jest executes the test that contains the debugger statement, execution will pause and you can examine the current scope and call stack.

> Note: the --runInBand cli option makes sure Jest runs test in the same process rather than spawning processes for individual tests. Normally Jest parallelizes test runs across processes but it is hard to debug many processes at the same time.

Debugging Tests in Visual Studio Code

Debugging Jest tests is supported out of the box for Visual Studio Code.

Use the following `launch.json` configuration file:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Debug CRA Tests",
      "type": "node",
      "request": "launch",
      "runtimeExecutable": "${workspaceRoot}/node_modules/.bin/react-scripts",
      "args": [
        "test",
        "--runInBand",
        "--no-cache",
        "--env=jsdom"
      ],
      "cwd": "${workspaceRoot}",
      "protocol": "inspector",
      "console": "integratedTerminal",
      "internalConsoleOptions": "neverOpen"
    }
  ]
}
```

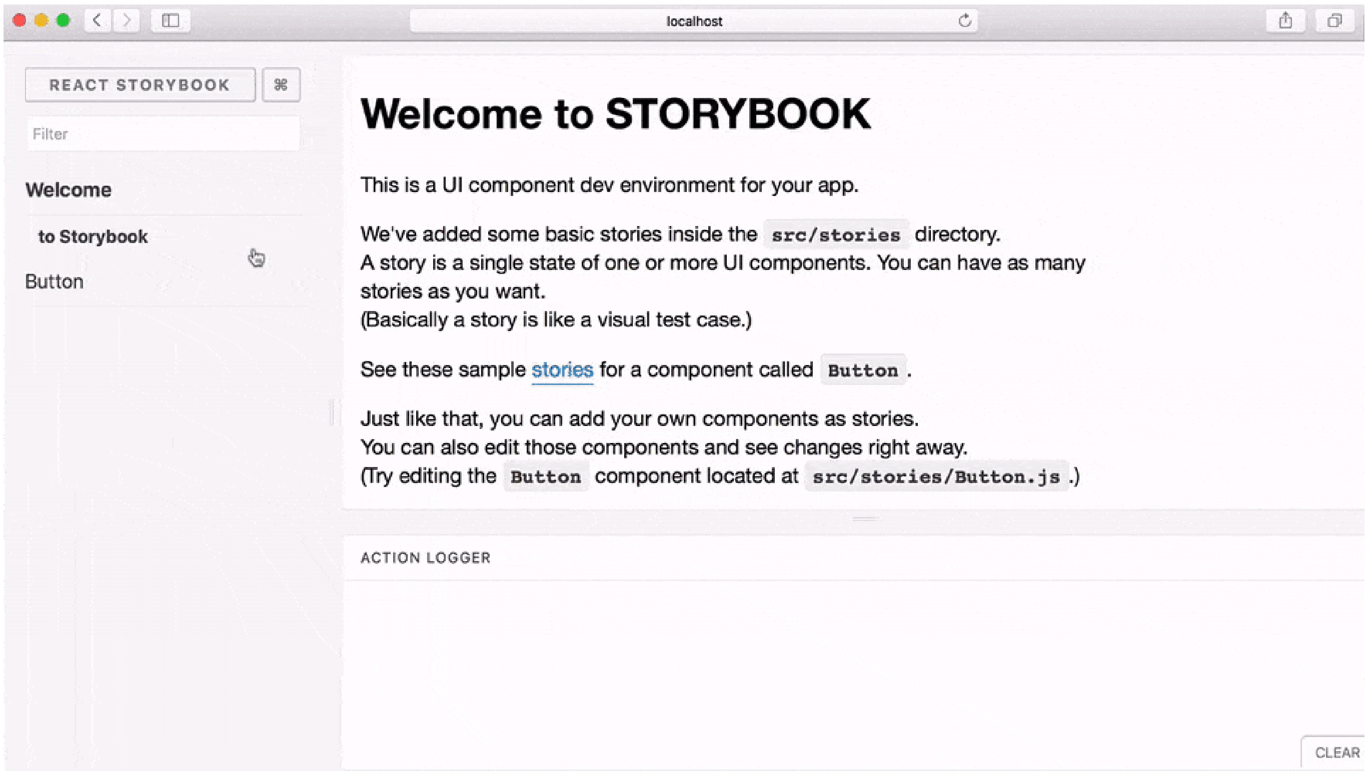
Developing Components in Isolation

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:

- In a regular state, with a text label.
- In the disabled mode.
- In a loading state.

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 Storybook for React (source) or React Styleguidist (source) to your project. **These are third-party tools that let you develop components and see all their states in isolation from your app**.



Welcome to STORYBOOK

This is a UI component dev environment for your app.

We've added some basic stories inside the `src/stories` directory.
A story is a single state of one or more UI components. You can have as many stories as you want.
(Basically a story is like a visual test case.)

See these sample stories for a component called `Button`.

Just like that, you can add your own components as stories.
You can also edit those components and see changes right away.
(Try editing the `Button` component located at `src/stories/Button.js`.)

You can also deploy your Storybook or style guide 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.

Getting Started with Storybook

Storybook is a development environment for React UI components. It allows you to browse a component library, view the different states of each component, and interactively develop and test components.

First, install the following npm package globally:

```
npm install -g @storybook/cli
```

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
- GitHub Repo
- Documentation
- Snapshot Testing UI with Storybook + addon/storyshot

Getting Started with Styleguidist

Styleguidist combines a style guide, where all your components are presented on a single page with their props documentation and usage examples, with an environment for developing components in isolation, similar to Storybook. In Styleguidist you write examples in Markdown, where each code snippet is rendered as a live editable playground.

First, install Styleguidist:

```
npm install --save react-styleguidist
```

Alternatively you may use `yarn`:

```
yarn add react-styleguidist
```

Then, add these scripts to your `package.json`:

```
   "scripts": {
+    "styleguide": "styleguidist server",
+    "styleguide:build": "styleguidist build",
     "start": "react-scripts start",
```

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

```
npm run styleguide
```

After that, follow the instructions on the screen.

Learn more about React Styleguidist:

- GitHub Repo

- Documentation

Publishing Components to npm

Create React App doesn't provide any built-in functionality to publish a component to npm. If you're ready to extract a component from your project so other people can use it, we recommend moving it to a separate directory outside of your project and then using a tool like nwb to prepare it for publishing.

Making a Progressive Web App

By default, the production build is a fully functional, offline-first Progressive Web App.

Progressive Web Apps are faster and more reliable than traditional web pages, and provide an engaging mobile experience:

- All static site assets are cached so that your page loads fast on subsequent visits, regardless of network connectivity (such as 2G or 3G). Updates are downloaded in the background.
- Your app will work regardless of network state, even if offline. This means your users will be able to use your app at 10,000 feet and on the subway.
- On mobile devices, your app can be added directly to the user's home screen, app icon and all. You can also re-engage users using web **push notifications**. This eliminates the need for the app store.

The `sw-precache-webpack-plugin` is integrated into production configuration, and it will take care of generating a service worker file that will automatically precache all of your local assets and keep them up to date as you deploy updates. The service worker will use a cache-first strategy for handling all requests for local assets, including the initial HTML, ensuring that your web app is reliably fast, even on a slow or unreliable network.

Opting Out of Caching

If you would prefer not to enable service workers prior to your initial production deployment, then remove the call to `registerServiceWorker()` from `src/index.js`.

If you had previously enabled service workers in your production deployment and have decided that you would like to disable them for all your existing users, you can swap out the call to `registerServiceWorker()` in `src/index.js` first by modifying the service worker import:

```
import { unregister } from './registerServiceWorker';
```

and then call `unregister()` instead. After the user visits a page that has `unregister()`, the service worker will be uninstalled. Note that depending on how `/service-worker.js` is served, it may take up to 24 hours for the cache to be invalidated.

Offline-First Considerations

1. Service workers require HTTPS, although to facilitate local testing, that policy does not apply to `localhost`. If your production web server does not support HTTPS, then the service worker registration will fail, but the rest of your web app will remain functional.

2. Service workers are not currently supported in all web browsers. Service worker registration won't be attempted on browsers that lack support.

3. The service worker is only enabled in the production environment, e.g. the output of `npm run build`. It's recommended that you do not enable an offline-first service worker in a development environment, as it can lead to frustration when previously cached assets are used and do not include the latest changes you've made locally.

4. If you *need* to test your offline-first service worker locally, build the application (using `npm run build`) and run a simple http server from your build directory. After running the build script, `create-react-app` will give instructions for one way to test your production build locally and the deployment instructions have instructions for using other methods. *Be sure to always use an incognito window to avoid complications with your browser cache.*

5. If possible, configure your production environment to serve the generated `service-worker.js` with HTTP caching disabled. If that's not possible—GitHub Pages, for instance, does not allow you to change the default 10 minute HTTP cache lifetime—then be aware that if you visit your production site, and then revisit again before `service-worker.js` has expired from your HTTP cache, you'll continue to get the previously cached assets from the service worker. If you have an immediate need to view your updated production deployment, performing a shift-refresh will temporarily disable the service worker and retrieve all assets from the network.

6. Users aren't always familiar with offline-first web apps. It can be useful to let the user know when the service worker has finished populating your caches (showing a "This web app works offline!" message) and also let them know when the service worker has fetched the latest updates that will be available the next time they load the page (showing a "New content is available; please refresh." message). Showing this messages is currently left as an exercise to the developer, but as a starting point, you can make use of the logic included in `src/registerServiceWorker.js`, which demonstrates which service worker lifecycle events to listen for to detect each scenario, and which as a default, just logs appropriate messages to the JavaScript console.

7. By default, the generated service worker file will not intercept or cache any cross-origin traffic, like HTTP API requests, images, or embeds loaded from a different domain. If you would like to use a runtime caching strategy for those requests, you can `eject` and then configure the `runtimeCaching` option in the `SWPrecacheWebpackPlugin` section of `webpack.config.prod.js`.

Progressive Web App Metadata

The default configuration includes a web app manifest located at `public/manifest.json`, that you can customize with details specific to your web application.

When a user adds a web app to their homescreen using Chrome or Firefox on Android, the metadata in `manifest.json` determines what icons, names, and branding colors to use when the web app is displayed. The Web App Manifest guide provides more context about what each field means, and how your customizations will affect your users' experience.

Analyzing the Bundle Size

Source map explorer analyzes JavaScript bundles using the source maps. This helps you understand where code bloat is coming from.

To add Source map explorer to a Create React App project, follow these steps:

```
npm install --save source-map-explorer
```

Alternatively you may use `yarn`:

```
yarn add source-map-explorer
```

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

```
   "scripts": {
+    "analyze": "source-map-explorer build/static/js/main.*",
     "start": "react-scripts start",
     "build": "react-scripts build",
     "test": "react-scripts test --env=jsdom",
```

Then to analyze the bundle run the production build then run the analyze script.

```
npm run build
npm run analyze
```

Deployment

`npm run build` creates a `build` directory with a production build of your app. Set up your favorite 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, the easiest way to handle this would be to install 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'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 and Express:

```
const express = require('express');
const path = require('path');
const app = express();

app.use(express.static(path.join(__dirname, '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 under the hood (for example, 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(path.join(__dirname, 'build')));

-app.get('/', function (req, res) {
+app.get('/*', function (req, res) {
   res.sendFile(path.join(__dirname, 'build', 'index.html'));
 });
```

If you’re using Apache HTTP Server, 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`.

If you’re using Apache Tomcat, you need to follow this Stack Overflow answer.

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

On a production build, and in a browser that supports service workers, the service worker will automatically handle all navigation requests, like for `/todos/42`, by serving the cached copy of your `index.html`. This service worker navigation routing can be configured or disabled by `ejecting` and then modifying the `navigateFallback` and `navigateFallbackWhitelist` options of the `SWPreachePlugin` configuration.

When users install your app to the homescreen of their device the default configuration will make a shortcut to `/index.html`. This may not work for client-side routers which expect the app to be served from `/`. Edit the web app manifest at `public/manifest.json` and change `start_url` to match the required URL scheme, for example:

```
"start_url": ".",
```

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.

Note: If you are using `react-router@^4`, you can root `<Link>`s using the `basename` prop on any `<Router>`.
More information here.

For example:

```
<BrowserRouter basename="/calendar"/>
<Link to="/today"/> // renders <a href="/calendar/today">
```

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 blog post on how to deploy your React app to Microsoft Azure.

See this blog post or this repo for a way to use automatic deployment to Azure App Service.

Firebase

Install the Firebase CLI if you haven't already by running `npm install -g firebase-tools`. Sign up for a Firebase account 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!
```

IMPORTANT: you need to set proper HTTP caching headers for `service-worker.js` file in `firebase.json` file or you will not be able to see changes after first deployment (issue #2440). It should be added inside `"hosting"` key like next:

```
{
  "hosting": {
    ...
    "headers": [
      {"source": "/service-worker.js", "headers": [{"key": "Cache-Control",
"value": "no-cache"}]}
    ]
    ...
```

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.

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 for your project:

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

or for a GitHub user page:

```
  "homepage": "https://myusername.github.io",
```

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 gh-pages
```

Alternatively you may use `yarn`:

```
yarn add gh-pages
```

Add the following scripts in your `package.json`:

```
  "scripts": {
+   "predeploy": "npm run build",
+   "deploy": "gh-pages -d build",
    "start": "react-scripts start",
    "build": "react-scripts build",
```

The `predeploy` script will run automatically before `deploy` is run.

If you are deploying to a GitHub user page instead of a project page you'll need to make two additional modifications:

1. First, change your repository's source branch to be any branch other than **master**.
2. Additionally, tweak your `package.json` scripts to push deployments to **master**:

```
  "scripts": {
    "predeploy": "npm run build",
-   "deploy": "gh-pages -d build",
+   "deploy": "gh-pages -b master -d build",
```

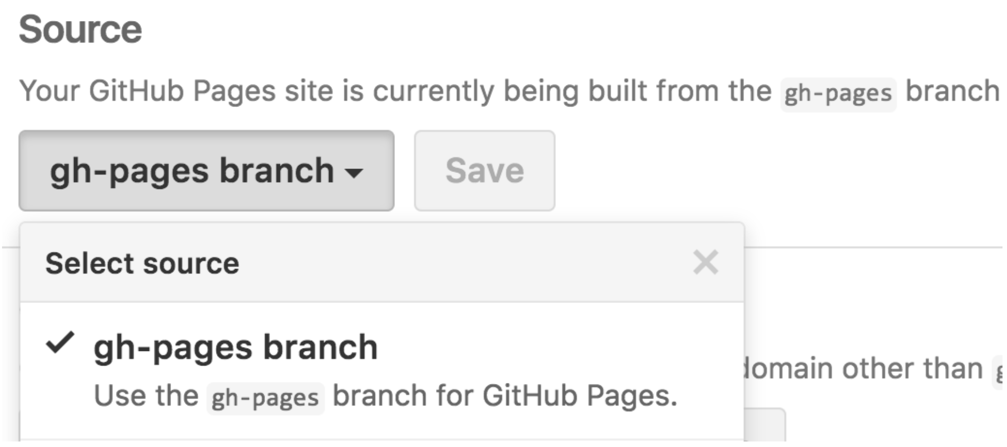
Step 3: Deploy the site by running `npm run deploy`

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:


Source

Your GitHub Pages site is currently being built from the `gh-pages` branch

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 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.

Troubleshooting

"/dev/tty: No such a device or address"

If, when deploying, you get `/dev/tty: No such a device or address` or a similar error, try the follwing:

1. Create a new Personal Access Token
2. `git remote set-url origin https://<user>:<token>@github.com/<user>/<repo>` .
3. Try `npm run deploy again`

Heroku

Use the Heroku Buildpack for Create React App.
You can find instructions in 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`.

Netlify

To do a manual deploy to Netlify's CDN:

```
npm install netlify-cli -g
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
2. Pick your Git hosting service and select your repository
3. Set `yarn build` as the build command and `build` as the publish directory
4. Click `Deploy 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 offers a zero-configuration single-command deployment. You can use `now` to deploy your app for free.

1. Install the `now` command-line tool either via the recommended desktop tool or via node with `npm install -g now`.

2. Build your app by running `npm run build`.

3. Move into the build directory by running `cd build`.
4. Run `now --name your-project-name` from within the build directory. You will see a **now.sh** URL in your output like this:

   ```
   > Ready! https://your-project-name-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.

S3 and CloudFront

See this blog post on how to deploy your React app to Amazon Web Services S3 and 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.

When asked about the project path, make sure to specify the `build` folder, for example:

```
       project path: /path/to/project/build
```

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.

Advanced Configuration

You can adjust various development and production settings by setting environment variables in your shell or with .env.

Variable	Development	Production	Usage
BROWSER	✅	❌	By default, Create React App will open the default system browser, favoring Chrome on macOS. Specify a browser to override this behavior, or set it to `none` to disable it completely. If you need to customize the way the browser is launched, you can specify a node script instead. Any arguments passed to `npm start` will also be passed to this script, and the url where your app is served will be the last argument. Your script's file name must have the `.js` extension.

Variable	Development	Production	Usage
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`). 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.
REACT_EDITOR	✅	❌	When an app crashes in development, you will see an error overlay with clickable stack trace. When you click on it, Create React App will try to determine the editor you are using based on currently running processes, and open the relevant source file. You can send a pull request to detect your editor of choice. Setting this environment variable overrides the automatic detection. If you do it, make sure your systems PATH environment variable points to your editor's bin folder. You can also set it to `none` to disable it completely.
CHOKIDAR_USEPOLLING	✅	❌	When set to `true`, the watcher runs in polling mode, as necessary inside a VM. Use this option if `npm start` isn't detecting changes.
GENERATE_SOURCEMAP	❌	✅	When set to `false`, source maps are not generated for a production build. This solves OOM issues on some smaller machines.

Variable	Development	Production	Usage
NODE_PATH	✅	✅	Same as `NODE_PATH` in Node.js, but only relative folders are allowed. Can be handy for emulating a monorepo setup by setting `NODE_PATH=src`.

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 `index.js` and you're referencing it by the folder name, you need to restart the watcher 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 "Adjusting Your Text Editor".
- If your project path contains parentheses, try moving the project to a path without them. This is caused by a Webpack watcher bug.
- On Linux and macOS, you might need to tweak system settings 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.

`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 installation as described in facebookincubator/create-react-app#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
- facebook/watchman#358
- ember-cli/ember-cli#6259

It is reported that installing Watchman 4.7.0 or newer fixes the issue. If you use Homebrew, you can run these commands to update it:

```
watchman shutdown-server
brew update
brew reinstall watchman
```

You can find other installation methods 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` exits too early

It is reported that `npm run build` can fail on machines with limited memory and no swap space, which is common in cloud environments. Even with small projects this command can increase RAM usage in your system by hundreds of megabytes, so if you have less than 1 GB of available memory your build is likely to fail with the following message:

> The build failed because the process exited too early. This probably means the system ran out of memory or someone called `kill -9` on the process.

If you are completely sure that you didn't terminate the process, 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.

Moment.js locales are missing

If you use a Moment.js, you might notice that only the English locale is available by default. This is because the locale files are large, and you probably only need a subset of all the locales provided by Moment.js.

To add a specific Moment.js locale to your bundle, you need to import it explicitly.
For example:

```js
import moment from 'moment';
import 'moment/locale/fr';
```

If import multiple locales this way, you can later switch between them by calling `moment.locale()` with the locale name:

```js
import moment from 'moment';
import 'moment/locale/fr';
import 'moment/locale/es';

// ...

moment.locale('fr');
```

This will only work for locales that have been explicitly imported before.

`npm run build` fails to minify

Some third-party packages don't compile their code to ES5 before publishing to npm. This often causes problems in the ecosystem because neither browsers (except for most modern versions) nor some tools currently support all ES6 features. We recommend to publish code on npm as ES5 at least for a few more years.

To resolve this:

1. Open an issue on the dependency's issue tracker and ask that the package be published pre-compiled.

- Note: Create React App can consume both CommonJS and ES modules. For Node.js compatibility, it is recommended that the main entry point is CommonJS. However, they can optionally provide an ES module entry point with the `module` field in `package.json`. Note that **even if a library provides an ES Modules version, it should still precompile other ES6 features to ES5 if it intends to support older browsers**.

2. Fork the package and publish a corrected version yourself.

3. If the dependency is small enough, copy it to your `src/` folder and treat it as application code.

In the future, we might start automatically compiling incompatible third-party modules, but it is not currently supported. This approach would also slow down the production builds.

Alternatives to Ejecting

Ejecting lets you customize anything, but from that point on you have to maintain the configuration and scripts yourself. This can be daunting if you have many similar projects. In such cases instead of ejecting we recommend to *fork* `react-scripts` and any other packages you need. This article dives into how to do it in depth. You can find more discussion in this issue.

Something Missing?

If you have ideas for more “How To” recipes that should be on this page, let us know or contribute some!