How to code in React.js

Joe Morgan

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

ISBN 978-1-7358317-4-9

How To Code in React.js

Joe Morgan

DigitalOcean, New York City, New York, USA

2021-07

How To Code in React.js

1. About DigitalOcean
2. Introduction
3. How To Set Up a React Project with Create React App
4. How To Create React Elements with JSX
5. How To Create Custom Components in React
6. How To Customize React Components with Props
7. How To Create Wrapper Components in React with Props
8. How To Style React Components
9. How To Manage State on React Class Components
10. How To Manage State with Hooks on React Components
11. How To Share State Across React Components with Context
12. How To Debug React Components Using React Developer Tools
13. How To Handle DOM and Window Events with React
14. How To Build Forms in React
15. How To Handle Async Data Loading, Lazy Loading, and Code Splitting with React
16. How To Call Web APIs with the useEffect Hook in React
17. How To Manage State in React with Redux
18. How To Handle Routing in React Apps with React Router
19. How To Add Login Authentication to React Applications
20. How To Avoid Performance Pitfalls in React with memo, useMemo, and useCallback
21. How To Deploy a React Application with Nginx on Ubuntu 20.04

22. How To Deploy a React Application to DigitalOcean App Platform

About DigitalOcean

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale. It provides highly available, secure and scalable compute, storage and networking solutions that help developers build great software faster. Founded in 2012 with offices in New York and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available. For more information, please visit https://www.digitalocean.com or follow @digitalocean on Twitter.

Introduction

About this Book

React seems to be everywhere. Companies and projects large and small are using it to build their applications. The popularity comes from the fact that React builds on core web development skills. That's not to say you will learn it in a day or that every feature is easy to understand on the first try. Instead, React excels precisely because it minimizes the amount of React-specific knowledge you need. You don't need to learn about templates or controllers or complex patterns. Instead, most of the code you write will be JavaScript combined with standard HTML. It can get complicated from there. The HTML, for example, is really a markup language called JSX that is parsed by React before going into the DOM. But as you take each step in your learning you will be building on solid foundations of web development. That means you gain a double benefit as your learn React. Not only will you be building world class applications, you will be strengthening your own knowledge of JavaScript and web standards. You will develop transferable skills that you can use in any future web-based application whether it's built with React or not.

This book is an introduction to React that works from the foundations upward. Each chapter takes you a little deeper into the React ecosystem, building on your previous knowledge. Along the way, you'll maintain internal state, pass information between parts of an application, and explore different options for styling your application. Whether you are completely new to React or if you've worked with it before, this series will be

accessible to you. Every chapter is self contained, so you can jump between chapters or skip whole sections. The book is designed for you to take a concept and explore it by building a small project that mirrors what you will see in everyday development.

Learning Goals and Outcomes

By the end of the book, you'll have a strong understanding of the different parts of a React application and you'll be able to combine the parts together to build individual components and whole applications. You'll be able to build small applications that use external data and respond to user actions. You'll also learn how to debug and optimize your application to make the best user experience.

How to Use This Book

You can read the book in any order, but if you are new to React, start with the first chapter that shows you how to create a new project using a tool called Create React App. Every subsequent chapter will start with a new project, so it will be useful to learn how to bootstrap a new application. After that, continue straight through or skip to the chapters that interest you. If something is unfamiliar, back up and you'll find a whole tutorial dedicated to the concept.

How To Set Up a React Project with Create React App

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

React is a popular JavaScript framework for creating front-end applications. Originally created by Facebook, it has gained popularity by allowing developers to create fast applications using an intuitive programming paradigm that ties JavaScript with an HTML-like syntax known as JSX.

Starting a new React project used to be a complicated multi-step process that involved setting up a build system, a code transpiler to convert modern syntax to code that is readable by all browsers, and a base directory structure. But now, Create React App includes all the JavaScript packages you need to run a React project, including code transpiling, basic linting, testing, and build systems. It also includes a server with hot reloading that will refresh your page as you make code changes. Finally, it will create a structure for your directories and components so you can jump in and start coding in just a few minutes.

In other words, you don’t have to worry about configuring a build system like Webpack. You don’t need to set up Babel to transpile you code to be cross-browser usable. You don’t have to worry about most of the complicated systems of modern front-end development. You can start writing React code with minimal preparation.

By the end of this tutorial, you’ll have a running React application that you can use as a foundation for any future applications. You’ll make your first changes to React code, update styles, and run a build to create a fully minified version of your application. You’ll also use a server with hot reloading to give you instant feedback and will explore the parts of a React project in depth. Finally, you will begin writing custom components and creating a structure that can grow and adapt with your project.

Prerequisites

To follow this tutorial, you'll need the following:

- Node.js version 10.16.0 installed on your computer. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- It will also help to have a basic understanding of JavaScript, which you can find in the How To Code in JavaScript series, along with a basic knowledge of HTML and CSS.

Step 1 — Creating a New Project with Create React App

In this step, you'll create a new application using the npm package manager to run a remote script. The script will copy the necessary files into a new directory and install all dependencies.

When you installed Node, you also installed a package managing application called `npm`. `npm` will install JavaScript packages in your project and also keep track of details about the project. If you'd like to learn more about `npm`, take a look at our How To Use Node.js Modules with npm and package.json tutorial.

`npm` also includes a tool called `npx`, which will run executable packages. What that means is you will run the Create React App code without first downloading the project.

The executable package will run the installation of `create-react-app` into the directory that you specify. It will start by making a new project in a directory, which in this tutorial will be called `digital-ocean-tutorial`. Again, this directory does not need to exist beforehand; the executable package will create it for you. The script will also run `npm install` inside the project directory, which will download any additional dependencies.

To install the base project, run the following command:

```
npx create-react-app digital-ocean-tutorial
```

This command will kick off a build process that will download the base code along with a number of dependencies.

When the script finishes you will see a success message that says:

Output

```
...
Success! Created digital-ocean-tutorial at your_file_path/digital-ocean-tut
orial
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can’t go back!

We suggest that you begin by typing:

  cd digital-ocean-tutorial
  npm start

Happy hacking!
```

`your_file_path` will be your current path. If you are a macOS user, it will be something like `/Users/your_username`; if you are on an Ubuntu server, it will say something like `/home/your_username`.

You will also see a list of `npm` commands that will allow you to run, build, start, and test your application. You'll explore these more in the next section.

> **Note:** There is another package manager for JavaScript called `yarn`. It's supported by Facebook and does many of the same things as `npm`. Originally, `yarn` provided new functionality such as lock files, but now these are implemented in `npm` as well. `yarn` also includes a few other features such as offline caching. Further differences can be found on the `yarn` documentation.
>
> If you have previously installed `yarn` on your system, you will see a list of `yarn` commands such as `yarn start` that work the same as `npm` commands. You can run `npm` commands even if you have `yarn` installed. If you prefer `yarn`, just replace `npm` with `yarn` in any future commands. The results will be the same.

Now your project is set up in a new directory. Change into the new directory:

```
cd digital-ocean-tutorial
```

You are now inside the root of your project. At this point, you've created a new project and added all of the dependencies. But you haven't take any actions to run the project. In the next section, you'll run custom scripts to build and test the project.

Step 2 — Using `react-scripts`

In this step, you will learn about the different `react-scripts` that are installed with the repo. You will first run the `test` script to execute the test code. Then you will run the `build` script to create a minified version. Finally, you'll look at how the `eject` script can give you complete control over customization.

Now that you are inside the project directory, take a look around. You can either open the whole directory in your text editor, or if you are on the terminal you can list the files out

with the following command:

```
ls -a
```

The `-a` flag ensures that the output also includes hidden files.

Either way, you will see a structure like this:

```
Output
node_modules/
public/
src/
.gitignore
README.md
package-lock.json
package.json
```

Let’s explain these one by one:

- `node_modules/` contains all of the external JavaScript libraries used by the application. You will rarely need to open it.
- The `public/` directory contains some base HTML, JSON, and image files. These are the roots of your project. You’ll have an opportunity to explore them more in Step 4.
- The `src/` directory contains the React JavaScript code for your project. Most of the work you do will be in that directory. You’ll explore this directory in detail in Step 5.
- The `.gitignore` file contains some default directories and files that git—your source control—will ignore, such as the `node_modules` directory. The ignored items tend to be larger directories or log files that you would not need in source control. It also will include some directories that you’ll create with some of the React scripts.

- `README.md` is a markdown file that contains a lot of useful information about Create React App, such as a summary of commands and links to advanced configuration. For now, it's best to leave the `README.md` file as you see it. As your project progresses, you will replace the default information with more detailed information about your project.

The last two files are used by your package manager. When you ran the initial `npx` command, you created the base project, but you also installed the additional dependencies. When you installed the dependencies, you created a `package-lock.json` file. This file is used by `npm` to ensure that the packages match exact versions. This way if someone else installs your project, you can ensure they have identical dependencies. Since this file is created automatically, you will rarely edit this file directly.

The last file is a `package.json`. This contains metadata about your project, such as the title, version number, and dependencies. It also contains scripts that you can use to run your project.

Open the `package.json` file in your favorite text editor:

```
nano package.json
```

When you open the file, you will see a JSON object containing all the metadata. If you look at the `scripts` object, you'll find four different scripts: `start`, `build`, `test`, and `eject`.

These scripts are listed in order of importance. The first script starts the local development environment; you'll get to that in the next step. The second script will build your project. You'll explore this in detail in Step 4, but it's worth running now to see what happens.

The `build` Script

To run any npm script, you just need to type `npm run script_name` in your terminal. There are a few special scripts where you can omit the `run` part of the command, but it's always

fine to run the full command. To run the `build` script, type the following in your terminal:

```
npm run build
```

You will immediately see the following message:

Output

```
> digital-ocean-tutorial@0.1.0 build your_file_path/digital-ocean-tutorial
> react-scripts build

Creating an optimized production build...
...
```

This tells you that Create React App is compiling your code into a usable bundle.

When it’s finished, you’ll see the following output:

Output

```
...
Compiled successfully.

File sizes after gzip:

  39.85 KB  build/static/js/9999.chunk.js
  780 B      build/static/js/runtime-main.99999.js
  616 B     build/static/js/main.9999.chunk.js
  556 B     build/static/css/main.9999.chunk.css

The project was built assuming it is hosted at the server root.
You can control this with the homepage field in your package.json.
For example, add this to build it for GitHub Pages:

  "homepage" : "http://myname.github.io/myapp",

The build folder is ready to be deployed.
You may serve it with a static server:

  serve -s build

Find out more about deployment here:

  bit.ly/CRA-deploy
```

List out the project contents and you will see some new directories:

```
ls -a
```

Output

```
build/
node_modules/
public/
src/
.gitignore
README.md
package-lock.json
package.json
```

You now have a `build` directory. If you opened the `.gitignore` file, you may have noticed that the `build` directory is ignored by git. That's because the `build` directory is just a minified and optimized version of the other files. There's no need to use version control since you can always run the `build` command. You'll explore the output more later; for now, it's time to move on to the `test` script.

The `test` Script

The `test` script is one of those special scripts that doesn't require the `run` keyword, but works even if you include it. This script will start up a test runner called Jest. The test runner looks through your project for any files with a `.spec.js` or `.test.js` extension, then runs those files.

To run the `test` script, type the following command:

```
npm test
```

After running this script your terminal will have the output of the test suite and the terminal prompt will disappear. It will look something like this:

Output

```
 PASS  src/App.test.js
  ✓ renders learn react link (67ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        4.204s
Ran all test suites.

Watch Usage
 › Press f to run only failed tests.
 › Press o to only run tests related to changed files.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press Enter to trigger a test run.
```

There are a few things to notice here. First, as noted before, it automatically detects any files with test extensions including `.test.js` and `.spec.js`. In this case, there is only one test suite—that is, only one file with a `.test.js` extension—and that test suite contains only one test. Jest can detect tests in your code hierarchy, so you can nest tests in a directory and Jest will find them.

Second, Jest doesn't run your test suite once and then exit. Rather, it continues running in the terminal. If you make any changes in the source code, it will rerun the tests again.

You can also limit which tests you run by using one of the keyboard options. If you type `o`, for example, you will only run the tests on files that have changed. This can save you lots of time as your test suites grow.

Finally, you can exit the test runner by typing `q`. Do this now to regain your command prompt.

The `eject` Script

The final script is `npm eject`. This script copies your dependencies and configuration files into your project, giving you full control over your code but ejecting the project from the Create React App integrated toolchain. You will not run this now because, once you run this script, you can't undo this action and you will lose any future Create React App updates.

The value in Create React App is that you don't have to worry about a significant amount of configuration. Building modern JavaScript applications requires a lot of tooling from build systems, such as Webpack, to compilation tools, such as Babel. Create React App handles all the configuration for you, so ejecting means dealing with this complexity yourself.

The downside of Create React App is that you won't be able to fully customize the project. For most projects that's not a problem, but if you ever want to take control of all aspects of the build process, you'll need to eject the code. However, as mentioned before, once you eject the code you will not be able to update to new versions of Create React App, and you'll have to manually add any enhancements on your own.

At this point, you've executed scripts to build and test your code. In the next step, you'll start the project on a live server.

Step 3 — Starting the Server

In this step, you will initialize a local server and run the project in your browser.

You start your project with another `npm` script. Like `npm test`, this script does not need the `run` command. When you run the script you will start a local server, execute the project code, start a watcher that listens for code changes, and open the project in a web browser.

Start the project by typing the following command in the root of your project. For this tutorial, the root of your project is the `digital-ocean-tutorial` directory. Be sure to open this in a separate terminal or tab, because this script will continue running as long as you allow it:

```
npm start
```

You'll see some placeholder text for a brief moment before the server starts up, giving this output:

Output

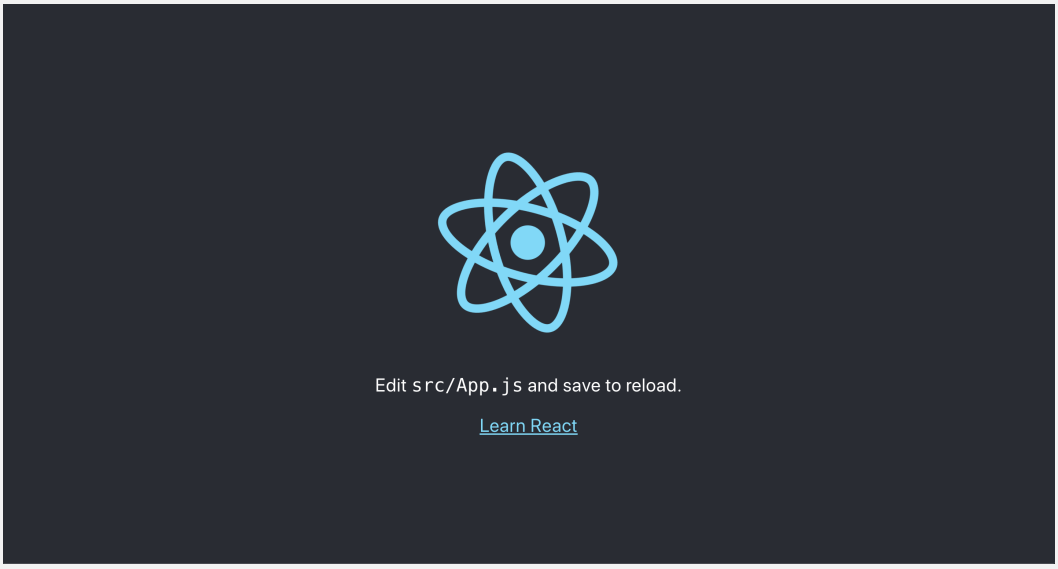
```
Compiled successfully!

You can now view digital-ocean-tutorial in the browser.

  http://localhost:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

If you are running the script locally, it will open the project in your browser window and shift the focus from the terminal to the browser.

If that doesn't happen, you can visit `http://localhost:3000/` to see the site in action. If you already happen to have another server running on port `3000`, that's fine. Create React App will detect the next available port and run the server with that. In other words, if you already have one project running on port `3000`, this new project will start on port `3001`.

If you are running this from a remote server you can still see your site without any additional configuration. The address will be `http://your_server_ip:3000`. If you have a firewall configured, you'll need to open up the port on your remote server.

In the browser, you will see the following React template project:



React template project

As long as the script is running, you will have an active local server. To stop the script, either close the terminal window or tab or type `CTRL+C` or `⌘-+c` in the terminal window or tab that is running your script.

At this point, you have started the server and are running your first React code. But before you make any changes to the React JavaScript code, you will see how React renders to the page in the first place.

Step 4 — Modifying the Homepage

In this step, you will modify code in the `public/` directory. The `public` directory contains your base HTML page. This is the page that will serve as the root to your project. You will rarely edit this directory in the future, but it is the base from which the project starts and a crucial part of a React project.

If you cancelled your server, go ahead and restart it with `npm start`, then open `public/` in your favorite text editor in a new terminal window:

```
nano public/
```

Alternatively, you can list the files with the `ls` command:

```
ls public/
```

You will see a list of files such as this:

```
Output
favicon.ico
logo192.png
manifest.json
index.html
logo512.png
robots.txt
```

`favicon.ico`, `logo192.png`, and `logo512.png` are icons that a user would see either in the tab of their browser or on their phone. The browser will select the proper-sized icons. Eventually, you'll want to replace these with icons that are more suited to your project. For now, you can leave them alone.

The `manifest.json` is a structured set of metadata that describes your project. Among other things, it lists which icon will be used for different size options.

The `robots.txt` file is information for web crawlers. It tells crawlers which pages they are or are not allowed to index. You will not need to change either file unless there is a compelling reason to do so. For instance, if you wanted to give some users a URL to special content that you do not want easily accessible, you can add it to `robots.txt` and it will still be publicly available, but not indexed by search engines.

The `index.html` file is the root of your application. This is the file the server reads, and it is the file that your browser will display. Open it up in your text editor and take a look.

If you are working from the command line, you can open it with the following command:

```
nano public/index.html
```

Here’s what you will see:

digital-ocean-tutorial/public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <!--
      manifest.json provides metadata used when your web app is installed on a
      user's mobile device or desktop.
       See https://developers.google.com/web/fundamentals/web-app-manifest/
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the `public` folder during the build.
      Only files inside the `public` folder can be referenced from the HTML.

      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
      work correctly both with client-side routing and a non-root public URL.
      Learn how to configure a non-root public URL by running `npm run build`.
    -->
    <title>React App</title>
  </head>
  <body>
```

```
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.

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

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

The file is pretty short. There are no images or words in the `<body>`. That's because React builds the entire HTML structure itself and injects it with JavaScript. But React needs to know where to inject the code, and that's the role of `index.html`.

In your text editor, change the `<title>` tag from `React App` to `Sandbox`:

digital-ocean-tutorial/public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    ...

<title>Sandbox</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.

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

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

Save and exit your text editor. Check your browser. The title is the name located on the browser tab. It will update automatically. If not, refresh the page and notice the change.

Now go back to your text editor. Every React project starts from a root element. There can be multiple root elements on a page, but there needs to be at least one. This is how React knows where to put the generated HTML code. Find the element `<div id="root">`. This is the `div` that React will use for all future updates. Change the `id` from `root` to `base`:

digital-ocean-tutorial/public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    ...
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="base"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.

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

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

Save the changes.

You will see an error in your browser:

Error message saying “Target container is not a DOM element”

React was looking for an element with an `id` of `root`. Now that it is gone, React can’t start the project.

Change the name back from `base` to `root`:

digital-ocean-tutorial/public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    ...
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.

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

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

Save and exit `index.html`.

At this point, you've started the server and made a small change to the root HTML page. You haven't yet changed any JavaScript code. In the next section, you will update the React JavaScript code.

Step 5 — Modifying the Heading Tag and Styling

In this step, you will make your first change to a React component in the `src/` directory. You'll make a small change to the CSS and the JavaScript code that will automatically update in your browser using the built-in hot reloading.

If you stopped the server, be sure to restart it with `npm start`. Now, take some time to see the parts of the `src/` directory. You can either open the full directory in your favorite text editor, or you can list out the project in a terminal with the following command:

```
ls src/
```

You will see the following files in your terminal or text editor.

```
Output
App.css
App.js
App.test.js
index.css
index.js
logo.svg
serviceWorker.js
setupTests.js
```

Let's go through these files one at a time.

You will not spend much time with the `serviceWorker.js` file at first, but it can be important as you start to make progressive web applications. The service worker can do many things including push notifications and offline caching, but for now it's best to leave it alone.

The next files to look at are `setupTests.js` and `App.test.js`. These are used for test files. In fact, when you ran `npm test` in Step 2, the script ran these files. The `setupTests.js`

file is short; all it includes is a few custom `expect` methods. You'll learn more about these in future tutorials in this series.

Open `App.test.js`:

```
nano src/App.test.js
```

When you open it, you'll see a basic test:

digital-ocean-tutorial/src/App.test.js

```
import React from 'react';
import { render } from '@testing-library/react';
import App from './App';

test('renders learn react link', () => {
  const { getByText } = render(<App />);
  const linkElement = getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

The test is looking for the phrase `learn react` to be in the document. If you go back to the browser running your project, you'll see the phrase on the page. React testing is different from most unit tests. Since components can include visual information, such as markup, along with logic for manipulating data, traditional unit tests do not work as easily. React testing is closer to a form of functional or integration testing.

Next, you'll see some styling files: `App.css`, `index.css`, and `logo.svg`. There are multiple ways of working with styling in React, but the easiest is to write plain CSS since that requires no additional configuration.

There are multiple CSS files because you can import the styles into a component just like they were another JavaScript file. Since you have the power to import CSS directly into a component, you might as well split the CSS to only apply to an individual component. What you are doing is separating concerns. You are not keeping all the CSS separate from the JavaScript. Instead you are keeping all the related CSS, JavaScript, markup, and images grouped together.

Open `App.css` in your text editor. If you are working from the command line, you can open it with the following command:

```
nano src/App.css
```

This is the code you'll see:

digital-ocean-tutorial/src/App.css

```
.App {
  text-align: center;
}

.App-logo {
  height: 40vmin;
  pointer-events: none;
}

@media (prefers-reduced-motion: no-preference) {
  .App-logo {
    animation: App-logo-spin infinite 20s linear;
  }
}

.App-header {
  background-color: #282c34;
  min-height: 100vh;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
  color: white;
}

.App-link {
  color: #61dafb;
}
```

```
@keyframes App-logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}
```

This is a standard CSS file with no special CSS preprocessors. You can add them later if you want, but at first, you only have plain CSS. Create React App tries to be unopinionated while still giving an out-of-the-box environment.

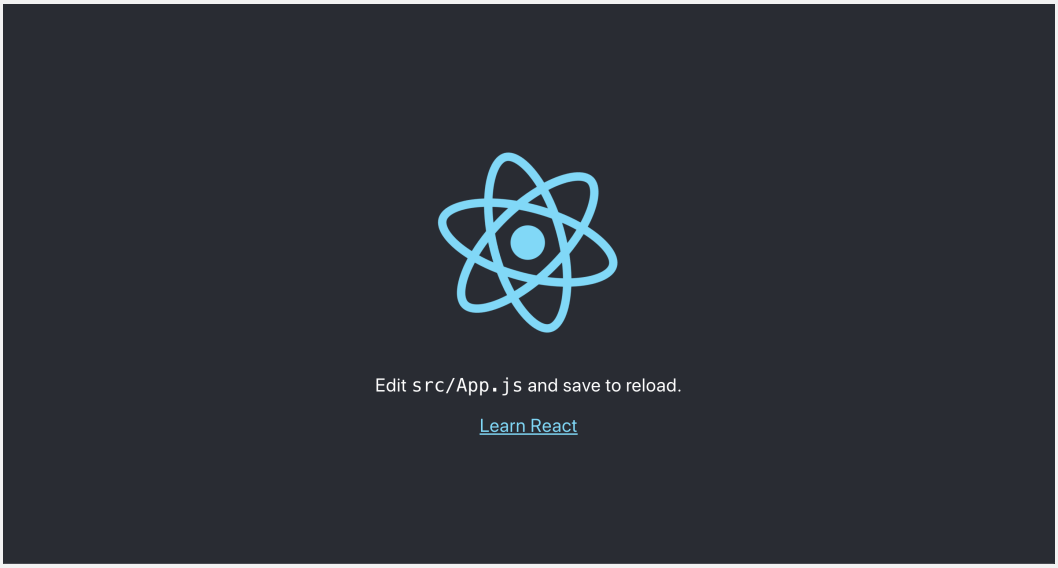
Back to `App.css`, one of the benefits of using Create React App is that it watches all files, so if you make a change, you'll see it in your browser without reloading.

To see this in action make a small change to the `background-color` in `App.css`. Change it from `#282c34` to `blue` then save the file. The final style will look like this:

digital-ocean-tutorial/src/App.css

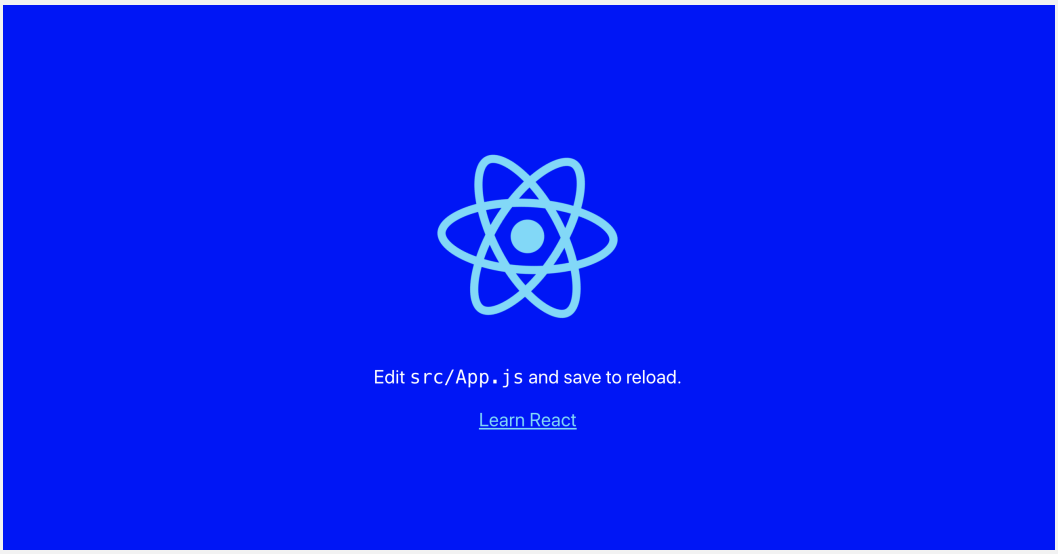
```
.App {
  text-align: center;
}
...
.App-header {
  background-color: blue
  min-height: 100vh;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
  color: white;
}
...

@keyframes App-logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}
```

Check out your browser. Here’s how it looked before:



React app with dark background

Here's how it will look after the change:



React app with blue background

Go ahead and change `background-color` back to `#282c34`.

digital-ocean-tutorial/src/App.css

```
.App {
  text-align: center;

...

.App-header {
  background-color: #282c34
  min-height: 100vh;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
  color: white;
}

...

@keyframes App-logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}
```

Save and exit the file.

You've made a small CSS change. Now it's time to make changes to the React JavaScript code. Start by opening `index.js`.

```
nano src/index.js
```

Here's what you'll see:

digital-ocean-tutorial/src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

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

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

At the top, you are importing `React`, `ReactDOM`, `index.css`, `App`, and `serviceWorker`. By importing `React`, you are actually pulling in code to convert JSX to JavaScript. JSX are the HTML-like elements. For example, notice how when you use `App`, you treat it like an HTML element `<App />`. You'll explore this more in future tutorials in this series.

`ReactDOM` is the code that connects your React code to the base elements, like the `index.html` page you saw in `public/`. Look at the following highlighted line:

digital-ocean-tutorial/src/index.js

```
...
import * as serviceWorker from './serviceWorker';


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

This code instructs React to find an element with an `id` of `root` and inject the React code there. `<App/>` is your root element, and everything will branch from there. This is the beginning point for all future React code.

At the top of the file, you'll see a few imports. You import `index.css`, but don't actually do anything with it. By importing it, you are telling Webpack via the React scripts to include that CSS code in the final compiled bundle. If you don't import it, it won't show up.

Exit from `src/index.js`.

At this point, you still haven't seen anything that you are viewing in your browser. To see this, open up `App.js`:

```
nano src/App.js
```

The code in this file will look like a series of regular HTML elements. Here's what you'll see:

digital-ocean-tutorial/src/App.js

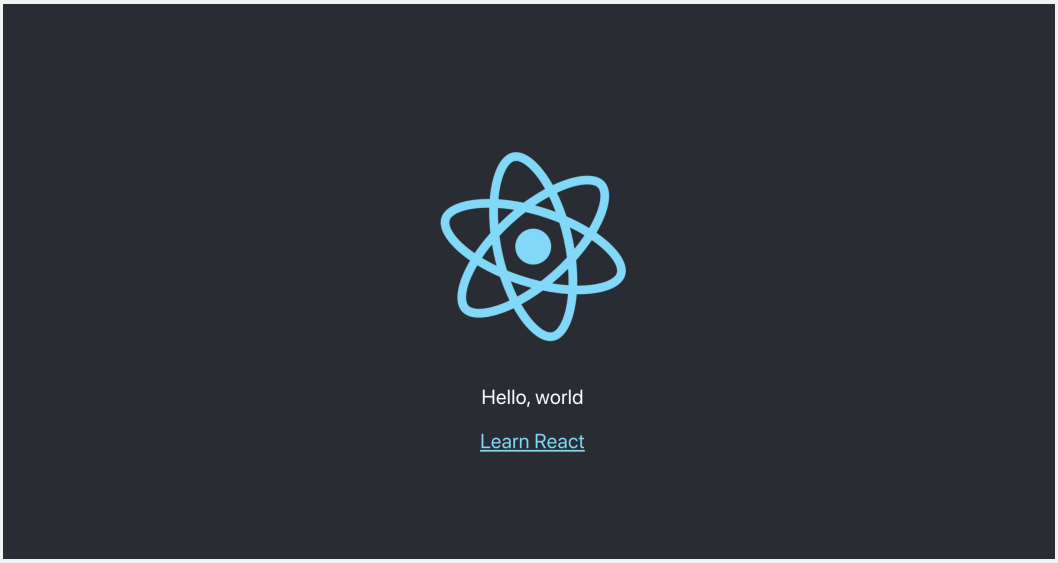
```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Change the contents of the `<p>` tag from `Edit <code>src/App.js</code> and save to reload.` to `Hello, world` and save your changes.

digital-ocean-tutorial/src/App.js

```
...

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
            Hello, world
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
...
```

Head over to your browser and you'll see the change:



React app with “Hello, world” in paragraph tag

You’ve now made your first update to a React component.

Before you go, notice a few more things. In this component, you import the `logo.svg` file and assign it to a variable. Then in the `<img>` element, you add that code as the `src`.

There are a few things going on here. Look at the `img` element:

digital-ocean-tutorial/src/App.js

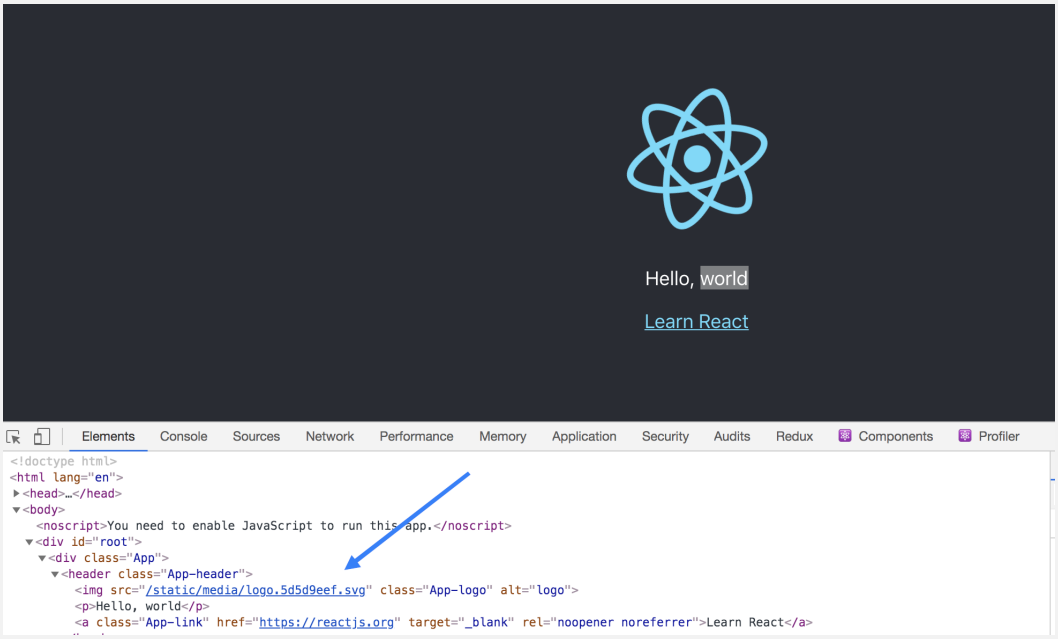
```
...
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
            Hello, world
        </p>
...
```

Notice how you pass the `logo` into curly braces. Anytime you are passing attributes that are not strings or numbers, you need to use the curly braces. React will treat those as JavaScript instead of strings. In this case, you are not actually importing the image; instead you are referencing the image. When Webpack builds the project it will handle the image and set the source to the appropriate place.

Exit the text editor.

If you look at the DOM elements in your browser, you'll see it adds a path. If you are using Chrome, you can inspect the element by right-clicking the element and selecting **Inspect**.

Here's how it would look in the browser:



Inspecting element with chrome dev tools

The DOM has this line:

```
<img src="/static/media/logo.5d5d9eef.svg" class="App-logo" alt="logo">
```

Your code will be slightly different since the logo will have a different name. Webpack wants to make sure the image path is unique. So even if you import images with the same name, they will be saved with different paths.

At this point, you've made a small change to the React JavaScript code. In the next step, you'll use the `build` command to minify the code into a small file that can be deployed to a server.

Step 6 — Building the Project

In this step, you will build the code into a bundle that can be deployed to external servers.

Head back to your terminal and build the project. You ran this command before, but as a reminder, this command will execute the `build` script. It will create a new directory with

the combined and minified files. To execute the build, run the following command from the root of your project:

```
npm run build
```

There will be a delay as the code compiles and when it’s finished, you’ll have a new directory called `build/`.

Open up `build/index.html` in a text editor.

```
nano build/index.html
```

You will see something like this:

digital-ocean-tutorial/build/index.html

```
<!doctype html><html lang="en"><head><meta charset="utf-8"/><link rel="ico
n" href="/favicon.ico"/><meta name="viewport" content="width=device-width,i
nitial-scale=1"/><meta name="theme-color" content="#000000"/><meta name="de
scription" content="Web site created using create-react-app"/><link rel="ap
ple-touch-icon" href="/logo192.png"/><link rel="manifest" href="/manifest.j
son"/><title>React App</title><link href="/static/css/main.d1b05096.chunk.c
ss" rel="stylesheet"></head><body><noscript>You need to enable JavaScript t
o run this app.</noscript><div id="root"></div><script>!function(e){functio
n r(r){for(var n,a,p=r[0],l=r[1],c=r[2],i=0,s=[];i<p.length;i++)a=p[i],Obje
ct.prototype.hasOwnProperty.call(o,a)&&o[a]&&s.push(o[a][0]),o[a]=0;for(n i
n l)Object.prototype.hasOwnProperty.call(l,n)&&(e[n]=l[n]);for(f&&f(r);s.le
ngth;)s.shift()();return u.push.apply(u,c||[]),t()}function t(){for(var e,r
=0;r<u.length;r++){for(var t=u[r],n=!0,p=1;p<t.length;p++){var l=t[p];0!==o
[l]&&(n=!1)}n&&(u.splice(r--,1),e=a(a.s=t[0]))}return e}var n={},o={1:0},u=
[];function a(r){if(n[r])return n[r].exports;var t=n[r]={i:r,l:!1,exports:
{}};return e[r].call(t.exports,t,t.exports,a),t.l=!0,t.exports}a.m=e,a.c=n,
a.d=function(e,r,t){a.o(e,r)||Object.defineProperty(e,r,{enumerable:!0,get:
t})},a.r=function(e){"undefined"!=typeof Symbol&&Symbol.toStringTag&&Objec
t.defineProperty(e,Symbol.toStringTag,{value:"Module"}),Object.defineProper
ty(e,"__esModule",{value:!0})},a.t=function(e,r){if(1&r&&(e=a(e)),8&r)retur
n e;if(4&r&&"object"==typeof e&&e&&e.__esModule)return e;var t=Object.creat
e(null);if(a.r(t),Object.defineProperty(t,"default",{enumerable:!0,value:
e}),2&r&&"string"!=typeof e)for(var n in e)a.d(t,n,function(r){return e
[r]}.bind(null,n));return t},a.n=function(e){var r=e&&e.__esModule?function
(){return e.default}:function(){return e};return a.d(r,"a",r),r},a.o=functi
on(e,r){return Object.prototype.hasOwnProperty.call(e,r)},a.p="/";var p=thi
s["webpackJsonpdo-create-react-app"]=this["webpackJsonpdo-create-react-ap
p"]||[],l=p.push.bind(p);p.push=r,p=p.slice();for(var c=0;c<p.length;c++)r
(p[c]);var f=l;t()}([])</script><script src="/static/js/2.c0be6967.chunk.j
```

```
s"></script><script src="/static/js/main.bac2dbd2.chunk.js"></script></body
></html>
```

The build directory takes all of your code and compiles and minifies it into the smallest usable state. It doesn't matter if a human can read it, since this is not a public-facing piece of code. Minifying like this will make the code take up less space while still allowing it to work. Unlike some languages like Python, the whitespace doesn't change how the computer interprets the code.

Conclusion

In this tutorial, you have created your first React application, configuring your project using JavaScript build tools without needing to go into the technical details. That's the value in Create React App: you don't need to know everything to get started. It allows you to ignore the complicated build steps so you can focus exclusively on the React code.

You've learned the commands to start, test, and build a project. You'll use these commands regularly, so take note for future tutorials. Most importantly, you updated your first React component.

If you would like to see React in action, try our How To Display Data from the DigitalOcean API with React tutorial.

How To Create React Elements with JSX

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

In this tutorial, you'll learn how to describe elements with JSX. JSX is an abstraction that allows you to write HTML-like syntax in your JavaScript code and will enable you to build React components that look like standard HTML markup. JSX is the templating language of React elements, and is therefore the foundation for any markup that React will render into your application.

Since JSX enables you to also write JavaScript in your markup, you'll be able to take advantage of JavaScript functions and methods, including array mapping and short-circuit evaluation for conditionals.

As part of the tutorial, you'll capture click events on buttons directly in the markup and catch instances when the syntax does not match exactly to standard HTML, such as with CSS classes. At the end of this tutorial, you'll have a working application that uses a variety of JSX features to display a list of elements that have a built-in click listener. This is a common pattern in React applications that you will use often in the course of learning the framework. You'll also be able to mix standard HTML elements along with JavaScript to see how React gives you the ability to create small, reusable pieces of code.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.19.0 and npm version 6.13.4. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- You will need to be able to create apps with Create React App. You can find instructions for installing an application with Create React App at How To Set Up a React Project with Create React App.

- You will also need a basic knowledge of JavaScript, which you can find in How To Code in JavaScript, along with a basic knowledge of HTML and CSS. A good resource for HTML and CSS is the Mozilla Developer Network.

Step 1 — Adding Markup to a React Element

As mentioned earlier, React has a special markup language called JSX. It is a mix of HTML and JavaScript syntax that looks something like this:

```
<div>
  {inventory.filter(item => item.available).map(item => (
    <Card>
        <div className="title">{item.name}</div>
        <div className="price">{item.price}</div>
    </Card>
```

```
    ))
  }
</div>
```

You will recognize some JavaScript functionality such as `.filter` and `.map`, as well as some standard HTML like `<div>`. But there are other parts that look like both HTML and JavaScript, such as `<Card>` and `className`.

This is JSX, the special markup language that gives React components the feel of HTML with the power of JavaScript.

In this step, you'll learn to add basic HTML-like syntax to an existing React element. To start, you'll add standard HTML elements into a JavaScript function, then see the compiled code in a browser. You'll also group elements so that React can compile them with minimal markup leaving clean HTML output.

To start, make a new project. On your command line run the following script to install a fresh project using `create-react-app`:

```
npx create-react-app jsx-tutorial
```

After the project is finished, change into the directory:

```
cd jsx-tutorial
```

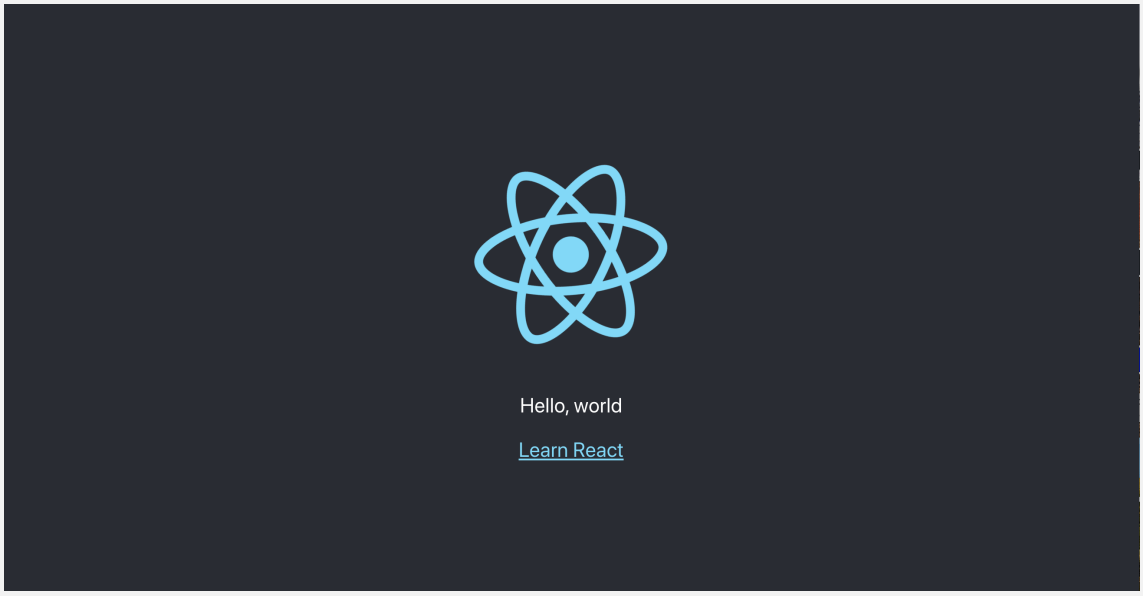
In a new terminal tab or window, start the project using the Create React App `start` script. The browser will autorefresh on changes, so leave this

script running the whole time that you work:

```
npm start
```

You will get a running local server. If the project did not open in a browser window, you can find it at `http://localhost:3000/`. If you are running this from a remote server, the address will be `http://your_IP_address:3000`.

Your browser will load with a React application included as part of Create React App.



React template project

You will be building a completely new set of custom components, so you'll need to start by clearing out some boilerplate code so that you can have an empty project. To start open `App.js` in a text editor. This is the root

component that is injected into the page. All components will start from here.

In a new terminal, move into the project folder and open `src/App.js` with the following command:

```
nano src/App.js
```

You will see a file like this:

jsx-tutorial/src/App.js

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

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
```

```
export default App;
```

Now, delete the line `import logo from './logo.svg` and everything after the return statement in the function. Change it to return `null`. The final code will look like this:

jsx-tutorial/src/App.js

```
import React from 'react';
import './App.css';

function App() {
  return null;
}

export default App;
```

Save and exit the text editor.

Finally, delete the logo. In the terminal window type the following command:

```
rm src/logo.svg
```

You won’t be using this SVG file in your application, and you should remove unused files as you work. It will better organize your code in the long run.

Now that these parts of your project are removed, you can move on to exploring the facets of JSX. This markup language is compiled by React and eventually becomes the HTML you see on a web page. Without going too deeply into the internals, React takes the JSX and creates a model of what your page will look like, then creates the necessary elements and adds them to the page.

What that means is that you can write what looks like HTML and expect that the rendered HTML will be similar. However, there are a few catches.

First, if you look at the tab or window running your server, you’ll see this:

Output

```
...
./src/App.js
  Line 1:8:  'React' is defined but never used  no-unused-vars
...
```

That’s the linter telling you that you aren’t using the imported React code. When you add the line `import React from 'react'` to your code, you are importing JavaScript code that converts the JSX to React code. If there’s no JSX, there’s no need for the import.

Let’s change that by adding a small amount of JSX. Start by replacing `null` with a `Hello, World` example:

jsx-tutorial/src/App.js

```
import React from 'react';
import './App.css';

function App() {
  return <h1>Hello, World</h1>;
}

export default App;
```

Save the file. If you look at the terminal with the server running, the warning message will be gone. If you visit your browser, you will see the message as an `h1` element.

browser screen showing “Hello, World”

Next, below the `<h1>` tag, add a paragraph tag that contains the string `I am writing JSX`. The code will look like this:

jsx-tutorial/src/App.js

```
import React from 'react';
import './App.css';

function App() {
  return(
    <h1>Hello, World</h1>
    <p>I am writing JSX</p>
  )
}

export default App;
```

Since the JSX spans multiple lines, you'll need to wrap the expression in parentheses.

Save the file. When you do you'll see an error in the terminal running your server:

Output

```
./src/App.js
  Line 7:5:  Parsing error: Adjacent JSX elements must be wrap
ped in an enclosing tag. Did you want a JSX fragment <>...</>?

   5 |   return(
   6 |     <h1>Hello, World</h1>
>  7 |     <p>I am writing JSX</p>
     |     ^
   8 |   )
   9 | }
  10 |
```

When you return JSX from a function or statement, you must return a single element. That element may have nested children, but there must be a single top-level element. In this case, you are returning two elements.

The fix is a small code change. Surround the code with an empty tag. An empty tag is an HTML element without any words. It looks like this: `<></>`.

Go back to `./src/App.js` in your editor and add the empty tag:

jsx-tutorial/src/App.js

```
import React from 'react';
import './App.css';

function App() {
  return(
    <>
      <h1>Hello, World</h1>
      <p>I am writing JSX</p>
    </>
  )
}

export default App;
```

The empty tag creates a single element, but when the code is compiled, it is not added to the final markup. This will keep your code clean while still giving React a single element.

Note: You could have also wrapped the code with a `div` instead of empty tags, as long as the code returns one element. In this example, an empty tag has the advantage of not adding extra markup to the parsed output.

Save the code and exit the file. Your browser will refresh and show the updated page with the paragraph element. In addition, when the code is converted the empty tags are stripped out:

Browser showing markup and devtools showing mar kup without empty tags

You've now added some basic JSX to your component and learned how all JSX needs to be nested in a single component. In the next step, you'll add some styling to your component.

Step 2 — Adding Styling to an Element with Attributes

In this step, you'll style the elements in your component to learn how HTML attributes work with JSX. There are many styling options in React. Some of them involve writing CSS in Javascript, others use preprocessors. In this tutorial you'll work with imported CSS and CSS classes.

Now that you have your code, it's time to add some styling. Open `App.css` in your text editor:

```
nano src/App.css
```

Since you are starting with new JSX, the current CSS refers to elements that no longer exist. Since you don't need the CSS, you can delete it.

After deleting the code, you'll have an empty file.

Next, you will add in some styling to center the text. In `src/App.css`, add the following code:

jsx-tutorial/src/App.css

```
.container {
    display: flex;
    flex-direction: column;
    align-items: center;
}
```

In this code block, you created a CSS class selector called `.container` and used that to center the content using `display: flex`.

Save the file and exit. The browser will update, but nothing will change. Before you can see the change, you need to add the CSS class to your React component. Open the component JavaScript code:

```
nano src/App.js
```

The CSS code is already imported with the line `import './App.css'`. That means that webpack will pull in the code to make a final style sheet, but to apply the CSS to your elements, you need to add the classes.

First, in your text editor, change the empty tags, `<>`, to `<div>`.

jsx-tutorial/src/App.js

```
import React from 'react';
import './App.css';

function App() {
  return(
    <div>
      <h1>Hello, World</h1>
      <p>I am writing JSX</p>
    </div>
  )
}

export default App;
```

In this code, you replaced the empty tags—`<>`—with `div` tags. Empty tags are useful for grouping your code without adding any extra tags, but here you need to use a `div` because empty tags do not accept any HTML attributes.

Next, you need to add the class name. This is where JSX will start to diverge from HTML. If you wanted to add a class to a usual HTML element you would do it like this:

```
<div class="container">
```

But since JSX is JavaScript, it has a few limitations. One of the limitations is that JavaScript has reserved keywords. That means you can't use certain words in any JavaScript code. For example, you can't make a variable called `null` because that word is already reserved.

One of the reserved words is `class`. React gets around this reserved word by changing it slightly. Instead of adding the attribute `class`, you will add the attribute `className`. As a rule, if an attribute is not working as expected, try adding the camel case version. Another attribute that is slightly different is the `for` attribute that you'd use for labels. There are a few other cases, but fortunately the list is fairly short.

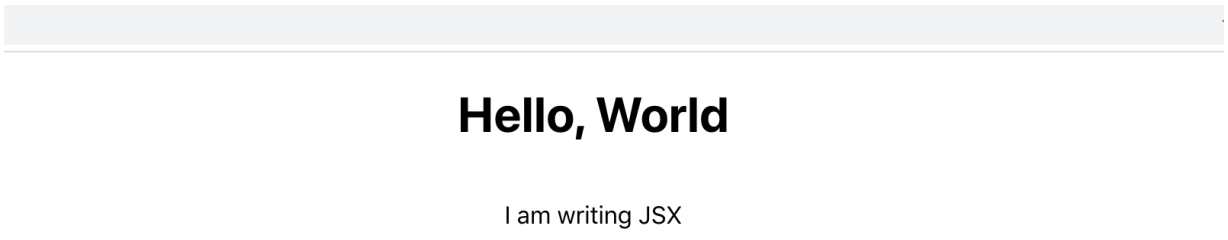
> **Note:** In React, attributes are often called props. Props are pieces of data that you can pass to other custom components. They look the same as attributes except that they do not match any HTML specs. In this tutorial, we'll call them attributes since they are mainly used like standard HTML attributes. This will distinguish them from props that do not behave like HTML attributes, which will be covered later in this series.

Now that you know how the `class` attribute is used in React, you can update your code to include the styles. In your text editor, add `className="container"` to your opening `div` tag:

jsx-tutorial/src/App.js

```
import React from 'react';
import './App.css';

function App() {
  return(
    <div className="container">
      <h1>Hello, World</h1>
      <p>I am writing JSX</p>
    </div>
  )
}

export default App;
```

Save the file. When you do, the page will reload and the content will be centered.



.

The `className` attribute is unique in React. You can add most HTML attributes to JSX without any change. As an example, go back to your text editor and add an `id` of `greeting` to your `<h1>` element. It will look like standard HTML:

jsx-tutorial/src/App.js

```
import React from 'react';
import './App.css';

function App() {
  return(
    <div className="container">
      <h1 id="greeting">Hello, World</h1>
      <p>I am writing JSX</p>
    </div>
  )
}

export default App;
```

Save the page and reload the browser. It will be the same.

So far, JSX looks like standard markup, but the advantage of JSX is that even though it looks like HTML, it has the power of JavaScript. That means you can assign variables and reference them in your attributes. To reference an attribute, wrap it with curly braces—`{}`—instead of quotes.

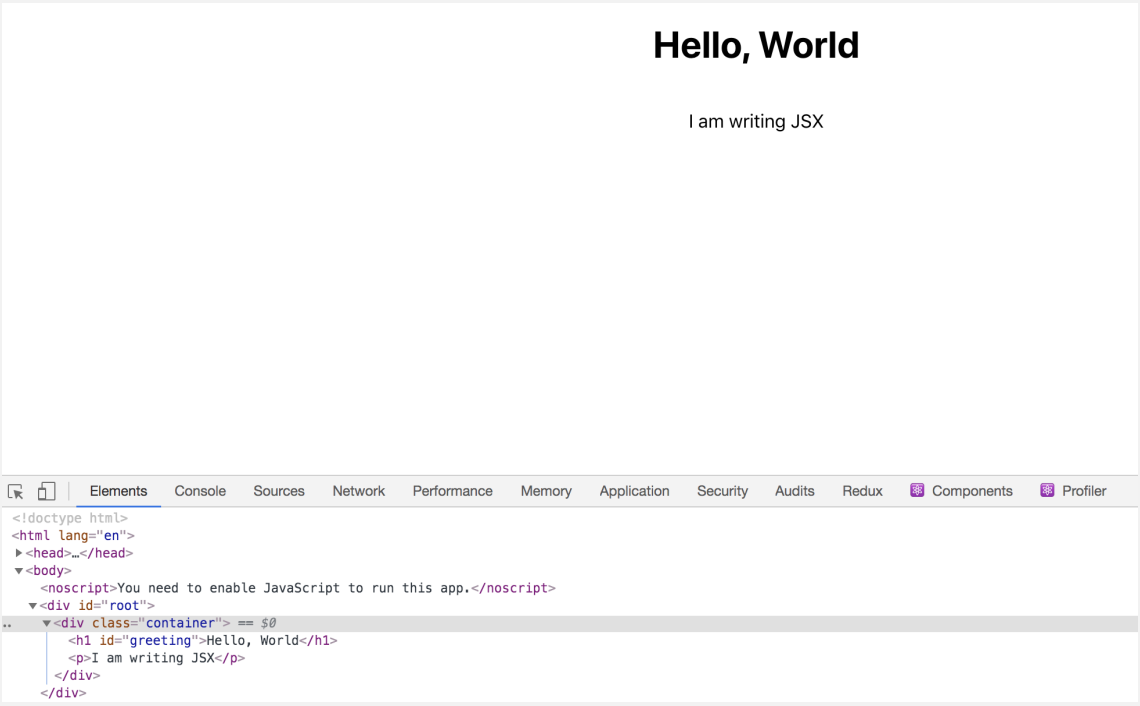
In your text editor, add the following highlighted lines to reference an attribute:

jsx-tutorial/src/App.js

```
import React from 'react';
import './App.css';

function App() {
  const greeting = "greeting";
  return(
    <div className="container">
     <h1 id={greeting}>Hello, World</h1>
      <p>I am writing JSX</p>
    </div>
  )
}

export default App;
```

In this code, you created a variable above the `return` statement called `greeting` with the value of `"greeting"`, then referenced the variable in the `id` attribute of your `<h1>` tag.

Save and exit the file. The page will be the same, but with an `id` tag.



Page with id tag highlighted in the developer tools

So far you've worked with a few elements on their own, but you can also use JSX to add many HTML elements and nest them to create complex pages.

To demonstrate this, you'll make a page with a list of emoji. These emoji will be wrapped with a `<button>` element. When you click on the emoji, you'll get their CLDR Short Name.

To start, you'll need to add a few more elements to the page. Open `src/App.js` in your text editor. Keep it open during this step.

```
nano src/App.js
```

First, add a list of emojis by adding the following highlighted lines:

jsx-tutorial/src/App.js

```
import React from 'react';
import './App.css';

function App() {
  const greeting = "greeting";
  return(
    <div className="container">
      <h1 id={greeting}>Hello, World</h1>
      <p>I am writing JSX</p>
      <ul>
        <li>
            <button>
              <span role="img" aria-label="grinning face" id="g
            </button>
        </li>
        <li>
          <button>
              <span role="img" aria-label="party popper" id="pa
          </button>
        </li>
        <li>
          <button>
              <span role="img" aria-label="woman dancing" id="w
          </button>
        </li>
```

```
      </ul>
    </div>
  )
}

export default App;
```

Here you created a `<ul>` tag to hold a list of emojis. Each emoji is in a separate `<li>` element and is surrounded with a `<button>` element. In the next step you'll add an event to this button.

You also surrounded the emoji with a `<span>` tag that has a few more attributes. Each `span` has the `role` attribute set to the `img` role. This will signal to accessibility software that the element is acting like an image. In addition, each `<span>` also has an `aria-label` and an `id` attribute with the name of the emoji. The `aria-label` will tell visitors with screen readers what is displayed. You will use the `id` when writing events in the next step.

When you write code this way, you are using semantic elements, which will help keep the page accessible and easy to parse for screen readers.

Save and exit the file. Your browser will refresh and you will see this:

browser with emoji as a list

Now add a little styling. Open the CSS code in your text editor:

```
nano src/App.css
```

Add the following highlighted code to remove the default background and border for the buttons while increasing the font size:

jsx-tutorial/src/App.css

```
.container {
    display: flex;
    flex-direction: column;
    align-items: center;
}

button {
    font-size: 2em
    border: 0;
    padding: 0;
    background: none;
    cursor: pointer;
}

ul {    display: flex;
    padding: 0;
}

li {
    margin: 0 20px;
    list-style: none;
    padding: 0;
}
```

In this code, you used `font-size`, `border`, and other parameters to adjust the look of your buttons and change the font. You also removed the list styles and added `display: flex` to the `<ul>` element to make it horizontal.

Save and close the CSS file. Your browser will refresh and you will see this:

`list with default styles removed`

You've now worked with several JSX elements that look like regular HTML. You've added classes, ids, and aria tags, and have worked with data as strings and variables. But React also uses attributes to define how your elements should respond to user events. In the next step, you'll start to make the page interactive by adding events to the button.

Step 3 — Adding Events to Elements

In this step, you'll add events to elements using special attributes and capture a click event on a button element. You'll learn how to capture information from the event to dispatch another action or use other information in the scope of the file.

Now that you have a basic page with information, it's time to add a few events to it. There are many event handlers that you can add to HTML elements. React gives you access to all of these. Since your JavaScript code is coupled with your markup, you can quickly add the events while keeping your code well-organized.

To start, add the `onclick` event handler. This lets you add some JavaScript code directly to your element rather than attaching an event listener:

jsx-tutorial/src/App.js

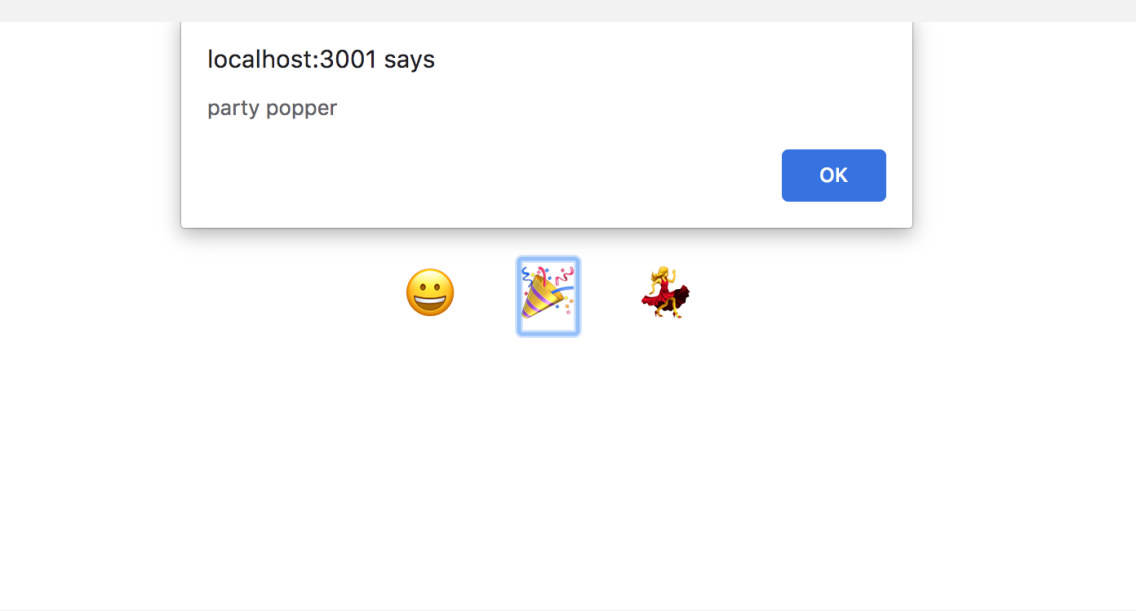
```
import React from 'react';
import './App.css';

function App() {
  const greeting = "greeting";
  return(
    <div className="container">
      <h1 id={greeting}>Hello, World</h1>
      <p>I am writing JSX</p>
      <ul>
        <li>
          <button
            onClick={event => alert(event.target.id)}
          >
            <span role="img" aria-label="grinning face" id="gri
          </button>
        </li>
        <li>
          <button
            onClick={event => alert(event.target.id)}
          >
              <span role="img" aria-label="party popper" id="pa
          </button>
        </li>
        <li>
```

```
            <button
              onClick={event => alert(event.target.id)}
            >
              <span role="img" aria-label="woman dancing" id="w
          </button>
        </li>
      </ul>
    </div>
  )
}

export default App;
```

Since this is JSX, you camelCased `onclick`, which means you added it as `onClick`. This `onClick` attribute uses an anonymous function to retrieve information about the item that was clicked.

You added an anonymous arrow function that will get the event from the clicked button, and the event will have a target that is the `<span>` element. The information you need is in the `id` attribute, which you can access with `event.target.id`. You can trigger the alert with the `alert()` function.

Save the file. In your browser, click on one of the emoji and you will get an alert with the name.



Alert for party popper

You can reduce a duplication by declaring the function once and passing it to each `onClick` action. Since the function does not rely on anything other than inputs and outputs, you can declare it outside the main component function. In other words, the function does not need to access the scope of the component. The advantage to keeping them separate is that your component function is slightly shorter and you could move the function out to a separate file later if you wanted to.

In your text editor, create a function called `displayEmojiName` that takes the event and calls the `alert()` function with an id. Then pass the function to each `onClick` attribute:

jsx-tutorial/src/App.js

```
import React from 'react';
import './App.css';

const displayEmojiName = event => alert(event.target.id);

function App() {
  const greeting = "greeting";
  return(
    <div className="container">
      <h1 id={greeting}>Hello, World</h1>
      <p>I am writing JSX</p>
      <ul>
        <li>
          <button
            onClick={displayEmojiName}
          >
            <span role="img" aria-label="grinning face" id="gri
          </button>
        </li>
        <li>
          <button
            onClick={displayEmojiName}
          >
              <span role="img" aria-label="party popper" id="pa
          </button>
```

```
        </li>
        <li>
            <button
              onClick={displayEmojiName}
            >
              <span role="img" aria-label="woman dancing" id="w
          </button>
        </li>
      </ul>
    </div>
  )
}

export default App;
```

Save the file. In your browser, click on an emoji and you will see the same alert.

In this step, you added events to each element. You also saw how JSX uses slightly different names for element events, and you started writing reusable code by taking the function and reusing it on several elements. In the next step, you will write a reusable function that returns JSX elements rather than writing each element by hand. This will further reduce duplication.

Step 4 — Mapping Over Data to Create Elements

In this step, you'll move beyond using JSX as simple markup. You'll learn to combine it with JavaScript to create dynamic markup that reduces code and improves readability. You'll refactor your code into an array that you will loop over to create HTML elements.

JSX doesn't limit you to an HTML-like syntax. It also gives you the ability to use JavaScript directly in your markup. You tried this a little already by passing functions to attributes. You also used variables to reuse data. Now it's time to create JSX directly from data using standard JavaScript code.

In your text editor, you will need to create an array of the emoji data in the `src/App.js` file. Reopen the file if you have closed it:

```
nano src/App.js
```

Add an array that will contain objects that have the emoji and the emoji name. Note that emojis need to be surrounded by quote marks. Create this array above the `App` function:

jsx-tutorial/src/App.js

```
import React from 'react';
import './App.css';

const displayEmojiName = event => alert(event.target.id);
const emojis = [
  {
    emoji: "😀",
    name: "grinning face"
  },
  {
    emoji: "🎉",
    name: "party popper"
  },
  {
    emoji: "💃",
    name: "woman dancing"
  }
];

function App() {
...
}

export default App;
```

Now that you have the data you can loop over it. To use JavaScript inside of JSX, you need to surround it with curly braces: `{}`. This is the same as when you added functions to attributes.

To create React components, you'll need to convert the data to JSX elements. To do this, you'll map over the data and return a JSX element. There are a few things you'll need to keep in mind as you write the code.

First, a group of items needs to be surrounded by a container `<div>`. Second, every item needs a special property called `key`. The `key` needs to be a unique piece of data that React can use to keep track of the elements so it can know when to update the component. The key will be stripped out of the compiled HTML, since it is for internal purposes only. Whenever you are working with loops you will need to add a simple string as a key.

Here's a simplified example that maps a list of names into a containing `<di v>`:

```
...
const names = [
    "Atul Gawande",
    "Stan Sakai",
    "Barry Lopez"
];

return(
    <div>
        {names.map(name => <div key={name}>{name}</div>)}
    </div>
)
...
```

The resulting HTML would look like this:

```
...
<div>
    <div>Atul Gawande</div>
    <div>Stan Sakai</div>
    <div>Barry Lopez</div>
</div>
...
```

Converting the emoji list will be similar. The `<ul>` will be the container. You'll map over data and return a `<li>` with a key of the emoji short name. You will replace the hard-coded data in the `<button>` and `<span>` tags with information from the loop.

In your text editor, add the following:

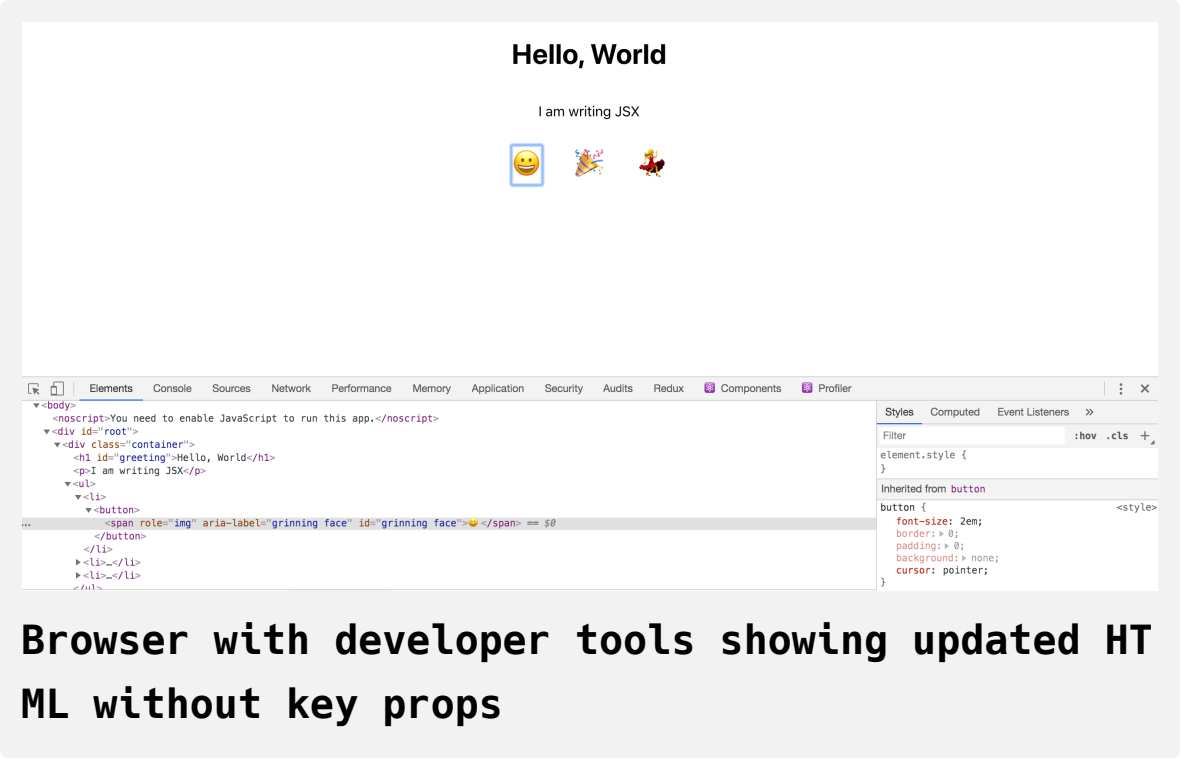
jsx-tutorial/src/App.js

```
import React from 'react';
import './App.css';

const displayEmojiName = event => alert(event.target.id);
const emojis = [
  {
    emoji: '😀',
    name: "test grinning face"
  },
  {
    emoji: '🎉',
    name: "party popper"
  },
  {
    emoji: '💃',
    name: "woman dancing"
  }
];

function App() {
  const greeting = "greeting";
  return(
    <div className="container">
      <h1 id={greeting}>Hello, World</h1>
      <p>I am writing JSX</p>
```

```
      <ul>
        {
          emojis.map(emoji => (
            <li key={emoji.name}>
              <button
                onClick={displayEmojiName}
              >
                <span role="img" aria-label={emoji.name}
                   id={emoji.name}>{emoji.emoji}</span>
              </button>
            </li>
          ))
        }
      </ul>
    </div>
  )
}


export default App;
```

In the code, you mapped over the `emojis` array in the `<ul>` tag and returned a `<li>`. In each `<li>` you used the emoji name as the `key` prop. The button will have the same function as normal. In the `<span>` element, replace the `aria-label` and `id` with the `name`. The content of the `<span>` tag should be the emoji.

Save the file. Your window will refresh and you'll see the data. Notice that the key is not present in the generated HTML.



Browser with developer tools showing updated HTML without key props

Combining JSX with standard JavaScript gives you a lot of tools to create content dynamically, and you can use any standard JavaScript you want. In this step, you replaced hard-coded JSX with an array and a loop to create HTML dynamically. In the next step, you'll conditionally show information using short circuiting.

Step 5 — Conditionally Showing Elements with Short Circuiting

In this step, you'll use short circuiting to conditionally show certain HTML elements. This will let you create components that can hide or show HTML

based on additional information giving your components flexibility to handle multiple situations.

There are times when you will need a component to show information in some cases and not others. For example, you may only want to show an alert message for the user if certain cases are true, or you may want to display some account information for an admin that you wouldn't want a normal user to see.

To do this you will use short circuting. This means that you will use a conditional, and if the first part is truthy, it will return the information in the second part.

Here's an example. If you wanted to show a button only if the user was logged in, you would surround the element with curly braces and add the condition before.

```
{isLoggedIn && <button>Log Out</button>}
```

In this example, you are using the `&&` operator, which returns the last value if everything is truthy. Otherwise, it returns `false`, which will tell React to return no additional markup. If `isLoggedIn` is truthy, React will display the button. If `isLoggedIn` is falsy, it will not show the button.

To try this out, add the following highlighted lines:

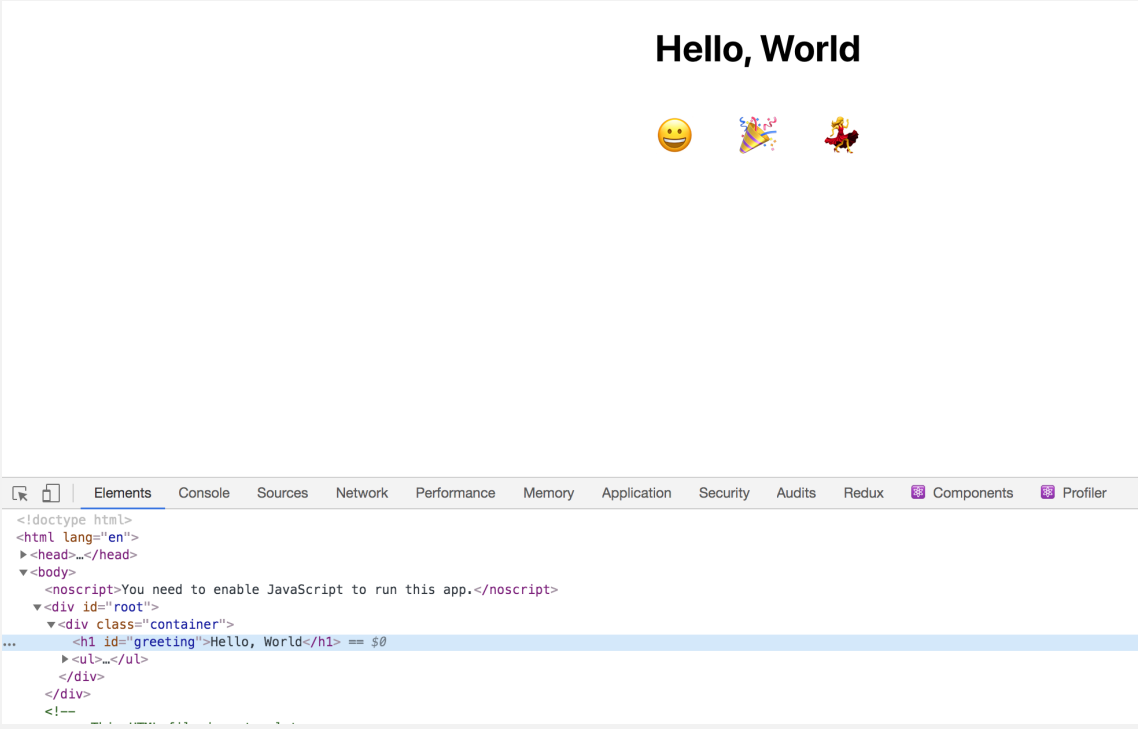
jsx-tutorial/src/App.js

```
import React from 'react';
import './App.css';


...


function App() {
  const greeting = "greeting";
  const displayAction = false;
  return(
    <div className="container">
      <h1 id={greeting}>Hello, World</h1>
      {displayAction && <p>I am writing JSX</p>}
      <ul>
...
      </ul>
    </div>
  )
}


export default App;
```

In your text editor, you created a variable called `displayAction` with a value of `false`. You then surrounded the `<p>` tag with curly braces. At the

start of the curly braces, you added `displayAction &&` to create the conditional.

Save the file and you will see the element disappear in your browser. Crucially, it will also not appear in the generated HTML. This is not the same as hiding an element with CSS. It won't exist at all in the final markup.



Browser with developer tools showing no paragraph element

Right now the value of `displayAction` is hard-coded, but you can also store that value as a state or pass it as a prop from a parent component.

In this step, you learned how to conditionally show elements. This gives you the ability to create components that are customizable based on other information.

Conclusion

At this point, you've created a custom application with JSX. You've learned how to add HTML-like elements to your component, add styling to those elements, pass attributes to create semantic and accessible markup, and add events to the components. You then mixed JavaScript into your JSX to reduce duplicate code and to conditionally show and hide elements.

This is the basis you need to make future components. Using a combination of JavaScript and HTML, you can build dynamic components that are flexible and allow your application to grow and change.

If you'd like to learn more about React, check out our React topic page.

How To Create Custom Components in React

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

In this tutorial, you'll learn to create custom components in React. Components are independent pieces of functionality that you can reuse in your application, and are the building blocks of all React applications. Often, they can be simple JavaScript functions and classes, but you use them as if they were customized HTML elements. Buttons, menus, and any other front-end page content can all be created as components. Components can also contain state information and display markdown.

After learning how to create components in React, you'll be able to split complex applications into small pieces that are easier to build and maintain.

In this tutorial, you'll create a list of emojis that will display their names on click. The emojis will be built using a custom component and will be called from inside another custom component. By the end of this tutorial, you'll have made custom components using both JavaScript classes and JavaScript functions, and you'll understand how to separate existing code into reusable pieces and how to store the components in a readable file structure.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.20.1 and npm version 6.14.4. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- You will need to be able to create apps with Create React App. You can find instructions for installing an application with Create React App at How To Set Up a React Project with Create React App.

- You will be using JSX syntax, which you can learn about in our How To Create Elements with JSX tutorial.

- You will also need a basic knowledge of JavaScript, which you can find in How To Code in JavaScript, along with a basic knowledge of HTML and CSS. A good resource for HTML and CSS is the Mozilla Developer Network.

Step 1 — Setting Up the React Project

In this step, you'll create a base for your project using Create React App. You will also modify the default project to create your base project by mapping over a list of emojis and adding a small amount of styling.

First, create a new project. Open a terminal, then run the following command:

```
npx create-react-app tutorial-03-component
```

Once this is finished, change into the project directory:

```
cd tutorial-03-component
```

Open the `App.js` code in a text editor:

```
nano src/App.js
```

Next, take out the template code created by Create React App, then replace the contents with new React code that displays a list of emojis:

tutorial-03-component/src/App.js

```
import React from 'react';
import './App.css';

const displayEmojiName = event => alert(event.target.id);
const emojis = [
  {
    emoji: '😀',
    name: "test grinning face"
  },
  {
    emoji: '🎉',
    name: "party popper"
  },
  {
    emoji: '💃',
    name: "woman dancing"
  }
];

function App() {
  const greeting = "greeting";
  const displayAction = false;
  return(
    <div className="container">
      <h1 id={greeting}>Hello, World</h1>
```

```
      {displayAction && <p>I am writing JSX</p>}
      <ul>
        {
          emojis.map(emoji => (
            <li key={emoji.name}>
              <button
                onClick={displayEmojiName}
              >
                <span role="img" aria-label={emoji.name}
                    id={emoji.name}>{emoji.emoji}</span>
              </button>
            </li>
          ))
        }
      </ul>
    </div>
  )
}

export default App;
```

This code uses JSX syntax to `map()` over the `emojis` array and list them as `<li>` list items. It also attaches `onClick` events to display emoji data in the browser. To explore the code in more detail, check out How to Create React Elements with JSX, which contains a detailed explanation of the JSX.

Save and close the file. You can now delete the `logo.svg` file, since it was part of the template and you are not referencing it anymore:

```
rm src/logo.svg
```

Now, update the styling. Open `src/App.css`:

```
nano src/App.css
```

Replace the contents with the following CSS to center the elements and adjust the font:

tutorial-03-component/src/App.css

```css
.container {
    display: flex;
    flex-direction: column;
    align-items: center;
}

button {
    font-size: 2em;
    border: 0;
    padding: 0;
    background: none;
    cursor: pointer;
}

ul {
    display: flex;
    padding: 0;
}

li {
    margin: 0 20px;
    list-style: none;
    padding: 0;
}
```

This uses `flex` to center the main `<h1>` and list elements. It also removes default button styles and `<li>` styles so the emojis line up in a row. More details can be found at How to Create React Elements with JSX.

Save and exit the file.

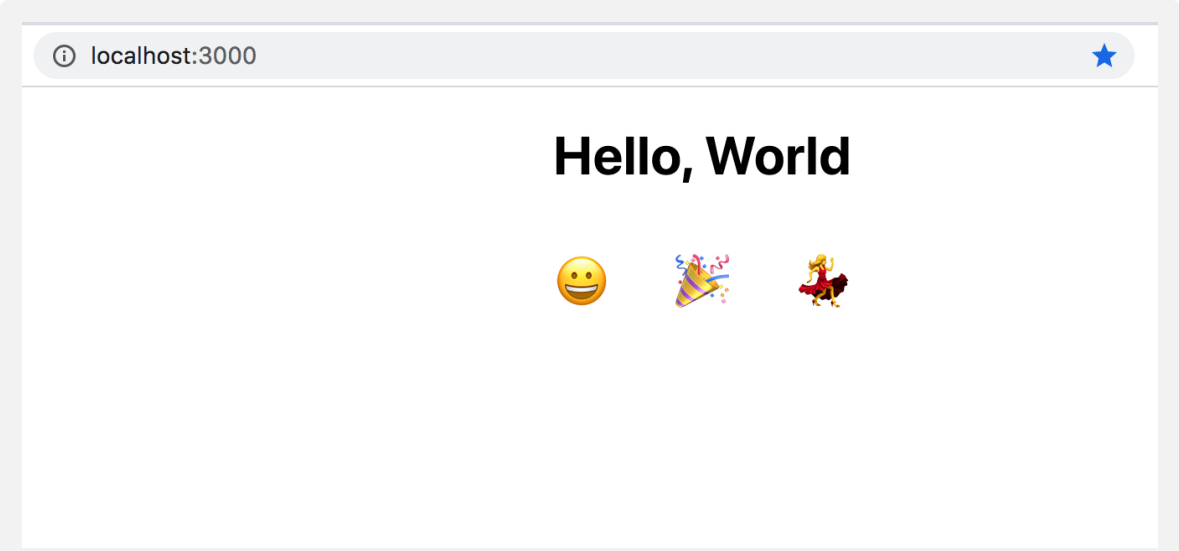
Open another terminal window in the root of your project. Start the project with the following command:

```
npm start
```

After the command runs, you'll see the project running in your web browser at `http://localhost:3000`.

Leave this running the entire time you work on your project. Every time you save the project, the browser will auto-refresh and show the most up-to-date code.

You will see your project page with **Hello, World** and the three emojis that you listed in your `App.js` file:



Browser with emoji

Now that you've set up your code, you can now start putting together components in React.

Step 2 — Creating an Independent Component with React Classes

Now that you have your project running, you can start making your custom component. In this step, you'll create an independent React component by extending the base React `Component` class. You'll create a new class, add methods, and use the render function to show data.

React components are self-contained elements that you can reuse throughout a page. By making small, focused pieces of code, you can move and reuse pieces as your application grows. The key here is that they are self-contained and focused, allowing you to separate out code into logical pieces. In fact, you have already been working with logically separated

components: The `App.js` file is a functional component, one that you will see more of in Step 3.

There are two types of custom component: class-based and functional. The first component you are going to make is a class-based component. You will make a new component called `Instructions` that explains the instructions for the emoji viewer.

> **Note:** Class-based components used to be the most popular way of creating React components. But with the introduction of React Hooks, many developers and libraries are shifting to using functional components.
>
> Though functional components are now the norm, you will often find class components in legacy code. You don't need to use them, but you do need to know how to recognize them. They also give a clear introduction to many future concepts, such as state management. In this tutorial, you'll learn to make both class and functional components.

To start, create a new file. By convention, component files are capitalized:

```
touch src/Instructions.js
```

Then open the file in your text editor:

```
nano src/Instructions.js
```

First, import `React` and the `Component` class and export `Instructions` with the following lines:

tutorial-03-component/src/Instructions.js

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


export default class Instructions extends Component {}
```

Importing `React` will convert the JSX. `Component` is a base class that you'll extend to create your component. To extend that, you created a class that has the name of your component (`Instructions`) and extended the base `Com ponent` with the `export` line. You're also exporting this class as the default with `export default` keywords at the start of the class declaration.

The class name should be capitalized and should match the name of the file. This is important when using debugging tools, which will display the name of the component. If the name matches the file structure, it will be easier to locate the relevant component.

The base `Component` class has several methods you can use in your custom class. The most important method, and the only one you'll use in this tutorial, is the `render()` method. The `render()` method returns the JSX code that you want to display in the browser.

To start, add a little explanation of the app in a `<p>` tag:

tutorial-03-component/src/Instructions.js

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

export class Instructions extends Component {

  render() {
    return(
      <p>Click on an emoji to view the emoji short name.</p>
    )
  }

}
```

Save and close the file. At this point, there's still no change to your browser. That's because you haven't used the new component yet. To use the component, you'll have to add it into another component that connects to the root component. In this project, `<App>` is the root component in `index.js`. To make it appear in your application, you'll need to add to the `<App>` component.

Open `src/App.js` in a text editor:

```
nano src/App.js
```

First, you'll need to import the component:

tutorial-03-component/src/App.js

```
import React from 'react';

import Instructions from './Instructions';

import './App.css';

...

export default App;
```

Since it's the default import, you could import to any name you wanted. It's best to keep the names consistent for readability—the import should match the component name, which should match the file name—but the only firm rule is that the component must start with a capital letter. That's how React knows it's a React component.

Now that you've imported the component, add it to the rest of your code as if it were a custom HTML element:

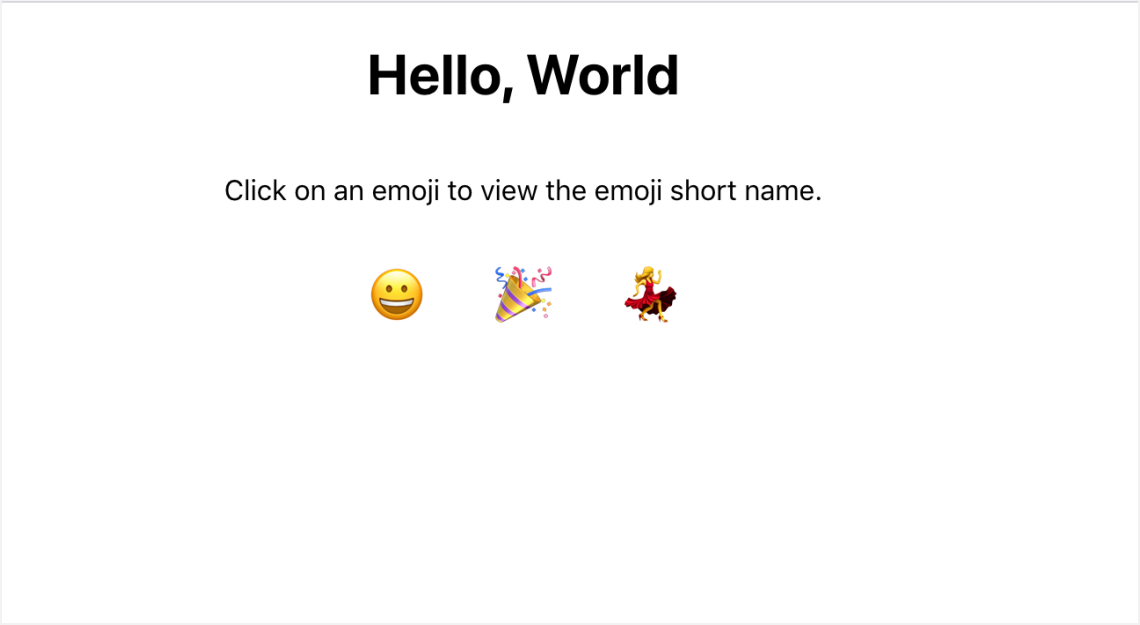
tutorial-03-component/src/App.js

```
import React from 'react';


import Instructions from './Instructions.js'


...
function App() {
  const greeting = "greeting";
  const displayAction = false;
  return(
    <div className="container">
      <h1 id={greeting}>Hello, World</h1>
      {displayAction && <p>I am writing JSX</p>}
      <Instructions />
      <ul>
        {
          emojis.map(emoji => (
            <li key={emoji.name}>
              <button
                onClick={displayEmojiName}
              >
                <span role="img" aria-label={emoji.name}
                    id={emoji.name}>{emoji.emoji}</span>
              </button>
            </li>
          ))
```

```
        }
      </ul>
    </div>
  )
}

export default App;
```

In this code, you wrapped the component with angle brackets. Since this component doesn’t have any children, it can be self closing by ending with `/>`.

Save the file. When you do, the page will refresh and you’ll see the new component.



Browser with instruction text

Now that you have some text, you can add an image. Download an emoji image from wikimedia and save it in the `src` directory as `emoji.svg` with the following command:

```
curl -o src/emoji.svg https://upload.wikimedia.org/wikipedia/commons/3/33/Twemoji_1f602.svg
```

`curl` makes the request to the URL, and the `-o` flag allows you to save the file as `src/emoji.svg`.

Next, open your component file:

```
nano src/Instructions.js
```

Import the emoji and add it to your custom component with a dynamic link:

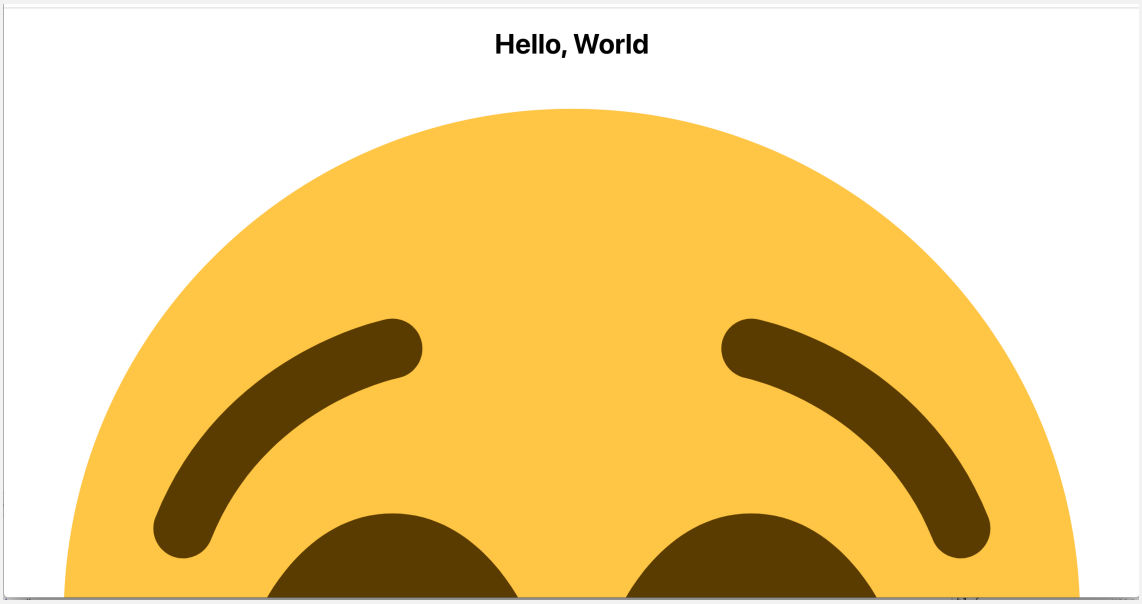
tutorial-03-component/src/Instructions.js

```
import React, { Component } from 'react';
import emoji from './emoji.svg'

export default class Instructions extends Component {

  render() {
    return(
      <>
        <img alt="laughing crying emoji" src={emoji} />
        <p>Click on an emoji to view the emoji short name.</p>
      </>
    )
  }
}
```

Notice that you need to include the file extension `.svg` when importing. When you import, you are importing a dynamic path that is created by webpack when the code compiles. For more information, refer to How To Set Up a React Project with Create React App.

You also need to wrap the `<img>` and `<p>` tags with empty tags to ensure that you are returning a single element.

Save the file. When you reload, the image will be very large compared to the rest of the content:



Browser window with large emoji image

To make the image smaller, you'll need to add some CSS and a `className` to your custom component.

First, in `Instructions.js`, change the empty tags to a div and give it a `className` of `instructions`:

tutorial-03-component/src/Instructions.js

```
import React, { Component } from 'react';
import emoji from './emoji.svg'

export default class Instructions extends Component {

  render() {
    return(
      <div className="instructions">
        <img alt="laughing crying emoji" src={emoji} />
        <p>Click on an emoji to view the emoji short name.</p>
      </div>
    )
  }
}
```

Save and close the file. Next open `App.css`:

```
nano src/App.css
```

Create rules for the `.instructions` class selector:

tutorial-03-component/src/App.css

```
.container {
    display: flex;
    flex-direction: column;
    align-items: center;
}

...

.instructions {
    display: flex;
    flex-direction: column;
    align-items: center;
}
```

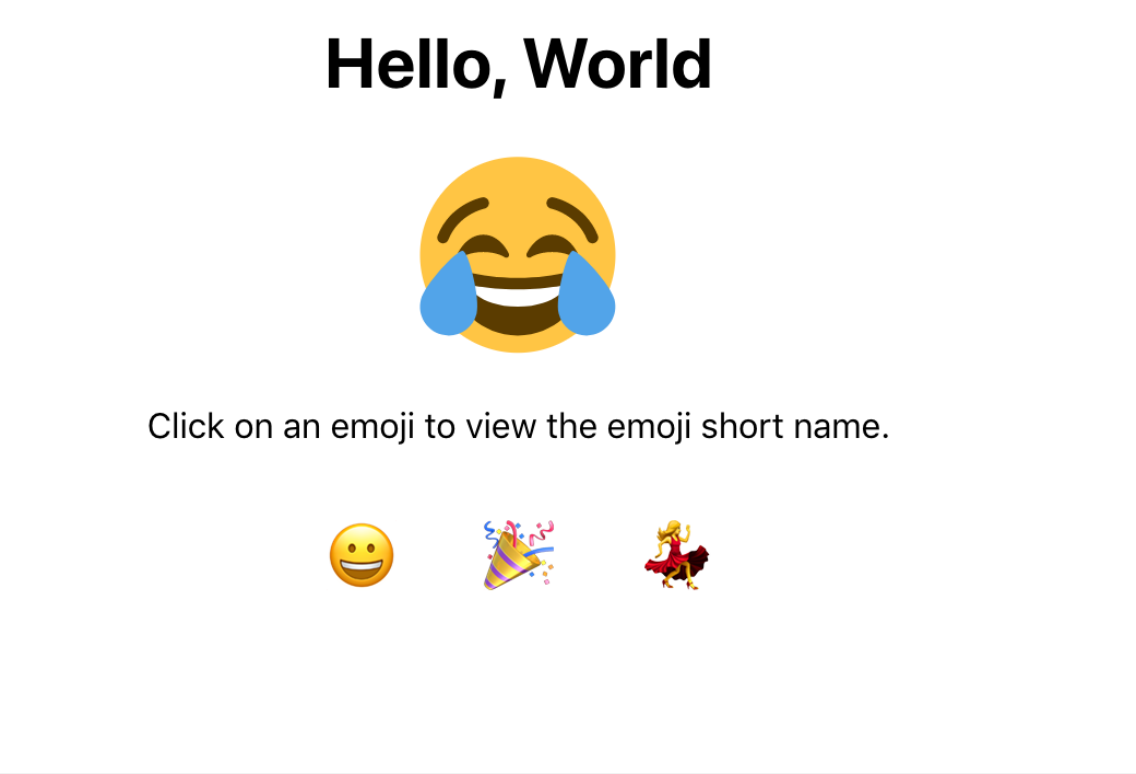
When you add a `display` of `flex` styling, you make the `img` and the `p` centered with flexbox. You changed the direction so that everything lines up vertically with `flex-direction: column;`. The line `align-items: center;` will center the elements on the screen.

Now that your elements are lined up, you need to change the image size. Give the `img` inside the `div` a `width` and `height` of `100px`.

tutorial-03-component/src/App.css

```
.container {
    display: flex;
    flex-direction: column;
    align-items: center;
}

...

.instructions {
    display: flex;
    flex-direction: column;
    align-items: center;
}

.instructions img {
    width: 100px;
    height: 100px;
}
```

Save and close the file. The browser will reload and you'll see the image is much smaller:



Browser window with smaller image

At this point, you've created an independent and reusable custom component. To see how it's reusable, add a second instance to `App.js`.

Open `App.js`:

```
nano src/App.js
```

In `App.js`, add a second instance of the component:

tutorial-03-component/src/App.js

```
import React from 'react';


import Instructions from './Instructions.js'


...


function App() {
  const greeting = "greeting";
  const displayAction = false;
  return(
    <div className="container">
      <h1 id={greeting}>Hello, World</h1>
      {displayAction && <p>I am writing JSX</p>}
      <Instructions />
      <Instructions />
      <ul>
        {
          emojis.map(emoji => (
            <li key={emoji.name}>
              <button
                onClick={displayEmojiName}
              >
                <span role="img" aria-label={emoji.name}
                    id={emoji.name}>{emoji.emoji}</span>
              </button>
```

```
            </li>
          ))
        }
      </ul>
    </div>
  )
}

export default App;
```

Save the file. When the browser reloads, you’ll see the component twice.

Browser with two instances of the Instructions component

In this case, you wouldn't want two instances of `Instructions`, but you can see that the component can be efficiently reused. When you create custom buttons or tables, you will likely use them multiple times on one page, making them perfect for custom components.

For now, you can delete the extra image tag. In your text editor, delete the second `<Instructions />` and save the file:

tutorial-03-component/src/App.js

```
import React from 'react';

import Instructions from './Instructions.js'

...

function App() {
  const greeting = "greeting";
  const displayAction = false;
  return(
    <div className="container">
      <h1 id={greeting}>Hello, World</h1>
      {displayAction && <p>I am writing JSX</p>}
      <Instructions />
      <ul>
        {
          emojis.map(emoji => (
            <li key={emoji.name}>
              <button
                onClick={displayEmojiName}
              >
                <span role="img" aria-label={emoji.name}
                    id={emoji.name}>{emoji.emoji}</span>
              </button>
            </li>
```

```
          ))
        }
      </ul>
    </div>
  )
}

export default App;
```

Now you have a reusable, independent component that you can add to a parent component multiple times. The structure you have now works for a small number of components, but there is a slight problem. All of the files are mixed together. The image for `<Instructions>` is in the same directory as the assets for `<App>`. You also are mixing the CSS code for `<App>` with the CSS for `<Instructions>`.

In the next step, you'll create a file structure that will give each component independence by grouping their functionality, styles, and dependencies together, giving you the ability to move them around as you need.

Step 3 — Creating a Readable File Structure

In this step, you'll create a file structure to organize your components and their assets, such as images, CSS, and other JavaScript files. You'll be grouping code by component, not by asset type. In other words, you won't have a separate directory for CSS, images, and JavaScript. Instead you'll

have a separate directory for each component that will contain the relevant CSS, JavaScript, and images. In both cases, you are separating concerns.

Since you have an independent component, you need a file structure that groups the relevant code. Currently, everything is in the same directory. List out the items in your `src` directory:

```
ls src/
```

The output will show that things are getting pretty cluttered:

```
Output
App.css                 Instructions.js         index.js
App.js                  emoji.svg               serviceWorker.
js
App.test.js             index.css               setupTests.js
```

You have code for the `<App>` component (`App.css`, `App.js`, and `App.test.js`) sitting alongside your root component (`index.css` and `index.js`) and your custom component `Instructions.js`.

React is intentionally agnostic about file structure. It does not recommend a particular structure, and the project can work with a variety of different file hierarchies. But we recommend to add some order to avoid overloading your root directory with components, CSS files, and images that will be difficult to navigate. Also, explicit naming can make it easier to see which

pieces of your project are related. For example, an image file named `Logo.svg` may not clearly be part of a component called `Header.js`.

One of the simplest structures is to create a `components` directory with a separate directory for each component. This will allow you to group your components separately from your configuration code, such as `serviceWorker`, while grouping the assets with the components.

Creating a `Components` Directory

To start, create a directory called `components`:

```
mkdir src/components
```

Next, move the following components and code into the directory: `App.css`, `App.js`, `App.test.js`, `Instructions.js`, and `emoji.svg`:

```
mv src/App.* src/components/
mv src/Instructions.js src/components/
mv src/emoji.svg src/components/
```

Here, you are using a wildcard (`*`) to select all files that start with `App.`.

After you move the code, you'll see an error in your terminal running `npm start`.

Output

```
Failed to compile.


./src/App.js
Error: ENOENT: no such file or directory, open 'your_file_path
/tutorial-03-component/src/App.js'
```

Remember, all of the code is importing using relative paths. If you change the path for some files, you'll need to update the code.

To do that, open `index.js`.

```
nano src/index.js
```

Then change the path of the `App` import to import from the `components/` directory.

tutorial-03-component/src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './components/App';
import * as serviceWorker from './serviceWorker';

...

serviceWorker.unregister();
```

Save and close the file. Your script will detect the changes and the error will disappear.

Now you have components in a separate directory. As your applications become more complex, you may have directories for API services, data stores, and utility functions. Separating component code is the first step, but you still have CSS code for `Instructions` mixed in the `App.css` file. To create this logical separation, you will first move the components into separate directories.

Moving Components to Individual Directories

First, make a directory specifically for the `<App>` component:

```
mkdir src/components/App
```

Then move the related files into the new directory:

```
mv src/components/App.* src/components/App
```

When you do you'll get a similar error to the last section:

```
Output
Failed to compile.


./src/components/App.js
Error: ENOENT: no such file or directory, open 'your_file_path
/tutorial-03-component/src/components/App.js'
```

In this case, you'll need to update two things. First, you'll need to update the path in `index.js`.

Open the `index.js` file:

```
nano src/index.js
```

Then update the import path for App to point to the `App` component in the `App` directory.

tutorial-03-component/src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './components/App/App';
import * as serviceWorker from './serviceWorker';

...

serviceWorker.unregister();
```

Save and close the file. The application still won't run. You'll see an error like this:

Output

```
Failed to compile.

./src/components/App/App.js

Module not found: Can't resolve './Instructions.js' in 'your_file_path/tutorial-03-component/src/components/App'
```

Since `<Instructions>` is not on the same directory level as the `<App>` component, you'll need to change the import path. Before that, create a

directory for `Instructions`. Make a directory called `Instructions` in the `src/components` directory:

```
mkdir src/components/Instructions
```

Then move `Instructions.js` and `emoji.svg` into that directory:

```
mv src/components/Instructions.js src/components/Instructions
mv src/components/emoji.svg src/components/Instructions
```

Now that the `Instructions` component directory has been created, you can finish updating the file paths to connect your component to your app.

Updating `import` Paths

Now that components are in individual directories, you can adjust the import path in `App.js`.

Open `App.js`:

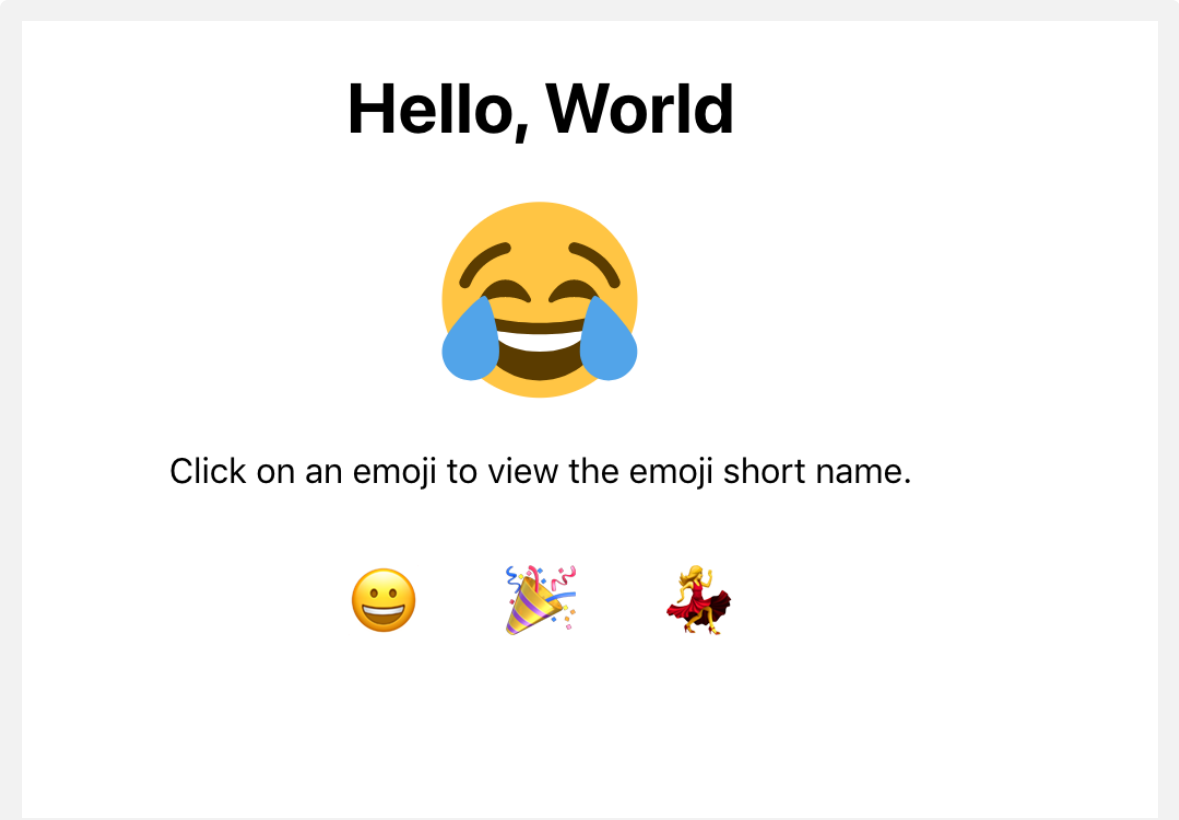
```
nano src/components/App/App.js
```

Since the path is relative, you'll need to move up one directory—`src/components`—then into the `Instructions` directory for `Instructions.js`, but since this is a JavaScript file, you don't need the final import.

tutorial-03-component/src/components/App/App.js

```
import React from 'react';

import Instructions from '../Instructions/Instructions.js';

import './App.css';

...

export default App;
```

Save and close the file. Now that your imports are all using the correct path, you're browser will update and show the application.



Browser window with smaller image

Note: You can also call the root file in each directory `index.js`. For example, instead of `src/components/App/App.js` you could create `src/components/App/index.js`. The advantage to this is that your imports are slightly smaller. If the path points to a directory, the import will look for an `index.js` file. The import for `src/components/App/index.js` in the `src/index.js` file would be `import ./components/App`. The disadvantage of this approach is that you have a lot of files with the same name, which can make it difficult to read in some text editors. Ultimately, it's a personal and team decision, but it's best to be consistent.

Separating Code in Shared Files

Now each component has its own directory, but not everything is fully independent. The last step is to extract the CSS for `Instructions` to a separate file.

First, create a CSS file in `src/components/Instructions`:

```
touch src/components/Instructions/Instructions.css
```

Next, open the CSS file in your text editor:

```
nano src/components/Instructions/Instructions.css
```

Add in the instructions CSS that you created in an earlier section:

tutorial-03-component/src/components/Instructions/Instructions.css

```
.instructions {
    display: flex;
    flex-direction: column;
    align-items: center;
}

.instructions img {
    width: 100px;
    height: 100px;
}
```

Save and close the file. Next, remove the instructions CSS from `src/components/App/App.css`.

```
nano src/components/App/App.css
```

Remove the lines about `.instructions`. The final file will look like this:

tutorial-03-component/src/components/App/App.css

```
.container {
    display: flex;
    flex-direction: column;
    align-items: center;
}

button {
    font-size: 2em;
    border: 0;
    padding: 0;
    background: none;
    cursor: pointer;
}

ul {
    display: flex;
    padding: 0;
}

li {
    margin: 0 20px;
    list-style: none;
    padding: 0;
}
```

Save and close the file. Finally, import the CSS in `Instructions.js`:

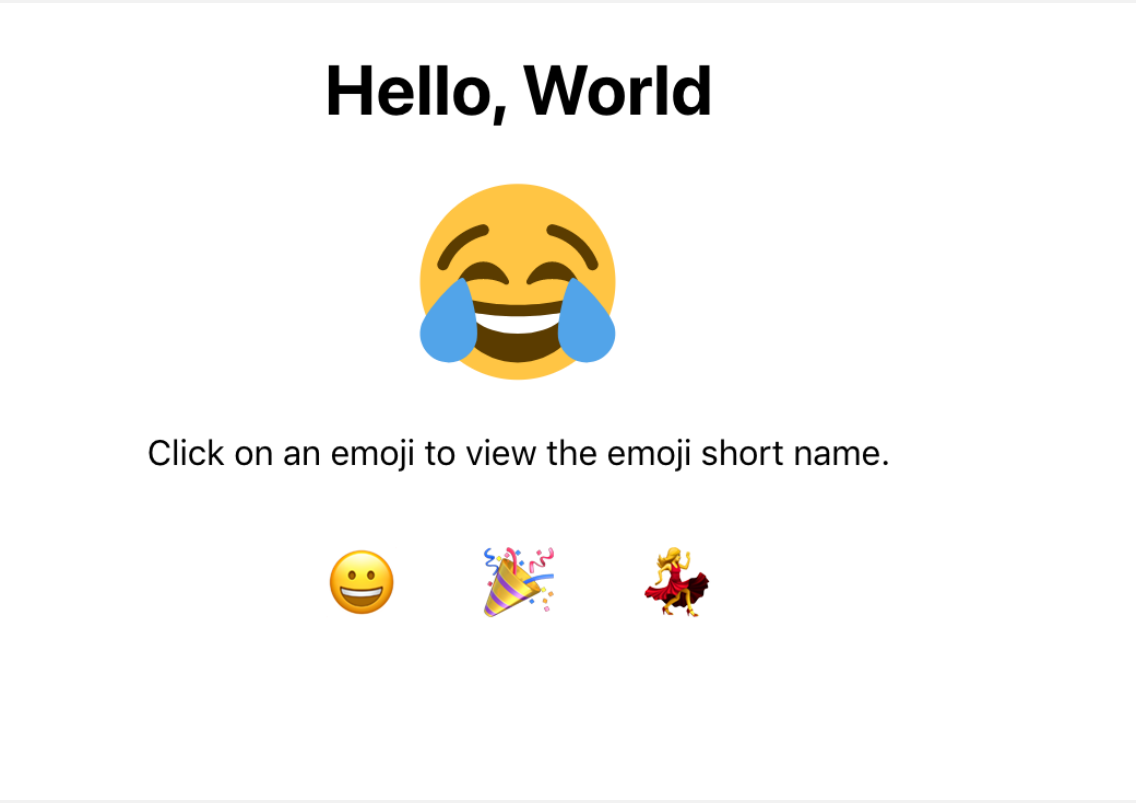
```
nano src/components/Instructions/Instructions.js
```

Import the CSS using the relative path:

tutorial-03-component/src/components/Instructions/Instructions.js

```
import React, { Component } from 'react';
import './Instructions.css';
import emoji from './emoji.svg'

export default class Instructions extends Component {

  render() {
    return(
      <div className="instructions">
        <img alt="laughing crying emoji" src={emoji} />
        <p>Click on an emoji to view the emoji short name.</p>
      </div>
    )
  }
}
```

Save and close the file. Your browser window will look as it did before, except now all the file assets are grouped in the same directory.



Browser window with smaller image

Now, take a final look at the structure. First, the `src/` directory:

```
ls src
```

You have the root component `index.js` and the related CSS `index.css` next to the `components/` directory and utility files such as `serviceWorker.js` and `setupTests.js`:

Output

```
components              serviceWorker.js
index.css               setupTests.js
index.js
```

Next, look inside `components`:

```
ls src/components
```

You'll see a directory for each component:

Output

```
App             Instructions
```

If you look inside each component, you'll see the component code, CSS, test, and image files if they exist.

```
ls src/components/App
```

Output

```
App.css         App.js          App.test.js
```

```
ls src/components/Instructions
```

```
Output
Instructions.css        Instructions.js        emoji.svg
```

At this point, you've created a solid structure for your project. You moved a lot of code around, but now that you have a structure, it will scale easier.

This is not the only way to compose your structure. Some file structures can take advantage of code splitting by specifying a directory that will be split into different packages. Other file structures split by route and use a common directory for components that are used across routes.

For now, stick with a less complex approach. As a need for another structure emerges, it's always easier to move from simple to complex. Starting with a complex structure before you need it will make refactoring difficult.

Now that you have created and organized a class-based component, in the next step you'll create a functional component.

Step 4 — Building a Functional Component

In this step, you'll create a functional component. Functional components are the most common component in contemporary React code. These components tend to be shorter, and unlike class-based components, they can use React hooks, a new form of state and event management.

A functional component is a JavaScript function that returns some JSX. It doesn't need to extend anything and there are no special methods to

memorize.

To refactor `<Instructions>` as a functional component, you need to change the class to a function and remove the render method so that you are left with only the return statement.

To do that, first open `Instructions.js` in a text editor.

```
nano src/components/Instructions/Instructions.js
```

Change the `class` declaration to a `function` declaration:

tutorial-03-component/src/components/Instructions/Instructions.js

```
import React, { Component } from 'react';
import './Instructions.css';
import emoji from './emoji.svg'

export default function Instructions() {
  render() {
    return(
      <div className="instructions">
        <img alt="laughing crying emoji" src={emoji} />
        <p>Click on an emoji to view the emoji short name.</p>
      </div>
    )
  }
}
```

Next, remove the import of `{ Component }`:

tutorial-03-component/src/components/Instructions/Instructions.js

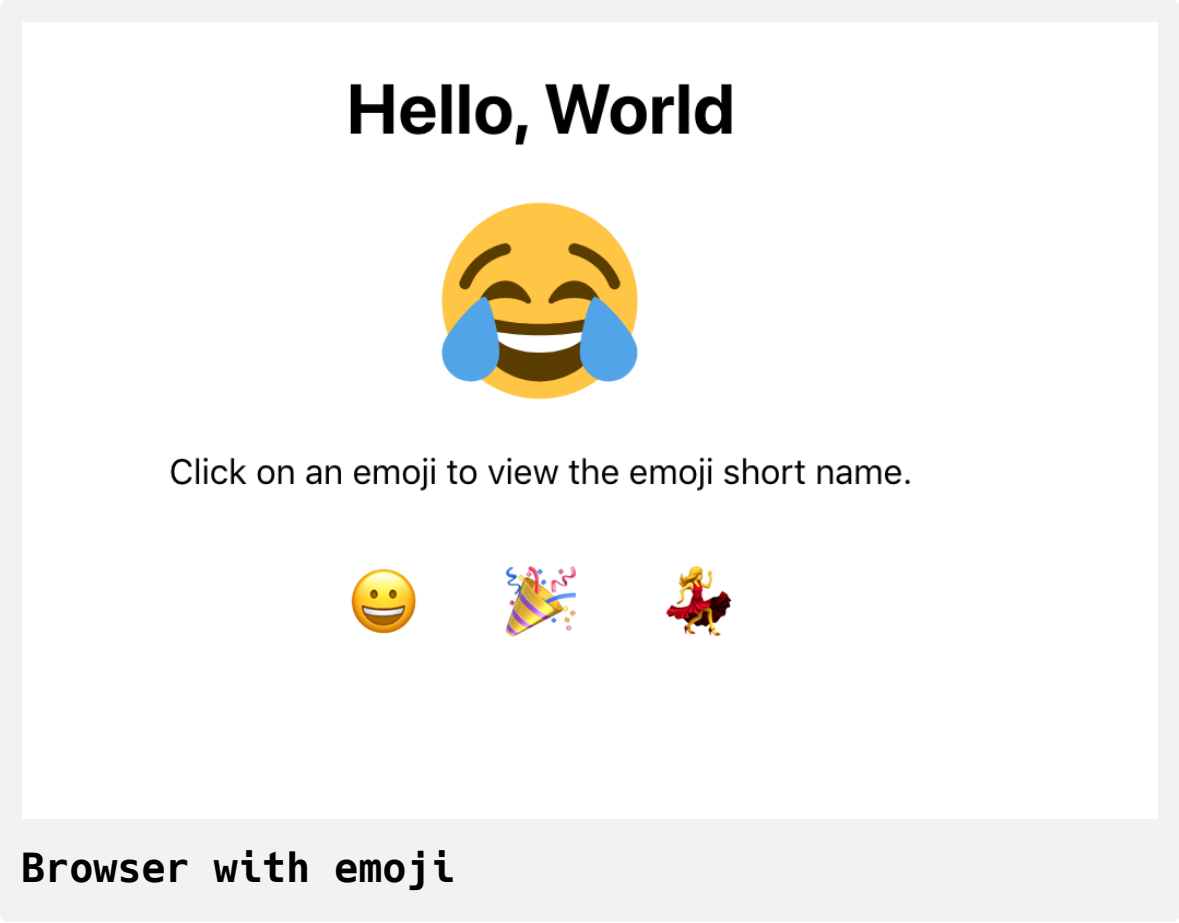
```
import React from 'react';
import './Instructions.css';
import emoji from './emoji.svg'

export default function Instructions() {

  render() {
    return(
      <div className="instructions">
        <img alt="laughing crying emoji" src={emoji} />
        <p>Click on an emoji to view the emoji short name.</p>
      </div>
    )
  }
}
```

Finally, remove the `render()` method. At that point, you are only returning JSX.

tutorial-03-component/src/components/Instructions/Instructions.js

```
import React from 'react';
import './Instructions.css';
import emoji from './emoji.svg'

export default function Instructions() {
  return(
    <div className="instructions">
        <img alt="laughing crying emoji" src={emoji} />
        <p>Click on an emoji to view the emoji short name.</p>
    </div>
  )
}
```

Save the file. The browser will refresh and you'll see your page as it was before.



Browser with emoji

You could also rewrite the function as an arrow function using the implicit return. The main difference is that you lose the function body. You will also need to first assign the function to a variable and then export the variable:

tutorial-03-component/src/components/Instructions/Instructions.js

```
import React from 'react';
import './Instructions.css';
import emoji from './emoji.svg'

const Instructions = () => (
  <div className="instructions">
    <img alt="laughing crying emoji" src={emoji} />
    <p>Click on an emoji to view the emoji short name.</p>
  </div>
)

export default Instructions;
```

Simple functional components and class-based components are very similar. When you have a simple component that doesn't store state, it's best to use a functional component. The real difference between the two is how you store a component's state and use properties. Class-based components use methods and properties to set state and tend to be a little longer. Functional components use hooks to store state or manage changes and tend to be a little shorter.

Conclusion

Now you have a small application with independent pieces. You created two major types of components: functional and class. You separated out parts of the components into directories so that you could keep similar pieces of code grouped together. You also imported and reused the components.

With an understanding of components, you can start to look at your applications as pieces that you can take apart and put back together. Projects become modular and interchangable. The ability to see whole applications as a series of components is an important step in thinking in React. If you would like to look at more React tutorials, take a look at our React Topic page, or return to the How To Code in React.js series page.

How To Customize React Components with Props

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

In this tutorial, you'll create custom components by passing props to your component. Props are arguments that you provide to a JSX element. They look like standard HTML props, but they aren't predefined and can have many different JavaScript data types including numbers, strings, functions, arrays, and even other React components. Your custom components can use props to display data or use the data to make the components interactive. Props are a key part of creating components that are adaptable to different situations, and learning about them will give you the tools to develop custom components that can handle unique situations.

After adding props to your component, you will use `PropTypes` to define the type of data you expect a component to receive. `PropTypes` are a simple type system to check that data matches the expected types during runtime. They serve as both documentation and an error checker that will help keep your application predictable as it scales.

By the end of the tutorial, you'll use a variety of `props` to build a small application that will take an array of animal data and display the information, including the name, scientific name, size, diet, and additional information.

Note: The first step sets up a blank project on which you will build the tutorial exercise. If you already have a working project and want to go directly to working with props, start with **Step 2**.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.20.1 and npm version 6.14.4. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- In following this tutorial, you will use Create React App. You can find instructions for installing an application with Create React App at How To Set Up a React Project with Create React App. This tutorial also assumes a knowledge of React components, which you can learn about in our How To Create Custom Components in React tutorial.

- You will also need to know the basics of JavaScript, which you can find in How To Code in JavaScript, along with a basic knowledge of HTML and CSS. A good resource for HTML and CSS is the Mozilla Developer Network.

Step 1 — Creating an Empty Project

In this step, you'll create a new project using Create React App. Then you will delete the sample project and related files that are installed when you bootstrap the project. Finally, you will create a simple file structure to organize your components.

To start, make a new project. In your command line, run the following script to install a fresh project using `create-react-app`:

```
npx create-react-app prop-tutorial
```

After the project is finished, change into the directory:

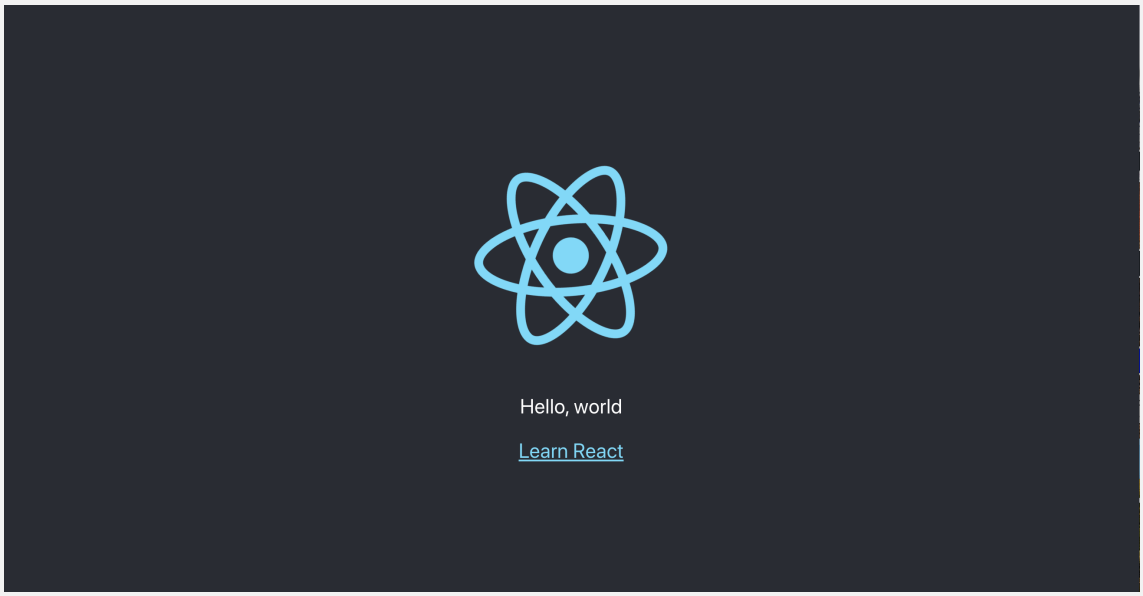
```
cd prop-tutorial
```

In a new terminal tab or window, start the project using the Create React App start script. The browser will autorefresh on changes, so leave this script running the whole time that you work:

```
npm start
```

You will get a running local server. If the project did not open in a browser window, you can open it by navigating to `http://localhost:3000/`. If you are running this from a remote server, the address will be `http://your_domain:3000`.

Your browser will load with a simple React application included as part of Create React App:



React template project

You will be building a completely new set of custom components. You'll start by clearing out some boilerplate code so that you can have an empty project.

To start, open `src/App.js` in a text editor. This is the root component that is injected into the page. All components will start from here. You can find more information about `App.js` at How To Set Up a React Project with Create React App.

Open `src/App.js` with the following command:

```
nano src/App.js
```

You will see a file like this:

prop-tutorial/src/App.js

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

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
```

```
export default App;
```

Delete the line `import logo from './logo.svg';`. Then replace everything in the `return` statement to return a set of empty tags: `<></>`. This will give you a validate page that returns nothing. The final code will look like this:

prop-tutorial/src/App.js

```
import React from 'react';
import './App.css';

function App() {
  return <></>;
}

export default App;
```

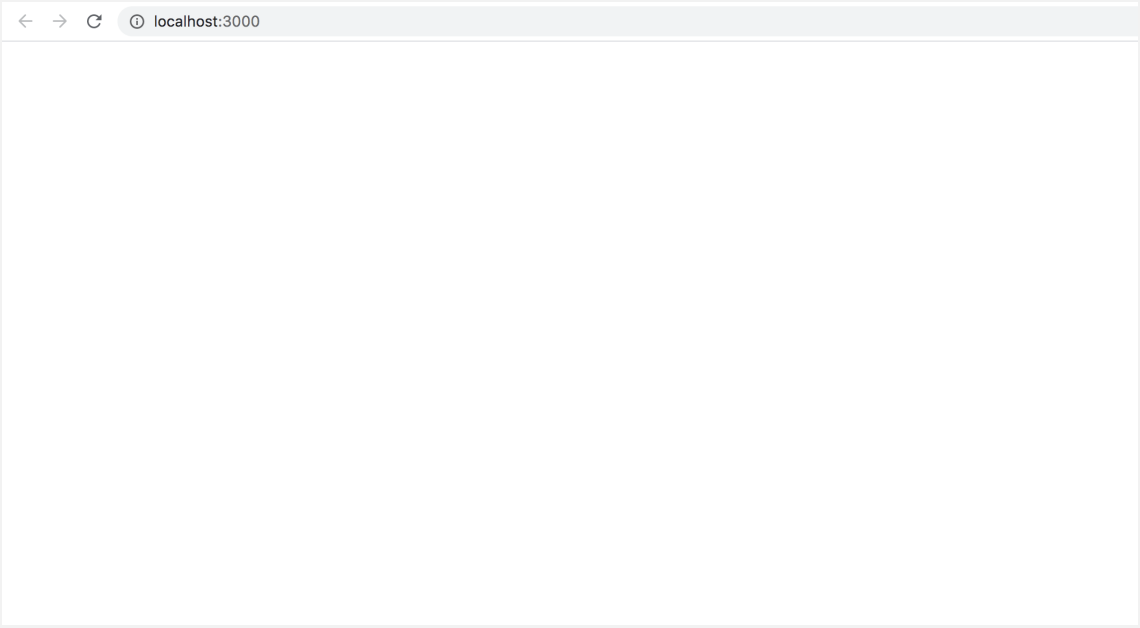
Save and exit the text editor.

Finally, delete the logo. You won't be using it in your application and you should remove unused files as you work. It will save you from confusion in the future.

In the terminal window type the following command:

```
rm src/logo.svg
```

If you look at your browser, you will see a blank screen.



blank screen in chrome

Now that you have cleared out the sample Create React App project, create a simple file structure. This will help you keep your components isolated and independent.

Create a directory called `components` in the `src` directory. This will hold all of your custom components.

```
mkdir src/components
```

Each component will have its own directory to store the component file along with the styles, images if there are any, and tests.

Create a directory for `App`:

```
mkdir src/components/App
```

Move all of the `App` files into that directory. Use the wildcard, `*`, to select any files that start with `App.` regardless of file extension. Then use the `mv` command to put them into the new directory.

```
mv src/App.* src/components/App
```

Finally, update the relative import path in `index.js`, which is the root component that bootstraps the whole process.

```
nano src/index.js
```

The import statement needs to point to the `App.js` file in the `App` directory, so make the following highlighted change:

prop-tutorial/src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './components/App/App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want your app to work offline and load faster, you ca
// unregister() to register() below. Note this comes with some
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Save and exit the file.

Now that the project is set up, you can create your first component.

Step 2 — Building Dynamic Components with Props

In this step, you will create a component that will change based on the input information called props. Props are the arguments you pass to a function or class, but since your components are transformed into HTML-like objects with JSX, you will pass the props like they are HTML attributes. Unlike HTML elements, you can pass many different data types, from strings, to arrays, to objects, and even functions.

Here you will create a component that will display information about animals. This component will take the name and scientific name of the animal as strings, the size as an integer, the diet as an array of strings, and additional information as an object. You'll pass the information to the new component as props and consume that information in your component.

By the end of this step, you'll have a custom component that will consume different props. You'll also reuse the component to display an array of data using a common component.

Adding Data

First, you need some sample data. Create a file in the `src/App` directory called data.

```
touch src/components/App/data.js
```

Open the new file in your text editor:

```
nano src/components/App/data.js
```

Next, add an array of objects you will use as sample data:

prop-tutorial/src/components/App/data.js

```
export default [
  {
    name: 'Lion',
    scientificName: 'Panthero leo',
    size: 140,
    diet: ['meat'],
  },
  {
    name: 'Gorilla',
    scientificName: 'Gorilla beringei',
    size: 205,
    diet: ['plants', 'insects'],
    additional: {
      notes: 'This is the eastern gorilla. There is also a west
 that is a different species.'
    }
  },
  {
    name: 'Zebra',
    scientificName: 'Equus quagga',
    size: 322,
    diet: ['plants'],
    additional: {

      notes: 'There are three different species of zebra.',
      link: 'https://en.wikipedia.org/wiki/Zebra'
```

```
    }
  }
]
```

The array of objects contains a variety of data and will give you an opportunity to try a variety of props. Each object is a separate animal with the name of the animal, the scientific name, size, diet, and an optional field called `additional`, which will contain links or notes. In this code, you also exported the array as the `default`.

Save and exit the file.

Creating Components

Next, create a placeholder component called `AnimalCard`. This component will eventually take props and display the data.

First, make a directory in `src/components` called `AnimalCard` then `touch` a file called `src/components/AnimalCard/AnimalCard.js` and a CSS file called `src/components/AnimalCard/AnimalCard.css`.

```
mkdir src/components/AnimalCard
touch src/components/AnimalCard/AnimalCard.js
touch src/components/AnimalCard/AnimalCard.css
```

Open `AnimalCard.js` in your text editor:

```
nano src/components/AnimalCard/AnimalCard.js
```

Add a basic component that imports the CSS and returns an `<h2>` tag.

prop-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import './AnimalCard.css'

export default function AnimalCard() {
  return <h2>Animal</h2>
}
```

Save and exit the file. Now you need to import the data and component into your base `App` component.

Open `src/components/App/App.js`:

```
nano src/components/App/App.js
```

Import the data and the component, then loop over the data returning the component for each item in the array:

prop-tutorial/src/components/App/App.js

```
import React from 'react';
import data from './data';
import AnimalCard from '../AnimalCard/AnimalCard';
import './App.css';

function App() {
  return (
    <div className="wrapper">
      <h1>Animals</h1>
      {data.map(animal => (
        <AnimalCard key={animal.name}/>
      ))}
    </div>
  )
}

export default App;
```

Save and exit the file. Here, you use the `.map()` array method to iterate over the data. In addition to adding this loop, you also have a wrapping `div` with a class that you will use for styling and an `<h1>` tag to label your project.

When you save, the browser will reload and you'll see a label for each card.

React project in the browser without styling

Next, add some styling to line up the items. Open `App.css`:

```
nano src/components/App/App.css
```

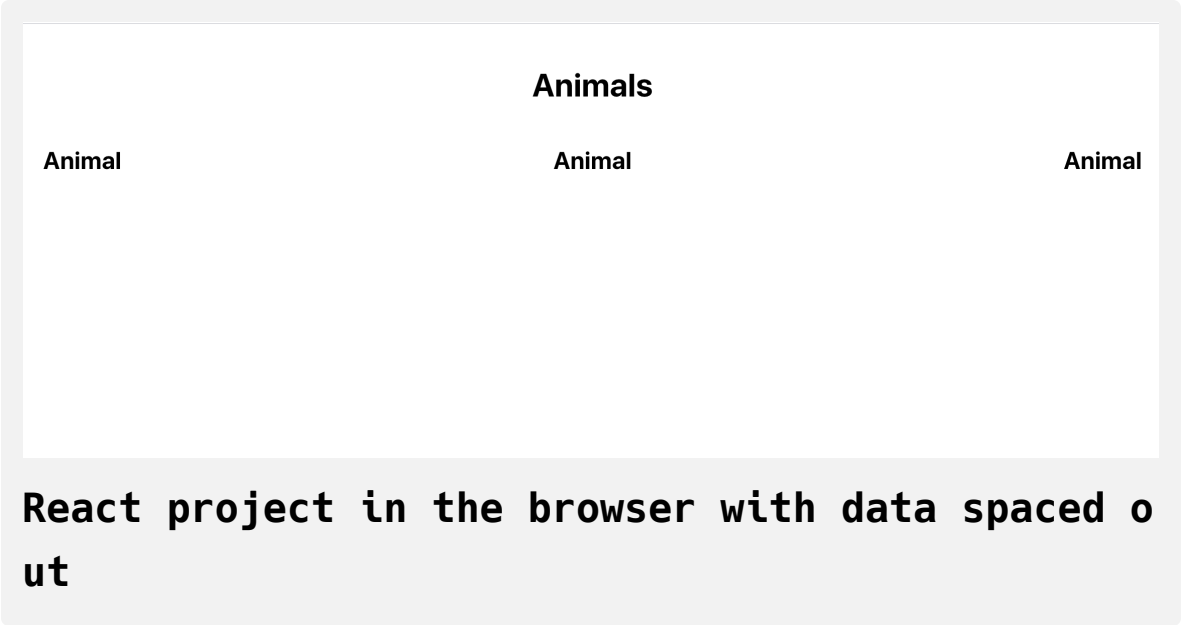
Replace the contents with the following to arrange the elements:

prop-tutorial/src/components/App/App.css

```
.wrapper {
    display: flex;
    flex-wrap: wrap;
    justify-content: space-between;
    padding: 20px;
}

.wrapper h1 {
    text-align: center;
    width: 100%;
}
```

This will use flexbox to rearrange the data so it will line up. The `padding` gives some space in the browser window. `justify-content` will spread out the extra space between elements, and `.wrapper h1` will give the `Animal` label the full width.

Save and exit the file. When you do, the browser will refresh and you'll see some data spaced out.



React project in the browser with data spaced out

Adding Props

Now that you have your components set up, you can add your first prop. When you looped over your data, you had access to each object in the `data` array and the items it contained. You will add each piece of the data to a separate prop that you will then use in your `AnimalCard` component.

Open `App.js`:

```
nano src/components/App/App.js
```

Add a prop of `name` to `AnimalCard`.

prop-tutorial/src/components/App/App.js

```
import React from 'react';
...
function App() {
  return (
    <div className="wrapper">
      <h1>Animals</h1>
      {data.map(animal => (
        <AnimalCard
          key={animal.name}
          name={animal.name}
        />
      ))}
    </div>
  )
}

export default App;
```

Save and exit the file. The `name` prop looks like a standard HTML attribute, but instead of a string, you'll pass the `name` property from the `animal` object in curly braces.

Now that you've passed one prop to the new component, you need to use it. Open `AnimalCard.js`:

```
nano src/components/AnimalCard/AnimalCard.js
```

All props that you pass into the component are collected into an object that will be the first argument of your function. Destructure the object to pull out individual props:

prop-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import './AnimalCard.css'

export default function AnimalCard(props) {
  const { name } = props;
  return (

    <h2>{name}</h2>
  );
}
```

Note that you do not need to destructure a prop to use it, but that this is a useful method for dealing with the sample data in this tutorial.

After you destructure the object, you can use the individual pieces of data. In this case, you'll use the title in an `<h2>` tag, surrounding the value with

curly braces so that React will know to evaluate it as JavaScript.

You can also use a property on the `prop` object using dot notation. As an example, you could create an `<h2>` element like this: `<h2>{props.title}</h2>`. The advantage of destructring is that you can collect unused props and use the object rest operator.

Save and exit the file. When you do, the browser will reload and you'll see the specific name for each animal instead of a placeholder.

React projects with animal names rendered

The `name` property is a string, but props can be any data type that you could pass to a JavaScript function. To see this at work, add the rest of the data.

Open the `App.js` file:

```
nano src/components/App/App.js
```

Add a prop for each of the following: `scientificName`, `size`, `diet`, and `additional`. These include strings, integers, arrays, and objects.

prop-tutorial/src/components/App/App.js

```
import React from 'react';
...

function App() {
  return (
    <div className="wrapper">
      <h1>Animals</h1>
      {albums.map(album => (
        <AnimalCard
          additional={animal.additional}
          diet={animal.diet}
          key={animal.name}
          name={animal.name}
          scientificName={animal.scientificName}
          size={animal.size}
        />
      ))}
    </div>
  )
}

export default App;
```

Since you are creating an object, you can add them in any order you want. Alphabetizing makes it easier to skim a list of props especially in a larger list. You also can add them on the same line, but separating to one per line keeps things readable.

Save and close the file. Open `AnimalCard.js`.

```
nano src/components/AnimalCard/AnimalCard.js
```

This time, destructure the props in the function parameter list and use the data in the component:

prop-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import './AnimalCard.css'


export default function AnimalCard({
  additional,
  diet,
  name,
  scientificName,
  size
}) {
  return (
    <div>
      <h2>{name}</h2>
      <h3>{scientificName}</h3>
      <h4>{size}kg</h4>
      <div>{diet.join(', ')}.</div>
    </div>
  );
}
```

After pulling out the data, you can add the `scientificName` and `size` into heading tags, but you'll need to convert the array into a string so that React

can display it on the page. You can do that with `join(', ')`, which will create a comma separated list.

Save and close the file. When you do, the browser will refresh and you'll see the structured data.

React project with animals with full data

You could create a similar list with the `additional` object, but instead add a function to alert the user with the data. This will give you the chance to pass functions as props and then use data inside a component when you call a function.

Open `App.js`:

```
nano src/components/App/App.js
```

Create a function called `showAdditionalData` that will convert the object to a string and display it as an alert.

prop-tutorial/src/components/App/App.js

```
import React from 'react';
...

function showAdditional(additional) {
  const alertInformation = Object.entries(additional)
    .map(information => `${information[0]}: ${information[1]}`)
    .join('\n');
  alert(alertInformation)
};

function App() {
  return (
    <div className="wrapper">
      <h1>Animals</h1>
      {data.map(animal => (
        <AnimalCard
          additional={animal.additional}
          diet={animal.diet}
          key={animal.name}
          name={animal.name}
          scientificName={animal.scientificName}
          showAdditional={showAdditional}
          size={animal.size}
        />
      ))}
```

```
    </div>
  )
}

export default App;
```

The function `showAdditional` converts the object to an array of pairs where the first item is the key and the second is the value. It then maps over the data converting the key-pair to a string. Then it joins them with a line break —`\n`—before passing the complete string to the alert function.

Since JavaScript can accept functions as arguments, React can also accept functions as props. You can therefore pass `showAdditional` to `AnimalCard` as a prop called `showAdditional`.

Save and close the file. Open `AnimalCard`:

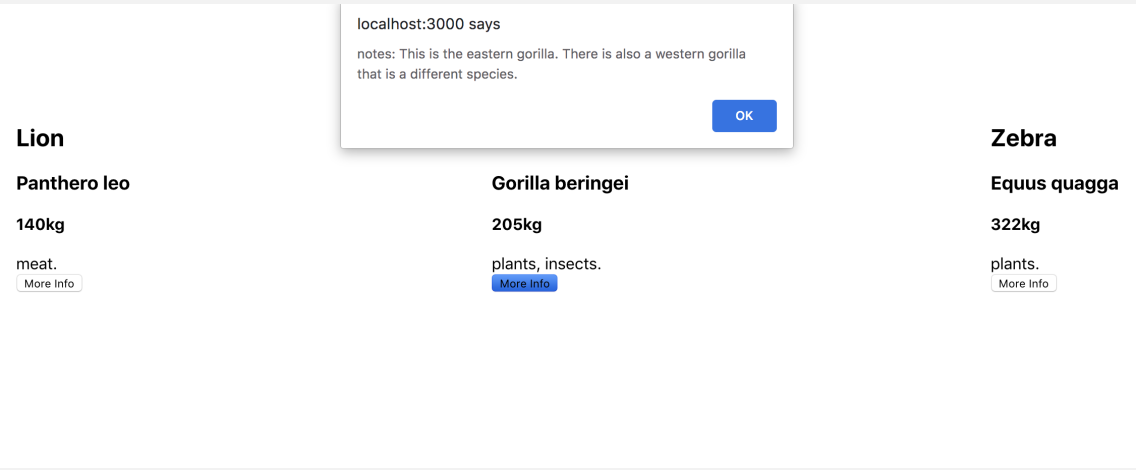
```
nano src/components/AnimalCard/AnimalCard.js
```

Pull the `showAdditional` function from the props object, then create a `<button>` with an `onClick` event that calls the function with the `additional` object:

prop-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import './AnimalCard.css'


export default function AnimalCard({
  additional,
  diet,
  name,
  scientificName,
  showAdditional,
  size
}) {
  return (
    <div>
      <h2>{name}</h2>
      <h3>{scientificName}</h3>
      <h4>{size}kg</h4>
      <div>{diet.join(', ')}.</div>
      <button onClick={() => showAdditional(additional)}>More I
    </div>
  );
}
```

Save the file. When you do, the browser will refresh and you'll see a button after each card. When you click the button, you'll get an alert with the additional data.



Alert with information

If you try clicking **More Info** for the `Lion`, you will get an error. That's because there is no additional data for the lion. You'll see how to fix that in Step 3.

Finally, add some styling to the music card. Add a `className` of `animal-wrapper` to the div in `AnimalCard`:

prop-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import './AnimalCard.css'

export default function AnimalCard({
...
  return (
    <div className="animal-wrapper">
...
    </div>
  )
}
```

Save and close the file. Open `AnimalCard.css`:

```
nano src/components/AnimalCard/AnimalCard.css
```

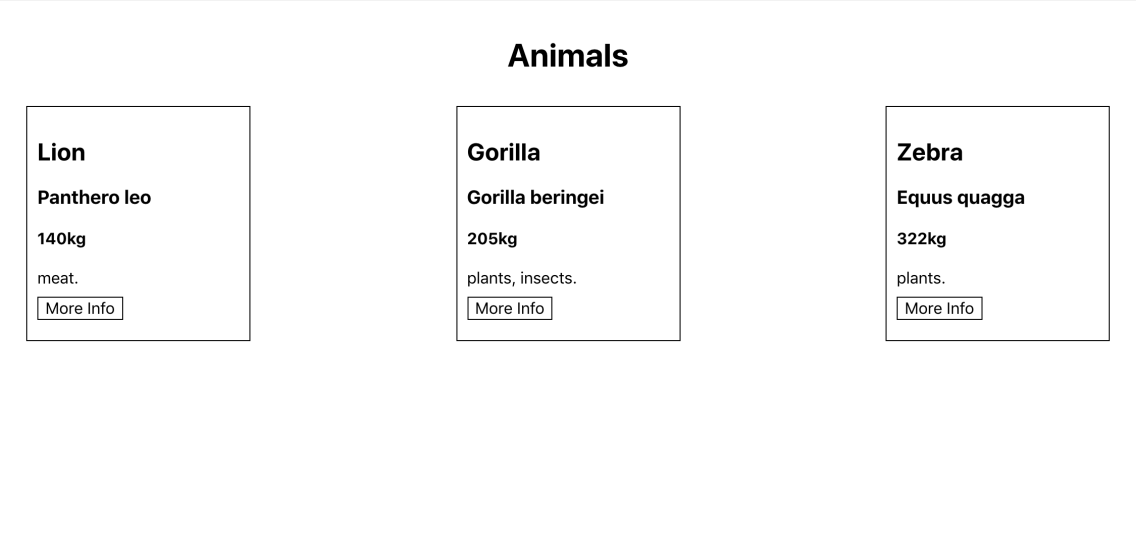
Add CSS to give the cards and the button a small border and padding:

prop-tutorial/src/components/AnimalCard/AnimalCard.css

```
.animal-wrapper {
    border: solid black 1px;
    margin: 10px;
    padding: 10px;
    width: 200px;
}

.animal-wrapper button {
    font-size: 1em;
    border: solid black 1px;
    padding: 10;
    background: none;
    cursor: pointer;
    margin: 10px 0;
}
```

This CSS will add a slight border to the card and replace the default button styling with a border and padding. `cursor: pointer` will change the cursor when you hover over the button.

Save and close the file. When you do the browser will refresh and you'll see the data in individual cards.



React project with styled animal cards

At this point, you've created two custom components. You've passed data to the second component from the first component using props. The props included a variety of data, such as strings, integers, arrays, objects, and functions. In your second component, you used the props to create a dynamic component using JSX.

In the next step, you'll use a type system called `prop-types` to specify the structure your component expects to see, which will create predictability in your app and prevent bugs.

Step 3 — Creating Predictable Props with `PropTypes` and `defaultProps`

In this step, you'll add a light type system to your components with `PropTypes`. `PropTypes` act like other type systems by explicitly defining the type of data you expect to receive for a certain prop. They also give you the

chance to define default data in cases where the prop is not always required. Unlike most type systems, `PropTypes` is a runtime check, so if the props do not match the type the code will still compile, but will also display a console error.

By the end of this step, you'll add predictability to your custom component by defining the type for each prop. This will ensure that the next person to work on the component will have a clear idea of the structure of the data the component will need.

The `prop-types` package is included as part of the Create React App installation, so to use it, all you have to do is import it into your component.

Open up `AnimalCard.js`:

```
nano src/components/AnimalCard/AnimalCard.js
```

Then import `PropTypes` from `prop-types`:

prop-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import PropTypes from 'prop-types';
import './AnimalCard.css'


export default function AnimalCard({
...
}
```

Add `PropTypes` directly to the component function. In JavaScript, functions are objects, which means you can add properties using dot syntax. Add the following `PropTypes` to `AnimalCard.js`:

prop-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import PropTypes from 'prop-types';
import './AnimalCard.css'

export default function AnimalCard({
...
}

AnimalCard.propTypes = {
  additional: PropTypes.shape({
    link: PropTypes.string,
    notes: PropTypes.string
  }),
  diet: PropTypes.arrayOf(PropTypes.string).isRequired,
  name: PropTypes.string.isRequired,
  scientificName: PropTypes.string.isRequired,
  showAdditional: PropTypes.func.isRequired,
  size: PropTypes.number.isRequired,
}
```

Save and close the file.

As you can see, there are many different `PropTypes`. This is only a small sample; see the official React documentation to see the others you can use.

Let's start with the `name` prop. Here, you are specifying that `name` must be a `string`. The property `scientificName` is the same. `size` is a `number`, which can include both floats such as `1.5` and integers such as `6`. `showAdditional` is a function (`func`).

`diet`, on the other hand, is a little different. In this case, you are specifying that `diet` will be an `array`, but you also need to specify what this array will contain. In this case, the array will only contain strings. If you want to mix types, you can use another prop called `oneOfType`, which takes an array of valid `PropTypes`. You can use `oneOfType` anywhere, so if you wanted `size` to be either a number or a string you could change it to this:

```
size: PropTypes.oneOfType([PropTypes.number, PropTypes.string])
```

The prop `additional` is also a little more complex. In this case, you are specifying an object, but to be a little more clear, you are stating what you want the object to contain. To do that, you use `PropTypes.shape`, which takes an object with additional fields that will need their own `PropTypes`. In this case, `link` and `notes` are both `PropTypes.string`.

Currently, all of the data is well-formed and matches the props. To see what happens if the `PropTypes` don't match, open up your data:

```
nano src/components/App/data.js
```

Change the size to a string on the first item:

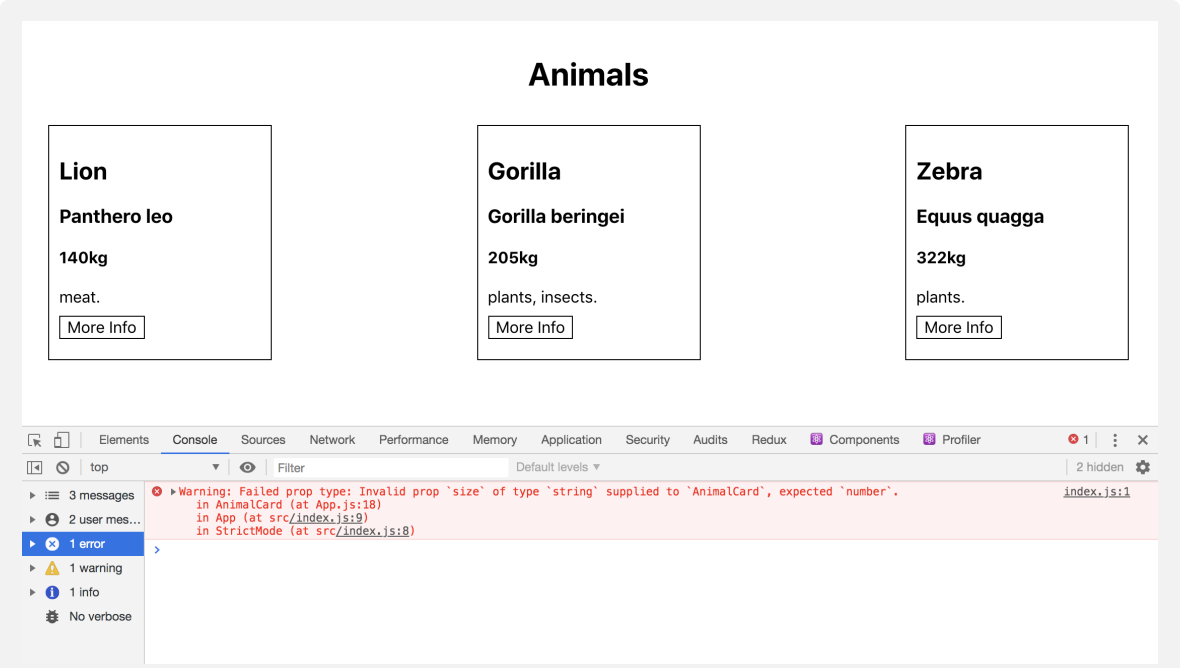
prop-tutorial/src/components/App/data.js

```
export default [
  {
    name: 'Lion',
    scientificName: 'Panthero leo',
    size: '140',
    diet: ['meat'],
  },
...
]
```

Save the file. When you do the browser will refresh and you'll see an error in the console.

Error

```
index.js:1 Warning: Failed prop type: Invalid prop `size` of t
ype `string` supplied to `AnimalCard`, expected `number`.
    in AnimalCard (at App.js:18)
    in App (at src/index.js:9)
    in StrictMode (at src/index.js:8)
```



Browser with type error

Unlike other type systems such as TypeScript, `PropTypes` will not give you a warning at build time, and as long as there are no code errors, it will still compile. This means that you could accidentally publish code with prop errors.

Change the data back to the correct type:

prop-tutorial/src/components/App/data.js

```
export default [
  {
    name: 'Lion',
    scientificName: 'Panthero leo',
    size: 140,
    diet: ['meat'],
  },
...
]
```

Save and close the file.

Open up `AnimalCard.js`:

```
nano src/components/AnimalCard/AnimalCard.js
```

Every prop except for `additional` has the `isRequired` property. That means, that they are required. If you don't include a required prop, the code will still compile, but you'll see a runtime error in the console.

If a prop isn't required, you can add a default value. It's good practice to always add a default value to prevent runtime errors if a prop is not required. For example, in the `AnimalCard` component, you are calling a

function with the `additional` data. If it’s not there, the function will try and modify an object that doesn’t exist and the application will crash.

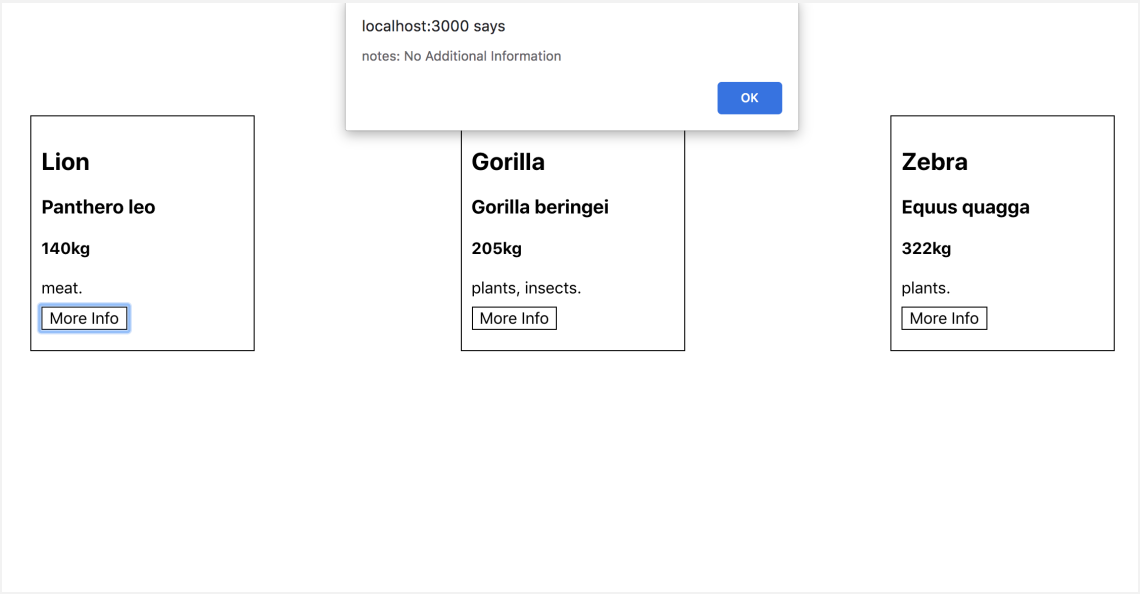
To prevent this problem, add a `defaultProp` for `additional`:

prop-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import PropTypes from 'prop-types';
import './AnimalCard.css'

export default function AnimalCard({
...
}

AnimalCard.propTypes = {
  additional: PropTypes.shape({
    link: PropTypes.string,
    notes: PropTypes.string
  }),
...
}

AnimalCard.defaultProps = {
  additional: {
    notes: 'No Additional Information'
  }
}
```

You add the `defaultProps` to the function using dot syntax just as you did with `propTypes`, then you add a default value that the component should use if the prop is `undefined`. In this case, you are matching the shape of `additional`, including a message that the there is no additional information.

Save and close the file. When you do, the browser will refresh. After it refreshes, click on the **More Info** button for **Lion**. It has no `additional` field in the data so the prop is `undefined`. But `AnimalCard` will substitute in the default prop.



Browser with default message in the alert

Now your props are well-documented and are either required or have a default to ensure predictable code. This will help future developers (including yourself) understand what props a component needs. It will make

it easier to swap and reuse your components by giving you full information about how the component will use the data it is receiving.

Conclusion

In this tutorial, you have created several components that use props to display information from a parent. Props give you the flexibility to begin to break larger components into smaller, more focused pieces. Now that you no longer have your data tightly coupled with your display information, you have the ability to make choices about how to segment your application.

Props are a crucial tool in building complex applications, giving the opportunity to create components that can adapt to the data they receive. With `PropTypes`, you are creating predictable and readable components that will give a team the ability to reuse each other's work to create a flexible and stable code base. If you would like to look at more React tutorials, take a look at our React Topic page, or return to the How To Code in React.js series page.

How To Create Wrapper Components in React with Props

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

In this tutorial, you'll create wrapper components with props using the React JavaScript library. Wrapper components are components that surround unknown components and provide a default structure to display the child components. This pattern is useful for creating user interface (UI) elements that are used repeatedly throughout a design, like modals, template pages, and information tiles.

To create wrapper components, you'll first learn to use the rest and spread operators to collect unused props to pass down to nested components. Then you'll create a component that uses the built-in `children` component to wrap nested components in JSX as if they were HTML elements. Finally, you'll pass components as props to create flexible wrappers that can embed custom JSX in multiple locations in a component.

During the tutorial, you'll build components to display a list of animal data in the form of cards. You'll learn to split data and refactor components as you create flexible wrapping components. By the end of this tutorial, you'll have a working application that will use advanced prop techniques to create reusable components that will scale and adapt as you application grows and changes.

Note: The first step sets up a blank project on which you will build the tutorial exercise. If you already have a working project and want to go directly to working with props, start with **Step 2**.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.20.1 and npm version 6.14.4. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- In this tutorial, you will create an app with Create React App. You can find instructions for installing an application with Create React App and general information about how it works at How To Set Up a React Project with Create React App.

- You will be using React components, which you can learn about in our How To Create Custom Components in React tutorial. It will also help to have a basic understanding of React props, which you can learn about in How to Customize React Components with Props.

- You will also need a basic knowledge of JavaScript, which you can find in our How To Code in JavaScript series, along with a basic

knowledge of HTML and CSS. A good resource for HTML and CSS is the Mozilla Developer Network.

Step 1 — Creating an Empty Project

In this step, you'll create a new project using Create React App. Then you will delete the sample project and related files that are installed when you bootstrap the project. Finally, you will create a simple file structure to organize your components. This will give you a solid basis on which to build this tutorial's wrapper application in the next step.

To start, make a new project. In your command line, run the following script to install a fresh project using `create-react-app`:

```
npx create-react-app wrapper-tutorial
```

After the project is finished, change into the directory:

```
cd wrapper-tutorial
```

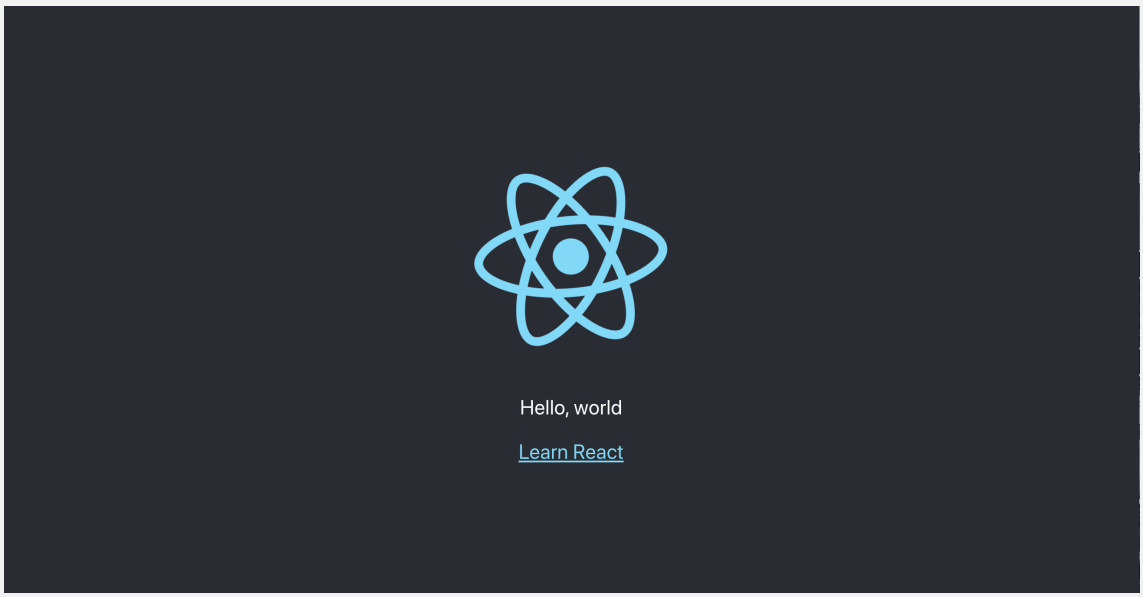
In a new terminal tab or window, start the project using the Create React App start script. The browser will auto-refresh on changes, so leave this script running while you work:

```
npm start
```

You will get a running local server. If the project did not open in a browser window, you can open it with `http://localhost:3000/`. If you are running

this from a remote server, the address will be `http://your_domain:3000`.

Your browser will load with a simple React application included as part of Create React App:



React template project

You will be building a completely new set of custom components, so you'll need to start by clearing out some boilerplate code so that you can have an empty project.

To start, open `src/App.js` in a text editor. This is the root component that is injected into the page. All components will start from here. You can find more information about `App.js` at How To Set Up a React Project with Create React App.

Open `src/App.js` with the following command:

```
nano src/App.js
```

You will see a file like this:

wrapper-tutorial/src/App.js

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

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
```

```
export default App;
```

Delete the line `import logo from './logo.svg';`. Then replace everything in the `return` statement to return a set of empty tags: `<></>`. This will give you a valid page that returns nothing. The final code will look like this:

wrapper-tutorial/src/App.js

```
import React from 'react';
import './App.css';

function App() {
  return <></>;
}

export default App;
```

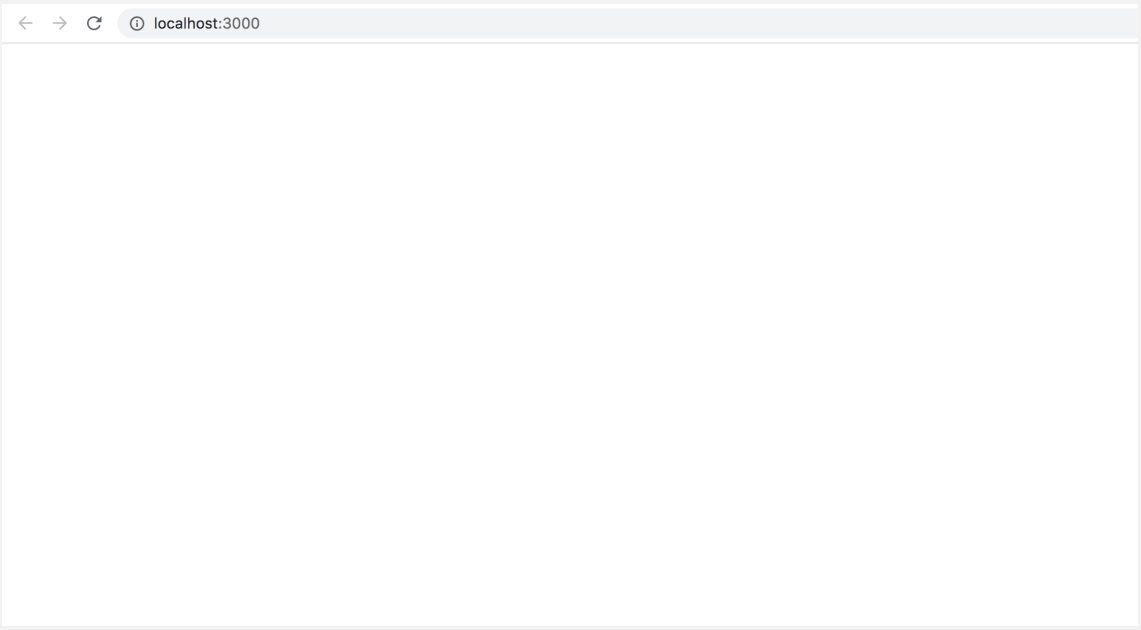
Save and exit the text editor.

Finally, delete the logo. You won't be using it in your application and you should remove unused files as you work. It will save you from confusion in the long run.

In the terminal window type the following command:

```
rm src/logo.svg
```

If you look at your browser, you will see a blank screen.



blank screen in chrome

Now that you have cleared out the sample Create React App project, create a simple file structure. This will help you keep your components isolated and independent.

Create a directory called `components` in the `src` directory. This will hold all of you custom components.

```
mkdir src/components
```

Each component will have its own directory to store the component file along with the styles, images if there are any, and tests.

Create a directory for `App`:

```
mkdir src/components/App
```

Move all of the `App` files into that directory. Use the wildcard, `*`, to select any files that start with `App.` regardless of file extension. Then use the `mv` command to put them into the new directory:

```
mv src/App.* src/components/App
```

Next, update the relative import path in `index.js`, which is the root component that bootstraps the whole process:

```
nano src/index.js
```

The import statement needs to point to the `App.js` file in the `App` directory, so make the following highlighted change:

wrapper-tutorial/src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './components/App/App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want your app to work offline and load faster, you ca
// unregister() to register() below. Note this comes with some
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Save and exit the file.

Now that the project is set up, you can create your first component.

Step 2 — Collecting Unused Props with `...props`

In this step, you'll create a component to display a set of data about a group of animals. Your component will contain a second nested component to display some information visually. To connect the parent and nested component, you'll use the rest and spread operators to pass unused props from the parent to the child without the parent needing to be aware of the names or types of the props.

By the end of this step, you'll have a parent component that can provide props to nested components without having to know what the props are. This will keep the parent component flexible, allowing you to update the child component without having to change the parent.

Creating an `AnimalCard` Component

To start, create a set of data for your animals. First, open a file containing the data set in the `components/App` directory:

```
nano src/components/App/data.js
```

Add the following data:

src/components/App/data.js

```
export default [
  {
    name: 'Lion',
    scientificName: 'Panthero leo',
    size: 140,
    diet: ['meat']
  },
  {
    name: 'Gorilla',
    scientificName: 'Gorilla beringei',
    size: 205,
    diet: ['plants', 'insects']
  },
  {
    name: 'Zebra',
    scientificName: 'Equus quagga',
    size: 322,
    diet: ['plants'],
  }
]
```

This list of animals is an array of objects that includes the animal's name, scientific name, weight, and diet.

Save and close the file.

Next, create a directory for the `AnimalCard` component:

```
mkdir src/components/AnimalCard
```

Open a new file in the directo:

```
nano src/components/AnimalCard/AnimalCard.js
```

Now add a component that will take the `name`, `diet`, and `size` as a prop and display it:

wrapper-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import PropTypes from 'prop-types';

export default function AnimalCard({ diet, name, size }) {
  return(
    <div>
      <h3>{name}</h3>
      <div>{size}kg</div>
      <div>{diet.join(', ')}.</div>
    </div>
  )
}

AnimalCard.propTypes = {
  diet: PropTypes.arrayOf(PropTypes.string).isRequired,
  name: PropTypes.string.isRequired,
  size: PropTypes.number.isRequired,
}
```

Here you are destructuring the props in the parameter list for the `AnimalCard` function, then displaying the data in a `div`. The `diet` data is listed as a

single string using the `join()` method. Each piece of data includes a corresponding `PropType` to make sure the data type is correct.

Save and close the file.

Now that you have your component and your data, you need to combine them together. To do that, import the component and the data into the root component of your project: `App.js`.

First, open the component:

```
nano src/components/App/App.js
```

From there, you can loop over the data and return a new `AnimalCard` with the relevant props. Add the highlighted lines to `App.js`:

wrapper-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';

import animals from './data';
import AnimalCard from '../AnimalCard/AnimalCard';

function App() {
  return (
    <div className="wrapper">
      {animals.map(animal =>
        <AnimalCard
          diet={animal.diet}
          key={animal.name}
          name={animal.name}
          size={animal.size}
        />
      )}
    </div>
  );
}

export default App;
```

Save and close the file.

As you work on more complex projects, your data will come from more varied places, such as APIs, `localStorage`, or static files. But the process for using each of these will be similar: assign the data to a variable and loop over the data. In this case, the data is from a static file, so you are importing directly to a variable.

In this code, you use the `.map()` method to iterate over `animals` and display the props. Notice that you do not have to use every piece of data. You are not explicitly passing the `scientificName` property, for example. You are also adding a separate `key` prop that React will use to keep track of the mapped data. Finally, you are wrapping the code with a `div` with a `className` of `wrapper` that you'll use to add some styling.

To add this styling, open `App.css`:

```
nano src/components/App/App.css
```

Remove the boilerplate styling and add flex properties to a class called `wrapper`:

prop-tutorial/src/components/App/App.css

```
.wrapper {
    display: flex;
    flex-wrap: wrap;
    justify-content: space-between;
    padding: 20px;
}
```

This will use flexbox layout to organize the data so it will line up. `padding` gives some space in the browser window, and `justify-content` spreads out the extra space between elements.

Save and exit the file. When you do, the browser will refresh and you'll see some data spaced out.

Browser with data spaced out

Creating a Details Component

You now have a simple component that displays the data. But let's say you wanted to give the `diet` data a little flair by converting the text to an emoji.

You can do this by converting the data in your component.

React is designed to be flexible, so when you are thinking about how to convert data, you have a few different options:

- You can create a function inside the component that converts the text to an emoji.
- You can create a function and store it in a file outside the component so that you can reuse the logic across different components.
- You can create a separate component that converts the text to an emoji.

Each approach is fine when applied to the right use case, and you'll find yourself switching between them as you build an application. To avoid premature abstraction and complexity, you should use the first option to start. If you find yourself wanting to reuse logic, you can pull the function out separately from the component. The third option is best if you want to have a reusable piece that includes the logic and the markup, or that you want to isolate to use across the application.

In this case, we'll make a new component, since we will want to add more data later and we are combining markup with conversion logic.

The new component will be called `AnimalDetails`. To make it, create a new directory:

```
mkdir src/components/AnimalDetails
```

Next, open `AnimalDetails.js` in your text editor:

```
nano src/components/AnimalDetails/AnimalDetails.js
```

Inside the file, make a small component that displays the `diet` as an emoji:

wrapper-tutorial/src/components/AnimalDetails/AnimalDetails.js

```
import React from 'react';
import PropTypes from 'prop-types';
import './AnimalDetails.css';

function convertFood(food) {
  switch(food) {
    case 'insects':
      return '🕷';
    case 'meat':
      return '🍖';
    case 'plants':
    default:
      return '🌱';
  }
}

export default function AnimalDetails({ diet }) {
  return(
    <div className="details">
      <h4>Details:</h4>
      <div>
        Diet: {diet.map(food => convertFood(food)).join(' ')}
      </div>
    </div>
```

```
  )
}


AnimalDetails.propTypes = {
  diet: PropTypes.arrayOf(PropTypes.string).isRequired,
}
```

The `AnimalDetails.propTypes` object sets up the function to take a prop of `diet` that is an array of strings. Then inside the component, the code loops over the `diet` and converts the string to an emoji using the switch statement.

Save and close the file.

You are also importing some CSS, so let's add that now.

Open `AnimalDetails.css`:

```
nano src/components/AnimalDetails/AnimalDetails.css
```

Add some CSS to give the element a border and margin to separate the details from the rest of the component:

wrapper-tutorial/src/components/AnimalDetails/AnimalDetails.css

```
.details {
    border-top: gray solid 1px;
    margin: 20px 0;
}
```

We use `.details` to match the rule to elements with a `className` of `details`.

Save and close the file.

Now that you have a new custom component, you can add it to your `AnimalCard` component. Open `AnimalCard.js`:

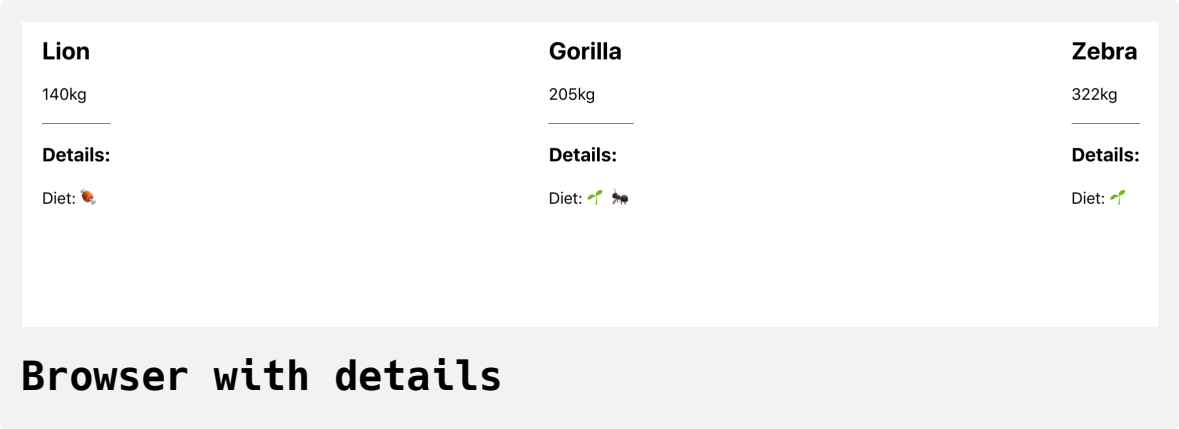
```
nano src/components/AnimalCard/AnimalCard.js
```

Replace the `diet.join` statement with the new `AnimalDetails` component and pass `diet` as a prop by adding the highlighted lines:

wrapper-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import PropTypes from 'prop-types';
import AnimalDetails from '../AnimalDetails/AnimalDetails';

export default function AnimalCard({ diet, name, size }) {
  return(
    <div>
      <h3>{name}</h3>
      <div>{size}kg</div>
      <AnimalDetails
        diet={diet}
      />
    </div>
  )
}

AnimalCard.propTypes = {
  diet: PropTypes.arrayOf(PropTypes.string).isRequired,
  name: PropTypes.string.isRequired,
  size: PropTypes.number.isRequired,
}
```

Save the file and you'll see the new details in the browser.



Browser with details

Passing Details Through a Component with `...props`

The components are working well together, but there's a slight inefficiency in `AnimalCard`. You are explicitly pulling `diet` out from the `props` argument, but you aren't using the data. Instead, you are passing it through to the component. There's nothing inherently wrong about this—in fact, it's often better to err on the side of too much communication. But in doing this, you make your code more difficult to maintain. Whenever you want to pass new data to `AnimalDetails`, you need to update three places: `App`, where you pass the props, `AnimalDetails`, which consumes the prop, and `AnimalCard`, which is the go-between.

A better way is to gather any unused props inside `AnimalCard` and then pass those directly to `AnimalDetails`. This gives you the chance to make changes to `AnimalDetails` without changing `AnimalCard`. In effect, `AnimalCard` doesn't need to know anything about the props or the `PropTypes` that are going into `AnimalDetails`.

To do that, you'll use the object rest operator. This operator collects any items that are not pulled out during destructuring and saves them into a new object.

Here's a simple example:

```
const dog = {
    name: 'dog',
    diet: ['meat']
}

const { name, ...props  } = dog;
```

In this case, the variable `name` will be `'dog'` and the variable `props` will be `{ diet: ['meat']}`.

Up till now, you've passed all props as if they were HTML attributes, but you can also use objects to send props. To use an object as a prop, you need to use the spread operator—`...props`—surrounded with curly braces. This will change each key-value pair into a prop.

Open `AnimalCard.js`:

```
nano src/components/AnimalCard/AnimalCard.js
```

Inside, remove `diet` from the destructured object and instead collect the rest of the props into a variable called `props`. Then pass those props directly to `AnimalDetails`:

wrapper-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import PropTypes from 'prop-types';
import AnimalDetails from '../AnimalDetails/AnimalDetails';

export default function AnimalCard({ name, size, ...props }) {
  return(
    <div>
      <h3>{name}</h3>
      <div>{size}kg</div>
      <AnimalDetails
        {...props}
      />
    </div>
  )
}

AnimalCard.propTypes = {
  name: PropTypes.string.isRequired,
  size: PropTypes.number.isRequired,
}
```

Notice that you can remove the `diet` `PropType` since you are not using the prop in this component.

In this case, you are only passing one prop to `AnimalDetails`. In cases where you have multiple props, the order will matter. A later prop will overwrite earlier props, so if you have a prop you want to take priority, make sure it is last. This can cause some confusion if your `props` object has a property that is also a named value.

Save and close the file. The browser will refresh and everything will look the same:

Browser with details

To see how the `...props` object adds flexibility, let's pass the `scientificName` to `AnimalDetails` via the `AnimalCard` component.

First, open `App.js`:

```
nano src/components/App/App.js
```

Then pass the `scientificName` as a prop:

wrapper-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';

import animals from './data';
import AnimalCard from '../AnimalCard/AnimalCard';

function App() {
  return (
    <div className="wrapper">
      {animals.map(animal =>
        <AnimalCard
          diet={animal.diet}
          key={animal.name}
          name={animal.name}
          size={animal.size}
          scientificName={animal.scientificName}
        />
      )}
    </div>
  );
}

export default App;
```

Save and close the file.

Skip over `AnimalCard`; you won't need to make any changes there. Then open `AnimalDetails` so you can consume the new prop:

```
nano src/components/AnimalDetails/AnimalDetails.js
```

The new prop will be a string, which you'll add to the `details` list along with a line declaring the `PropType`:

wrapper-tutorial/src/components/AnimalDetails/AnimalDetails.js

```
import React from 'react';
...
export default function AnimalDetails({ diet, scientificName })
  return(
    <div className="details">
      <h4>Details:</h4>
      <div>
        Scientific Name: {scientificName}.
      </div>
      <div>
        Diet: {diet.map(food => convertFood(food)).join(' ')}
      </div>
    </div>
  )
}

AnimalDetails.propTypes = {
  diet: PropTypes.arrayOf(PropTypes.string).isRequired,
  scientificName: PropTypes.string.isRequired,
}
```

Save and close the file. When you do, the browser will refresh and you'll see the new details without any changes to the `AnimalCard` component:

Browser with scientific name

In this step, you learned how to create flexible parent props that can take unknown props and pass them into nested components with the spread operator. This is a common pattern that will give you the flexibility you need to create components with focused responsibilities. In the next step, you'll create components that can take unknown components as a prop using the built in `children` prop.

Step 3 — Creating Wrapper Components with `children`

In this step, you'll create a wrapper component that can take an unknown group of components as a prop. This will give you the ability to nest components like standard HTML, and it will give you a pattern for creating reusable wrappers that will let you make a variety of components that need a common design but a flexible interior.

React gives you a built-in prop called `children` that collects any children components. Using this makes creating wrapper components intuitivie and readable.

To start, make a new component called `Card`. This will be a wrapper component to create a standard style for any new card components.

Create a new directory:

```
mkdir src/components/Card
```

Then open the `Card` component in your text editor:

```
nano src/components/Card/Card.js
```

Create a component that takes `children` and `title` as props and wraps them in a `div` by adding the following code:

wrapper-tutorial/src/components/Card/Card.js

```
import React from 'react';
import PropTypes from 'prop-types';
import './Card.css';

export default function Card({ children, title }) {
  return(
    <div className="card">
      <div className="card-details">
        <h2>{title}</h2>
      </div>
      {children}
    </div>
  )
}

Card.propTypes = {
  children: PropTypes.oneOfType([
    PropTypes.arrayOf(PropTypes.element),
    PropTypes.element.isRequired
  ]),
  title: PropTypes.string.isRequired,
}
```

The `PropTypes` for the `children` are new. The `children` prop can either be a JSX element or an array of JSX elements. The `title` is a string.

Save and close the file.

Next, add some styling. Open `Card.css`:

```
nano src/components/Card/Card.css
```

Your card will have a border and a line under the details.

wrapper-tutorial/src/components/Card/Card.css

```
.card {
    border: black solid 1px;
    margin: 10px;
    padding: 10px;
    width: 200px;
}

.card-details {
    border-bottom: gray solid 1px;
    margin-bottom: 20px;
}
```

Save and close the file. Now that you have your component you need to use it. You could wrap each `AnimalCard` with the `Card` component in `App.js`, but since the name `AnimalCard` implies it is already a `Card`, it would be better to use the `Card` component inside of `AnimalCard`.

Open up `AnimalCard`:

```
nano src/components/AnimalCard/AnimalCard.js
```

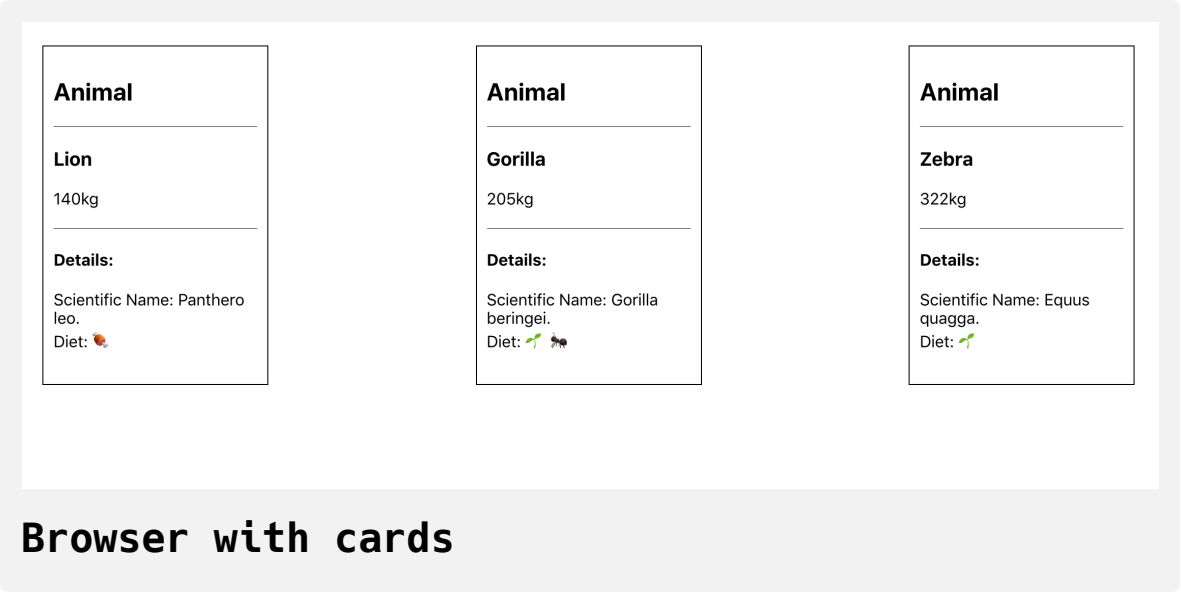
Unlike other props, you don’t pass `children` explicitly. Instead, you include the JSX as if they were HTML child elements. In other words, you just nest them inside of the element, like the following:

wrapper-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import PropTypes from 'prop-types';
import Card from '../Card/Card';
import AnimalDetails from '../AnimalDetails/AnimalDetails';

export default function AnimalCard({ name, size, ...props }) {
  return(
    <Card title="Animal">
      <h3>{name}</h3>
      <div>{size}kg</div>
      <AnimalDetails
        {...props}
      />
    </Card>
  )
}

AnimalCard.propTypes = {
  name: PropTypes.string.isRequired,
  size: PropTypes.number.isRequired,
}
```

Unlike a React component, you do not need to have a single root element as a child. That's why the `PropType` for `Card` specified it could be an array of elements or a single element. In addition to passing the `children` as nested components, you are giving the card a title of `Animal`.

Save and close the file. When you do, the browser will refresh and you'll see the updated card component.



Browser with cards

Now you have a reusable `Card` component that can take any number of nested children. The primary advantage of this is that you can reuse the `Card` with any arbitrary component. If you wanted to make a `Plant` card, you could do that by wrapping the plant information with the `Card` component. It doesn't even need to relate at all: If you wanted to reuse the `Card` component in a completely different applications that lists things like music or account data, you could do that, too. The `Card` component doesn't care

what the children are; you are just reusing the wrapper element, which in this case is the styled border and title.

The downside to using `children` is that you can only have one instance of the child prop. Occasionally, you'll want a component to have custom JSX in multiple places. Fortunately, you can do that by passing JSX and React components as props, which we will cover in the next step.

Step 4 — Passing Components as Props

In this step, you'll modify your `Card` component to take other components as props. This will give your component maximum flexibility to display unknown components or JSX in multiple locations throughout the page. Unlike `children`, which you can only use once, you can have as many components as props, giving your wrapper component the ability to adapt to a variety of needs while maintaining a standard look and structure.

By the end of this step, you'll have a component that can wrap children components and also display other components in the card. This pattern will give you flexibility when you need to create components that need information that is more complex than simple strings and integers.

Let's modify the `Card` component to take an arbitrary React element called `details`.

First, open the `Card` component:

```
nano src/components/Card/Card.js
```

Next, add a new prop called `details` and place it below the `<h2>` element:

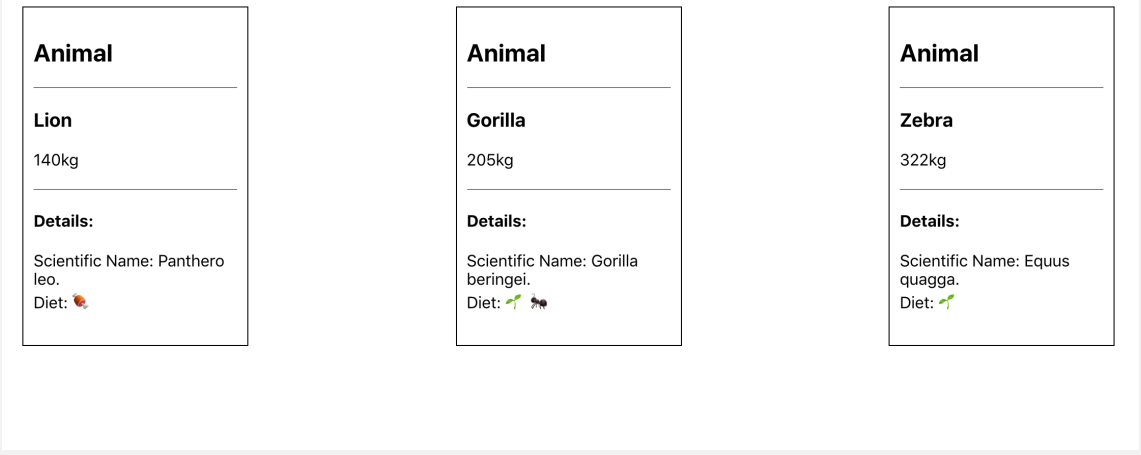
wrapper-tutorial/src/components/Card/Card.js

```
import React from 'react';
import PropTypes from 'prop-types';
import './Card.css';

export default function Card({ children, details, title }) {
  return(
    <div className="card">
      <div className="card-details">
        <h2>{title}</h2>
        {details}
      </div>
      {children}
    </div>
  )
}

Card.propTypes = {
  children: PropTypes.oneOfType([
    PropTypes.arrayOf(PropTypes.element),
    PropTypes.element.isRequired
  ]),
  details: PropTypes.element,
  title: PropTypes.string.isRequired,
}
```

```
Card.defaultProps = {
  details: null,
}
```

This prop will have the same type as `children`, but it should be optional. To make it optional, you add a default value of `null`. In this case, if a user passes no details, the component will still be valid and will not display anything extra.

Save and close the file. The page will refresh and you'll see the same image as before:



Browser with cards

Now add some details to the `AnimalCard`. First, open `AnimalCard`.

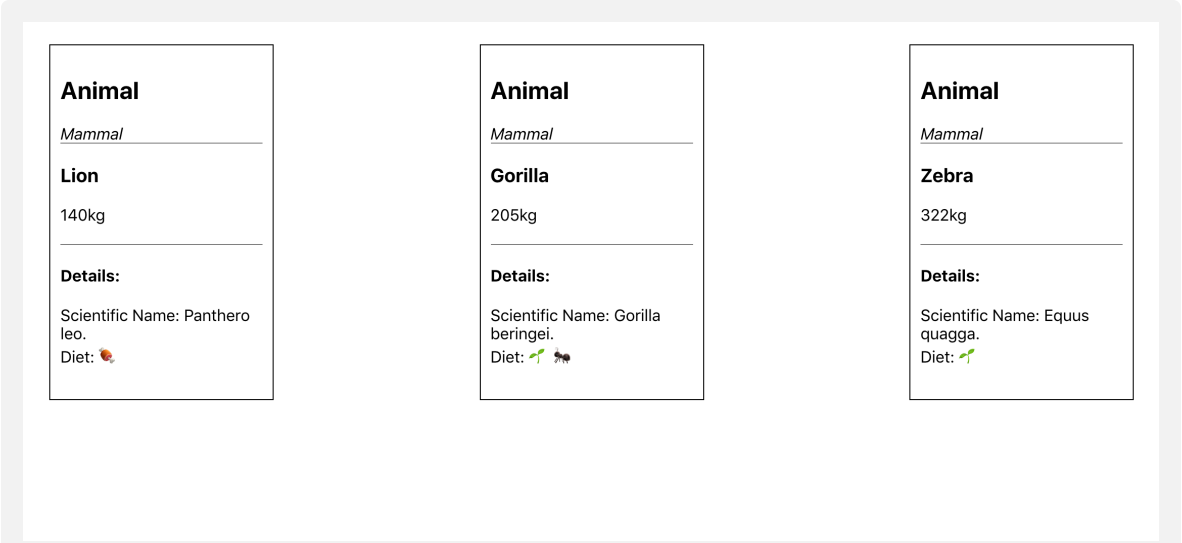
```
nano src/components/AnimalCard/AnimalCard.js
```

Since the `Card` component is already using `children`, you'll need to pass the new JSX component as a prop. Since these are all mammals, add that to the card, but wrap it in `<em>` tags to make it italic.

wrapper-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
...

export default function AnimalCard({ name, size, ...props }) {
  return(
    <Card title="Animal" details={<em>Mammal</em>}>
      <h3>{name}</h3>
      <div>{size}kg</div>
      <AnimalDetails
        {...props}
      />
    </Card>
  )
}
...
```

Save the file. When you do, the browser will refresh and you'll see the update, including the phrase **Mammal**.



Browser with card and details

This prop is already powerful because it can take JSX of any size. In this example, you added only a single element, but you could pass as much JSX as you wanted. It also doesn't have to be JSX. If you have a complicated markup for example, you wouldn't want to pass it directly in the prop; this would be difficult to read. Instead, you could create a separate component and then pass the component as a prop.

To see this at work, pass `AnimalDetails` to the `details` prop:

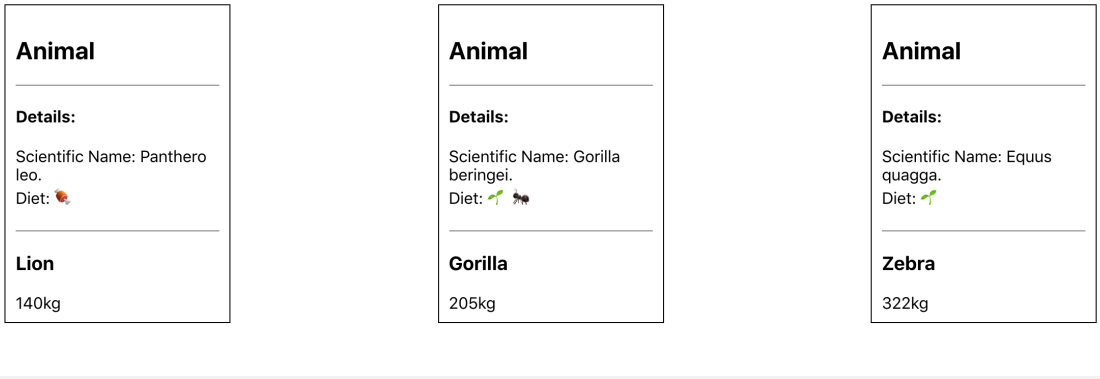
wrapper-tutorial/src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
...

export default function AnimalCard({ name, size, ...props }) {
  return(
    <Card
      title="Animal"
      details={
        <AnimalDetails
          {...props}
        />
      }
    >
      <h3>{name}</h3>
      <div>{size}kg</div>
    </Card>
  )
}
...
```

`AnimalDetails` is more complicated and has a number of lines of markup. If you were to add it directly to `details`, it would increase the prop

substantially and make it difficult to read.

Save and close the file. When you do, the browser will refresh and the details will appear at the top of the card.



Card with details at the top

Now you have a `Card` component that can take custom JSX and place it in multiple spots. You are not restricted to a single prop; you can pass elements to as many props as you want. This gives you the ability to create flexible wrapping components that can give other developers the opportunity to customize a component while retaining its overall style and functionality.

Passing a component as a prop isn't perfect. It's a little more difficult to read and isn't as clear as passing `children`, but they are just as flexible and you can use as many of them as you want in a component. You should use `children` first, but don't hesitate to fall back to props if that is not enough.

In this step, you learned how to pass JSX and React components as props to another component. This will give your component the flexibility to handle many situations where a wrapper component may need multiple props to handle JSX or components.

Conclusion

You have created a variety of wrapping components that can display data flexibly while keeping a predictable look and structure. You created components that can collect and pass unknown props to nested components. You also used the built-in `children` prop to create wrapper components that can handle an arbitrary number of nested elements. Finally, you created a component that can take JSX or React components as a prop so that your wrapper component can handle multiple instances of different customizations.

Wrapper components give you the ability to adapt to unknown circumstances while also maximizing code reuse and consistency. This pattern is useful for creating basic UI elements that you will reuse throughout an application including: buttons, alerts, modals, slide shows, and more. You'll find yourself returning to it many times.

If you would like to look at more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Style React Components

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

In this tutorial, you'll learn three different ways to style React components: plain Cascading Style Sheets (CSS), inline styles with JavaScript-style objects, and JSS, a library for creating CSS with JavaScript. These options each have advantages and disadvantages, some giving you more protection against style conflicts or allowing you to directly refer to props or other dynamic data. But all the options have one thing in common: They let you keep your component-specific styles close to the component, making components easier to reuse across a project or even across many unrelated projects.

Each of these options relies on CSS properties. To use plain CSS without any runtime data, you can import style sheets. If you want to create styles that are integrated with the component, you can use inline style objects that use CSS property names as keys and the style as the value. Finally, if you want a combination, you can use a third-party library such as JSS to write your CSS in JavaScript syntax, a software concept known as CSS-in-JS.

To illustrate these methods, you'll build an example `alert` component that will either show a success style or an error style depending on the prop. The `alert` component will take any number of children. This means you will need to be cautious about style conflicts, since you have no way of knowing

what styles the children components will have. After making the `alert` component, you will refactor it using each of the styling options so that you can see the similarities and differences between the approaches.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.20.1 and npm version 6.14.4. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- You will need to be able to create apps with Create React App. You can find instructions for installing an application with Create React App at How To Set Up a React Project with Create React App.

- You will be using React components, which you can learn about in our How To Create Custom Components in React tutorial.

- You will also need a basic knowledge of JavaScript, which you can find in the How To Code in JavaScript series, along with a basic knowledge of HTML and CSS. A good resource for HTML and CSS is the Mozilla Developer Network.

Step 1 — Creating an Empty Project

In this step, you'll create a new project using Create React App. Then you will delete the sample project and related files that are installed when you

bootstrap the project. Finally, you will create a simple file structure to organize your components. This will give you a solid basis on which to build this tutorial's sample application for styling in the next step.

To start, make a new project. In your terminal, run the following script to install a fresh project using `create-react-app`:

```
npx create-react-app styling-tutorial
```

After the project is finished, change into the directory:

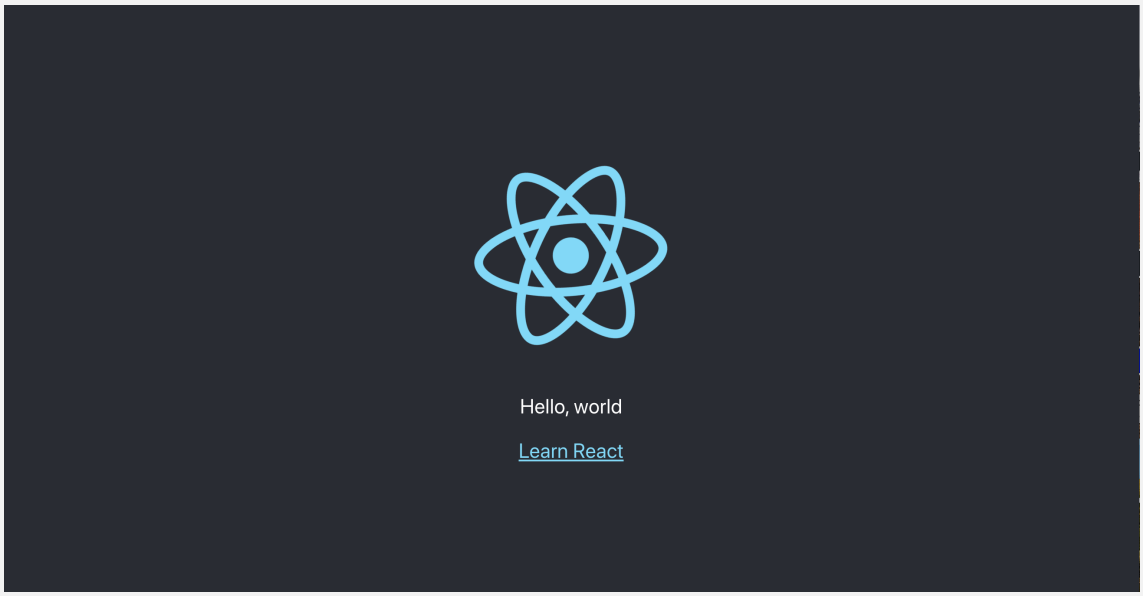
```
cd styling-tutorial
```

In a new terminal tab or window, start the project using the Create React App start script. The browser will auto-refresh on changes, so leave this script running while you work:

```
npm start
```

You will get a running local server. If the project did not open in a browser window, you can open it with `http://localhost:3000/`. If you are running this from a remote server, the address will be `http://your_domain:3000`.

Your browser will load with a simple React application included as part of Create React App:



React template project

You will be building a completely new set of custom components, so you'll need to start by clearing out some boilerplate code so that you can have an empty project.

To start, open `src/App.js` in a text editor. This is the root component that is injected into the page. All components will start from here. You can find more information about `App.js` at How To Set Up a React Project with Create React App.

Open `src/App.js` with the following command:

```
nano src/App.js
```

You will see a file like this:

styling-tutorial/src/App.js

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

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
```

```
export default App;
```

Delete the line `import logo from './logo.svg';`. Then replace everything in the `return` statement to return a set of empty tags: `<></>`. This will give you a valid page that returns nothing. The final code will look like this:

styling-tutorial/src/App.js

```
import React from 'react';
import './App.css';

function App() {
  return <></>;
}

export default App;
```

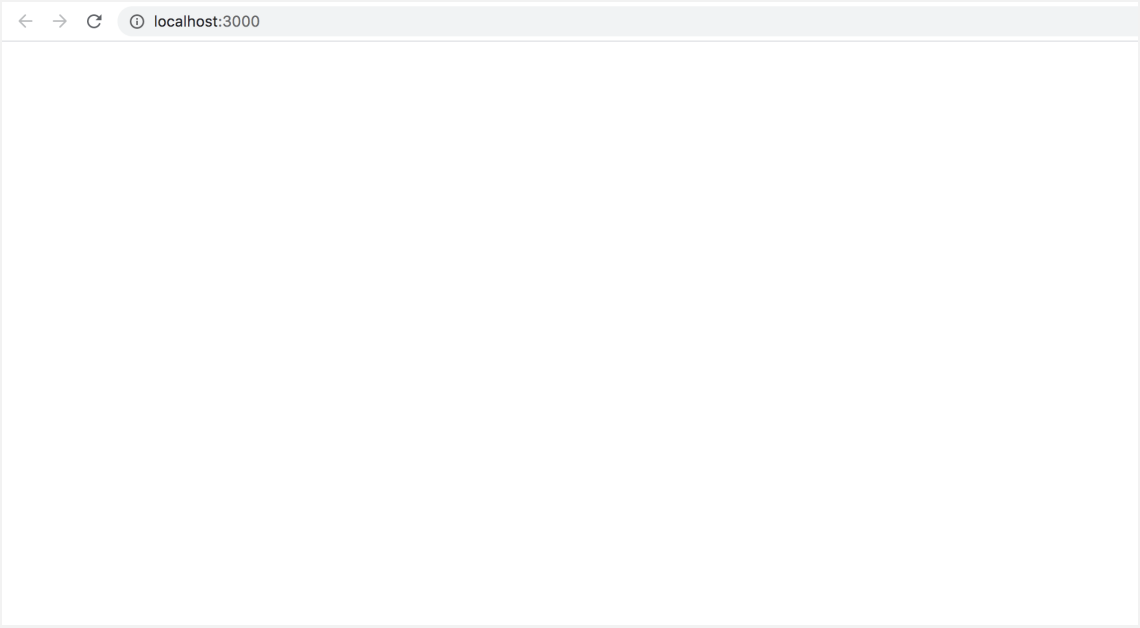
Save and exit the text editor.

Finally, delete the logo. You won't be using it in your application and you should remove unused files as you work. It will save you from confusion in the long run.

In the terminal window type the following command:

```
rm src/logo.svg
```

If you look at your browser, you will see a blank screen.



blank screen in chrome

Now that you have cleared out the sample Create React App project, create a simple file structure. This will help you keep your components isolated and independent.

Create a directory called `components` in the `src` directory. This will hold all of your custom components.

```
mkdir src/components
```

Each component will have its own directory to store the component file along with the styles, images, and tests.

Create a directory for `App`:

```
mkdir src/components/App
```

Move all of the `App` files into that directory. Use the wildcard, `*`, to select any files that start with `App.` regardless of file extension. Then use the `mv` command to put them into the new directory:

```
mv src/App.* src/components/App
```

Next, update the relative import path in `index.js`, which is the root component that bootstraps the whole process:

```
nano src/index.js
```

The import statement needs to point to the `App.js` file in the `App` directory, so make the following highlighted change:

styling-tutorial/src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './components/App/App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want your app to work offline and load faster, you ca
// unregister() to register() below. Note this comes with some
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Save and exit the file.

Now that the project is set up, you can create your first component.

Step 2 — Styling a Component with Plain CSS

In this step, you'll build a sample `Alert` component that will display an alert on a web page. You'll style this using plain CSS, which you will import directly into the component. This will ensure that the component's styles remain closely coupled with the component's JavaScript and JSX. You'll also create a component that will implement the `Alert` component to see how styles can affect children and how you can use props to change styles dynamically.

By the end of this step, you'll have created several components that use plain CSS imported directly into the component.

Building an Alert Component

To start, create a new `Alert` component. First, make the directory:

```
mkdir src/components/Alert
```

Next, open `Alert.js`:

```
nano src/components/Alert/Alert.js
```

Add a basic component that returns the string `Alert`:

styling-tutorial/src/components/Alert/Alert.js

```
import React from 'react';


export default function Alert() {
  return <div>Alert</div>
}
```

Save and close the file.

Next, open `App.js`:

```
nano src/components/App/App.js
```

Import the `Alert` component and render it inside a `<div>` by adding the highlighted code:

styling-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';

import Alert from '../Alert/Alert';

function App() {
  return (
    <div className="wrapper">
      <Alert />
    </div>
  )
}

export default App;
```

In this code, you gave the `<div>` a `className` of `wrapper`, which will be used for styling later.

Save and close the file. When you do, the browser will refresh and you'll see your component:

Browser with Alert

Next, you will style the `App` component to give it some padding so that the `Alert` component is not so close to the edge. Open the `App.css` file:

```
nano src/components/App/App.css
```

This file uses standard CSS. To add padding to the wrapper, replace the default code with a rule as you would for CSS in a plain HTML project. In this case, add a `padding` of `20px`:

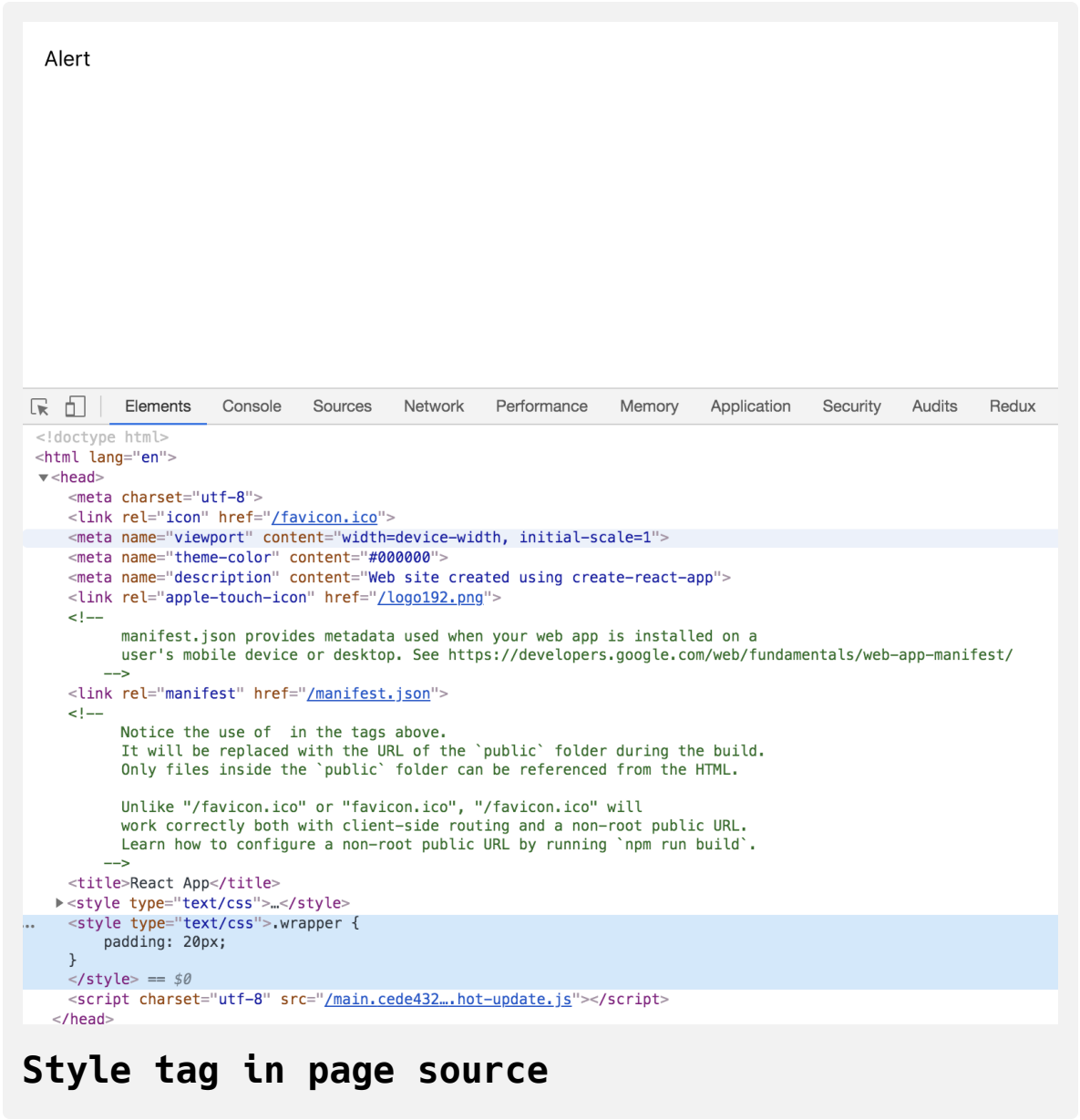
styling-tutorial/src/components/App/App.css

```
.wrapper {
    padding: 20px;
}
```

Save and close the file. When you do, the page will refresh and you'll see the extra padding:

Browser with extra padding

When you use Create React App, webpack will take the imported CSS and add it to a style tag at the top of the file rendered in the browser. If you look at the `<head>` element in your page source, you'll see the styles:



Style tag in page source

This means that you can keep the CSS alongside the component and it will be collected together during the build phase. It also means that your styles are global in scope, which can create potential name conflicts. With this method, each class name will need to be unique across all components.

To explore this problem, you will make some changes to the `Alert` component.

First, open the file:

```
nano src/components/Alert/Alert.js
```

Then add some React code that will take `children`, `type`, and `title` as props:

styling-tutorial/src/components/Alert/Alert.js

```
import React from 'react';
import PropTypes from 'prop-types';

export default function Alert({ children, title, type }) {
  return (
    <div>
      <h2>{title}</h2>
      {children}
    </div>
  )
}

Alert.propTypes = {
  children: PropTypes.oneOfType([
    PropTypes.arrayOf(PropTypes.element),
    PropTypes.element.isRequired
  ]),
  title: PropTypes.string.isRequired,
  type: PropTypes.string.isRequired,
}
```

The `title` in this code is in a `<h2>` tag, and `children` will allow you to display child components. You will soon use the `type` prop to set a success

and an error alert based on the `PropTypes` typing system.

Save and close the file. Next, update the `Alert` component in `App` to use the new props.

First, open `App.js`:

```
nano src/components/App/App.js
```

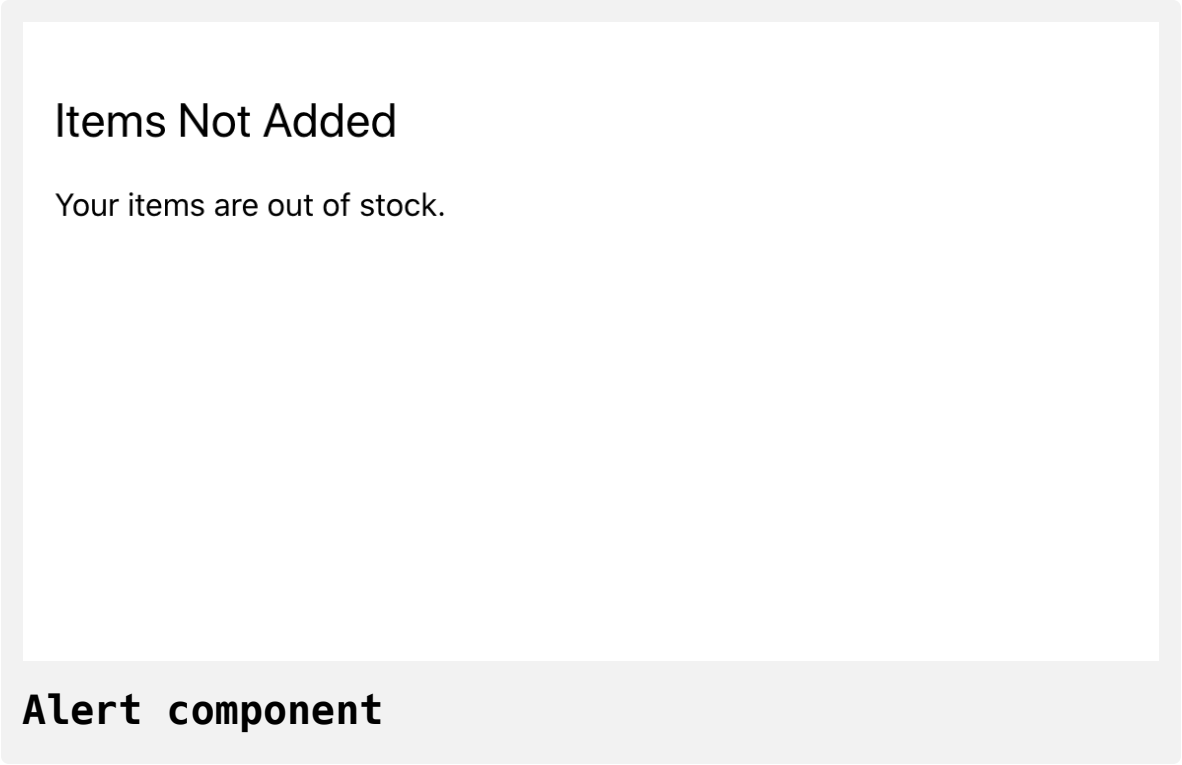
Make an alert that notifies a user that an attempt to add items to a shopping cart has failed:

styling-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';

import Alert from '../Alert/Alert';

function App() {
  return (
    <div className="wrapper">
      <Alert title="Items Not Added" type="error">
        <div>Your items are out of stock.</div>
      </Alert>
    </div>
  )
}

export default App;
```

In this code, you updated the `title` and `children` with a fail message, then added a `type` of `error`.

Save and close the file. When you do, the browser will refresh and you'll see your new component:



Alert component

Your alert is rendering, so the next step is to style the component with CSS.

Adding CSS to the `Alert` Component

Since the `Alert` component dislays an error, you will add a border and set the color of the border to a shade of red. You'll also give the `<h2>` tag the same color. But this presents a problem: You can't use the name `wrapper` on the outer `<div>` in your `Alert` component, because that name is already taken by the `App` component.

Class name conflicts aren't a new problem in CSS, and there have been a number of attempts to solve it using naming conventions such as BEM. But naming conventions can become verbose, and can still occasionally lead to conflicts in projects with a large number of components.

Rather than using a specific set of rules separated by naming convention, in this tutorial you will prefix the `wrapper` class name with the name of the component. Your new class name will be `alert-wrapper`. In addition, you will add the `type` of the alert as a class.

Open up the `Alert` component:

```
nano src/components/Alert/Alert.js
```

Next, add the following highlighted code:

styling-tutorial/src/components/Alert/Alert.js

```
import React from 'react';
import PropTypes from 'prop-types';
import './Alert.css';
...
export default function Alert({ children, type, title }) {
  return(
    <div className={`alert-wrapper ${type}`}>
      <h2>{title}</h2>
      {children}
    </div>
  )
}
...
```

In this case, you're combining `alert-wrapper` and the `type` variable into a single string using a template literal.

Save and close the file. Now you have a unique class name that changes dynamically based on the prop. The JSX in this code will resolve to a `div` with the class names of `alert-wrapper` and `error`. The compiled mark up would be this: `<div class="alert-wrapper error">`.

Now add the styles. First, open the CSS for the `Alert` component:

```
nano src/components/Alert/Alert.css
```

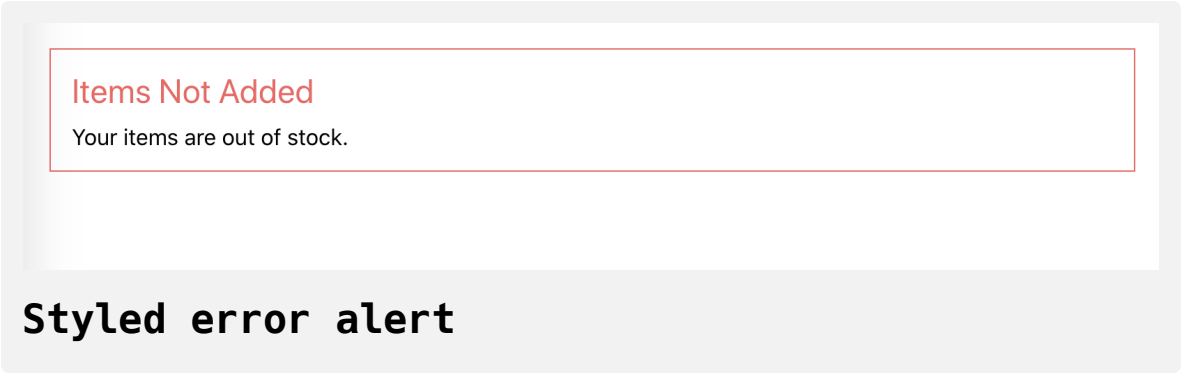
Add the following CSS to the `alert-wrapper`, `success`, and `error` classes:

styling-tutorial/src/components/Alert/Alert.css

```
.alert-wrapper {
    padding: 15px;
    margin-bottom: 15px;
}

.alert-wrapper h2 {
    margin: 0 0 10px 0;
}

.alert-wrapper.success {
    border: #6DA06F solid 1px;
}

.success h2 {
    color: #6DA06F;
}

.alert-wrapper.error {
    border: #F56260 solid 1px;
}

.error h2 {
    color: #F56260;
}
```

This code adds some margins and padding to the `alert-wrapper`. It then adds a border with a shade of red for the `error` class using the hexidecimal color code `#F56260`, and a shade of green (`#6DA06F`) for the `success` class. It also updates the `<h2>` color to red or green depending on the parent.

Save and close the file. When you do, the browser will refresh and you'll see the new styles:



Styled error alert

Now that you have a styled `Alert` component, you can create a new component that displays a list of items within the `Alert` component. Since the children will be more complex, there will be greater possibilities for style conflicts.

Creating a Success Message Component

First, create a directory for for the new component `CartSuccess`:

```
mkdir src/components/CartSuccess
```

Open `CartSuccess.js`:

```
nano src/components/CartSuccess/CartSuccess.js
```

Inside the component, import the `Alert` component and pass a `<div>` containing a series of items that a user has added to the cart:

styling-tutorial/src/components/CartSuccess/CartSuccess.js

```
import React from 'react';
import Alert from '../Alert/Alert';
import './CartSuccess.css';

export default function CartSuccess() {
  return(
    <Alert title="Added to Cart" type="success">
      <div className="cart-success-wrapper">
          <h2>
            You have added 3 items:
          </h2>
          <div className="item">
            <div>Bananas</div>
            <div>Quantity: 2</div>
          </div>
          <div className="item">
            <div>Lettuce</div>
            <div>Quantity: 1</div>
          </div>
      </div>
    </Alert>
  )
}
```

Notice how you needed to create a unique class name—`cart-success-wrapper`—for the outer `<div>`. Save and close the file.

Next, add some CSS to the custom message. Open `CartSuccess.css`:

```
nano src/components/CartSuccess/CartSuccess.css
```

Add a `display` of `flex` to the wrapper. You'll want most of the items to wrap, except for the `<h2>` element, which should take up the whole width:

styling-tutorial/src/components/CartSuccess/CartSuccess.css

```
.cart-success-wrapper {
    border-top: black solid 1px;
    display: flex;
    flex-wrap: wrap;
}

.cart-success-wrapper h2 {
    width: 100%;
}

.item {
    margin-right: 20px;
}
```

Here, you gave the `<h2>` a width of `100%`. In addition to flexing the element, you also added a small border to the top of the message, and added a margin to the `item` class to provide some space between items.

Save and close the file.

Now that you have a styled component, add it to your `App` component.

Open `App.js`:

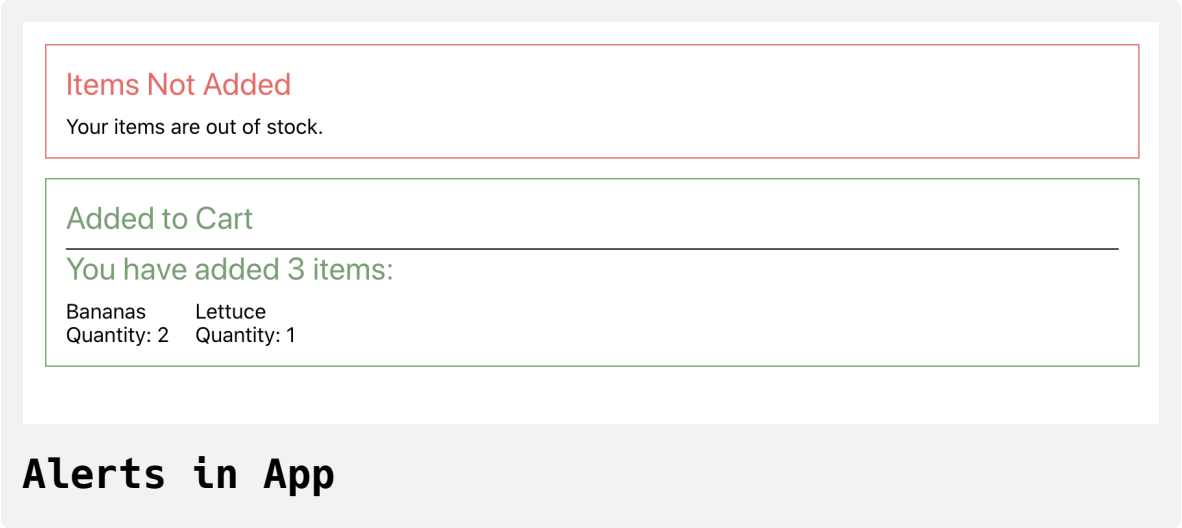
```
nano src/components/App/App.js
```

Import the component and add it after the current `Alert` component, as shown in the highlighted code:

styling-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';

import Alert from '../Alert/Alert';
import CartSuccess from '../CartSuccess/CartSuccess';

function App() {
  return(
    <div className="wrapper">
      <Alert title="Items Not Added" type="error">
        <div>Your items are out of stock.</div>
      </Alert>
      <CartSuccess />
    </div>
  )
}

export default App;
```

Save and close the file. When you do, the browser will refresh and you'll see your new component:



Alerts in App

This shows the new color and the message as intended, but the nested component received some unexpected styles. The rule for the `<h2>` in the `Alert` component is being applied to the nested `<h2>` tag in the `children` props.

Unexpected styles cascading to children are a common problem with CSS. However, since React gives you the opportunity to bundle and share components across projects, you have a greater potential for styles to inadvertently flow down to children components.

To fix this with pure CSS, make the `<h2>` rule for the `Alert` component a little more specific.

Open the `Alert.css` file:

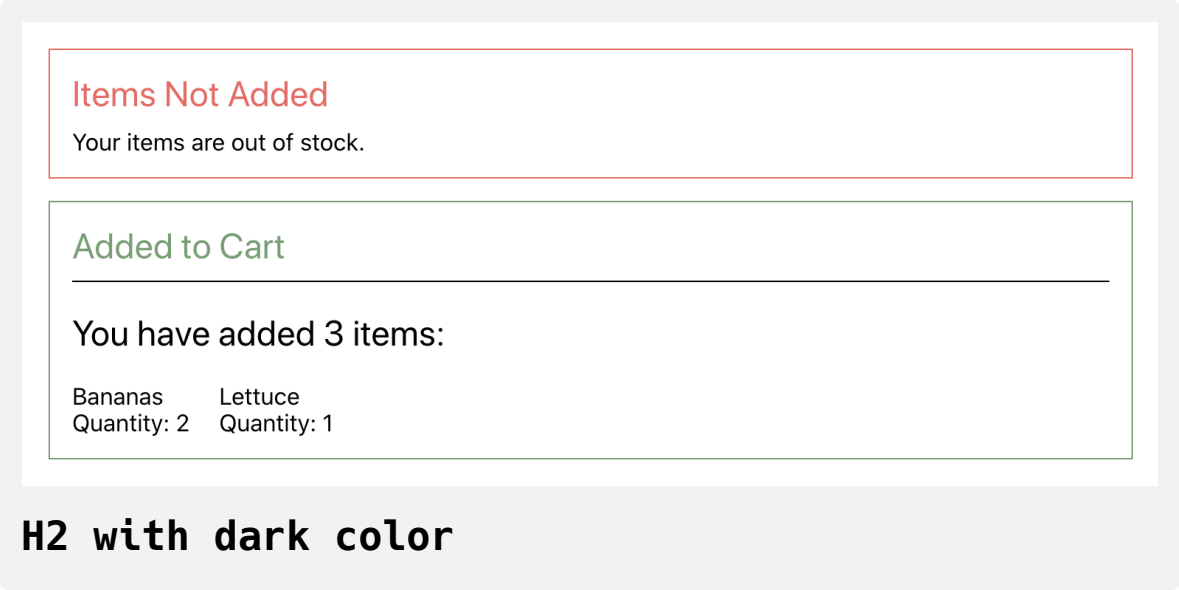
```
nano src/components/Alert/Alert.css
```

Change the rules so that the `<h2>` styling only applies to the direct children of the classes rather than all children using the CSS `>` child combinator:

styling-tutorial/src/components/Alert/Alert.css

```css
.alert-wrapper {
    padding: 15px;
    margin-bottom: 15px;
}

.alert-wrapper > h2 {
    margin: 0 0 10px 0;
}

.alert-wrapper.success {
    border: #6da06f solid 1px;
}

.success > h2 {
    color: #6da06f;
}

.alert-wrapper.error {
    border: #f56260 solid 1px;
}

.error > h2 {
    color: #f56260;
}
```

Save and close the file. When you do, the page will refresh and you'll see the `<h2>` element in `CartSuccess` retain the default color:



Now the styles for the `Alert` component will only affect the immediate children and will not apply to other child nodes or components. This method works well in this case, but in circumstances where components are more complex, it can be difficult to write rules that apply to all cases without leaking outside the component.

In this step, you styled a component using CSS stylesheets imported directly into a component. Styling React elements with standard CSS is a quick way to create components with associated styles using standard CSS files. The ease of use makes it a good first step when you are working on new or small projects, but as the projects grow it can cause problems.

As you built the components, you encountered two common styling problems: class name conflicts and unintended style application. You can

work around them with standard CSS, but there are other styling approaches that give you tools for handling these problems programmatically instead of with naming conventions. In the next step, you will explore solving these problems with style objects.

Step 3 — Styling with Style Objects

In this step, you'll style your components using style objects, which are JavaScript objects that use CSS properties as keys. As you work on your components, you'll update keys to match the JavaScript syntax and learn how to dynamically set style properties based on component props.

Separate CSS is the most common way to style HTML. This method is fast, and browsers are efficient at applying styles quickly and consistently. But this is not the only option for styling HTML. In standard HTML, you can set inline styles directly on an element using the style attribute with a string containing the styles you wanted to apply.

One of the best uses of style objects is for calculating styles dynamically. This is particularly useful if you need to know the element's current position, since this is not determined until the elements are rendered and thus can only be handled dynamically.

Writing style strings manually is difficult to do and can introduce bugs. A missing color or semicolon will break the entire string. Fortunately, in JSX, you aren't limited to just a string. The style attribute can also accept an object containing the styles. These style names will need to be `camelCase` rather than `kebab-case`.

The biggest advantage to using inline styles like this is that, since you are building styles with JavaScript, you can dynamically set CSS properties instead of dynamically setting classes. This means you can write code without CSS classes at all, avoiding any potential name conflicts and allowing you to calculate styles at runtime.

To use style objects, start by refactoring `App.js`. First, open the file:

```
nano src/components/App/App.js
```

Inside the component, remove the imported `App.css` file, and then create an object that has a `padding` of `20` and pass the object to the `<div>` using the `style` attribute:

styling-tutorial/src/components/App/App.js

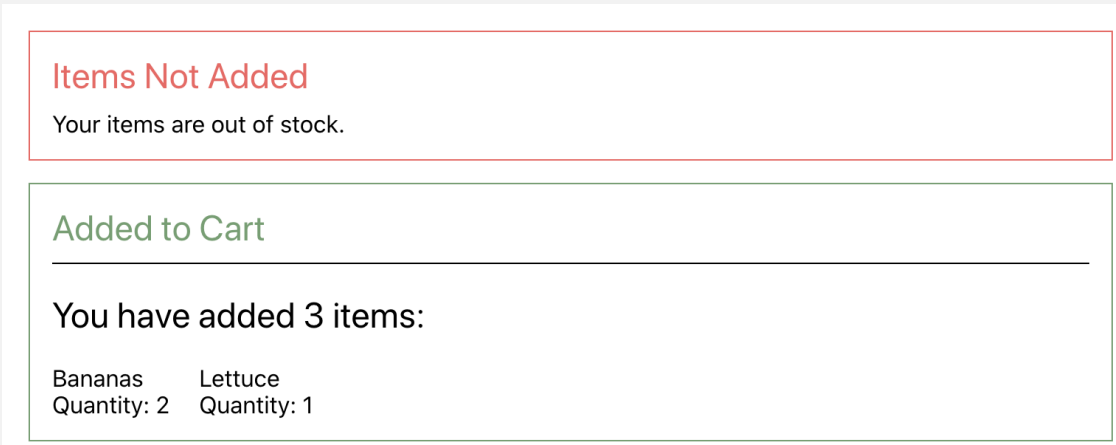
```
import React from 'react';

import Alert from '../Alert/Alert';
import CartSuccess from '../CartSuccess/CartSuccess';

function App() {
  const wrapper = {
    padding: 20
  };

  return(
    <div style={wrapper}>
      <Alert title="Items Not Added" type="error">
        <div>Your items are out of stock.</div>
      </Alert>
      <CartSuccess />
    </div>
  )
}

export default App;
```

Notice that you do not have to specify pixels as the unit for `padding`. React will convert this to a string of pixels by default. If you want a specific unit,

pass it as a string. So if you wanted the padding to be a percentage for example, it would be `padding: '20%'`.

Most numbers will be automatically converted to pixels. There are exceptions, however. The property `line-height` can take plain numbers without a unit. If you wanted to use the pixels unit in that case, you'd need to specify pixels as a string.

Save and close the file. When you do, the browser will refresh and you'll see the page as it was before:



Page with style object

Next, refactor `CartSuccess`. First, open the file:

```
nano src/components/CartSuccess/CartSuccess.js
```

As with `App.js`, remove the imported CSS (`CartSuccess.css`) and create a style object for each item that previously had a class:

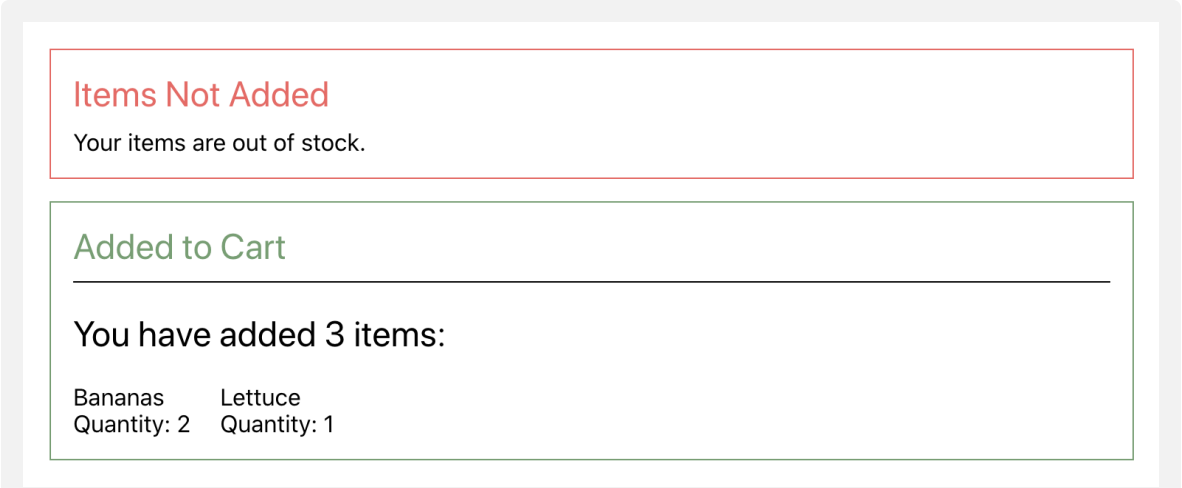
styling-tutorial/src/components/CartSuccess/CartSuccess.js

```
import React from 'react';
import Alert from '../Alert/Alert';

export default function CartSuccess() {
  const styles = {
    header: {
      width: '100%'
    },
    item: {
      marginRight: 20
    },
    wrapper: {
      borderTop: 'black solid 1px',
      display: 'flex',
      flexWrap: 'wrap'
    }
  }

  return(
    <Alert title="Added to Cart" type="success">
      <div style={styles.wrapper}>
          <h2 style={styles.header}>
            You have added 3 items:
          </h2>
```

```
          <div style={styles.item}>
            <div>Bananas</div>
            <div>Quantity: 2</div>
          </div>
          <div style={styles.item}>
            <div>Lettuce</div>
            <div>Quantity: 1</div>
          </div>
      </div>
    </Alert>
  )
}
```

In this case, you didn't create multiple, separate objects; instead, you created a single object that contains other objects. Notice also that you needed to use camel case for the properties of `margin-right`, `border-top`, and `flex-wrap`.

Save and close the file. When you do, the page will refresh and you'll see the page with the same styles:



Page with style object

Since you are not using classes, you don't have to worry about any name conflicts. Another advantage of creating styles with JavaScript is that you can take advantage of any JavaScript syntax such as variables and template literals. With modern CSS, you can use variables, which is a major improvement, but may not be fully available depending on your browser support requirements. In particular, they are not supported in any version of Internet Explorer, although you can use a polyfill to add support.

Since style objects are created at runtime, they are more predictable and can use any supported JavaScript.

To see how style objects can help in this situation, refactor `Alert.js` to use style objects. First, open `Alert.js`:

```
nano src/components/Alert/Alert.js
```

Inside `Alert.js`, remove `import './Alert.css';` and create an object called `colors` that has a property for the error color and a property for the success color. Then convert the CSS to a JavaScript object using the `type` prop to dynamically set the color:

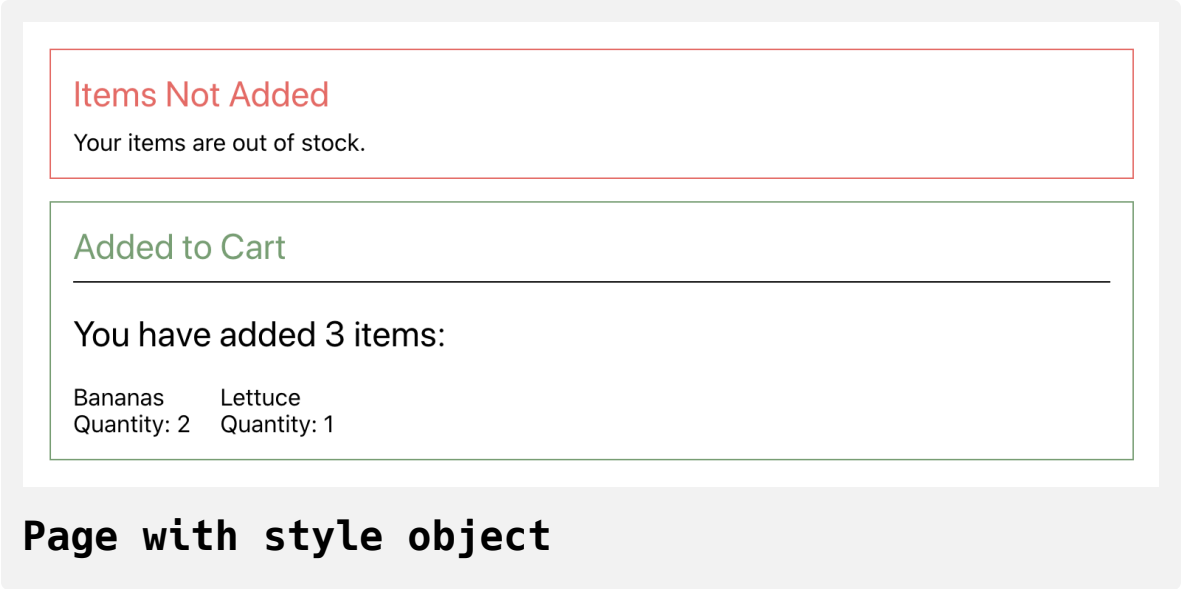
styling-tutorial/src/components/Alert/Alert.js

```
import React from 'react';
import PropTypes from 'prop-types';

export default function Alert({ children, type, title }) {
  const colors = {
    success: '#6da06f',
    error: '#f56260',
  }

  const style = {
    heading: {
      color: colors[type],
      margin: '0 0 10px 0',
    },
    wrapper: {
      border: `${colors[type]} solid 1px`,
      marginBottom: 15,
      padding: 15,
      position: 'relative',
    }
  }

  return(
    <div style={style.wrapper}>
      <h2 style={style.heading}>{title}</h2>
```

```
      {children}
    </div>
  )
}


...
```

There are a few changes here. First, you use a single style declaration for `wrapper` and then dynamically set the color based on the type. You are not styling `<h2>` elements in general, but instead are styling these particular elements, which happen to be `<h2>` elements. Since you are not applying styles to an element type, there is no danger that the styles will flow down to child elements.

Save and close the file. When you do, the browser will refresh and you'll see the applied styles.



Page with style object

Style objects solve many problems, but they do have disadvantages. First, there is a performance cost for inline styles. Browsers were designed to handle CSS efficiently, and styles objects that apply inline styles can not take advantage of these optimizations. The other problem is that it's more difficult to apply styles to child elements with style objects. In this case, you did not want a style to apply to children, but it is often the case that you do want styles to cascade. For example, setting a custom font family on every element or applying a custom size to every `<h2>` element would be easier if you used a less specific styling strategy.

There is, however, a middle ground between these approaches. Several third-party libraries are designed to find this middle ground. In the next step, you'll create styles using a hybrid approach called CSS-in-JS using a library called JSS.

Step 4 — Styling with JSS

In this step, you'll style objects using the popular library JSS. You'll install the new library and convert your style objects to JSS objects. You'll then refactor your code to use dynamically generated class names that will prevent conflicts between class names across modules. You'll also build JavaScript style objects that dynamically set styles and use nested properties to create specific style rules.

JSS is a library for creating CSS-in-JS. This methodology has many different use cases and options, but the main advantage in this tutorial is that it will create dynamic class names that avoid conflicts between components. You also will be able to take advantage of JavaScript syntax, which means you will be able to use variables and create styles based off of React props.

To begin, install the React specific version of JSS. This tutorial will use version `10.1.1`:

```
npm install react-jss
```

The package will install several dependencies, including a number of JSS plugins that will give you the ability to write concise style rules.

When the installation is complete, you'll see a success message:

Output

```
+ react-jss@10.1.1
added 281 packages from 178 contributors, removed 142 package
s, updated 1392 packages and audited 1025251 packages in 144.8
72s
```

Your output will vary slightly depending on your Node version and other dependencies.

Now that the library is installed, convert `App.js` to use JSS. First, open `App.js`:

```
nano src/components/App/App.js
```

There are two steps to use JSS. First, you have to import a function to create a custom hook. Hooks are functions that React will run on every component render. With JSS, you have to create a hook by passing in the style definitions, outside of the component. This will prevent the code from running on every re-render; since the style definitions are static, there's no reason to run the code more then once.

Create the hook and the style object by making the highlighted changes:

styling-tutorial/src/components/App/App.js

```
import React from 'react';
import { createUseStyles } from 'react-jss';

import Alert from '../Alert/Alert';
import CartSuccess from '../CartSuccess/CartSuccess';

const useStyles = createUseStyles({
  wrapper: {
    padding: 20,
  }
});

function App() {
  return(
    <div>
      <Alert title="Items Not Added" type="error">
        <div>Your items are out of stock.</div>
      </Alert>
      <CartSuccess />
    </div>
  )
}

export default App;
```

Notice in this case that your style object contains another object called `wrapper`, which contains the styles using the same camel case format. The name of the object—`wrapper`—is the basis for creating the dynamic class name.

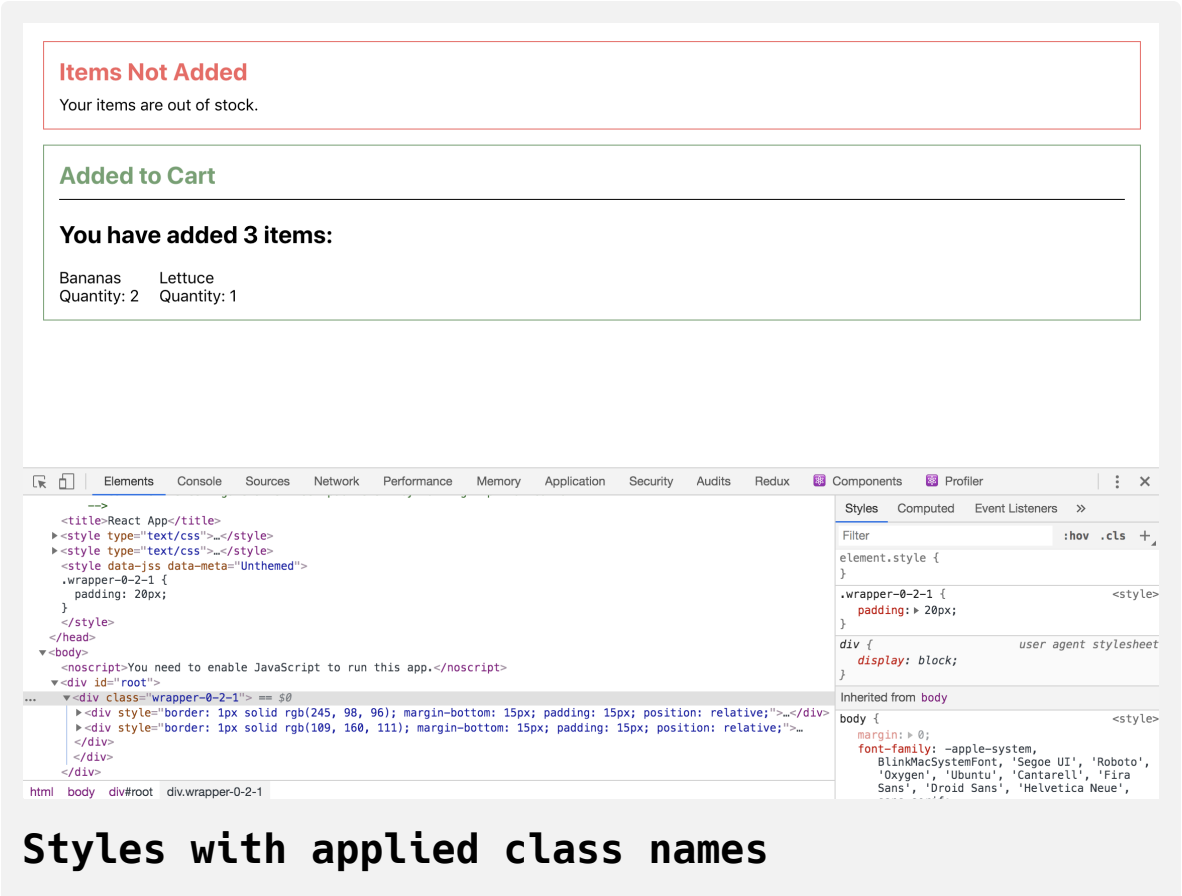
After you create the hook, you consume it by executing the function inside the component. This registers the hook and creates the styles dynamically. Make the following highlighted change:

styling-tutorial/src/components/App/App.js

```
import React from 'react';
import { createUseStyles } from 'react-jss'

import Alert from '../Alert/Alert';
import CartSuccess from '../CartSuccess/CartSuccess';

const useStyles = createUseStyles({
  wrapper: {
    padding: 20,
  }
});

function App() {
  const classes = useStyles()
  return(
    <div className={classes.wrapper}>
      <Alert title="Items Not Added" type="error">
        <div>Your items are out of stock.</div>
      </Alert>
      <CartSuccess />
    </div>
  )
}

export default App;
```

In this code, you call the function and assign the results to a variable called `classes`. The new variable `classes` will be an object that contains the dynamic class names. You then apply the appropriate class name to your element by using the same name that you defined on the object. Here you used `classes.wrapper`.

Save and close the file. When you do the browser will refresh and you'll see the same styles as before. But if you look at the console, you'll see that the class name does not perfectly match the one you defined in your object:



Styles with applied class names

In this case, the class name is `wrapper-0-2-1`, but your class name may be different. You'll also see that the styles are converted from an object to CSS and placed in a `<style>` tag. Contrast this to the inline styles, which are not converted to CSS and do not have any class names.

JSS creates the class names dynamically so they will not conflict with similar names in other files. To see this at work, refactor `CartSuccess.js` to use JSS styling.

Open the file:

```
nano src/components/CartSuccess/CartSuccess.js
```

Inside the file, create a custom hook using `createUseStyles`. Instead of applying a class to the `<h2>` element, you'll create a rule for the `<h2>` elements inside of a wrapper. To do that with plain CSS, you add a space between the class and the element—`.wrapper h2`. This applies the style to all `<h2>` elements that are children of the `.wrapper` class.

With JSS, you can create a similar rule by creating another object inside of the containing element. To link them up, start the object name with the `&` symbol:

styling-tutorial/src/components/CartSuccess/CartSuccess.js

```
import React from 'react';
import { createUseStyles } from 'react-jss';
import Alert from '../Alert/Alert';

const useStyles = createUseStyles({
  item: {
    marginRight: 20
  },
  wrapper: {
    borderTop: 'black solid 1px',
    display: 'flex',
    flexWrap: 'wrap',
    '& h2': {
      width: '100%'
    }
  }
})

export default function CartSuccess() {
  const classes = useStyles();
  return(
    <Alert title="Added to Cart" type="success">
      <div className={classes.wrapper}>
          <h2>
```

```
            You have added 3 items:
          </h2>
          <div className={classes.item}>
            <div>Bananas</div>
            <div>Quantity: 2</div>
          </div>
          <div className={classes.item}>
            <div>Lettuce</div>
            <div>Quantity: 1</div>
          </div>
      </div>
    </Alert>
  )
}
```

In addition to creating rules for the wrapper, you also created a rule for `item`. After creating the custom hook, you passed the custom class names to the `className` property.

Save the file. Notice that you are using the same name—`wrapper`—in both this component and the `App` component. But when the browser reloads, there will be no naming conflict; everything will look correct. If you inspect the elements in your browser, you'll see that even though they started with the same name, they each have a unique class:

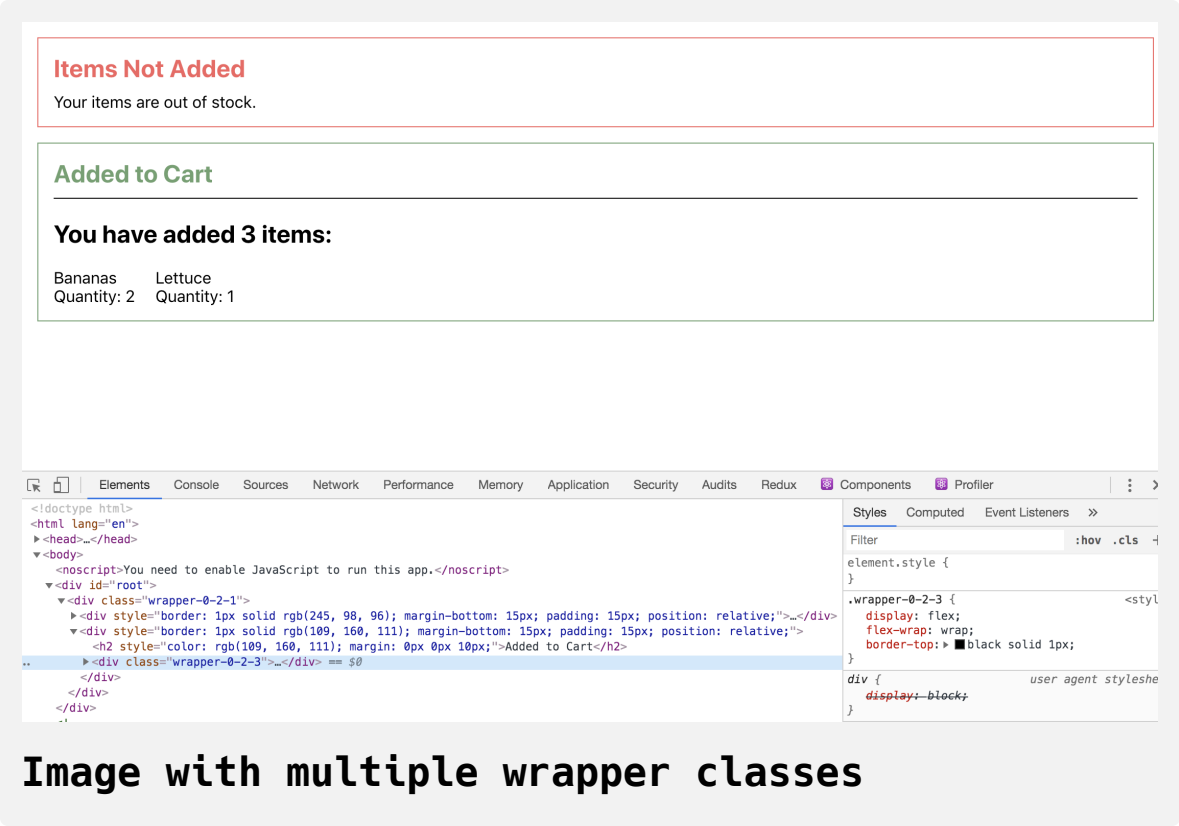


Image with multiple wrapper classes

In this case, the class for the outer component is `wrapper-0-2-1`, which was generated in the `App` component. The class for `CartSuccess` is `wrapper-0-2-3`. Your component names may be slightly different, but they will be unique.

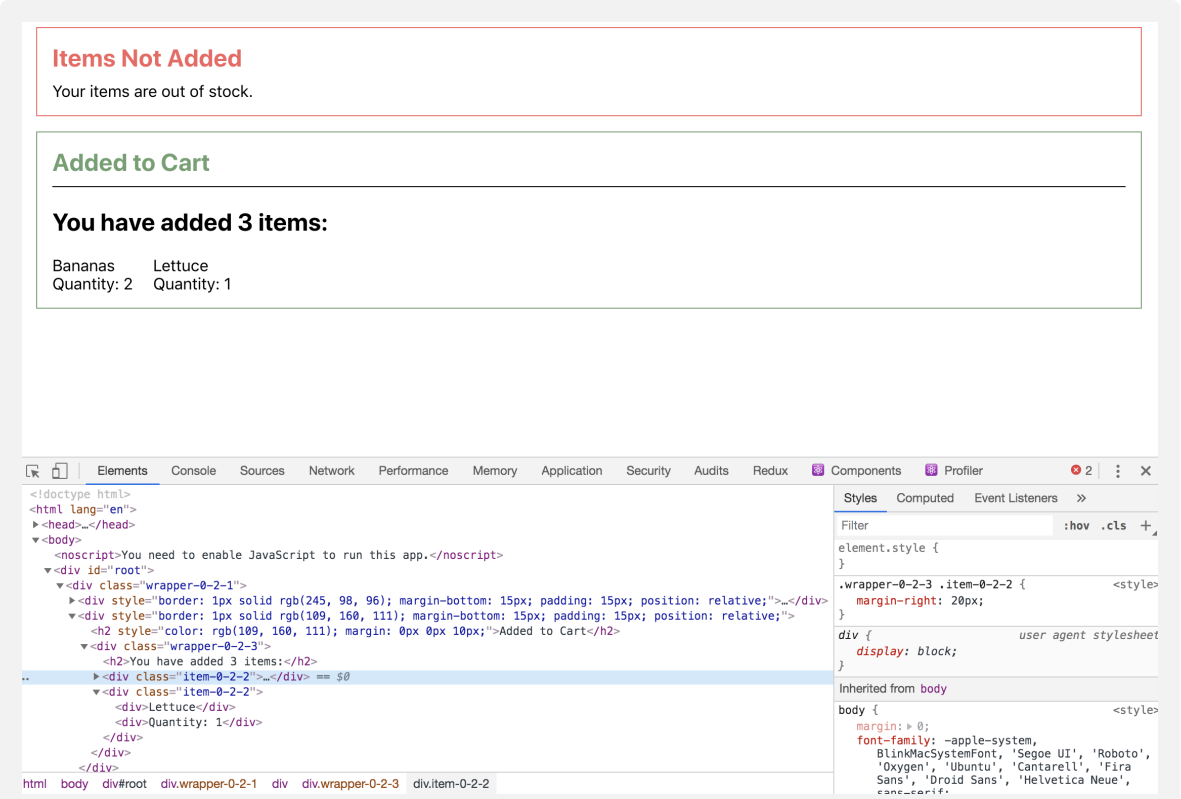
In some situations, you may need to make a specific selector to override other styles. For example, let's say you only want the `item` styling to apply when the element is a child of the `wrapper` class. To do this, first create the class on the object with no properties. Then inside the `wrapper` class, reference the new class with a `$` symbol:

styling-tutorial/src/components/CartSuccess/CartSuccess.js

```
import React from 'react';
import { createUseStyles } from 'react-jss'
import Alert from '../Alert/Alert';

const useStyles = createUseStyles({
  item: {},
  wrapper: {
    borderTop: 'black solid 1px',
    display: 'flex',
    flexWrap: 'wrap',
    '& h2': {
      width: '100%'
    },
    '& $item': {
      marginRight: 20
    }
  }
})

export default function CartSuccess() {
  const classes = useStyles()
  return(
    <Alert title="Added to Cart" type="success">
      <div className={classes.wrapper}>
```

```
          <h2>
            You have added 3 items:
          </h2>
          <div className={classes.item}>
            <div>Bananas</div>
            <div>Quantity: 2</div>
          </div>
          <div className={classes.item}>
            <div>Lettuce</div>
            <div>Quantity: 1</div>
          </div>
      </div>
    </Alert>
  )
}
```

Save and close the file. When the browser reloads, the page will look the same, but the `item` CSS will be applied more specifically to items under the wrapper component:



Item class applied

JSS gives you the ability to create rules with the same level of focus that you'd create with regular CSS, but will do so while creating unique class names that won't clash.

One final advantage of JSS is that you have the ability to use variables and other JavaScript language features. Since you are using `react-jss`, you can pass props to the style object to create dynamic styles. To test this out, refactor the `Alert.js` component to use props and variables to create dynamic properties.

First, open the file:

```
nano src/components/Alert/Alert.js
```

Create a style object like you did in the last refactored code. Be sure to move the object defining the colors outside of the component function so it is in the same scope as the `createUseStyles` function:

styling-tutorial/src/components/Alert/Alert.js

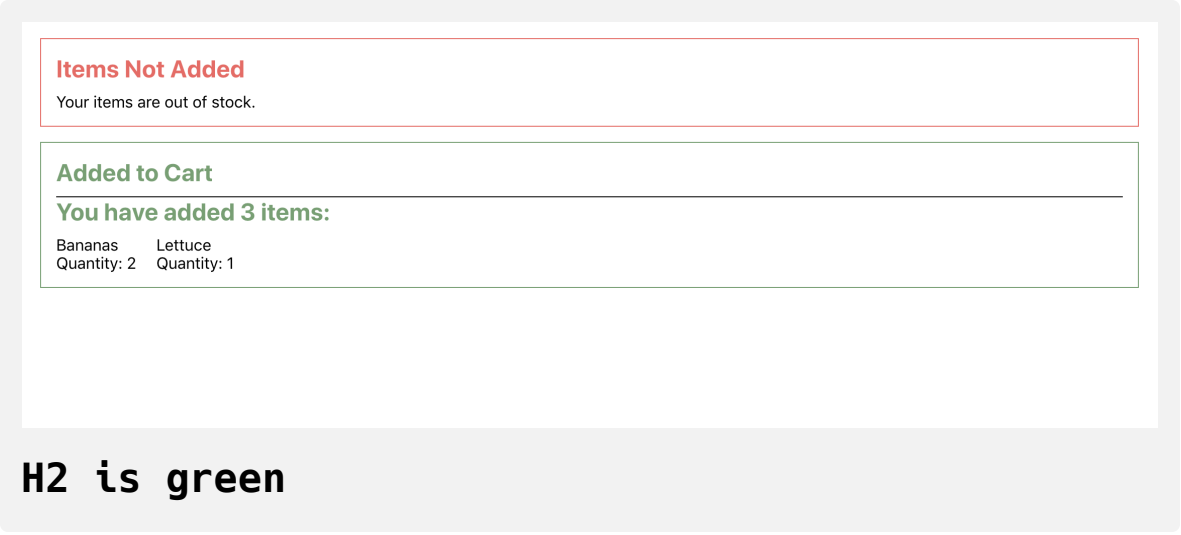
```
import React from 'react';
import PropTypes from 'prop-types';
import { createUseStyles } from 'react-jss';

const colors = {
  success: '#6da06f',
  error: '#f56260',
};

const useStyles = createUseStyles({
    wrapper: {
      border: ({ type }) => `${colors[type]} solid 1px`,
      marginBottom: 15,
      padding: 15,
      position: 'relative',
      '& h2': {
        color: ({ type }) => colors[type],
        margin: [0, 0, 10, 0],
      }
    }
});

export default function Alert({ children, type, title }) {
  const classes = useStyles({ type })
  return(
```

```
    <div className={classes.wrapper}>
      <h2>{title}</h2>
      {children}
    </div>
  )
}

...
```

To pass props, you make the style rule a function. The function accepts the props as an argument then returns a rule. To create a dynamic border, you add `border` as the property name and an arrow function that takes `type` and returns a string: `({ type }) => `${colors[type]} solid 1px`,`. Then after you create your hook, you pass in the props you want to reference when creating the classes object. As before, you style the `<h2>` tag by element instead of creating a specific class. You also pass an array of values for `margin` rather than a string such as `0px 0px 10px 10px`.

Save the file. Notice that you don't have to pass all the props into the function. In this case, you only want to use `type`, so that's all you need to pass. However, you can pass more or even pass unknown props using the rest operator to collect props and then pass them as a group. You do need to pass it as an object; however, since that's the standard way to pass props, it will make extending the arguments easier in the future.

When the page reloads, you'll see the correct colors, but there will be a slight problem: the green success color is now updating the `<h2>` element in `CartSuccess`:

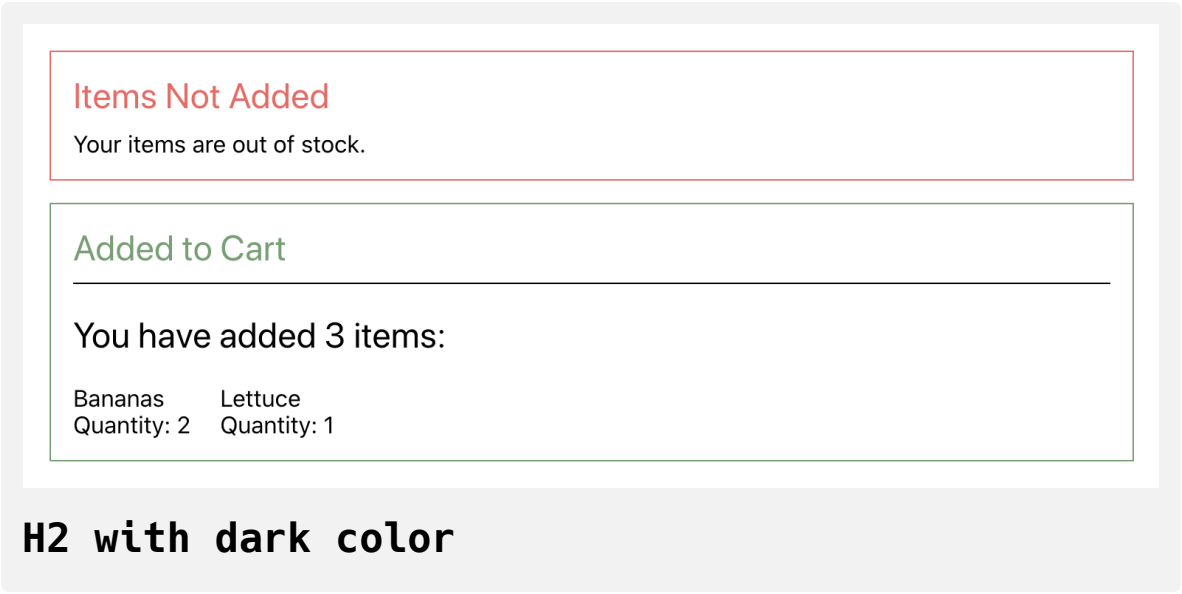


JSS solves many problems, but it still creates standard CSS. That means that styles can apply to child elements if you are not careful. To fix this, add the `>` symbol to make the CSS only apply to immediate children:

styling-tutorial/src/components/Alert/Alert.js

```
import React from 'react';
...
const useStyles = createUseStyles({
    wrapper: {
      border: ({ type }) => `${colors[type]} solid 1px`,
      marginBottom: 15,
      padding: 15,
      position: 'relative',
      '& > h2': {
        color: ({ type }) => colors[type],
        margin: [0, 0, 10, 0],
      }
    }
});

export default function Alert({ children, type, title }) {

...

}

...
```

Save and close the file. When you do the browser will reload and you'll see the correct styles:



There is much more to JSS beyond what is covered in this tutorial. One important advantage that we haven't touched on is theming. JSS gives you the ability to create styles based off of pre-defined theme objects. That means that instead of creating a color red from a hard coded value, you can make the alert border the `alert` color, which will likely be a shade of red, but could be different depending on the theme definition. This is useful when creating white label products or creating reusable components that need to work across projects.

In this step, you styled components using a third-party library called `react-jss`. You also created style object and used JSS to convert those objects into dynamic classes to avoid conflicting with other components. Using this method, you can safely reuse simple class names without worrying about

conflicts later in the code. Finally, you learned how to create styles using functions and props to build dynamic styles that reference component props.

Conclusion

Throughout this tutorial, you have developed several reusable components that use different style techniques. You've learned how style objects and JSS create objects using names that closely mirror standard CSS properties, and have created components that can dynamically set styles based on incoming properties. You also learned how different approaches provide different options for handling name conflicts and reusability.

As with most React techniques, there is no single best solution. Instead, you can choose the styling option that is the best fit for your project. With these options in hand, you can start with something simple and refactor as the project grows or the requirements change, while remaining confident that your components will continue to meet your style goals.

If you would like to look at more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Manage State on React Class Components

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

In React, state refers to a structure that keeps track of how data changes over time in your application. Managing state is a crucial skill in React because it allows you to make interactive components and dynamic web applications. State is used for everything from tracking form inputs to capturing dynamic data from an API. In this tutorial, you'll run through an example of managing state on class-based components.

As of the writing of this tutorial, the official React documentation encourages developers to adopt React Hooks to manage state with functional components when writing new code, rather than using class-based components. Although the use of React Hooks is considered a more modern practice, it's important to understand how to manage state on class-based components as well. Learning the concepts behind state management will help you navigate and troubleshoot class-based state management in existing code bases and help you decide when class-based state management is more appropriate. There's also a class-based method called `componentDidCatch` that is not available in Hooks and will require setting state using class methods.

This tutorial will first show you how to set state using a static value, which is useful for cases where the next state does not depend on the first state, such as setting data from an API that overrides old values. Then it will run through how to set a state as the current state, which is useful when the next state depends on the current state, such as toggling a value. To explore these different ways of setting state, you'll create a product page component that you'll update by adding purchases from a list of options.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.20.1 and npm version 6.14.4. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- In this tutorial, you will create apps with Create React App. You can find instructions for installing an application with Create React App at How To Set Up a React Project with Create React App.

- You will also need a basic knowledge of JavaScript, which you can find in How To Code in JavaScript, along with a basic knowledge of HTML and CSS. A good resource for HTML and CSS is the Mozilla Developer Network.

Step 1 — Creating an Empty Project

In this step, you'll create a new project using Create React App. Then you will delete the sample project and related files that are installed when you bootstrap the project. Finally, you will create a simple file structure to organize your components. This will give you a solid basis on which to build this tutorial's sample application for managing state on class-based components.

To start, make a new project. In your terminal, run the following script to install a fresh project using `create-react-app`:

```
npx create-react-app state-class-tutorial
```

After the project is finished, change into the directory:

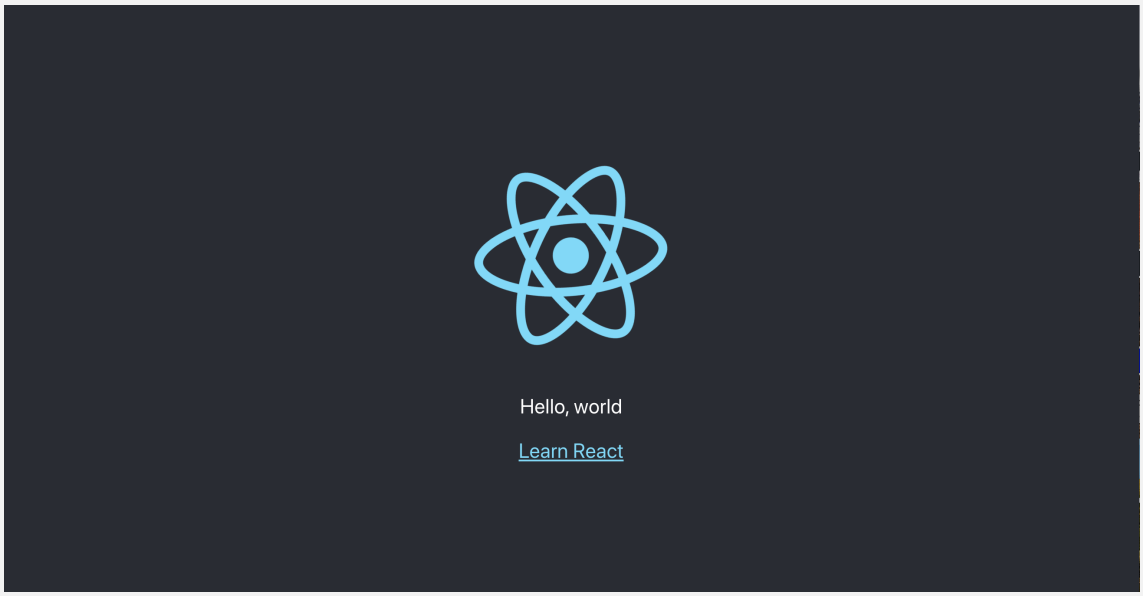
```
cd state-class-tutorial
```

In a new terminal tab or window, start the project using the Create React App start script. The browser will auto-refresh on changes, so leave this script running while you work:

```
npm start
```

You will get a running local server. If the project did not open in a browser window, you can open it with `http://localhost:3000/`. If you are running this from a remote server, the address will be `http://your_domain:3000`.

Your browser will load with a simple React application included as part of Create React App:



React template project

You will be building a completely new set of custom components, so you'll need to start by clearing out some boilerplate code so that you can have an empty project.

To start, open `src/App.js` in a text editor. This is the root component that is injected into the page. All components will start from here. You can find more information about `App.js` at How To Set Up a React Project with Create React App.

Open `src/App.js` with the following command:

```
nano src/App.js
```

You will see a file like this:

state-class-tutorial/src/App.js

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

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
```

```
export default App;
```

Delete the line `import logo from './logo.svg';`. Then replace everything in the `return` statement to return a set of empty tags: `<></>`. This will give you a valid page that returns nothing. The final code will look like this:

state-class-tutorial/src/App.js

```
import React from 'react';
import './App.css';

function App() {
  return <></>;
}

export default App;
```

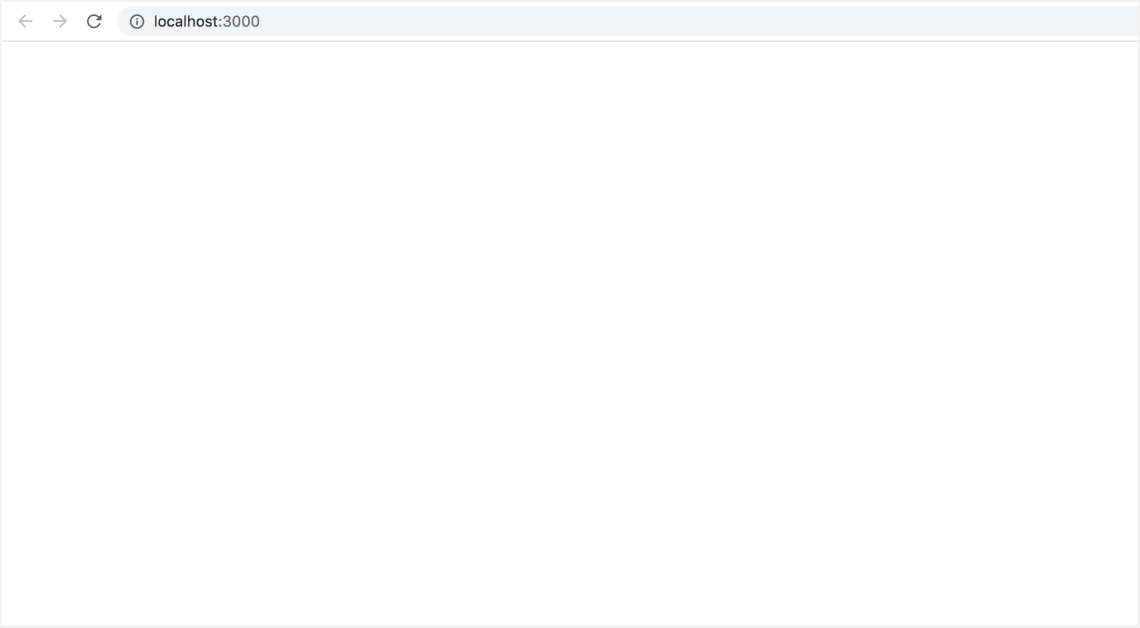
Save and exit the text editor.

Finally, delete the logo. You won't be using it in your application and you should remove unused files as you work. It will save you from confusion in the long run.

In the terminal window type the following command:

```
rm src/logo.svg
```

If you look at your browser, you will see a blank screen.



blank screen in chrome

Now that you have cleared out the sample Create React App project, create a simple file structure. This will help you keep your components isolated and independent.

Create a directory called `components` in the `src` directory. This will hold all of your custom components.

```
mkdir src/components
```

Each component will have its own directory to store the component file along with the styles, images, and tests.

Create a directory for `App`:

```
mkdir src/components/App
```

Move all of the `App` files into that directory. Use the wildcard, `*`, to select any files that start with `App.` regardless of file extension. Then use the `mv` command to put them into the new directory:

```
mv src/App.* src/components/App
```

Next, update the relative import path in `index.js`, which is the root component that bootstraps the whole process:

```
nano src/index.js
```

The import statement needs to point to the `App.js` file in the `App` directory, so make the following highlighted change:

state-class-tutorial/src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './components/App/App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want your app to work offline and load faster, you ca
// unregister() to register() below. Note this comes with some
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Save and exit the file.

Now that the project is set up, you can create your first component.

Step 2 — Using State in a Component

In this step, you'll set the initial state of a component on its class and reference the state to display a value. You'll then make a product page with a shopping cart that displays the total items in the cart using the state value. By the end of the step, you'll know the different ways to hold a value and when you should use state rather than a prop or a static value.

Building the Components

Start by creating a directory for `Product`:

```
mkdir src/components/Product
```

Next, open up `Product.js` in that directory:

```
nano src/components/Product/Product.js
```

Start by creating a component with no state. The component will have two parts: The cart, which has the number of items and the total price, and the product, which has a button to add and remove an item. For now, the buttons will have no actions.

Add the following code to `Product.js`:

state-class-tutorial/src/components/Product/Product.js

```
import React, { Component } from 'react';
import './Product.css';

export default class Product extends Component {
  render() {
    return(
      <div className="wrapper">
        <div>
          Shopping Cart: 0 total items.
        </div>
        <div>Total: 0</div>

        <div className="product"><span role="img" aria-label="i
        <button>Add</button> <button>Remove</button>
      </div>
    )
  }
}
```

You have also included a couple of `div` elements that have JSX class names so you can add some basic styling.

Save and close the file, then open `Product.css`:

```
nano src/components/Product/Product.css
```

Give some light styling to increase the `font-size` for the text and the emoji:

state-class-tutorial/src/components/Product/Product.css

```
.product span {
    font-size: 100px;
}

.wrapper {
    padding: 20px;
    font-size: 20px;
}

.wrapper button {
    font-size: 20px;
    background: none;
}
```

The emoji will need a much larger font size than the text, since it's acting as the product image in this example. In addition, you are removing the default

gradient background on buttons by setting the `background` to `none`.

Save and close the file.

Now, render the `Product` component in the `App` component so you can see the results in the browser. Open `App.js`:

```
nano src/components/App/App.js
```

Import the component and render it. You can also delete the CSS import since you won't be using it in this tutorial:

state-class-tutorial/src/components/App/App.js

```
import React from 'react';
import Product from '../Product/Product';

function App() {
  return <Product />
}

export default App;
```

Save and close the file. When you do, the browser will refresh and you'll see the `Product` component.

Shopping Cart: 0 total items.
Total: 0

Add Remove

Product Page

Setting the Initial State on a Class Component

There are two values in your component values that are going to change in your display: total number of items and total cost. Instead of hard coding them, in this step you'll move them into an object called `state`.

The `state` of a React class is a special property that controls the rendering of a page. When you change the state, React knows that the component is out-of-date and will automatically re-render. When a component re-renders, it modifies the rendered output to include the most up-to-date information in `state`. In this example, the component will re-render whenever you add a product to the cart or remove it from the cart. You can add other properties to a React class, but they won't have the same ability to trigger re-rendering.

Open `Product.js`:

```
nano src/components/Product/Product.js
```

Add a property called `state` to the `Product` class. Then add two values to the `state` object: `cart` and `total`. The `cart` will be an array, since it may eventually hold many items. The `total` will be a number. After assigning these, replace references to the values with `this.state.property`:

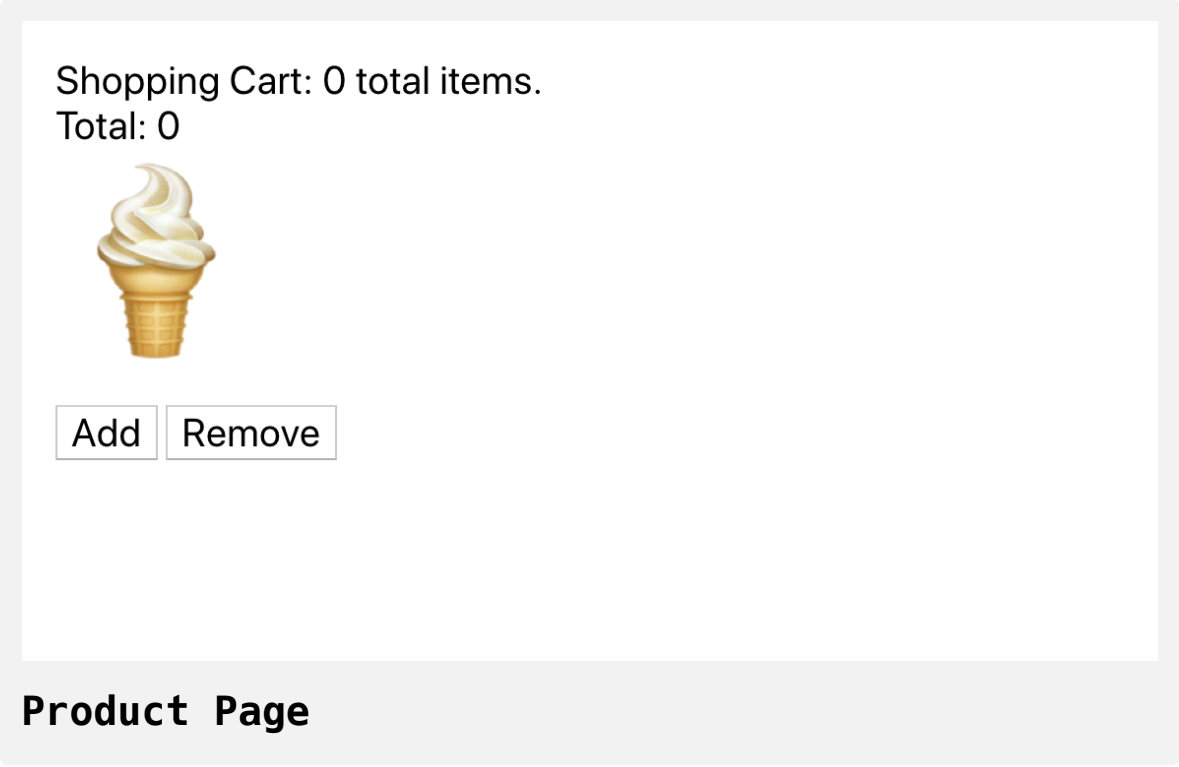
state-class-tutorial/src/components/Product/Product.js

```
import React, { Component } from 'react';
import './Product.css';

export default class Product extends Component {

  state = {
    cart: [],
    total: 0
  }

  render() {
    return(
      <div className="wrapper">
        <div>
          Shopping Cart: {this.state.cart.length} total items.
        </div>
        <div>Total {this.state.total}</div>

        <div className="product"><span role="img" aria-label="i
        <button>Add</button> <button>Remove</button>
      </div>
    )
```

```
  }
}
```

Notice that in both cases, since you are referencing JavaScript inside of your JSX, you need to wrap the code in curly braces. You are also displaying the `length` of the `cart` array to get a count of the number of items in the array.

Save the file. When you do, the browser will refresh and you'll see the same page as before.



Product Page

The `state` property is a standard class property, which means that it is accessible in other methods, not just the `render` method.

Next, instead of displaying the price as a static value, convert it to a string using the `toLocaleString` method, which will convert the number to a string that matches the way numbers are displayed in the browser's region.

Create a method called `getTotal()` that takes the `state` and converts it to a localized string using an array of `currencyOptions`. Then, replace the reference to `state` in the JSX with a method call:

state-class-tutorial/src/components/Product/Product.js

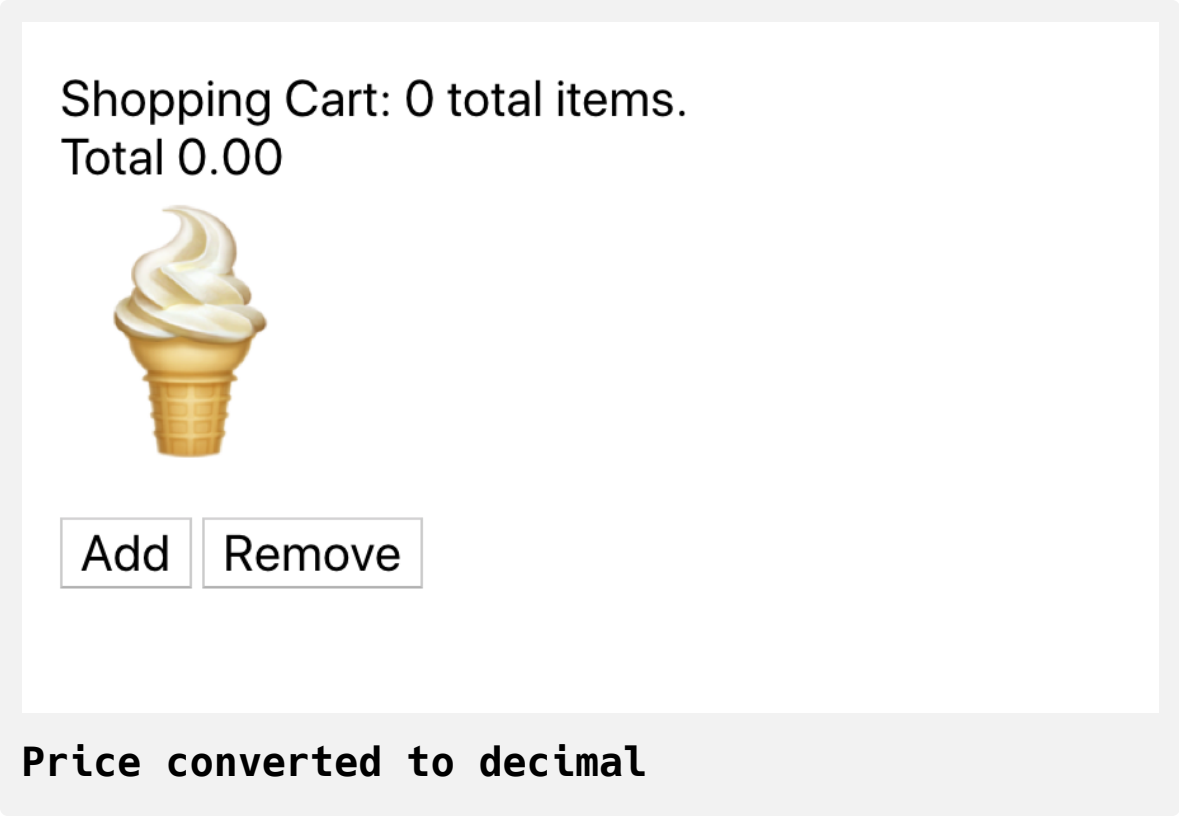
```
import React, { Component } from 'react';
import './Product.css';

export default class Product extends Component {

  state = {
    cart: [],
    total: 0
  }

  currencyOptions = {
    minimumFractionDigits: 2,
    maximumFractionDigits: 2,
  }

  getTotal = () => {
    return this.state.total.toLocaleString(undefined, this.curr
  }

  render() {
    return(
      <div className="wrapper">
        <div>
          Shopping Cart: {this.state.cart.length} total items.
```

```
        </div>
        <div>Total {this.getTotal()}</div>

        <div className="product"><span role="img" aria-label="i
        <button>Add</button> <button>Remove</button>
      </div>
    )
  }
}
```

Since `total` is a price for goods, you are passing `currencyOptions` that set the maximum and minimum decimal places for your `total` to two. Note that this is set as a separate property. Often, beginner React developers will put information like this in the `state` object, but it is best to only add information to `state` that you expect to change. This way, the information in `state` will be easier to keep strack of as your application scales.

Another important change you made was to create the `getTotal()` method by assigning an arrow function to a class property. Without using the arrow function, this method would create a new `this` binding, which would interfere with the current `this` binding and introduce a bug into our code. You'll see more on this in the next step.

Save the file. When you do, the page will refresh and you'll see the value converted to a decimal.




```
Price converted to decimal
```

You've now added state to a component and referenced it in your class. You also accessed values in the `render` method and in other class methods. Next, you'll create methods to update the state and show dynamic values.

Step 3 — Setting State from a Static Value

So far you've created a base state for the component and you've referenced that state in your functions and your JSX code. In this step, you'll update your product page to modify the `state` on button clicks. You'll learn how to pass a new object containing updated values to a special method called `setState`, which will then set the `state` with the updated data.

To update `state`, React developers use a special method called `setState` that is inherited from the base `Component` class. The `setState` method can take either an object or a function as the first argument. If you have a static value that doesn't need to reference the `state`, it's best to pass an object containing the new value, since it's easier to read. If you need to reference the current state, you pass a function to avoid any references to out-of-date `state`.

Start by adding an event to the buttons. If your user clicks **Add**, then the program will add the item to the `cart` and update the `total`. If they click **Remove**, it will reset the cart to an empty array and the `total` to `0`. For example purposes, the program will not allow a user to add an item more then once.

Open `Product.js`:

```
nano src/components/Product/Product.js
```

Inside the component, create a new method called `add`, then pass the method to the `onClick` prop for the **Add** button:

state-class-tutorial/src/components/Product/Product.js

```
import React, { Component } from 'react';
import './Product.css';

export default class Product extends Component {

  state = {
    cart: [],
    total: 0
  }

  add = () => {
    this.setState({
      cart: ['ice cream'],
      total: 5
    })
  }

  currencyOptions = {
    minimumFractionDigits: 2,
    maximumFractionDigits: 2,
  }

  getTotal = () => {
    return this.state.total.toLocaleString(undefined, this.curr
```

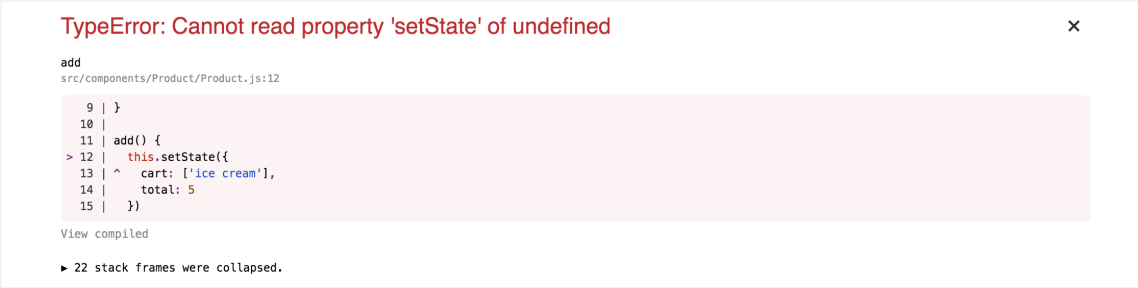
```
  }

  render() {
    return(
      <div className="wrapper">
        <div>
          Shopping Cart: {this.state.cart.length} total items.
        </div>
        <div>Total {this.getTotal()}</div>

        <div className="product"><span role="img" aria-label="i
        <button onClick={this.add}>Add</button>
        <button>Remove</button>
      </div>
    )
  }
}
```

Inside the `add` method, you call the `setState` method and pass an object containing the updated `cart` with a single item `ice cream` and the updated price of `5`. Notice that you again used an arrow function to create the `add` method. As mentioned before, this will ensure the function has the proper `this` context when running the update. If you add the function as a method without using the arrow function, the `setState` would not exist without binding the function to the current context.

For example, if you created the `add` function this way:

```
export default class Product extends Component {
...
  add() {
    this.setState({
      cart: ['ice cream'],
      total: 5
    })
  }
...
}
```

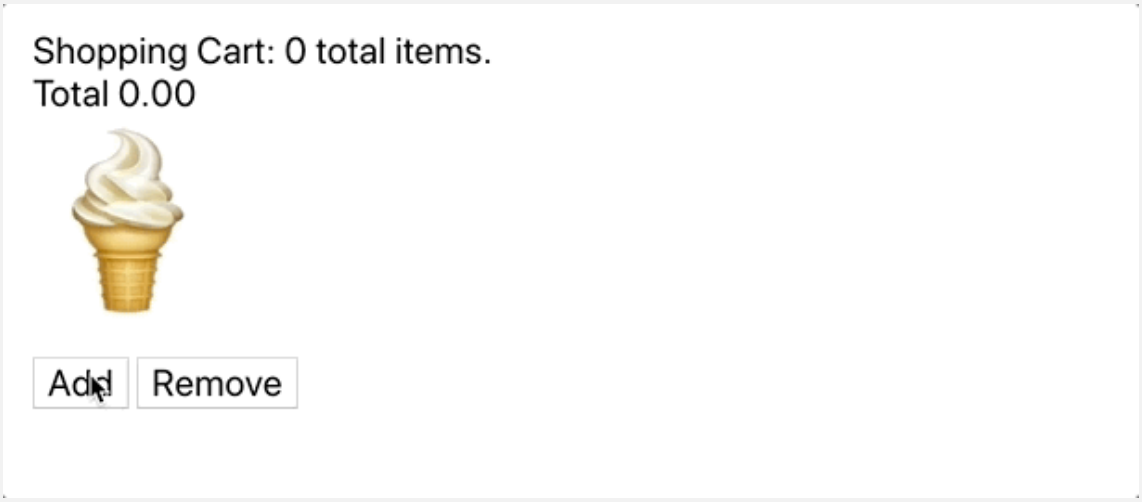
The user would get an error when they click on the **Add** button.



Context Error

Using an arrow function ensures that you'll have the proper context to avoid this error.

Save the file. When you do, the browser will reload, and when you click on the **Add** button the cart will update with the current amount.



`Click on the button and see state updated`

With the `add` method, you passed both properties of the `state` object: `cart` and `total`. However, you do not always need to pass a complete object. You only need to pass an object containing the properties that you want to update, and everything else will stay the same.

To see how React can handle a smaller object, create a new function called `remove`. Pass a new object containing just the `cart` with an empty array, then add the method to the `onClick` property of the **Remove** button:

state-class-tutorial/src/components/Product/Product.js

```
import React, { Component } from 'react';
import './Product.css';

export default class Product extends Component {

  ...
  remove = () => {
    this.setState({
      cart: []
    })
  }

  render() {
    return(
      <div className="wrapper">
        <div>
          Shopping Cart: {this.state.cart.length} total items.
        </div>
        <div>Total {this.getTotal()}</div>

        <div className="product"><span role="img" aria-label="i
        <button onClick={this.add}>Add</button>
        <button onClick={this.remove}>Remove</button>
      </div>
```

```
    )
  }
}
```

Save the file. When the browser refreshes, click on the **Add** and **Remove** buttons. You'll see the cart update, but not the price. The `total` state value is preserved during the update. This value is only preserved for example purposes; with this application, you would want to update both properties of the `state` object. But you will often have components with stateful properties that have different responsibilities, and you can make them persist by leaving them out of the updated object.

The change in this step was static. You knew exactly what the values would be ahead of time, and they didn't need to be recalculated from `state`. But if the product page had many products and you wanted to be able to add them multiple times, passing a static object would provide no guarantee of referencing the most up-to-date `state`, even if your object used a `this.state` value. In this case, you could instead use a function.

In the next step, you'll update `state` using functions that reference the current state.

Step 4 — Setting State Using Current State

There are many times when you'll need to reference a previous state to update a current state, such as updating an array, adding a number, or

modifying an object. To be as accurate as possible, you need to reference the most up-to-date `state` object. Unlike updating `state` with a predefined value, in this step you'll pass a function to the `setState` method, which will take the current state as an argument. Using this method, you will update a component's state using the current state.

Another benefit of setting `state` with a function is increased reliability. To improve performance, React may batch `setState` calls, which means that `this.state.value` may not be fully reliable. For example, if you update `state` quickly in several places, it is possible that a value could be out of date. This can happen during data fetches, form validations, or any situation where several actions are occurring in parallel. But using a function with the most up-to-date `state` as the argument ensures that this bug will not enter your code.

To demonstrate this form of state management, add some more items to the product page. First, open the `Product.js` file:

```
nano src/components/Product/Product.js
```

Next, create an array of objects for different products. The array will contain the product emoji, name, and price. Then loop over the array to display each product with an **Add** and **Remove** button:

state-class-tutorial/src/components/Product/Product.js

```
import React, { Component } from 'react';
import './Product.css';

const products = [
  {
    emoji: '🍦',
    name: 'ice cream',
    price: 5
  },
  {
    emoji: '🍩',
    name: 'donuts',
    price: 2.5,
  },
  {
    emoji: '🍉',
    name: 'watermelon',
    price: 4
  }
];

export default class Product extends Component {

  ...
```

```
  render() {
    return(
      <div className="wrapper">
        <div>
          Shopping Cart: {this.state.cart.length} total items.
        </div>
        <div>Total {this.getTotal()}</div>
        <div>
          {products.map(product => (
            <div key={product.name}>
              <div className="product">
                <span role="img" aria-label={product.name}>{pro
              </div>
              <button onClick={this.add}>Add</button>
              <button onClick={this.remove}>Remove</button>
            </div>
          ))}
        </div>
      </div>
    )
  }
}
```

In this code, you are using the `map()` array method to loop over the `products` array and return the JSX that will display each element in your browser.

Save the file. When the browser reloads, you'll see an updated product list:

Shopping Cart: 0 total items.
Total 0.00

Add Remove

Add Remove

Add Remove

```
Product list
```

Now you need to update your methods. First, change the `add()` method to take the `product` as an argument. Then instead of passing an object to `setState()`, pass a function that takes the `state` as an argument and returns an object that has the `cart` updated with the new product and the `total` updated with the new price:

state-class-tutorial/src/components/Product/Product.js

```
import React, { Component } from 'react';
import './Product.css';

...

export default class Product extends Component {

  state = {
    cart: [],
    total: 0
  }

  add = (product) => {
    this.setState(state => ({
      cart: [...state.cart, product.name],
      total: state.total + product.price
    }))
  }

  currencyOptions = {
    minimumFractionDigits: 2,
    maximumFractionDigits: 2,
  }
```

```
  getTotal = () => {
    return this.state.total.toLocaleString(undefined, this.curr
  }

  remove = () => {
    this.setState({
      cart: []
    })
  }

  render() {
    return(
      <div className="wrapper">
        <div>
          Shopping Cart: {this.state.cart.length} total items.
        </div>
        <div>Total {this.getTotal()}</div>

        <div>
          {products.map(product => (
            <div key={product.name}>
              <div className="product">
                <span role="img" aria-label={product.name}>{pro
              </div>
              <button onClick={() => this.add(product)}>Add</bu
              <button onClick={this.remove}>Remove</button>
            </div>
```

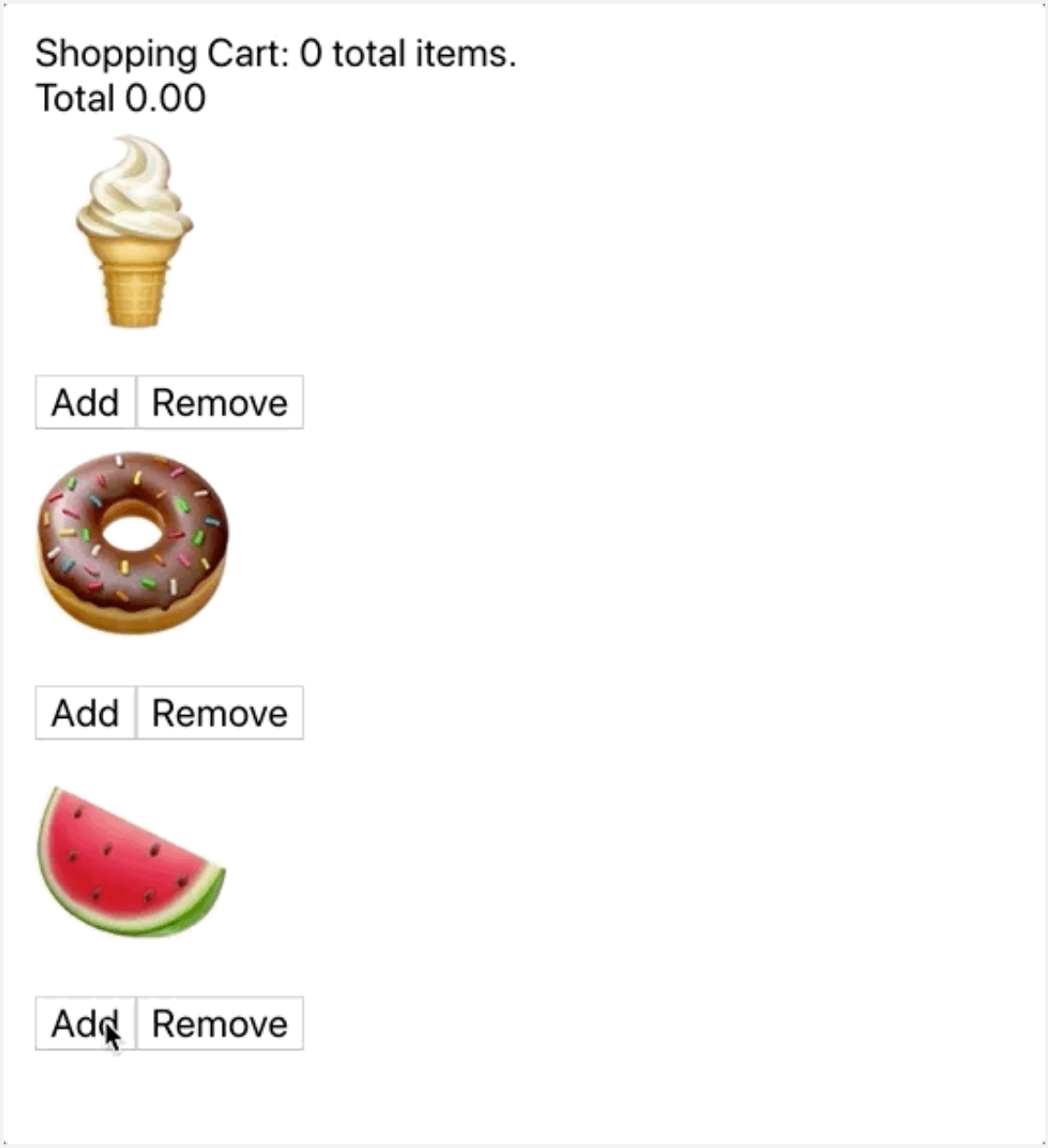
```
          ))}
        </div>
      </div>
    )
  }
}
```

Inside the anonymous function that you pass to `setState()`, make sure you reference the argument—`state`—and not the component's state—`this.state`. Otherwise, you still run a risk of getting an out-of-date `state` object. The `state` in your function will be otherwise identical.

Take care not to directly mutate state. Instead, when adding a new value to the `cart`, you can add the new `product` to the `state` by using the spread syntax on the current value and adding the new value onto the end.

Finally, update the call to `this.add` by changing the `onClick()` prop to take an anonymous function that calls `this.add()` with the relevant product.

Save the file. When you do, the browser will reload and you'll be able to add multiple products.



Adding products

Next, update the `remove()` method. Follow the same steps: convert `setState` to take a function, update the values without mutating, and update the `onChange()` prop:

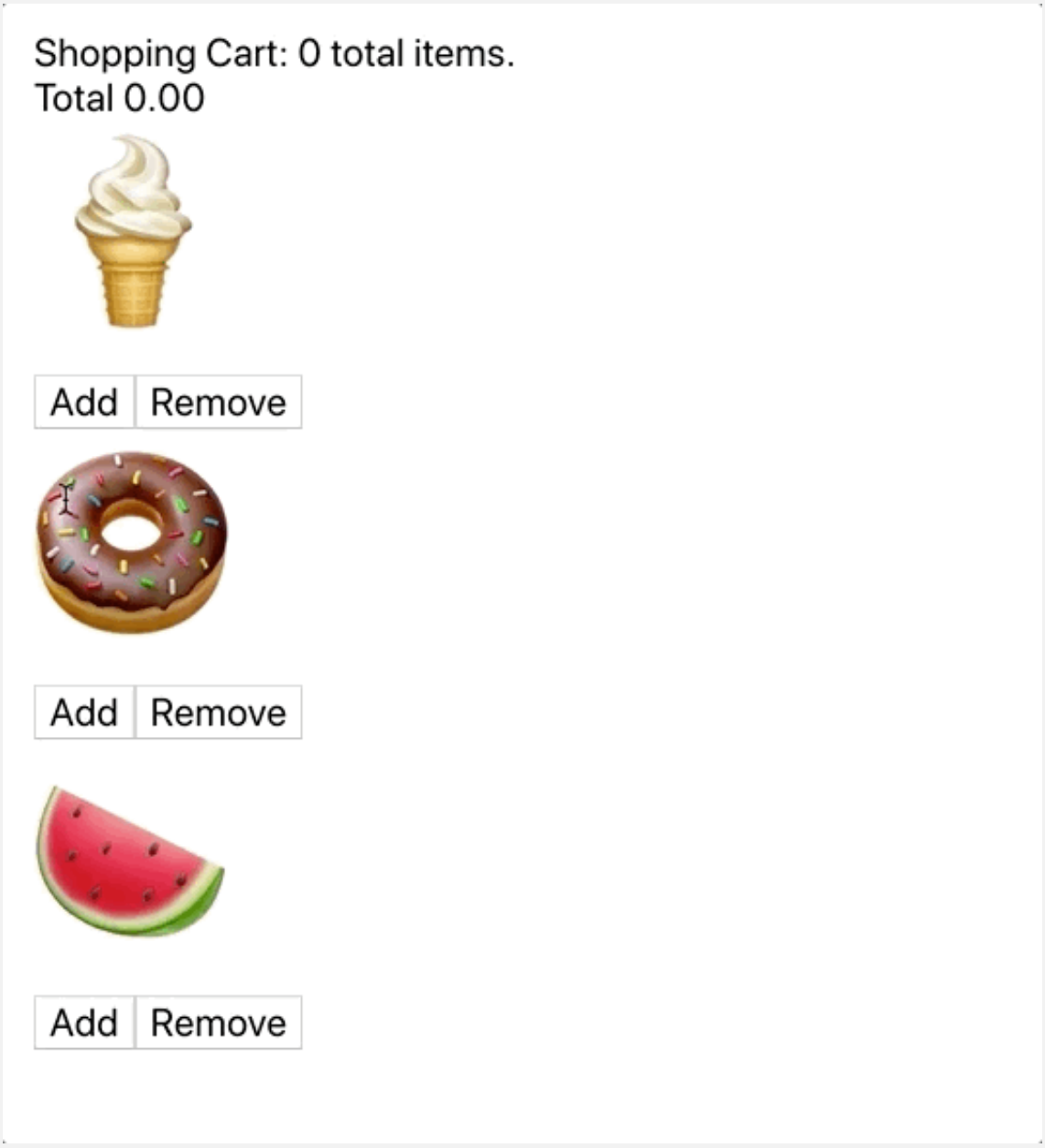
state-class-tutorial/src/components/Product/Product.js

```
import React, { Component } from 'react';
import './Product.css';

...

export default class Product extends Component {

...

  remove = (product) => {
    this.setState(state => {
      const cart = [...state.cart];
      cart.splice(cart.indexOf(product.name))
      return ({
        cart,
        total: state.total - product.price
      })
    })
  }

  render() {
    return(
      <div className="wrapper">
        <div>
```

```
          Shopping Cart: {this.state.cart.length} total items.
        </div>
        <div>Total {this.getTotal()}</div>
        <div>
          {products.map(product => (
            <div key={product.name}>
              <div className="product">
                <span role="img" aria-label={product.name}>{pro
              </div>
              <button onClick={() => this.add(product)}>Add</bu
              <button onClick={() => this.remove(product)}>Remo
            </div>
          ))}
        </div>
      </div>
    )
  }
}
```

To avoid mutating the state object, you must first make a copy of it using the `spread` operator. Then you can splice out the item you want from the copy and return the copy in the new object. By copying `state` as the first step, you can be sure that you will not mutate the `state` object.

Save the file. When you do, the browser will refresh and you'll be able to add and remove items:



Remove items

There is still a bug in this application: In the `remove` method, a user can subtract from the `total` even if the item is not in the `cart`. If you click **Remove** on the ice cream without adding it to your cart, your total will be **-5.00**.

You can fix the bug by checking for an item's existence before subtracting, but an easier way is to keep your state object small by only keeping references to the products and not separating references to products and total cost. Try to avoid double references to the same data. Instead, store the raw data in `state`— in this case the whole `product` object—then perform the calculations outside of the `state`.

Refactor the component so that the `add()` method adds the whole object, the `remove()` method removes the whole object, and the `getTotal` method uses the `cart`:

state-class-tutorial/src/components/Product/Product.js

```
import React, { Component } from 'react';
import './Product.css';

...

export default class Product extends Component {

  state = {
    cart: [],
  }

  add = (product) => {
    this.setState(state => ({
      cart: [...state.cart, product],
    }))
  }

  currencyOptions = {
    minimumFractionDigits: 2,
    maximumFractionDigits: 2,
  }

  getTotal = () => {
```

```
    const total = this.state.cart.reduce(
      (totalCost, item) => totalCost + item.price, 0);
    return total.toLocaleString(undefined, this.currencyOptions
  }

  remove = (product) => {
    this.setState(state => {
      const cart = [...state.cart];
      const productIndex = cart.findIndex(p => p.name === produ
      if(productIndex < 0) {
        return;
      }
      cart.splice(productIndex, 1)
      return ({
        cart
      })
    })
  }

  render() {
    ...
  }
}
```

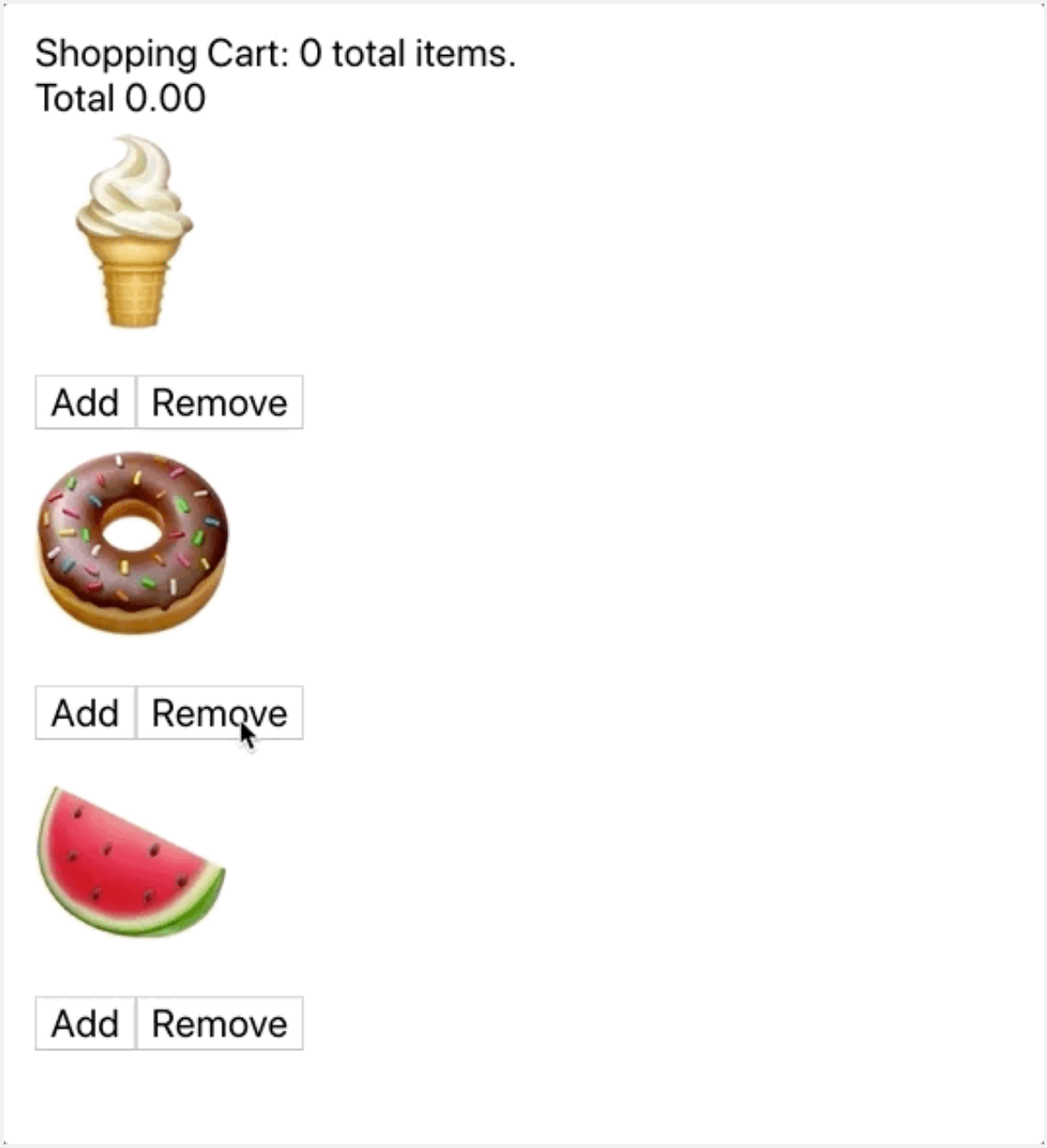
The `add()` method is similar to what it was before, except that reference to the `total` property has been removed. In the `remove()` method, you find

the index of the `product` with `findByIndex`. If the index doesn't exist, you'll get a `-1`. In that case, you use a conditional statement toreturn nothing. By returning nothing, React will know the `state` didn't change and won't trigger a re-render. If you return `state` or an empty object, it will still trigger a re-render.

When using the `splice()` method, you are now passing `1` as the second argument, which will remove one value and keep the rest.

Finally, you calculate the `total` using the `reduce()` array method.

Save the file. When you do, the browser will refresh and you'll have your final `cart`:



Add and remove

The `setState` function you pass can have an additional argument of the current props, which can be helpful if you have state that needs to reference the current props. You can also pass a callback function to `setState` as the second argument, regardless of if you pass an object or function for the first

argument. This is particularly useful when you are setting `state` after fetching data from an API and you need to perform a new action after the `state` update is complete.

In this step, you learned how to update a new state based on the current state. You passed a function to the `setState` function and calculated new values without mutating the current state. You also learned how to exit a `setState` function if there is no update in a manner that will prevent a re-render, adding a slight performance enhancement.

Conclusion

In this tutorial, you have developed a class-based component with a dynamic state that you've updated statically and using the current state. You now have the tools to make complex projects that respond to users and dynamic information.

React does have a way to manage state with Hooks, but it is helpful to understand how to use state on components if you need to work with components that must be class-based, such as those that use the `componentDidCatch` method.

Managing state is key to nearly all components and is necessary for creating interactive applications. With this knowledge you can recreate many common web components, such as sliders, accordions, forms, and more. You will then use the same concepts as you build applications using hooks or develop components that pull data dynamically from APIs.

If you would like to look at more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Manage State with Hooks on React Components

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

In React development, keeping track of how your application data changes over time is called state management. By managing the state of your application, you will be able to make dynamic apps that respond to user input. There are many methods of managing state in React, including class-based state management and third-party libraries like Redux. In this tutorial, you'll manage state on functional components using a method encouraged by the official React documentation: Hooks.

Hooks are a broad set of tools that run custom functions when a component's props change. Since this method of state management doesn't require you to use classes, developers can use Hooks to write shorter, more readable code that is easy to share and maintain. One of the main differences between Hooks and class-based state management is that there is no single object that holds all of the state. Instead, you can break up state into multiple pieces that you can update independently.

Throughout this tutorial, you'll learn how to set state using the `useState` and `useReducer` Hooks. The `useState` Hook is valuable when setting a value without referencing the current state; the `useReducer` Hook is useful when you need to reference a previous value or when you have different

actions the require complex data manipulations. To explore these different ways of setting state, you'll create a product page component with a shopping cart that you'll update by adding purchases from a list of options. By the end of this tutorial, you'll be comfortable managing state in a functional component using Hooks, and you'll have a foundation for more advanced Hooks such as `useEffect`, `useMemo`, and `useContext`.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.20.1 and npm version 6.14.4. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- A React development environment set up with Create React App, with the non-essential boilerplate removed. To set this up, follow **Step 1 — Creating an Empty Project** of the How To Manage State on React Class Components tutorial. This tutorial will use `hooks-tutorial` as the project name.

- You will also need a basic knowledge of JavaScript, which you can find in How To Code in JavaScript, along with a basic knowledge of HTML and CSS. A useful resource for HTML and CSS is the Mozilla Developer Network.

Step 1 – Setting Initial State in a Component

In this step, you'll set the initial state on a component by assigning the initial state to a custom variable using the `useState` Hook. To explore Hooks, you'll make a product page with a shopping cart, then display the initial values based on the state. By the end of the step, you'll know the different ways to hold a state value using Hooks and when to use state rather than a prop or a static value.

Start by creating a directory for a `Product` component:

```
mkdir src/components/Product
```

Next, open up a file called `Product.js` in the `Product` directory:

```
nano src/components/Product/Product.js
```

Start by creating a component with no state. The component will consist of two parts: the cart, which has the number of items and the total price, and the product, which has a button to add or remove the item from the cart. For now, these buttons will have no function.

Add the following code to the file:

hooks-tutorial/src/components/Product/Product.js

```
import React from 'react';
import './Product.css';

export default function Product() {
  return(
    <div className="wrapper">
      <div>
        Shopping Cart: 0 total items.
      </div>
      <div>Total: 0</div>

      <div className="product"><span role="img" aria-label="ice
      <button>Add</button> <button>Remove</button>
    </div>
  )
}
```

In this code, you used JSX to create the HTML elements for the `Product` component, with an ice cream emoji to represent the product. In addition, two of the `<div>` elements have class names so you can add some basic CSS styling.

Save and close the file, then create a new file called `Product.css` in the `Product` directory:

```
nano src/components/Product/Product.css
```

Add some styling to increase the font size for the text and the emoji:

hooks-tutorial/src/components/Product/Product.css

```
.product span {
    font-size: 100px;
}

.wrapper {
    padding: 20px;
    font-size: 20px;
}

.wrapper button {
    font-size: 20px;
    background: none;
    border: black solid 1px;
}
```

The emoji will need a much larger `font-size`, since it's acting as the product image. In addition, you are removing the default gradient background on the button by setting `background` to `none`.

Save and close the file. Now, add the component into the `App` component to render the `Product` component in the browser. Open `App.js`:

```
nano src/components/App/App.js
```

Import the component and render it. Also, delete the CSS import since you won't be using it in this tutorial:

hooks-tutorial/src/components/App/App.js

```
import React from 'react';
import Product from '../Product/Product';

function App() {
  return <Product />
}

export default App;
```

Save and close the file. When you do, the browser will refresh and you'll see the `Product` component:

Shopping Cart: 0 total items.
Total: 0

Add Remove

Product Page

Now that you have a working component, you can replace the hard-coded data with dynamic values.

React exports several Hooks that you can import directly from the main `React` package. By convention, React Hooks start with the word `use`, such as `useState`, `useContext`, and `useReducer`. Most third-party libraries follow the same convention. For example, Redux has a `useSelector` and a `useStore` Hook.

Hooks are functions that let you run actions as part of the React lifecycle. Hooks are triggered either by other actions or by changes in a component's props and are used to either create data or to trigger further changes. For example, the `useState` Hook generates a stateful piece of data along with a function for changing that piece of data and triggering a re-render. It will

create a dynamic piece of code and hook into the lifecycle by triggering re-renders when the data changes. In practice, that means you can store dynamic pieces of data in variables using the `useState` Hook.

For example, in this component, you have two pieces of data that will change based on user actions: the cart and the total cost. Each of these can be stored in state using the above Hook.

To try this out, open up `Product.js`:

```
nano src/components/Product/Product.js
```

Next, import the `useState` Hook from React by adding the highlighted code:

hooks-tutorial/src/components/Product/Product.js

```
import React, { useState } from 'react';
import './Product.css';

export default function Product() {
  return(
    <div className="wrapper">
      <div>
        Shopping Cart: 0 total items.
      </div>
      <div>Total: 0</div>

      <div className="product"><span role="img" aria-label="ice
      <button>Add</button> <button>Remove</button>
    </div>
  )
}
```

`useState` is a function that takes the initial state as an argument and returns an array with two items. The first item is a variable containing the state, which you will often use in your JSX. The second item in the array is a function that will update the state. Since React returns the data as an array, you can use destructuring to assign the values to any variable names you

want. That means you can call `useState` many times and never have to worry about name conflicts, since you can assign every piece of state and update function to a clearly named variable.

Create your first Hook by invoking the `useState` Hook with an empty array. Add in the following highlighted code:

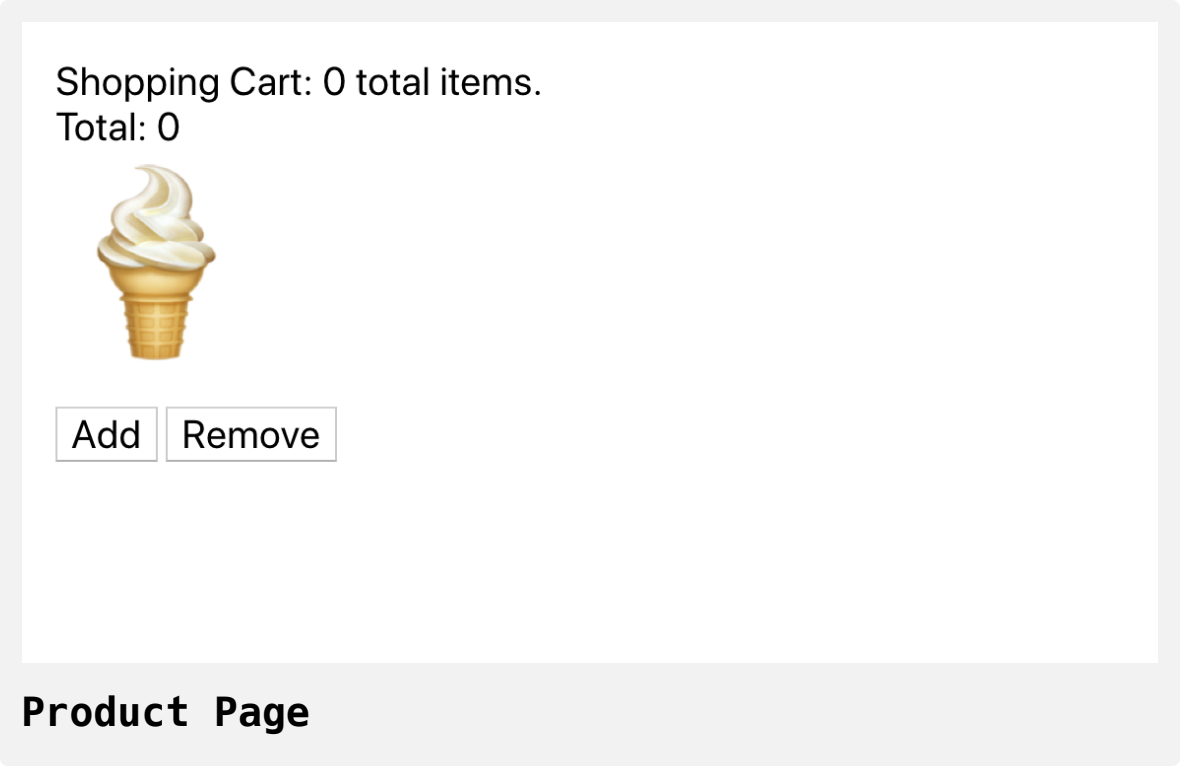
hooks-tutorial/src/components/Product/Product.js

```
import React, { useState } from 'react';
import './Product.css';

export default function Product() {
  const [cart, setCart] = useState([]);
  return(
    <div className="wrapper">
      <div>
        Shopping Cart: {cart.length} total items.
      </div>
      <div>Total: 0</div>

      <div className="product"><span role="img" aria-label="ice
      <button>Add</button> <button>Remove</button>
    </div>
  )
}
```

Here you assigned the first value, the state, to a variable called `cart`. `cart` will be an array that contains the products in the cart. By passing an empty array as an argument to `useState`, you set the initial empty state as the first value of `cart`.

In addition to the `cart` variable, you assigned the update function to a variable called `setCart`. At this point, you aren't using the `setCart` function, and you may see a warning about having an unused variable. Ignore this warning for now; in the next step, you'll use `setCart` to update the `cart` state.

Save the file. When the browser reloads, you'll see the page without changes:



One important difference between Hooks and class-based state management is that, in class-based state management, there is a single state object. With Hooks, state objects are completely independent of each other, so you can have as many state objects as you want. That means that if you want a new

piece of stateful data, all you need to do is call `useState` with a new default and assign the result to new variables.

Inside `Product.js`, try this out by creating a new piece of state to hold the `total`. Set the default value to `0` and assign the value and function to `tota l` and `setTotal`:

hooks-tutorial/src/components/Product/Product.js

```
import React, { useState } from 'react';
import './Product.css';

export default function Product() {
  const [cart, setCart] = useState([]);
  const [total, setTotal] = useState(0);
  return(
    <div className="wrapper">
      <div>
        Shopping Cart: {cart.length} total items.
      </div>
      <div>Total: {total}</div>

      <div className="product"><span role="img" aria-label="ice
      <button>Add</button> <button>Remove</button>
    </div>
  )
}
```

Now that you have some stateful data, you can standardize the displayed data to make a more predictable experience. For example, since the total in this example is a price, it will always have two decimal places. You can use

the `toLocaleString` method to convert `total` from a number to a string with two decimal places. It will also convert the number to a string according to the numerical conventions that match the browser's locale. You'll set the options `minimumFractionDigits` and `maximumFractionDigits` to give a consistent number of decimal places.

Create a function called `getTotal`. This function will use the in-scope variable `total` and return a localized string that you will use to display the total. Use `undefined` as the first argument to `toLocaleString` to use the system locale rather than specifying a locale:

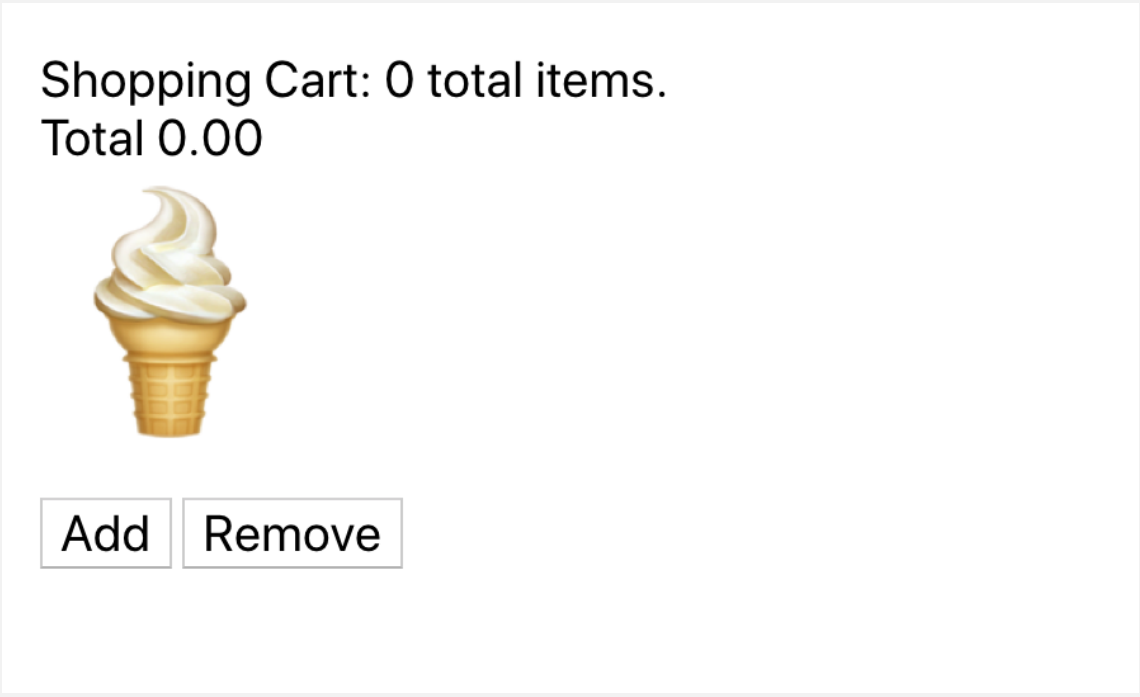
hooks-tutorial/src/components/Product/Product.js

```
import React, { useState } from 'react';
import './Product.css';

const currencyOptions = {
  minimumFractionDigits: 2,
  maximumFractionDigits: 2,
}

export default function Product() {
  const [cart, setCart] = useState([]);
  const [total, setTotal] = useState(0);

  function getTotal() {
    return total.toLocaleString(undefined, currencyOptions)
  }

  return(
    <div className="wrapper">
      <div>
        Shopping Cart: {cart.length} total items.
      </div>
      <div>Total: {getTotal()}</div>

      <div className="product"><span role="img" aria-label="ice
```

```
      <button>Add</button> <button>Remove</button>
    </div>
  )
}
```

You now have added some string processing to the displayed total. Even though `getTotal` is a separate function, it shares the same scope as the surrounding function, which means it can reference the variables of the component.

Save the file. The page will reload and you'll see the updated total with two decimal places:



Price converted to decimal

This function works, but as of now, `getTotal` can only operate in this piece of code. In this case, you can convert it to a pure function, which gives the same outputs when given the same inputs and does not rely on a specific environment to operate. By converting the function to a pure function, you make it more reusable. You can, for example, extract it to a separate file and use it in multiple components.

Update `getTotal` to take `total` as an argument. Then move the function outside of the component:

hooks-tutorial/src/components/Product/Product.js

```
import React, { useState } from 'react';
import './Product.css';

const currencyOptions = {
  minimumFractionDigits: 2,
  maximumFractionDigits: 2,
}

function getTotal(total) {
  return total.toLocaleString(undefined, currencyOptions)
}

export default function Product() {
  const [cart, setCart] = useState([]);
  const [total, setTotal] = useState(0);



  return(
    <div className="wrapper">
      <div>
        Shopping Cart: {cart.length} total items.
```

```
      </div>
      <div>Total: {getTotal(total)}</div>

      <div className="product"><span role="img" aria-label="ice
      <button>Add</button> <button>Remove</button>
    </div>
  )
}
```

Save the file. When you do, the page will reload and you'll see the component as it was before.

Functional components like this make it easier to move functions around. As long as there are no scope conflicts, you can move these conversion functions anywhere you want.

In this step, you set the default value for a stateful piece of data using `useState`. You then saved the stateful data and a function for updating the state to variables using array destructuring. In the next step, you'll use the update function to change the state value to re-render the page with updated information.

Step 2 — Setting State with `useState`

In this step, you'll update your product page by setting a new state with a static value. You have already created the function to update a piece of state, so now you'll create an event to update both stateful variables with predefined values. By the end of this step, you'll have a page with state that a user will be able to update at the click of a button.

Unlike class-based components, you cannot update several pieces of state with a single function call. Instead, you must call each function individually. This means there is a greater separation of concerns, which helps keep stateful objects focused.

Create a function to add an item to the cart and update the total with the price of the item, then add that functionality to the **Add** button:

hooks-tutorial/src/components/Product/Product.js

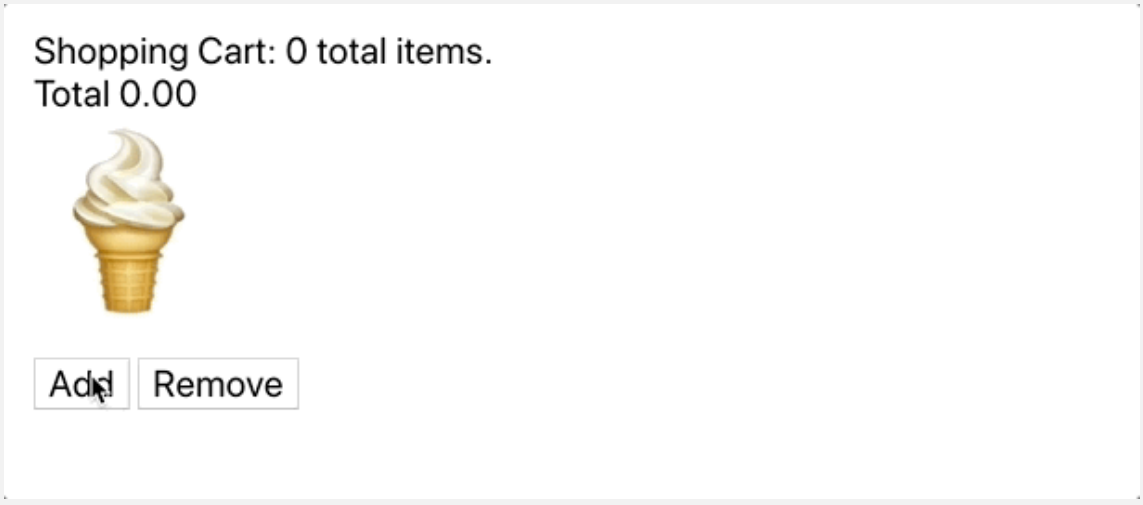
```
import React, { useState } from 'react';

...

export default function Product() {
  const [cart, setCart] = useState([]);
  const [total, setTotal] = useState(0);

  function add() {
    setCart(['ice cream']);
    setTotal(5);
  }

  return(
    <div className="wrapper">
      <div>
        Shopping Cart: {cart.length} total items.
      </div>
      <div>Total: {getTotal(total)}</div>

      <div className="product"><span role="img" aria-label="ice
      <button onClick={add}>Add</button>
      <button>Remove</button>
    </div>
```

```
  )
}
```

In this snippet, you called `setCart` with an array containing the word “ice cream” and called `setTotal` with `5`. You then added this function to the `onClick` event handler for the **Add** button.

Notice that the function must have the same scope as the functions to set state, so it must be defined inside the component function.

Save the file. When you do, the browser will reload, and when you click on the **Add** button the cart will update with the current amount:



Click on the button and see state updated

Since you are not referencing a `this` context, you can use either an arrow function or a function declaration. They both work equally well here, and

each developer or team can decide which style to use. You can even skip defining an extra function and pass the function directly into the `onClick` property.

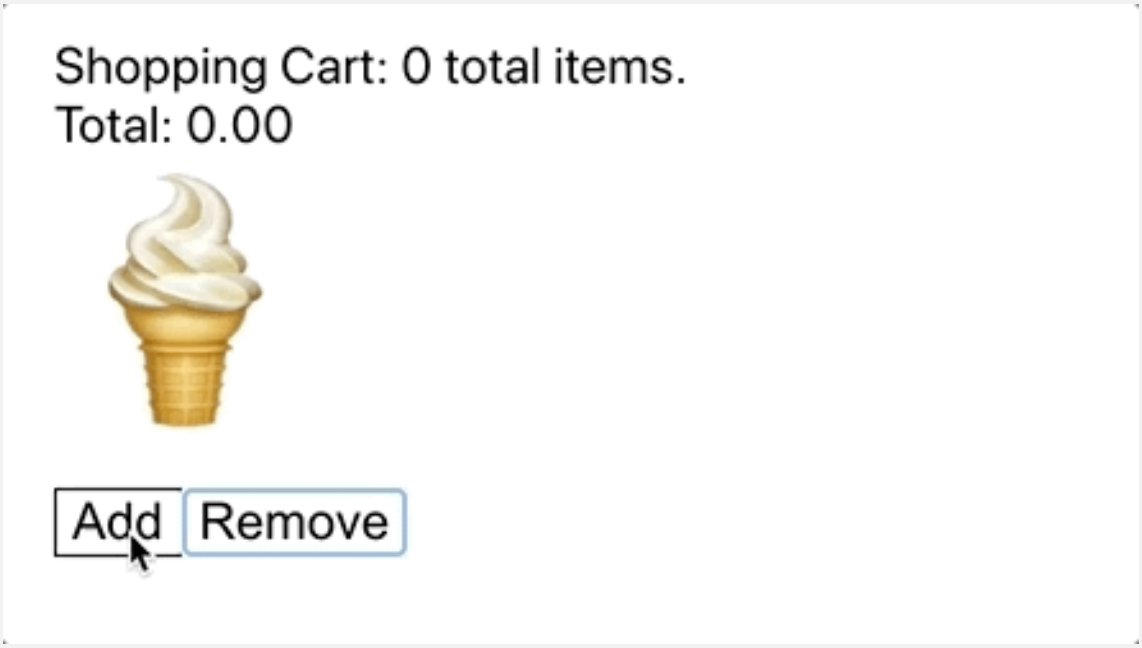
To try this out, create a function to remove the values by setting the cart to an empty object and the total to `0`. Create the function in the `onClick` prop of the **Remove** button:

hooks-tutorial/src/component/Product/Product.js

```
import React, { useState } from 'react';
...
export default function Product() {
  const [cart, setCart] = useState([]);
  const [total, setTotal] = useState(0);

  function add() {
    setCart(['ice cream']);
    setTotal(5);
  }

  return(
    <div className="wrapper">
      <div>
        Shopping Cart: {cart.length} total items.
      </div>
      <div>Total: {getTotal(total)}</div>

      <div className="product"><span role="img" aria-label="ice
      <button onClick={add}>Add</button>
      <button
        onClick={() => {
          setCart([]);
          setTotal(0);
        }}
```

```
      >
        Remove
      </button>
    </div>
  )
}
```

Save the file. When you do, you will be able to add and remove an item:



Add and Remove

Both strategies for assigning the function work, but there are some slight performance implications to creating an arrow function directly in a prop. In every re-render, React will create a new function, which would trigger a

prop change and cause the component to re-render. When you define a function outside of a prop, you can take advantage of another Hook called `useCallback`. This will memoize the function, meaning that it will only create a new function if certain values change. If nothing changes, the program will use the cached memory of the function instead of recalculating it. Some components may not need that level of optimization, but as a rule, the higher a component is likely to be in a tree, the greater the need for memoization.

In this step, you updated state data with functions created by the `useState` Hook. You created wrapping functions to call both functions to update the state of several pieces of data at the same time. But these functions are limited because they add static, pre-defined values instead of using the previous state to create the new state. In the next step, you'll update the state using the current state with both the `useState` Hook and a new Hook called `useReducer`.

Step 3 — Setting State Using Current State

In the previous step, you updated state with a static value. It didn't matter what was in the previous state—you always passed the same value. But a typical product page will have many items that you can add to a cart, and you'll want to be able to update the cart while preserving the previous items.

In this step, you'll update the state using the current state. You'll expand your product page to include several products and you'll create functions that update the cart and the total based on the current values. To update the

values, you'll use both the `useState` Hook and a new Hook called `useReducer`.

Since React may optimize code by calling actions asynchronously, you'll want to make sure that your function has access to the most up-to-date state. The most basic way to solve this problem is to pass a function to the state-setting function instead of a value. In other words, instead of calling `setState(5)`, you'd call `setState(previous => previous +5)`.

To start implementing this, add some more items to the product page by making a `products` array of objects, then remove the event handlers from the **Add** and **Remove** buttons to make room for the refactoring:

hooks-tutorial/src/component/Product/Product.js

```
import React, { useState } from 'react';
import './Product.css';


...


const products = [
  {
    emoji: '🍦',
    name: 'ice cream',
    price: 5
  },
  {
    emoji: '🍩',
    name: 'donuts',
    price: 2.5,
  },
  {
    emoji: '🍉',
    name: 'watermelon',
    price: 4
  }
];


export default function Product() {
  const [cart, setCart] = useState([]);
```

```
  const [total, setTotal] = useState(0);


  function add() {
    setCart(['ice cream']);
    setTotal(5);
  }


  return(
    <div className="wrapper">
      <div>
        Shopping Cart: {cart.length} total items.
      </div>
      <div>Total: {getTotal(total)}</div>
        <div>
        {products.map(product => (
          <div key={product.name}>
            <div className="product">
              <span role="img" aria-label={product.name}>{produ
            </div>
            <button>Add</button>
            <button>Remove</button>
          </div>
        ))}
      </div>
    </div>
  )
}
```

You now have some JSX that uses the `.map` method to iterate over the array and display the products.

Save the file. When you do, the page will reload and you'll see multiple products:

Shopping Cart: 0 total items.
Total 0.00

Add Remove

Add Remove

Add Remove

```
Product list
```

Currently, the buttons have no actions. Since you only want to add the specific product on click, you'll need to pass the product as an argument to the `add` function. In the `add` function, instead of passing the new item directly to the `setCart` and `setTotal` functions, you'll pass an anonymous function that takes the current state and returns a new updated value:

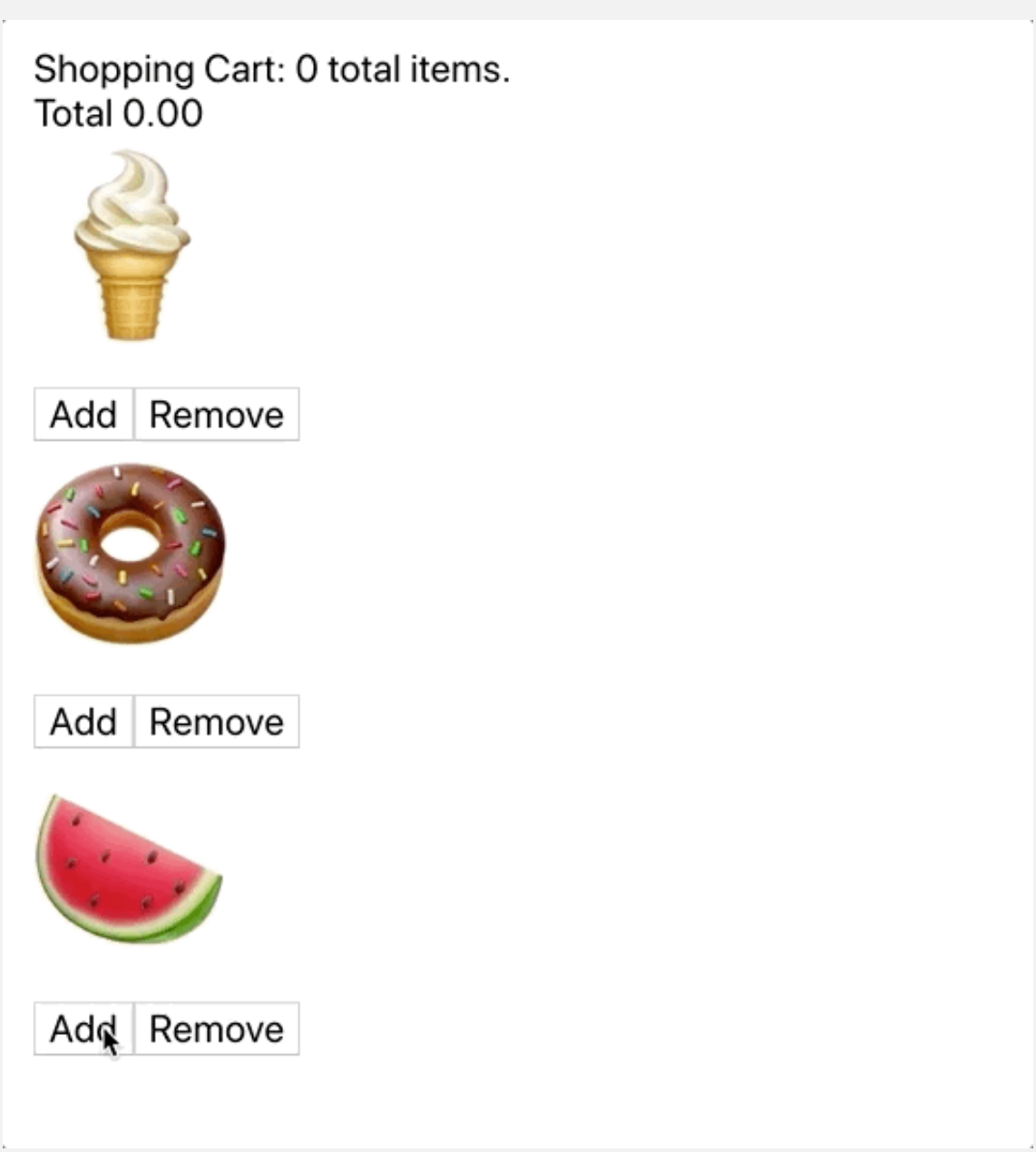
hooks-tutorial/src/component/Product/Product.js

```
import React, { useState } from 'react';
import './Product.css';
...
export default function Product() {
  const [cart, setCart] = useState([]);
  const [total, setTotal] = useState(0);

  function add(product) {
    setCart(current => [...current, product.name]);
    setTotal(current => current + product.price);
  }

  return(
    <div className="wrapper">
      <div>
        Shopping Cart: {cart.length} total items.
      </div>
      <div>Total: {getTotal(total)}</div>

      <div>
        {products.map(product => (
          <div key={product.name}>
            <div className="product">
              <span role="img" aria-label={product.name}>{produ
            </div>
```

```
            <button onClick={() => add(product)}>Add</button>
            <button>Remove</button>
          </div>
        ))}
      </div>
    </div>
  )
}
```

The anonymous function uses the most recent state—either `cart` or `total`—as an argument that you can use to create a new value. Take care, though, not to directly mutate state. Instead, when adding a new value to the cart you can add the new product to the state by spreading the current value and adding the new value onto the end.

Save the file. When you do, the browser will reload and you'll be able to add multiple products:



Adding products

There's another Hook called `useReducer` that is specially designed to update the state based on the current state, in a manner similar to the `.reduce` array method. The `useReducer` Hook is similar to `useState`, but when you initialize the Hook, you pass in a function the Hook will run when you

change the state along with the initial data. The function—referred to as the `reducer`—takes two arguments: the state and another argument. The other argument is what you will supply when you call the update function.

Refactor the cart state to use the `useReducer` Hook. Create a funciton called `cartReducer` that takes the `state` and the `product` as arguments. Replace `useState` with `useReducer`, then pass the `cartReducer` function as the first argument and an empty array as the second argument, which will be the initial data:

hooks-tutorial/src/component/Product/Product.js

```
import React, { useReducer, useState } from 'react';

...

function cartReducer(state, product) {
  return [...state, product]
}

export default function Product() {
  const [cart, setCart] = useReducer(cartReducer, []);
  const [total, setTotal] = useState(0);

  function add(product) {
    setCart(product.name);
    setTotal(current => current + product.price);
  }

  return(
...
  )
}
```

Now when you call `setCart`, pass in the product name instead of a function. When you call `setCart`, you will call the reducer function, and the product will be the second argument. You can make a similar change with the `total` state.

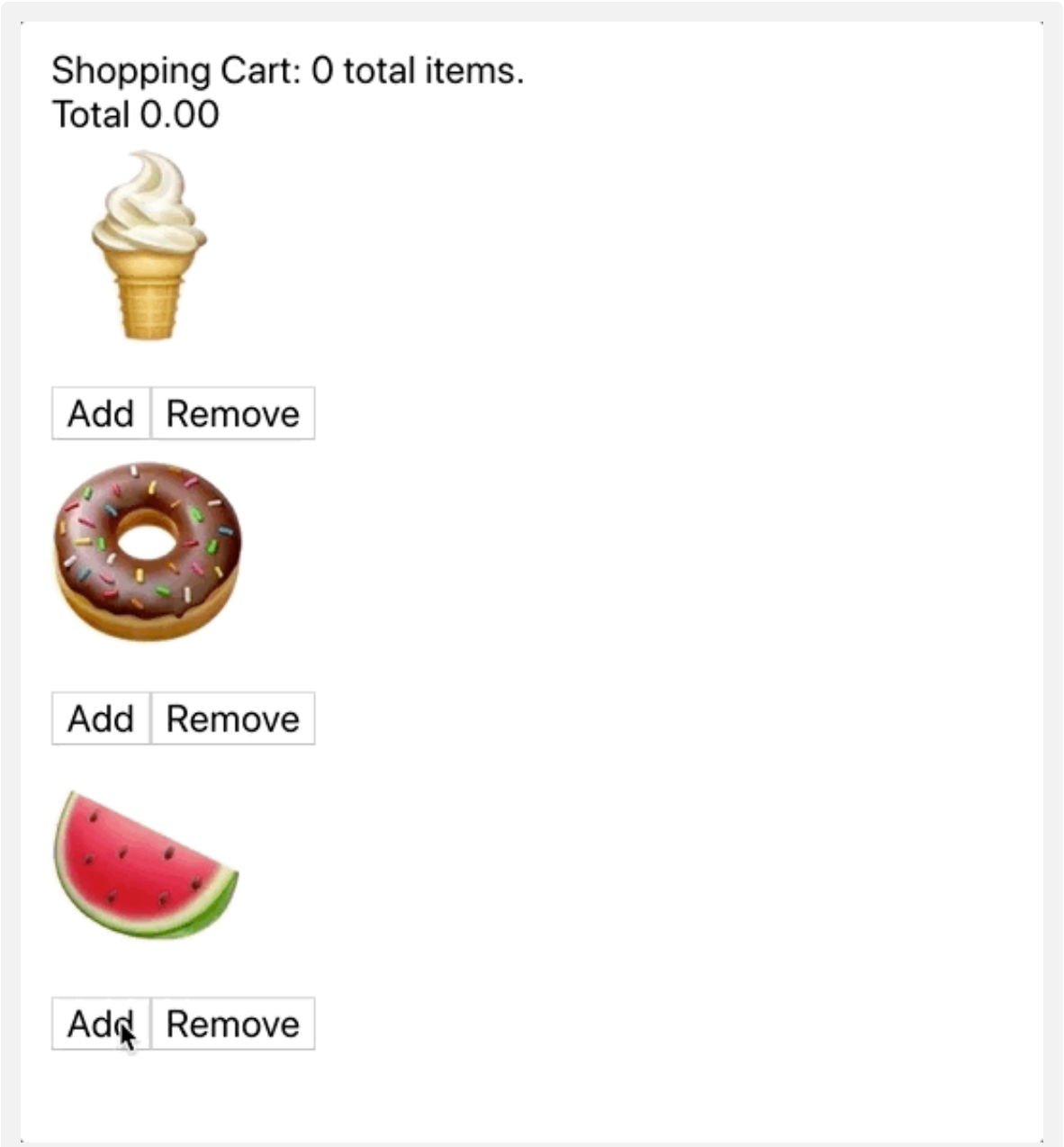
Create a function called `totalReducer` that takes the current state and adds the new amount. Then replace `useState` with `useReducer` and pass the new value `setCart` instead of a function:

hooks-tutorial/src/component/Product/Product.js

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

...

function totalReducer(state, price) {
  return state + price;
}

export default function Product() {
  const [cart, setCart] = useReducer(cartReducer, []);
  const [total, setTotal] = useReducer(totalReducer, 0);

  function add(product) {
    setCart(product.name);
    setTotal(product.price);
  }

  return(
    ...
  )
}
```

Since you are no longer using the `useState` Hook, you removed it from the import.

Save the file. When you do, the page will reload and you'll be able to add items to the cart:



Adding products

Now it’s time to add the `remove` function. But this leads to a problem: The reducer functions can handle adding items and updating totals, but it’s not clear how it will be able to handle removing items from the state. A common pattern in reducer functions is to pass an object as the second argument that contains the name of the action and the data for the action. Inside the reducer, you can then update the total based on the action. In this case, you will add items to the cart on an `add` action and remove them on a `remove` action.

Start with the `totalReducer`. Update the function to take an `action` as the second argument, then add a conditional to update the state based on the `action.type`:

hooks-tutorial/src/component/Product/Product.js

```
import React, { useReducer } from 'react';
import './Product.css';


...


function totalReducer(state, action) {
  if(action.type === 'add') {
    return state + action.price;
  }
  return state - action.price
}


export default function Product() {
  const [cart, setCart] = useReducer(cartReducer, []);
  const [total, setTotal] = useReducer(totalReducer, 0);

  function add(product) {
    const { name, price } = product;
    setCart(name);
    setTotal({ price, type: 'add' });
  }

  return(
    ...
```

```
  )
}
```

The `action` is an object with two properites: `type` and `price`. The type can be either `add` or `remove`, and the `price` is a number. If the type is `add`, it increases the total. If it is `remove`, it lowers the total. After updating the `totalReducer`, you call `setTotal` with a `type` of `add` and the `price`, which you set using destructuring assignment.

Next, you will update the `cartReducer`. This one is a little more complicated: You can use `if/then` conditionals, but it's more common to use a `switch` statement. Switch statements are particularly useful if you have a reducer that can handle many different actions because it makes those actions more readable in your code.

As with the `totalReducer`, you'll pass an object as the second item `type` and `name` properties. If the action is `remove`, update the state by splicing out the first instance of a product.

After updating the `cartReducer`, create a `remove` function that calls `setCart` and `setTotal` with objects containing `type: 'remove'` and either the `price` or the `name`. Then use a switch statement to update the data based on the action type. Be sure to return the final state:

hooks-tutorial/src/complicated/Product/Product.js

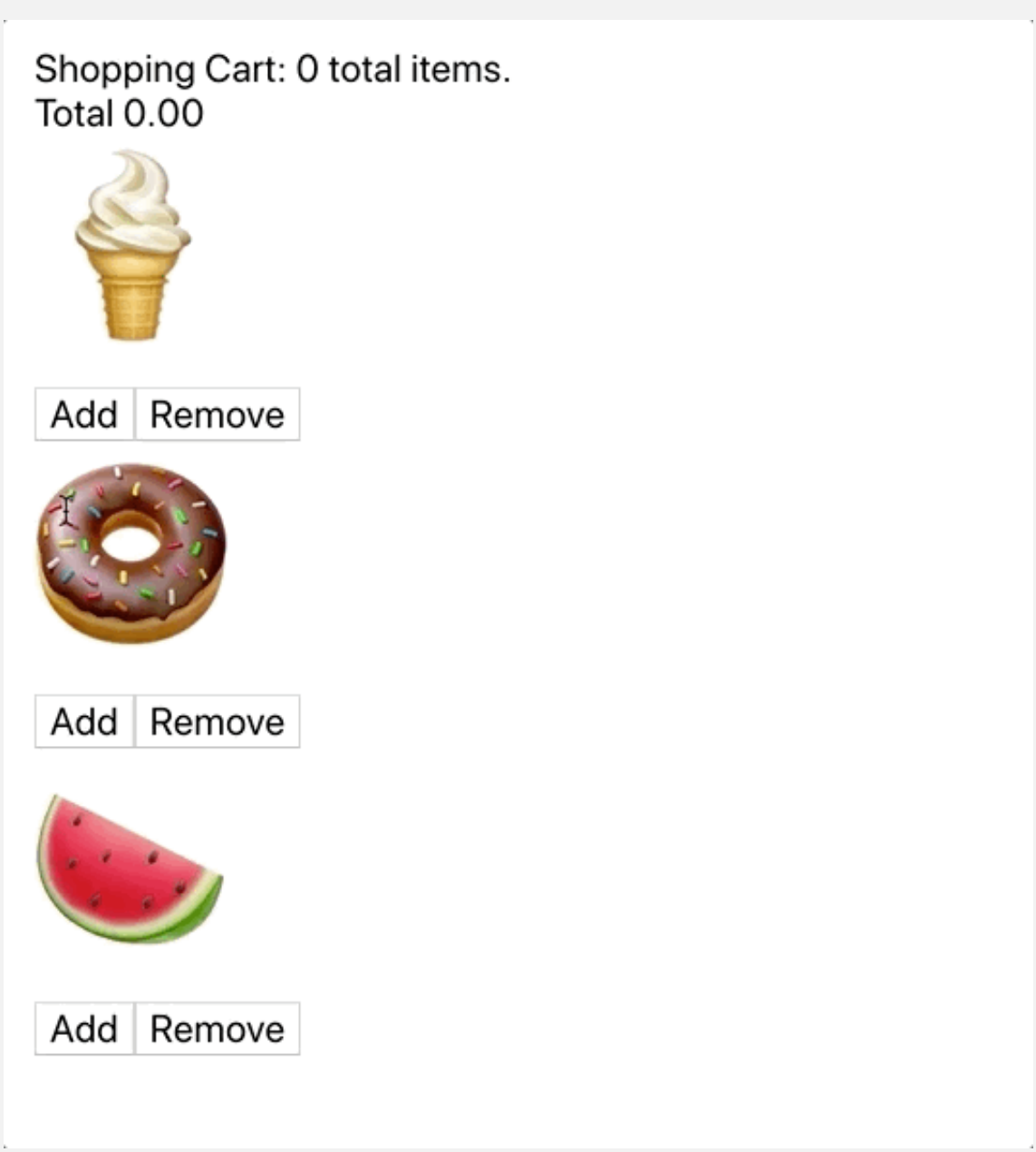
```
import React, { useReducer } from 'react';
import './Product.css';


...


function cartReducer(state, action) {
  switch(action.type) {
    case 'add':
      return [...state, action.name];
    case 'remove':
      const update = [...state];
      update.splice(update.indexOf(action.name), 1);
      return update;
    default:
      return state;
  }
}


function totalReducer(state, action) {
  if(action.type === 'add') {
    return state + action.price;
  }
  return state - action.price
}
```

```
export default function Product() {
  const [cart, setCart] = useReducer(cartReducer, []);
  const [total, setTotal] = useReducer(totalReducer, 0);

  function add(product) {
    const { name, price } = product;
    setCart({ name, type: 'add' });
    setTotal({ price, type: 'add' });
  }

  function remove(product) {
    const { name, price } = product;
    setCart({ name, type: 'remove' });
    setTotal({ price, type: 'remove' });
  }

  return(
    <div className="wrapper">
      <div>
        Shopping Cart: {cart.length} total items.
      </div>
      <div>Total: {getTotal(total)}</div>

      <div>
        {products.map(product => (
          <div key={product.name}>
```

```
            <div className="product">
              <span role="img" aria-label={product.name}>{produ
            </div>
            <button onClick={() => add(product)}>Add</button>
            <button onClick={() => remove(product)}>Remove</but
          </div>
        ))}
      </div>
    </div>
  )
}
```

As you work on your code, take care not to directly mutate the state in the reducer functions. Instead, make a copy before `splicing` out the object. Also note it is a best practice to add a `default` action on a switch statement in order to account for unforeseen edge cases. In this, case just return the object. Other options for the `default` are throwing an error or falling back to an action such as add or remove.

After making the changes, save the file. When the browser refreshes, you'll be able to add and remove items:



`Remove items`

There is still a subtle bug left in this product. In the `remove` method, you can subtract from a price even if the item is not in the cart. If you click **Remove** on the ice cream without adding it to your cart, your displayed total will be **-5.00**.

You can fix this bug by checking that an item exists before you subtract it, but a more efficient way is to minimize the different pieces of state by only saving related data in one place. In other words, try to avoid double references to the same data, in this case, the product. Instead, store the raw data in one state variable—the whole product object—then perform the calculations using that data.

Refactor the component so that the `add()` function passes the whole product to the reducer and the `remove()` function removes the whole object. The `getTotal` method will use the cart, and so you can delete the `totalReducer` function. Then you can pass the cart to `getTotal()`, which you can refactor to reduce the array to a single value:

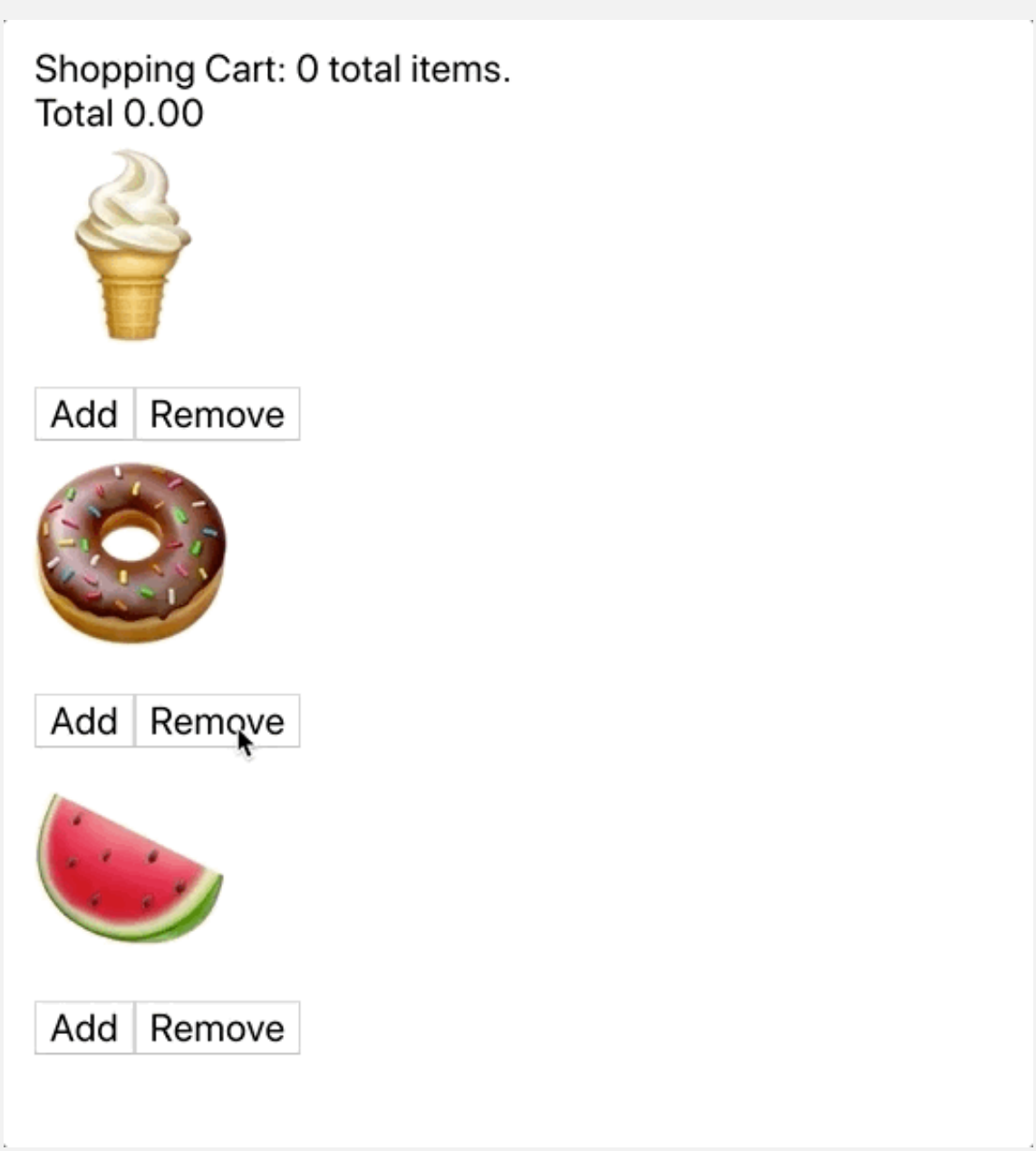
hooks-tutorial/src/component/Product/Product.js

```
import React, { useReducer } from 'react';
import './Product.css';

const currencyOptions = {
  minimumFractionDigits: 2,
  maximumFractionDigits: 2,
}

function getTotal(cart) {
  const total = cart.reduce((totalCost, item) => totalCost + it
  return total.toLocaleString(undefined, currencyOptions)
}

...

function cartReducer(state, action) {
  switch(action.type) {
    case 'add':
      return [...state, action.product];
    case 'remove':
      const productIndex = state.findIndex(item => item.name ==
      if(productIndex < 0) {
        return state;
      }
      const update = [...state];
```

```
      update.splice(productIndex, 1)
      return update
    default:
      return state;
  }
}

export default function Product() {
  const [cart, setCart] = useReducer(cartReducer, []);

  function add(product) {
    setCart({ product, type: 'add' });
  }

  function remove(product) {
    setCart({ product, type: 'remove' });
  }

  return(
    <div className="wrapper">
      <div>
        Shopping Cart: {cart.length} total items.
      </div>
      <div>Total: {getTotal(cart)}</div>

      <div>
        {products.map(product => (
```

```
          <div key={product.name}>
            <div className="product">
              <span role="img" aria-label={product.name}>{produ
            </div>
            <button onClick={() => add(product)}>Add</button>
            <button onClick={() => remove(product)}>Remove</but
          </div>
        ))}
      </div>
    </div>
  )
}
```

Save the file. When you do, the browser will refresh and you'll have your final cart:



Add and remove products

By using the `useReducer` Hook, you kept your main component body well-organized and legible, since the complex logic for parsing and splicing the array is outside of the component. You also could move the reducer outside the componet if you wanted to reuse it, or you can create a custom Hook to

use across multiple components. You can make custom Hooks as functions surrounding basic Hooks, such as `useState`, `useReducer`, or `useEffect`.

Hooks give you the chance to move the stateful logic in and out of the component, as opposed to classes, where you are generally bound to the component. This advantage can extend to other components as well. Since Hooks are functions, you can import them into multiple components rather then using inheritance or other complex forms of class composition.

In this step, you learned to set state using the current state. You created a component that updated state using both the `useState` and the `useReducer` Hooks, and you refactored the component to different Hooks to prevent bugs and improve reusability.

Conclusion

Hooks were a major change to React that created a new way to share logic and update components without using classes. Now that you can create components using `useState` and `useReducer`, you have the tools to make complex projects that respond to users and dynamic information. You also have a foundation of knowledge that you can use to explore more complex Hooks or to create custom Hooks.

If you would like to look at more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Share State Across React Components with Context

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

In this tutorial, you'll share state across multiple components using React context. React context is an interface for sharing information with other components without explicitly passing the data as props. This means that you can share information between a parent component and a deeply nested child component, or store site-wide data in a single place and access them anywhere in the application. You can even update data from nested components by providing update functions along with the data.

React context is flexible enough to use as a centralized state management system for your project, or you can scope it to smaller sections of your application. With context, you can share data across the application without any additional third-party tools and with a small amount of configuration. This provides a lighter weight alternative to tools like Redux, which can help with larger applications but may require too much setup for medium-sized projects.

Throughout this tutorial, you'll use context to build an application that use common data sets across different components. To illustrate this, you'll create a website where users can build custom salads. The website will use context to store customer information, favorite items, and custom salads.

You'll then access that data and update it throughout the application without passing the data via props. By the end of this tutorial, you'll learn how to use context to store data at different levels of the project and how to access and update the data in nested components.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.20.1 and npm version 6.14.4. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- A React development environment set up with Create React App, with the non-essential boilerplate removed. To set this up, follow **Step 1 — Creating an Empty Project** of the How To Manage State on React Class Components tutorial. This tutorial will use `state-context-tutorial` as the project name.

- You will also need a basic knowledge of JavaScript, which you can find in How To Code in JavaScript, along with a basic knowledge of HTML and CSS. A useful resource for HTML and CSS is the Mozilla Developer Network.

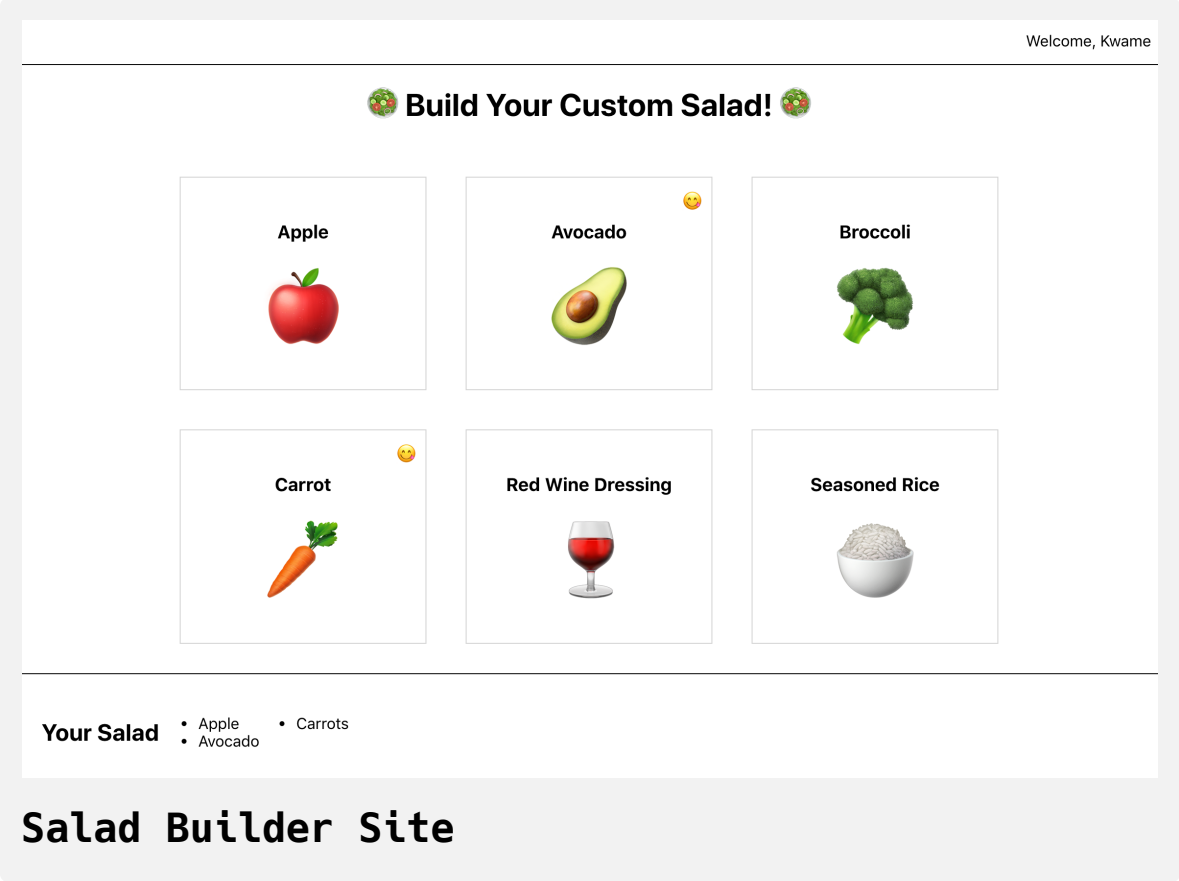
- You will be using React components, the `useState` Hook, and the `useReducer` Hook, which you can learn about in our tutorials How To

Create Custom Components in React and How To Manage State with Hooks on React Components.

Step 1 — Building the Basis for Your Application

In this step, you'll build the general structure of your custom salad builder. You'll create components to display possible toppings, a list of selected toppings, and customer information. As you build the application with static data, you'll find how different pieces of information are used in a variety of components and how to identify pieces of data that would be helpful in a context.

Here's an example of the application you will build:



Notice how there is information that you might need to use across components. For example, the username (which for this sample is **Kwame**) displays user data in a navigation area, but you may also need user information to identify favorite items or for a checkout page. The user information will need to be accessible by any component in the application. Looking at the salad builder itself, each salad ingredient will need to be able to update the **Your Salad** list at the bottom of the screen, so you'll need to store and update that data from a location that is accessible to each component as well.

Start by hard-coding all the data so that you can work out the structure of your app. Later, you'll add in the context starting in the next step. Context provides the most value as applications start to grow, so in this step you'll build several components to show how context works across a component tree. For smaller components or libraries, you can often use wrapping components and lower level state management techniques, like React Hooks and class-based management.

Since you are building a small app with multiple components, install JSS to make sure there won't be any class name conflicts and so that you can add styles in the same file as a component. For more on JSS, see Styling React Components.

Run the following command:

```
npm install react-jss
```

npm will install the component, and when it completes you'll see a message like this:

```
Output
+ react-jss@10.3.0
added 27 packages from 10 contributors, removed 10 packages an
daudited 1973 packages in 15.507s
```

Now that you have JSS installed, consider the different components you'll need. At the top of the page, you'll have a `Navigation` component to store the welcome message. The next component will be the `SaladMaker` itself. This will hold the title along with the builder and the **Your Salad** list at the bottom. The section with ingredients will be a separate component called the `SaladBuilder`, nested inside `SaladMaker`. Each ingredient will be an instance of a `SaladItem` component. Finally, the bottom list will be a component called `SaladSummary`.

Note: The components do not need to be divided this way. As you work on your applications, your structure will change and evolve as you add more functionality. This example is meant to give you a structure to explore how context affects different components in the tree.

Now that you have an idea of the components you'll need, make a directory for each one:

```
mkdir src/components/Navigation
mkdir src/components/SaladMaker
mkdir src/components/SaladItem
mkdir src/components/SaladBuilder
mkdir src/components/SaladSummary
```

Next, build the components from the top down starting with `Navigation`. First, open the component file in a text editor:

```
nano src/components/Navigation/Navigation.js
```

Create a component called `Navigation` and add some styling to give the `Navigation` a border and padding:

state-context-tutorial/src/components/Navigation/Navigation.js

```
import React from 'react';
import { createUseStyles } from 'react-jss';

const useStyles = createUseStyles({
  wrapper: {
    borderBottom: 'black solid 1px',
    padding: [15, 10],
    textAlign: 'right',
  }
});

export default function Navigation() {
  const classes = useStyles();
  return(
    <div className={classes.wrapper}>
      Welcome, Kwame
    </div>
  )
}
```

Since you are using JSS, you can create style objects directly in the component rather than a CSS file. The wrapper `div` will have a padding, a

`solid` `black` border, and align the text to the right with `textAlign`.

Save and close the file. Next, open `App.js`, which is the root of the project:

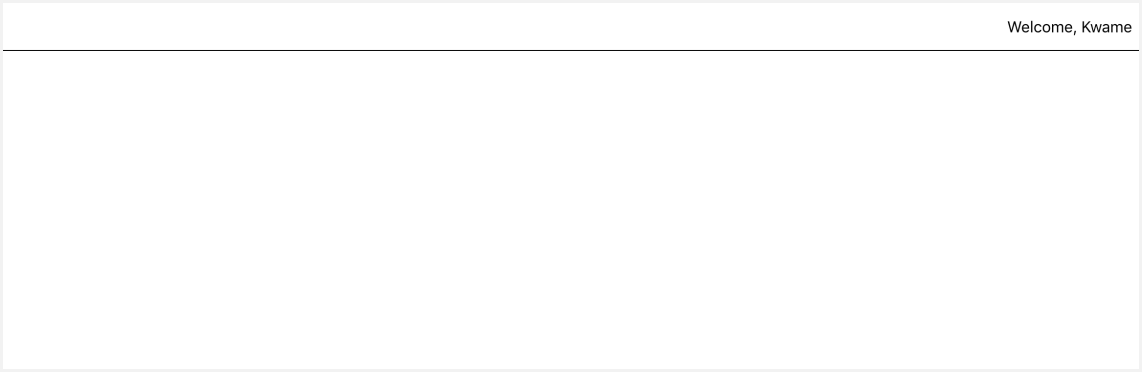
```
nano src/components/App/App.js
```

Import the `Navigation` component and render it inside empty tags by adding the highlighted lines:

state-context-tutorial/src/components/App/App.js

```
import React from 'react';
import Navigation from '../Navigation/Navigation';

function App() {
  return (
    <>
      <Navigation />
    </>
  );
}

export default App;
```

Save and close the file. When you do, the browser will refresh and you'll see the navigation bar:



Think of the navigation bar as a global component, since in this example it's serving as a template component that will be reused on every page.

The next component will be the `SaladMaker` itself. This is a component that will only render on certain pages or in certain states.

Open `SaladMaker.js` in your text editor:

```
nano src/components/SaladMaker/SaladMaker.js
```

Create a component that has an `<h1>` tag with the heading:

state-context-tutorial/src/components/SaladMaker/SaladMaker.js

```
import React from 'react';
import { createUseStyles } from 'react-jss';

const useStyles = createUseStyles({
  wrapper: {
    textAlign: 'center',
  }
});

export default function SaladMaker() {
  const classes = useStyles();
  return(
    <>
      <h1 className={classes.wrapper}>
        <span role="img" aria-label="salad">🥗 </span>
          Build Your Custom Salad!
          <span role="img" aria-label="salad"> 🥗</span>
      </h1>
    </>
  )
}
```

In this code, you are using `textAlign` to center the component on the page. The `role` and `aria-label` attributes of the `span` element will help with accessibility using Accessible Rich Internet Applications (ARIA).

Save and close the file. Open `App.js` to render the component:

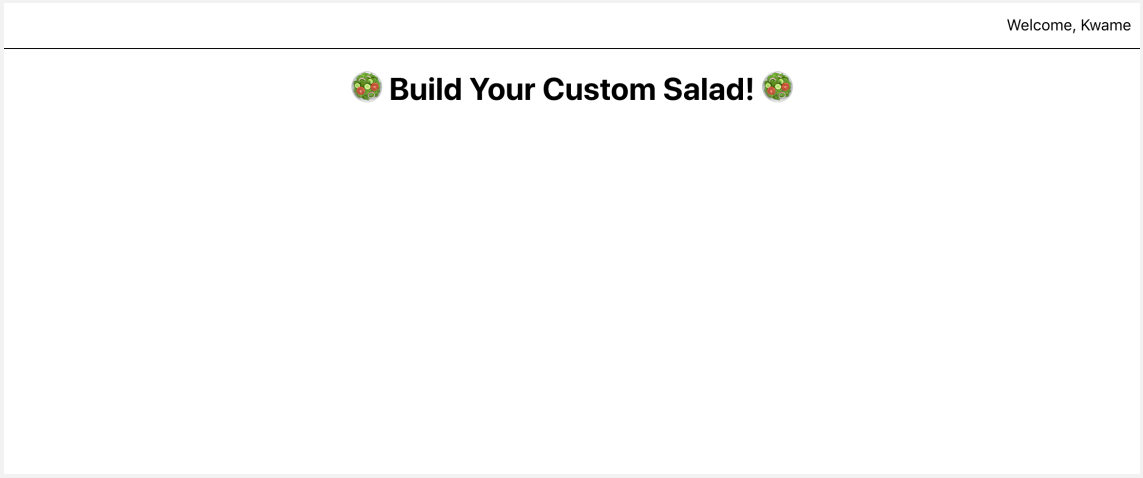
```
nano src/components/App/App.js
```

Import `SaladMaker` and render after the `Navigation` component:

state-context-tutorial/src/components/App/App.js

```
import React from 'react';
import Navigation from '../Navigation/Navigation';
import SaladMaker from '../SaladMaker/SaladMaker';

function App() {
  return (
    <>
      <Navigation />
      <SaladMaker />
    </>
  );
}

export default App;
```

Save and close the file. When you do, the page will reload and you'll see the heading:



Salad Maker Page

Next, create a component called `SaladItem`. This will be a card for each individual ingredient.

Open the file in your text editor:

```
nano src/components/SaladItem/SaladItem.js
```

This component will have three parts: the name of the item, an icon showing if the item is a favorite of the user, and an emoji placed inside a button that will add the item to the salad on click. Add the following lines to `SaladItem.js`:

state-context-tutorial/src/components/SaladItem/SaladItem.js

```
import React from 'react';
import PropTypes from 'prop-types';
import { createUseStyles } from 'react-jss';

const useStyles = createUseStyles({
  add: {
    background: 'none',
    border: 'none',
    cursor: 'pointer',
  },
  favorite: {
    fontSize: 20,
    position: 'absolute',
    top: 10,
    right: 10,
  },
  image: {
    fontSize: 80
  },
  wrapper: {
    border: 'lightgrey solid 1px',
    margin: 20,
    padding: 25,
    position: 'relative',
```

```
    textAlign: 'center',
    textTransform: 'capitalize',
    width: 200,
  }
});

export default function SaladItem({ image, name }) {
  const classes = useStyles();
  const favorite = true;
  return(
    <div className={classes.wrapper}>
        <h3>
          {name}
        </h3>
        <span className={classes.favorite}
          aria-label={favorite ? 'Favorite' : 'Not Favorite'}>
          {favorite ? '😊' : ''}
        </span>
        <button className={classes.add}>
          <span className={classes.image} role="img"
            aria-label={name}>{image}</span>
        </button>
    </div>
  )
}

SaladItem.propTypes = {
  image: PropTypes.string.isRequired,
```

```
  name: PropTypes.string.isRequired,
}
```

The `image` and `name` will be props. The code uses the `favorite` variable and ternary operators to conditionally determine if the `favorite` icon appears or not. The `favorite` variable will later be determined with context as part of the user's profile. For now, set it to `true`. The styling will place the favorite icon in the upper right corner of the card and remove the default border and background on the button. The `wrapper` class will add a small border and transform some of the text. Finally, `PropTypes` adds a weak typing system to provide some enforcement to make sure the wrong prop type is not passed.

Save and close the file. Now, you'll need to render the different items. You'll do that with a component called `SaladBuilder`, which will contain a list of items that it will convert to a series of `SaladItem` components:

Open `SaladBuilder`:

```
nano src/components/SaladBuilder/SaladBuilder.js
```

If this were a production app, this data would often come from an Application Programming Interface (API). But for now, use a hard-coded list of ingredients:

state-context-tutorial/src/components/SaladBuilder/SaladBuilder.js

```
import React from 'react';
import SaladItem from '../SaladItem/SaladItem';

import { createUseStyles } from 'react-jss';

const useStyles = createUseStyles({
  wrapper: {
    display: 'flex',
    flexWrap: 'wrap',
    padding: [10, 50],
    justifyContent: 'center',
  }
});

const ingredients = [
  {
    image: '🍏',
    name: 'apple',
  },
  {
    image: '🥑',
    name: 'avocado',
  },
  {
```

```
    image: '🥦',
    name: 'broccoli',
  },
  {
    image: '🥕',
    name: 'carrot',
  },
  {
    image: '🍷',
    name: 'red wine dressing',
  },
  {
    image: '🍚',
    name: 'seasoned rice',
  },
];

export default function SaladBuilder() {
  const classes = useStyles();
  return(
    <div className={classes.wrapper}>
      {
        ingredients.map(ingredient => (
          <SaladItem
            key={ingredient.name}
            image={ingredient.image}
            name={ingredient.name}
```

```
          />
        ))
      }
    </div>
  )
}
```

This snippet uses the `map()` array method to map over each item in the list, passing the `name` and `image` as props to a `SaladItem` component. Be sure to add a `key` to each item as you map. The styling for this component adds a display of `flex` for the flexbox layout, wraps the components, and centers them.

Save and close the file.

Finally, render the component in `SaladMaker` so it will appear in the page.

Open `SaladMaker`:

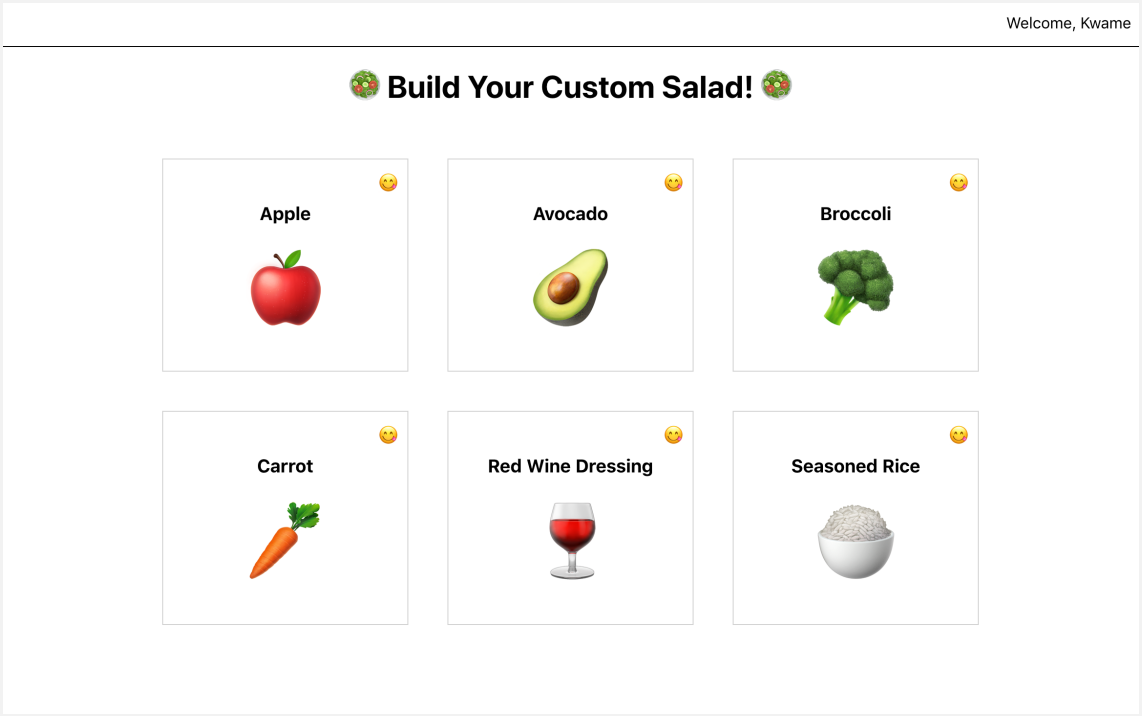
```
nano src/components/SaladMaker/SaladMaker.js
```

Then import `SaladBuilder` and render after the heading:

state-context-tutorial/src/components/SaladMaker/SaladMaker.js

```
import React from 'react';
import { createUseStyles } from 'react-jss';
import SaladBuilder from '../SaladBuilder/SaladBuilder';

const useStyles = createUseStyles({
  wrapper: {
    textAlign: 'center',
  }
});

export default function SaladMaker() {
  const classes = useStyles();
  return(
    <>
      <h1 className={classes.wrapper}>
        <span role="img" aria-label="salad">🥗 </span>
          Build Your Custom Salad!
          <span role="img" aria-label="salad"> 🥗</span>
      </h1>
      <SaladBuilder />
    </>
  )
}
```

Save and close the file. When you do the page will reload and you'll find the content:



Salad Builder with Items

The last step is to add the summary of the salad in progress. This component will show a list of items a user has selected. For now, you'll hard-code the items. You'll update them with context in Step 3.

Open `SaladSummary` in your text editor:

```
nano src/components/SaladSummary/SaladSummary.js
```

The component will be a heading and an unsorted list of items. You'll use flexbox to make them wrap:

state-context-tutorial/src/components/SaladSummary/SaladSummary.jss

```
import React from 'react';
import { createUseStyles } from 'react-jss';

const useStyles = createUseStyles({
  list: {
    display: 'flex',
    flexDirection: 'column',
    flexWrap: 'wrap',
    maxHeight: 50,
    '& li': {
      width: 100
    }
  },
  wrapper: {
    borderTop: 'black solid 1px',
    display: 'flex',
    padding: 25,
  }
});

export default function SaladSummary() {
  const classes = useStyles();
  return(
    <div className={classes.wrapper}>
```

```
      <h2>Your Salad</h2>
      <ul className={classes.list}>
        <li>Apple</li>
        <li>Avocado</li>
        <li>Carrots</li>
      </ul>
    </div>
  )
}
```

Save the file. Then open `SaladMaker` to render the item:

```
nano src/components/SaladMaker/SaladMaker.js
```

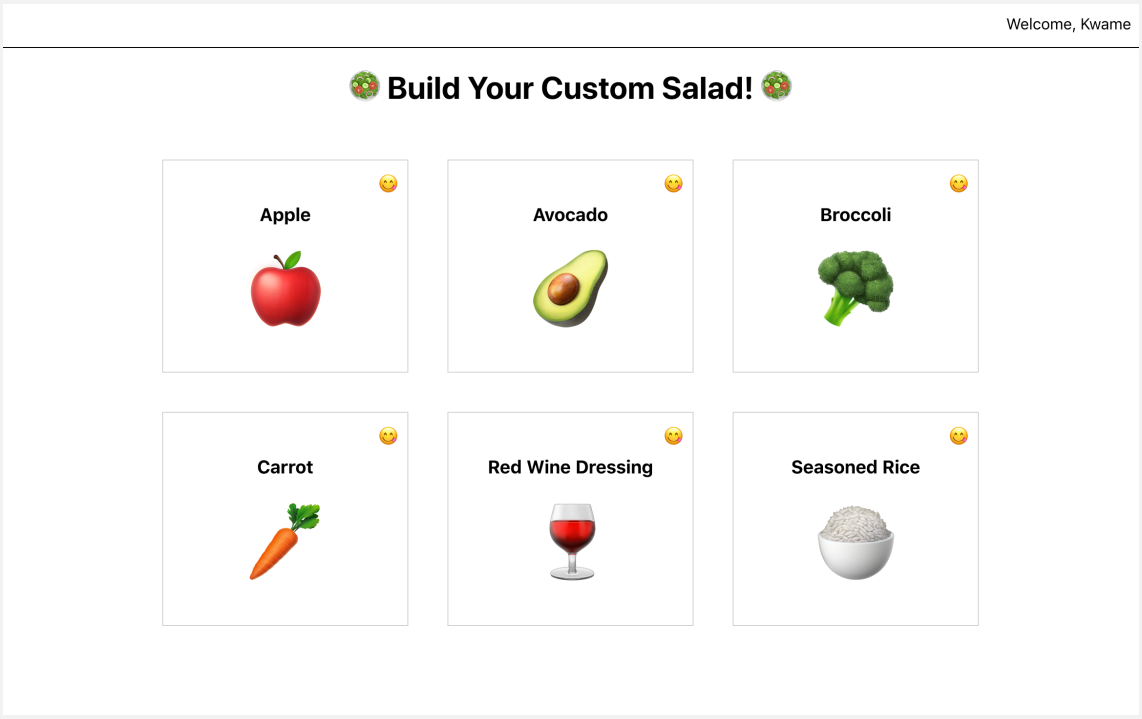
Import and add `SaladSummary` after the `SaladBuilder`:

state-context-tutorial/src/components/SaladMaker/SaladMaker.js

```
import React from 'react';
import { createUseStyles } from 'react-jss';
import SaladBuilder from '../SaladBuilder/SaladBuilder';
import SaladSummary from '../SaladSummary/SaladSummary';

const useStyles = createUseStyles({
  wrapper: {
    textAlign: 'center',
  }
});

export default function SaladMaker() {
  const classes = useStyles();
  return(
    <>
      <h1 className={classes.wrapper}>
        <span role="img" aria-label="salad">🥗 </span>
          Build Your Custom Salad!
          <span role="img" aria-label="salad"> 🥗</span>
      </h1>
      <SaladBuilder />
      <SaladSummary />
    </>
```

```
  )
}
```

Save and close the file. When you do, the page will refresh and you'll find the full application:



Salad Builder Site

There is shared data throughout the application. The `Navigation` component and the `SaladItem` component both need to know something about the user: their name and their list of favorites. The `SaladItem` also needs to update data that is accessible in the `SaladSummary` component. The

components share common ancestors, but passing the data down through the tree would be difficult and error prone.

That's where context comes in. You can declare the data in a common parent and then access later without explicitly passing it down the hierarchy of components.

In this step, you created an application to allow the user to build a salad from a list of options. You created a set of components that need to access or update data that is controlled by other components. In the next step, you'll use context to store data and access it in child components.

Step 2 — Providing Data from a Root Component

In this step, you'll use context to store the customer information at the root of the component. You'll create a custom context, then use a special wrapping component called a `Provider` that will store the information at the root of the project. You'll then use the `useContext` Hook to connect with the provider in nested components so you can display the static information. By the end of this step, you'll be able to provide centralized stores of information and use information stored in a context in many different components.

Context at its most basic is an interface for sharing information. Many applications have some universal information they need to share across the application, such as user preferences, theming information, and site-wide application changes. With context, you can store that information at the root

level then access it anywhere. Since you set the information in a parent, you know it will always be available and it will always be up-to-date.

To add a context, create a new directory called `User`:

```
mkdir src/components/User
```

`User` isn't going to be a traditional component, in that you are going to use it both as a component and as a piece of data for a special Hook called `useContext`. For now, keep the flat file structure, but if you use a lot of contexts, it might be worth moving them to a different directory structure.

Next, open up `User.js` in your text editor:

```
nano src/components/User/User.js
```

Inside the file, import the `createContext` function from React, then execute the function and export the result:

state-context-tutorial/src/components/User/User.js

```
import { createContext } from 'react';

const UserContext = createContext();
export default UserContext;
```

By executing the function, you have registered the context. The result, `UserContext`, is what you will use in your components.

Save and close the file.

The next step is to apply the context to a set of elements. To do that, you will use a component called a `Provider`. The `Provider` is a component that sets the data and then wraps some child components. Any wrapped child components will have access to data from the `Provider` with the `useContext` Hook.

Since the user data will be constant across the project, put it as high up the component tree as you can. In this application, you will put it at the root level in the `App` component:

Open up `App`:

```
nano src/components/App/App.js
```

Add in the following highlighted lines of code to import the context and pass along the data:

state-context-tutorial/src/components/App/App.js

```
import React from 'react';
import Navigation from '../Navigation/Navigation';
import SaladMaker from '../SaladMaker/SaladMaker';
import UserContext from '../User/User';

const user = {
  name: 'Kwame',
  favorites: [
    'avocado',
    'carrot'
  ]
}

function App() {
  return (
    <UserContext.Provider value={user}>
      <Navigation />
      <SaladMaker />
    </UserContext.Provider>
  );
}

export default App;
```

In a typical application, you would fetch the user data or have it stored during a server-side render. In this case, you hard-coded some data that you might receive from an API. You created an object called `user` that holds the username as a string and an array of favorite ingredients.

Next, you imported the `UserContext`, then wrapped `Navigation` and `Salad Maker` with a component called the `UserContext.Provider`. Notice how in this case `UserContext` is acting as a standard React component. This component will take a single prop called `value`. That prop will be the data you want to share, which in this case is the `user` object.

Save and close the file. Now the data is available throughout the application. However, to use the data, you'll need to once again import and access the context.

Now that you have set context, you can start replacing hard-coded data in your component with dynamic values. Start by replacing the hard-coded name in `Navigation` with the user data you set with `UserContext.Provider`.

Open `Navigation.js`:

```
nano src/components/Navigation/Navigation.js
```

Inside of `Navigation`, import the `useContext` Hook from React and `UserContext` from the component directory. Then call `useContext` using `UserContext` as an argument. Unlike the `UserContext.Provider`, you do not need to render `UserContext` in the JSX. The Hook will return the data that you

provided in the `value` prop. Save the data to a new variable called `user`, which is an object containing `name` and `favorites`. You can then replace the hard-coded name with `user.name`:

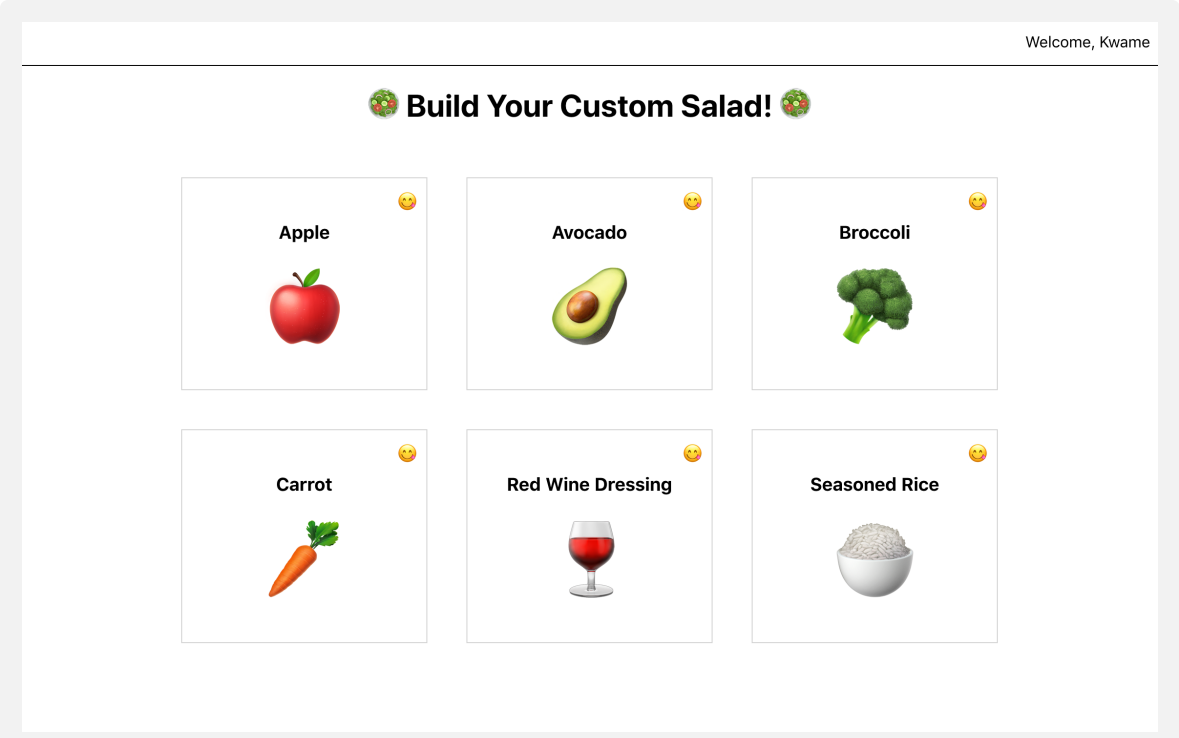
state-context-tutorial/src/components/Navigation/Navigation.js

```
import React, { useContext } from 'react';
import { createUseStyles } from 'react-jss';

import UserContext from '../User/User';

const useStyles = createUseStyles({
  wrapper: {
    outline: 'black solid 1px',
    padding: [15, 10],
    textAlign: 'right',
  }
});

export default function Navigation() {
  const user = useContext(UserContext);
  const classes = useStyles();
  return(
    <div className={classes.wrapper}>
      Welcome, {user.name}
    </div>
  )
}
```

`UserContext` worked as a component in `App.js`, but here you are using it more as a piece of data. However, it can still act as a component if you would like. You can access the same data by using a `Consumer` that is part of the `UserContext`. You retrieve the data by adding `UserContext.Consumer` to your JSX, then use a function as a child to access the data.

While it's possible to use the `Consumer` component, using Hooks can often be shorter and easier to read, while still providing the same up-to-date information. This is why this tutorial uses the Hooks approach.

Save and close the file. When you do, the page will refresh and you'll see the same name. But this time it has updated dynamically:



Salad Builder Site

In this case the data didn’t travel across many components. The component tree that represents the path that the data traveled would look like this:

```
| UserContext.Provider
  | Navigation
```

You could pass this username as a prop, and at this scale that could be an effective strategy. But as the application grows, there’s a chance that the `Navigation` component will move. There may be a component called `Header` that wraps the `Navigation` component and another component such as a `TitleBar`, or maybe you’ll create a `Template` component and then nest the `Navigation` in there. By using context, you won’t have to refactor `Navigation` as long as the `Provider` is up the tree, making refactoring easier.

The next component that needs user data is the `SaladItem` component. In the `SaladItem` component, you’ll need the user’s array of favorites. You’ll conditionally display the emoji if the ingredient is a favorite of the user.

Open `SaladItem.js`:

```
nano src/components/SaladItem/SaladItem.js
```

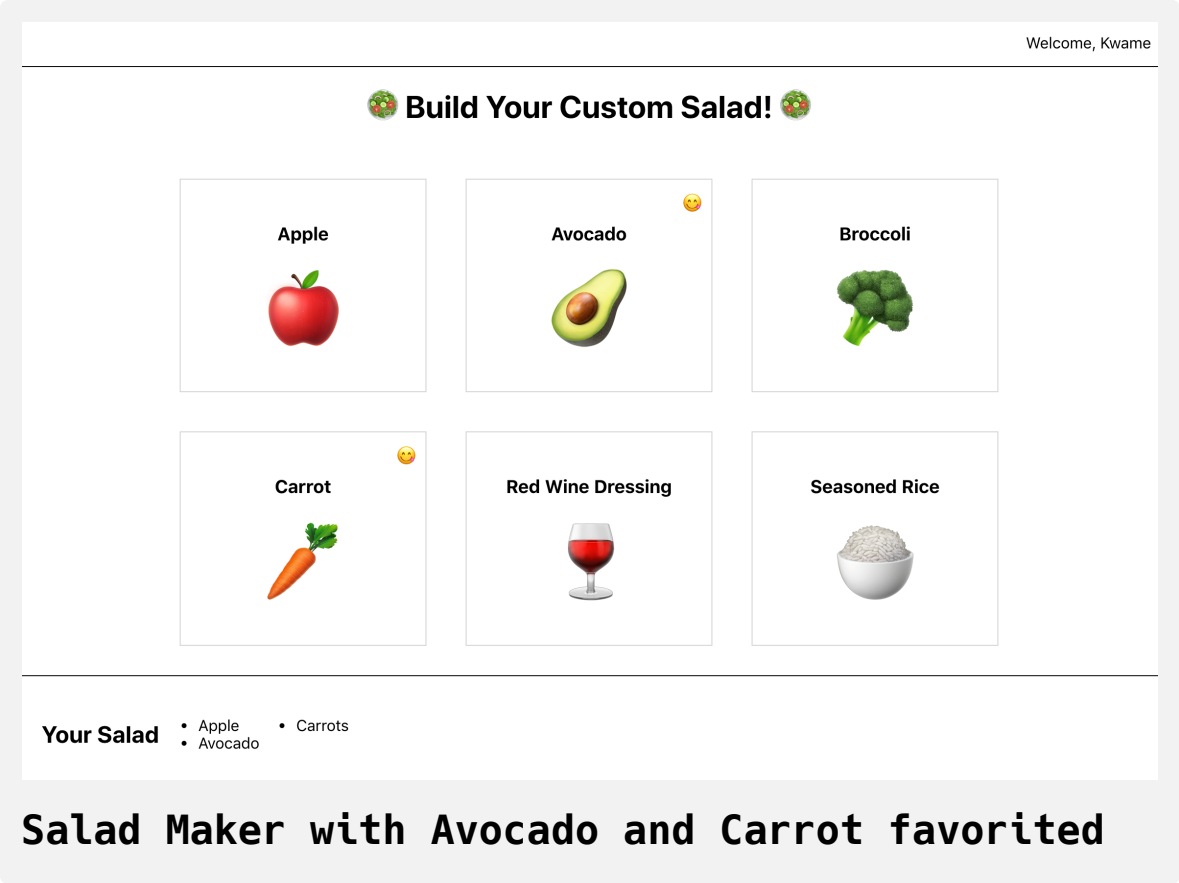
Import `useContext` and `UserContext`, then call `useContext` with `UserContext`. After that, check to see if the ingredient is in the `favorites` array using the `includes` method:

state-context-tutorial/src/components/SaladItem/SaladItem.js

```
import React, { useContext } from 'react';
import PropTypes from 'prop-types';
import { createUseStyles } from 'react-jss';

import UserContext from '../User/User';

const useStyles = createUseStyles({
...
});

export default function SaladItem({ image, name }) {
  const classes = useStyles();
  const user = useContext(UserContext);
  const favorite = user.favorites.includes(name);
  return(
    <div className={classes.wrapper}>
        <h3>
          {name}
        </h3>
        <span className={classes.favorite}
          aria-label={favorite ? 'Favorite' : 'Not Favorite'}>
          {favorite ? '😊' : ''}
        </span>
        <button className={classes.add}>
```

```
          <span className={classes.image} role="img"
            aria-label={name}>{image}</span>

        </button>

    </div>

  )
}


SaladItem.propTypes = {
  image: PropTypes.string.isRequired,
  name: PropTypes.string.isRequired,
}
```

Save and close the file. When you do, the browser will refresh and you'll see that only the favorite items have the emoji:



Salad Maker with Avocado and Carrot favorited

Unlike `Navigation`, the context is traveling much farther. The component tree would look something like this:

```
| User.Provider
  | SaladMaker
    | SaladBuilder
      | SaladItem
```

The information skipped over two intermediary components without any props. If you had to pass the data as a prop all the way through the tree, it would be a lot of work and you'd risk having a future developer refactor the code and forget to pass the prop down. With context, you can be confident the code will work as the application grows and evolves.

In this step, you created a context and used a `Provider` to set the data in the component tree. You also accessed context with the `useContext` Hook and used context across multiple components. This data was static and thus never changed after the initial set up, but there are going to be times when you need to share data and also modify the data across multiple components. In the next step, you'll update nested data using context.

Step 3 — Updating Data from Nested Components

In this step, you'll use context and the `useReducer` Hook to create dynamic data that nested components can consume and update. You'll update your `SaladItem` components to set data that the `SaladSummary` will use and display. You'll also set context providers outside of the root component. By the end of this step, you'll have an application that can use and update data across several components and you'll be able to add multiple context providers at different levels of an application.

At this point, your application is displaying user data across multiple components, but it lacks any user interaction. In the previous step, you used context to share a single piece of data, but you can also share a collection of data, including functions. That means you can share data and also share the function to update the data.

In your application, each `SaladItem` needs to update a shared list. Then your `SaladSummary` component will display the items the user has selected and add it to the list. The problem is that these components are not direct

descendants, so you can’t pass the data and the update functions as props. But they do share a common parent: `SaladMaker`.

One of the big differences between context and other state management solutions such as Redux is that context is not intended to be a central store. You can use it multiple times throughout an application and initiate it at the root level or deep in a component tree. In other words, you can spread your contexts throughout the application, creating focused data collections without worrying about conflicts.

To keep context focused, create `Providers` that wrap the nearest shared parent when possible. In this case, that means, rather than adding another context in `App`, you will add the context in the `SaladMaker` component.

Open `SaladMaker`:

```
nano src/components/SaladMaker/SaladMaker.js
```

Then create and export a new context called `SaladContext`:

state-context-tutorial/src/components/SaladMaker/SaladMaker.js

```
import React, { createContext } from 'react';
import { createUseStyles } from 'react-jss';
import SaladBuilder from '../SaladBuilder/SaladBuilder';
import SaladSummary from '../SaladSummary/SaladSummary';

const useStyles = createUseStyles({
  wrapper: {
    textAlign: 'center',
  }
});

export const SaladContext = createContext();

export default function SaladMaker() {
  const classes = useStyles();
  return(
    <>
      <h1 className={classes.wrapper}>
        <span role="img" aria-label="salad">🥙 </span>
          Build Your Custom Salad!
          <span role="img" aria-label="salad"> 🥙</span>
      </h1>
      <SaladBuilder />
      <SaladSummary />
```

```
    </>
  )
}
```

In the previous step, you made a separate component for your context, but in this case you are creating it in the same file that you are using it. Since `User` does not seem related directly to the `App`, it might make more sense to keep them separate. However, since the `SaladContext` is tied closely to the `SaladMaker` component, keeping them together will create more readable code.

In addition, you could create a more generic context called `OrderContext`, which you could reuse across multiple components. In that case, you'd want to make a separate component. For now, keep them together. You can always refactor later if you decide to shift to another pattern.

Before you add the `Provider` think about the data that you want to share. You'll need an array of items and a function for adding the items. Unlike other centralized state management tools, context does not handle updates to your data. It merely holds the data for use later. To update data, you'll need to use other state management tools such as Hooks. If you were collecting data for the same component, you'd use either the `useState` or `useReducer` Hooks. If you are new to these Hooks, check out How To Manage State with Hooks on React Components.

The `useReducer` Hook is a good fit since you'll need to update the most recent state on every action.

Create a `reducer` function that adds a new item to a `state` array, then use the `useReducer` Hook to create a `salad` array and a `setSalad` function:

state-context-tutorial/src/components/SaladMaker/SaladMaker.js

```
import React, { useReducer, createContext } from 'react';
import { createUseStyles } from 'react-jss';
import SaladBuilder from '../SaladBuilder/SaladBuilder';
import SaladSummary from '../SaladSummary/SaladSummary';

const useStyles = createUseStyles({
  wrapper: {
    textAlign: 'center',
  }
});

export const SaladContext = createContext();

function reducer(state, item) {
  return [...state, item]
}

export default function SaladMaker() {
  const classes = useStyles();
  const [salad, setSalad] = useReducer(reducer, []);
  return(
    <>
      <h1 className={classes.wrapper}>
        <span role="img" aria-label="salad">🥗 </span>
```

```
          Build Your Custom Salad!
          <span role="img" aria-label="salad">🥙</span>
      </h1>
      <SaladBuilder />
      <SaladSummary />
    </>
  )
}
```

Now you have a component that contains the `salad` data you want to share, a function called `setSalad` to update the data, and the `SaladContext` to share the data in the same component. At this point, you need to combine them together.

To combine, you'll need to create a `Provider`. The problem is that the `Provider` takes a single `value` as a prop. Since you can't pass `salad` and `setSalad` individually, you'll need to combine them into an object and pass the object as the `value`:

state-context-tutorial/src/components/SaladMaker/SaladMaker.js

```
import React, { useReducer, createContext } from 'react';
import { createUseStyles } from 'react-jss';
import SaladBuilder from '../SaladBuilder/SaladBuilder';
import SaladSummary from '../SaladSummary/SaladSummary';

const useStyles = createUseStyles({
  wrapper: {
    textAlign: 'center',
  }
});

export const SaladContext = createContext();

function reducer(state, item) {
  return [...state, item]
}

export default function SaladMaker() {
  const classes = useStyles();
  const [salad, setSalad] = useReducer(reducer, []);
  return(
    <SaladContext.Provider value={{ salad, setSalad }}>
      <h1 className={classes.wrapper}>
        <span role="img" aria-label="salad">🥗 </span>
```

```
            Build Your Custom Salad!
            <span role="img" aria-label="salad">🥙</span>
        </h1>
        <SaladBuilder />
        <SaladSummary />
      </SaladContext.Provider>
    )
}
```

Save and close the file. As with `Navigation`, it may seem unnecessary to create a context when the `SaladSummary` is in the same component as the context. Passing `salad` as a prop is perfectly reasonable, but you may end up refactoring it later. Using context here keeps the information together in a single place.

Next, go into the `SaladItem` component and pull the `setSalad` function out of the context.

Open the component in a text editor:

```
nano src/components/SaladItem/SaladItem.js
```

Inside `SaladItem`, import the context from `SaladMaker`, then pull out the `setSalad` function using destructuring. Add a click event to the button that will call the `setSalad` function. Since you want a user to be able to add an

item multiple times, you'll also need to create a unique id for each item so that the `map` function will be able to assign a unique `key`:

state-context-tutorial/src/components/SaladItem/SaladItem.js

```
import React, { useReducer, useContext } from 'react';
import PropTypes from 'prop-types';
import { createUseStyles } from 'react-jss';

import UserContext from '../User/User';
import { SaladContext } from '../SaladMaker/SaladMaker';

const useStyles = createUseStyles({
...
});

const reducer = key => key + 1;
export default function SaladItem({ image, name }) {
  const classes = useStyles();
  const { setSalad } = useContext(SaladContext)
  const user = useContext(UserContext);
  const favorite = user.favorites.includes(name);
  const [id, updateId] = useReducer(reducer, 0);
  function update() {
    setSalad({
      name,
      id: `${name}-${id}`
    })
    updateId();
```

```
  };
  return(
    <div className={classes.wrapper}>
        <h3>
          {name}
        </h3>
        <span className={classes.favorite}
          aria-label={favorite ? 'Favorite' : 'Not Favorite'}>
          {favorite ? '😊' : ''}
        </span>
        <button className={classes.add} onClick={update}>
          <span className={classes.image} role="img"
            aria-label={name}>{image}</span>
        </button>
    </div>
  )
}
...
```

To make the unique id, you'll use the `useReducer` Hook to increment a value on every click. For the first click, the id will be `0`; the second will be `1`, and so on. You'll never display this value to the user; this will just create a unique value for the mapping function later.

After creating the unique id, you created a function called `update` to increment the id and to call `setSalad`. Finally, you attached the function to the button with the `onClick` prop.

Save and close the file. The last step is to pull the dynamic data from the context in the `SaladSummary`.

Open `SaladSummary`:

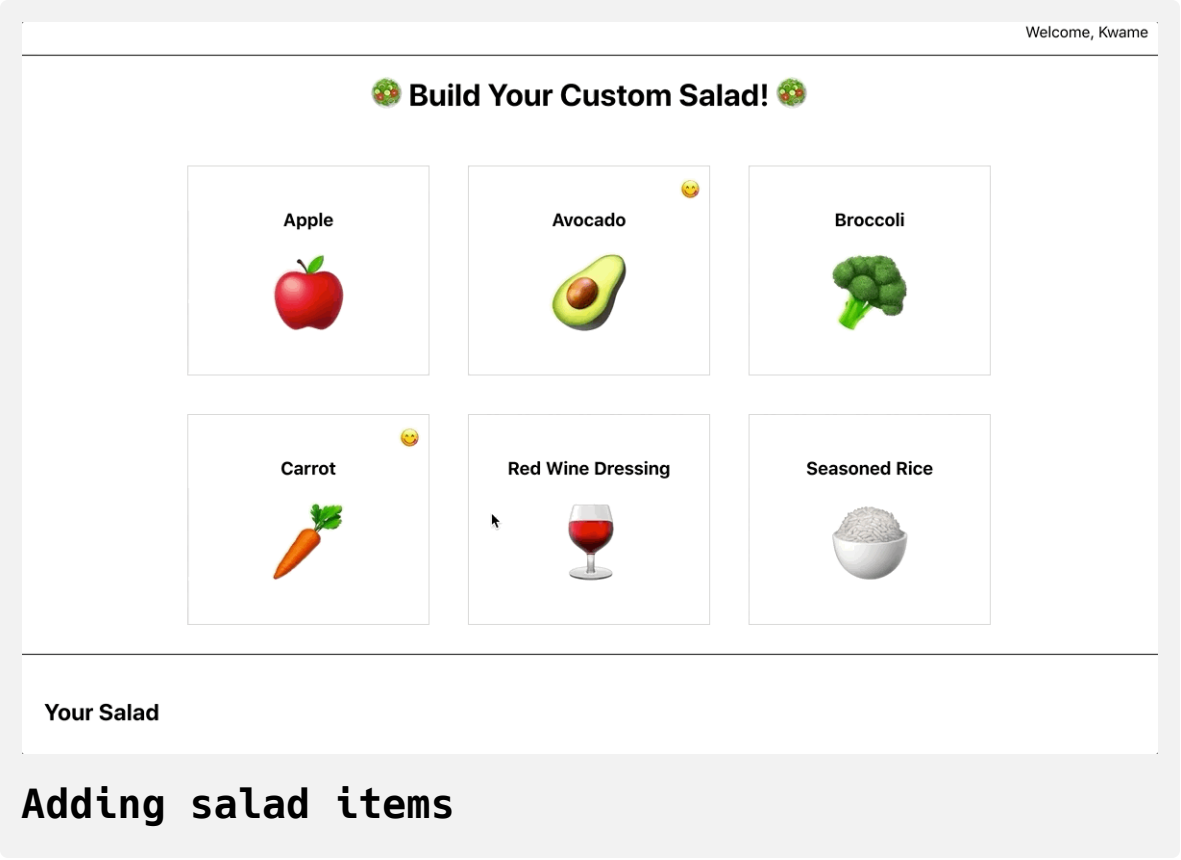
```
nano src/components/SaladSummary/SaladSummary.js
```

Import the `SaladContext` component, then pull out the `salad` data using destructuring. Replace the hard-coded list items with a function that maps over `salad`, converting the objects to `<li>` elements. Be sure to use the `id` as the `key`:

state-context-tutorial/src/components/SaladSummary/SaladSummary.js

```
import React, { useContext } from 'react';
import { createUseStyles } from 'react-jss';

import { SaladContext } from '../SaladMaker/SaladMaker';

const useStyles = createUseStyles({
...
});

export default function SaladSummary() {
  const classes = useStyles();
  const { salad } = useContext(SaladContext);
  return(
    <div className={classes.wrapper}>
      <h2>Your Salad</h2>
      <ul className={classes.list}>
        {salad.map(({ name, id }) => (<li key={id}>{name}</li>)
      </ul>
    </div>
  )
}
```

Save and close the file. When you do, you will be able to click on items and it will update the summary:



Notice how the context gave you the ability to share and update data in different components. The context didn't update the items itself, but it gave you a way to use the `useReducer` Hook across multiple components. In addition, you also had the freedom to put the context lower in the tree. It may seem like it's best to always keep the context at the root, but by keeping the context lower, you don't have to worry about unused state sticking around in a central store. As soon as you unmount a component, the data disappears. That can be a problem if you ever want to save the data, but in that case, you just need to raise the context up to a higher parent.

Another advantage of using context lower in your application tree is that you can reuse a context without worrying about conflicts. Suppose you had a larger app that had a sandwich maker and a salad maker. You could create a generic context called `OrderContext` and then you could use it at multiple points in your component without worrying about data or name conflicts. If you had a `SaladMaker` and a `SandwichMaker`, the tree would look something like this:

```
| App
  | Salads
    | OrderContext
      | SaladMaker
  | Sandwiches
    | OrderContext
      | SandwichMaker
```

Notice that `OrderContext` is there twice. That's fine, since the `useContext` Hook will look for the nearest provider.

In this step you shared and updated data using context. You also placed the context outside the root element so it's close to the components that need the information without cluttering a root component. Finally, you combined context with state management Hooks to create data that is dynamic and accessible across several components.

Conclusion

Context is a powerful and flexible tool that gives you the ability to store and use data across an application. It gives you the ability to handle distributed

data with built-in tools that do not require any additional third party installation or configuration.

Creating reusable contexts is important across a variety of common components such as forms that need to access data across elements or tab views that need a common context for both the tab and the display. You can store many types of information in contexts including themes, form data, alert messages, and more. Context gives you the freedom to build components that can access data without worrying about how to pass data through intermediary components or how to store data in a centralized store without making the store too large.

If you would like to look at more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Debug React Components Using React Developer Tools

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

Since React apps are made to scale and grow quickly, it's easy for subtle bugs to infiltrate your code. The React Developer Tools browser extension can help you track down these bugs by giving you more insight into the current state for each component. React Developer Tools gives you an interface for exploring the React component tree along with the current props, state, and context for individual components. React Developer Tools also lets you determine which components are re-rendering and can generate graphs to show how long individual components take to render. You can use this information to track down inefficient code or to optimize data-heavy components.

This tutorial begins by installing the React Developer Tools browser extension. You'll then build a text analyzer as a test application, which will take a block of text and display information such as word count, character count, and character usage. Finally, you will use React Developer Tools to explore the text analyzer's components and keep track of the changing props and context. The examples will use the Chrome browser, but you can also use the plugin for Firefox.

By the end of this tutorial, you'll be able to start using the React Developer Tools to debug and explore any React project.

Prerequisites

- To use the Chrome React Developer Tools extension, you will need to download and install the Google Chrome web browser or the open-source Chromium web browser. You can also following along using the React Developer Tools FireFox plugin for the FireFox web browser.

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.22.0 and npm version 6.14.6. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- A React development environment set up with Create React App. To set this up, follow **Step 1 — Creating an Empty Project** of the How To Manage State on React Class Components tutorial, which will remove the non-essential boilerplate. This tutorial will use `debug-tutorial` as the project name.

- You will be using React components and Hooks in this tutorial, including the `useState` and context Hooks. You can learn about components and Hooks in our tutorials How To Create Custom Components in React, How To Manage State with Hooks on React

Components, and How To Share State Across React Components with Context.

- You will also need a basic knowledge of JavaScript and HTML, which you can find in our How To Build a Website with HTML series and in How To Code in JavaScript. Basic knowledge of CSS would also be useful, which you can find at the Mozilla Developer Network.

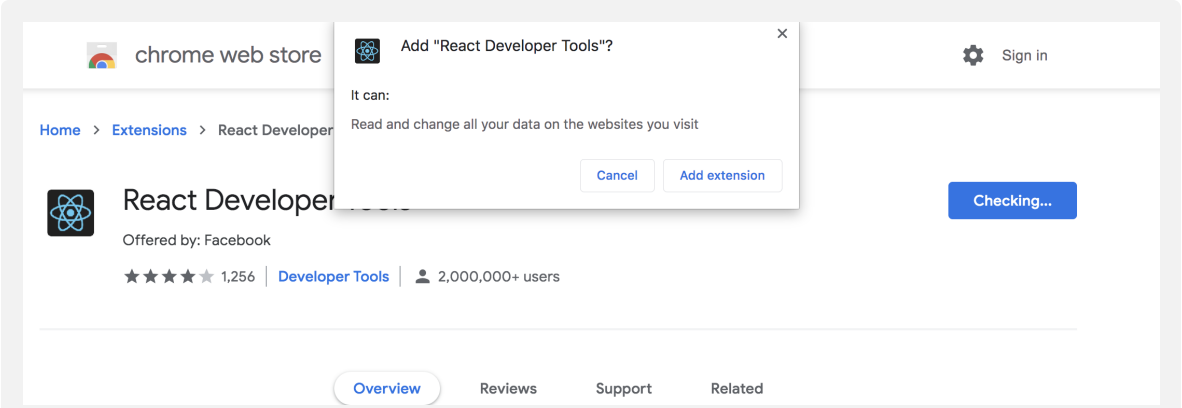
Step 1 — Installing the React Developer Tools Extension

In this step, you'll install the React Developer Tools broswer extension in Chrome. You'll use the developer tools in the Chrome JavaScript console to explore the component tree of the `debug-tutorial` project you made in the Prerequisites. This step will use Chrome, but the steps will be nearly identical for installing the React Developer Tools as an add-on in Firefox.

By the end of this step, you'll have the React Developer Tools installed in your browser and you'll be able to explore and filter components by name.

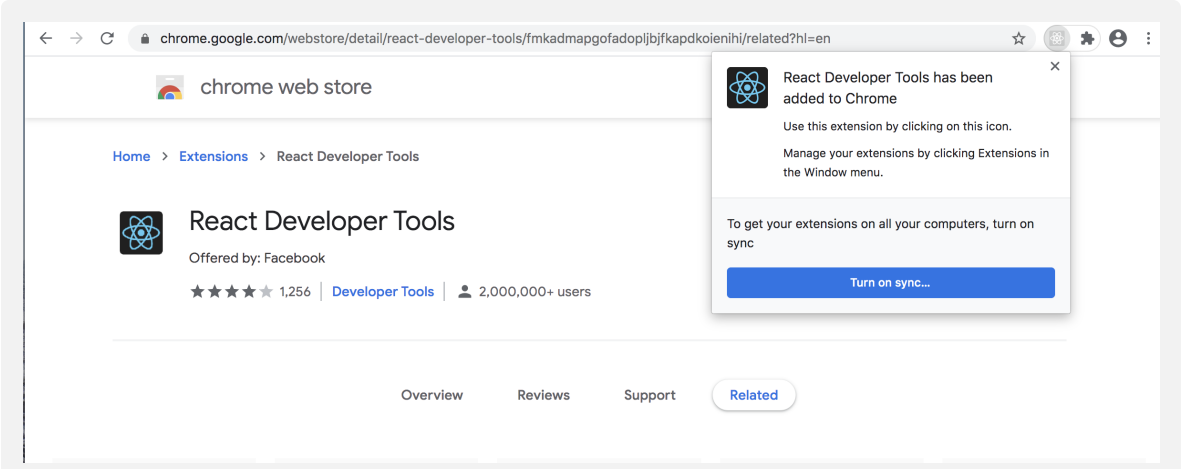
The React Developer Tools is a plugin for the Chrome and Firefox browser. When you add the extension, you are adding additional tools to the developer console. Visit the Chrome plugin page for React Developer Tools to install the extension.

Click on the **Add to Chrome** button. Then click on the **Add extension** button to confirm:



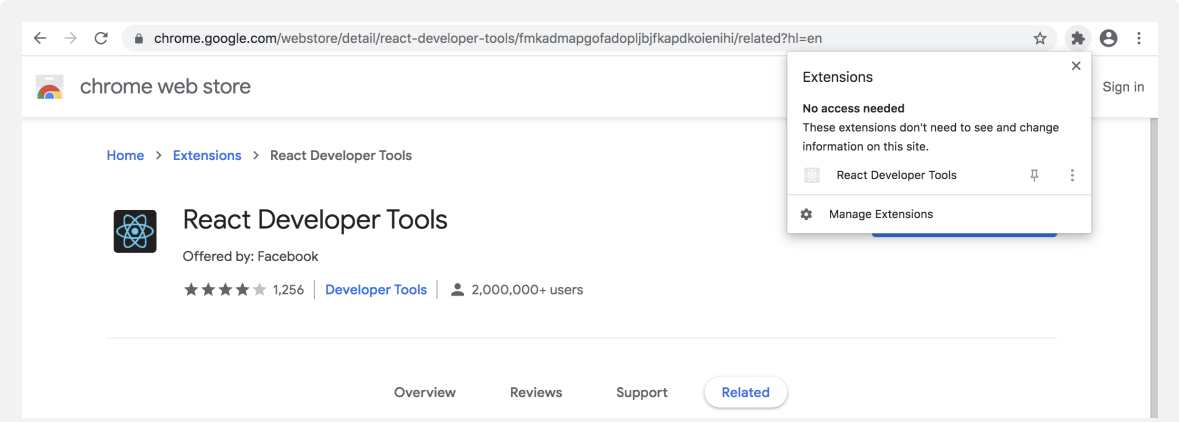
Chrome add extension button

Chrome will install the extension, and a success message and a new icon will appear in the upper right corner of your browser next to the address bar:



Chrome success message

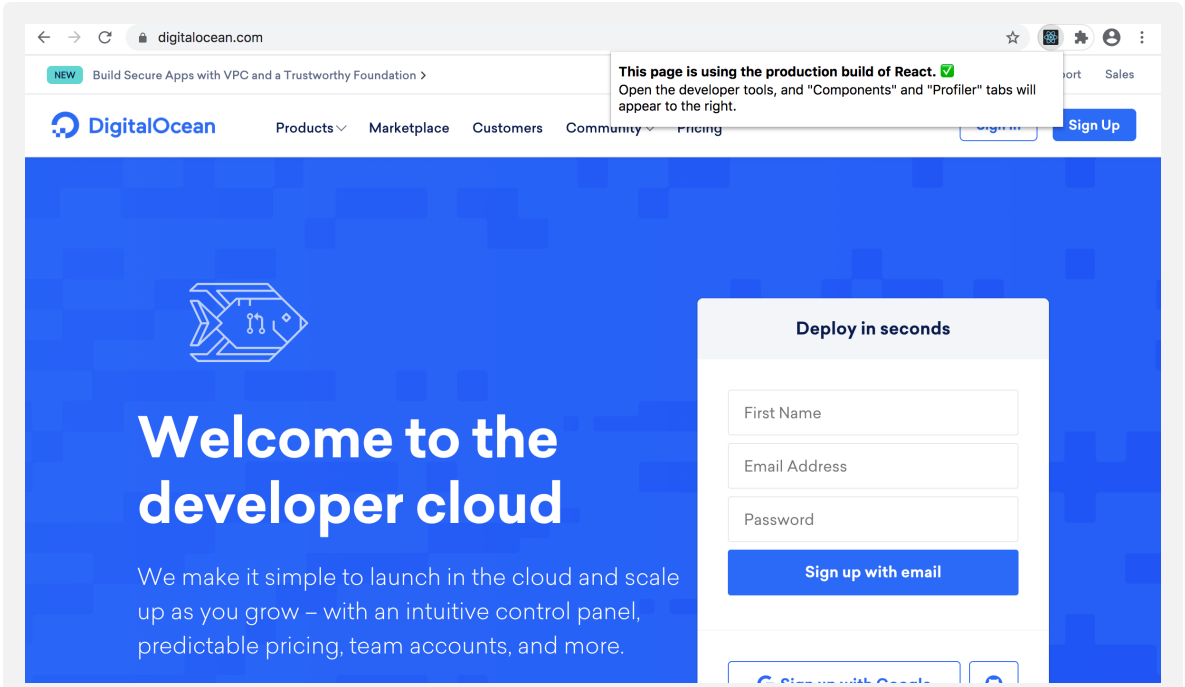
If the icon does not appear, you can add it by clicking on the puzzle piece, then clicking on the pushpin icon by the React Developer Tools:



Pin extension

When you are on a page that does not have any React components, the icon will appear gray. However, if you are on a page with React components, the icon will appear blue and green. If you click on the icon, it will indicate that the application is running a production version of React.

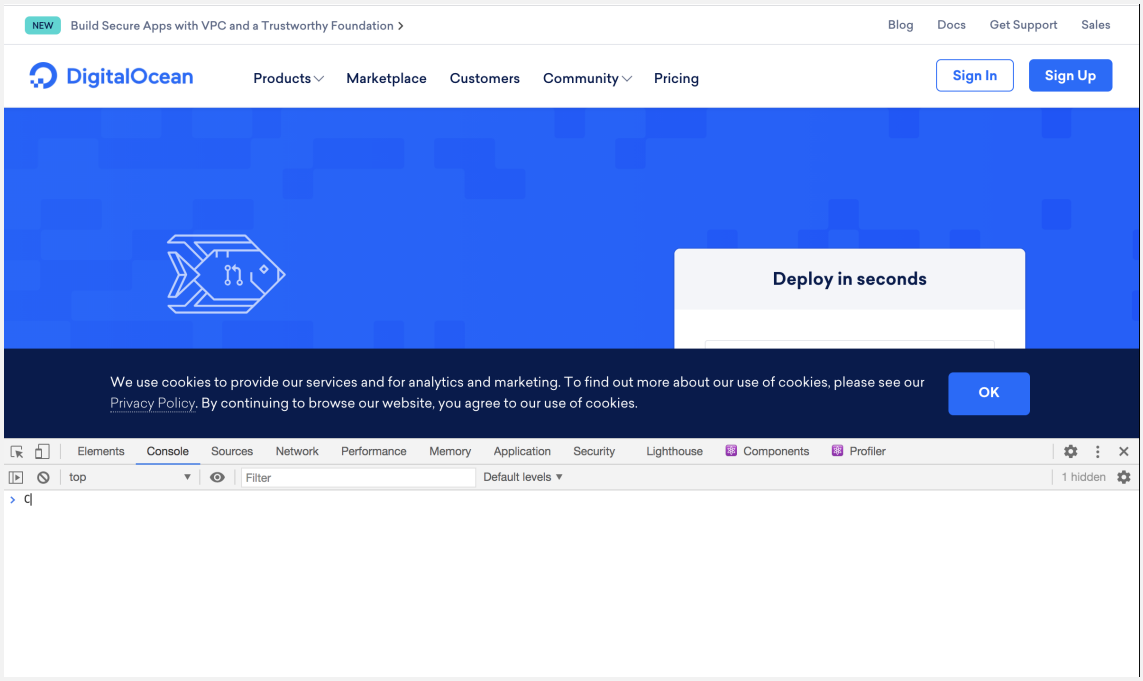
Visit `digitalocean.com`, to find that the homepage is running a production version of React:



DigitalOcean React Production Build information

Now that you are on a website that uses React, open the console to access the React Developer Tools. Open the console by either right-clicking and inspecting an element or by opening the toolbar by clicking **View > Developer > JavaScript console**.

When you open the console, you'll find two new tabs: **Components** and **Profiler**:

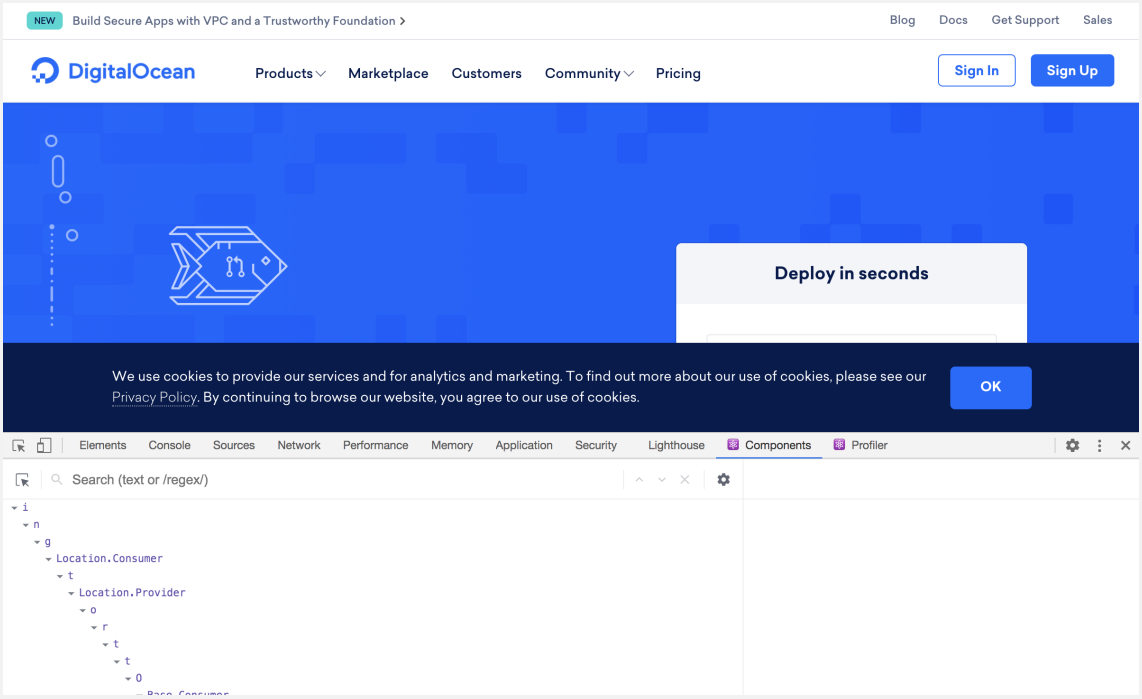


`Console Open`

The **Components** tab will show the current React component tree, along with any props, state, or context. The **Profiler** tab lets you record interactions and measure component rendering. You'll explore the **Profiler** tab in Step 3.

Click on the **Components** tab to see the current component tree.

Since this is a production build, the code will be minified and the components will not have descriptive names:



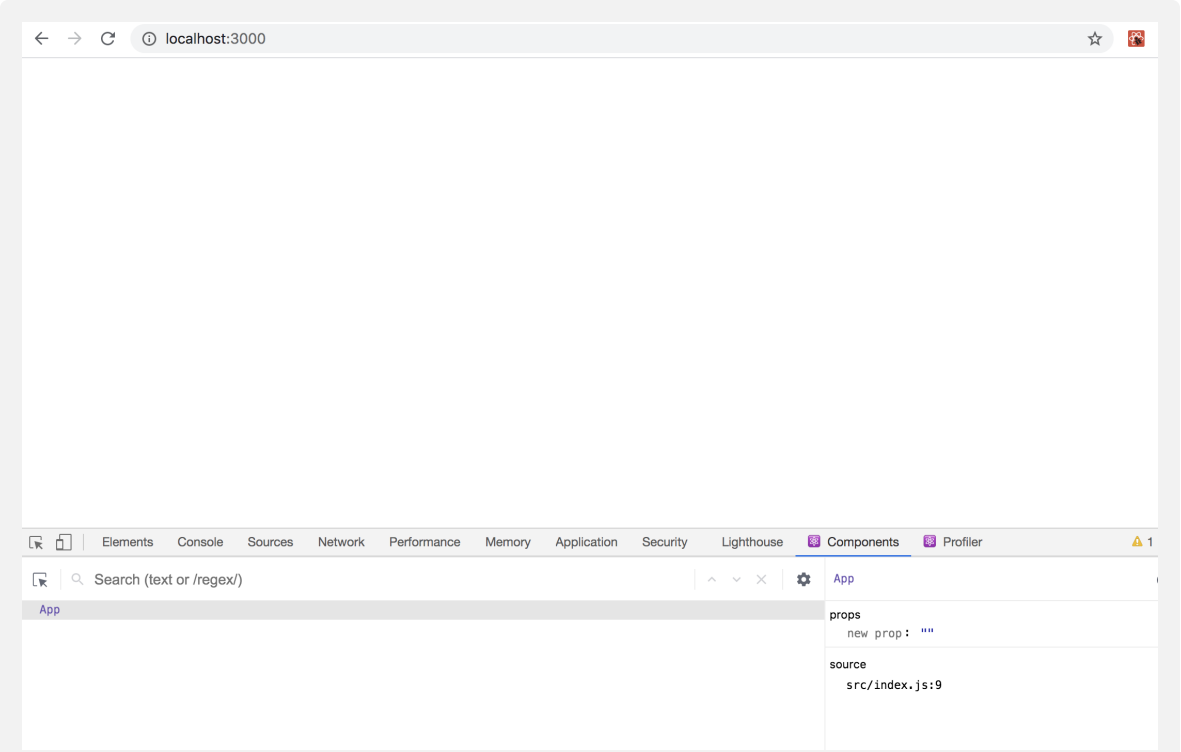
Components for digitalocean.com in the console

Now that you've tried out React Developer Tools on a working website, you can use it on your test application. If you haven't started your `debug-tutorial` application yet, go to a terminal window and run `npm start` from the root of the project.

Open a browser to `http://localhost:3000`.

Notice that the icon for React Developer Tools is now red and white. If you click on the React Developer Tools icon, you'll see a warning that the page is in development mode. Since you are still working on the sample application, this is expected.

Open the console and you'll find the name of the `App` component in the **Components** tab.



`Base Component`

There’s not a lot of information yet, but as you build out the project in the next step, you’ll see all of your components forming a navigable tree.

In this step, you added the React Developer Tools extension to Chrome. You activated the tools on both a production and a development page, and you briefly explored your `debug-tutorial` project in the **Components** tab. In the next step, you’ll build the text analyzer that you’ll use to try out the features of the React Developer Tools.

Step 2 — Identifying Real-Time Component Props and Context

In this step, you'll build a small application to analyze a block of text. The app will determine and report the word count, character count, and character frequency of the text in the input field. As you build the application, you'll use React Developer Tools to explore the current state and props of each component. You'll also use React Developer Tools to view the current context in deeply nested components. Finally, you'll use the tools to identify components that re-render as state changes.

By the end of this step, you'll be able to use the React Developer Tools to explore a live application and to observe the current props and state without console statements or debuggers.

To start, you will create an input component that will take a large amount of text.

Open the `App.js` file:

```
nano src/components/App/App.js
```

Inside the component, add a `div` with a class of `wrapper`, then create a `<label>` element surrounding a `<textarea>` element:

debug-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';

function App() {
  return(
    <div className="wrapper">
     <label htmlFor="text">
       Add Your Text Here:
       <br>
       <textarea
         id="text"
         name="text"
         rows="10"
         cols="100"
       >
       </textarea>
      </label>
    </div>
  )
}

export default App;
```

This will be the input area for your user. The `htmlFor` attribute links the `label` element to elements with an `id` of `text` using JSX. You also give the `<textarea>` component `10` rows and `100` columns to provide room for a large amount of text.

Save and close the file. Next, open `App.css`:

```
nano src/components/App/App.css
```

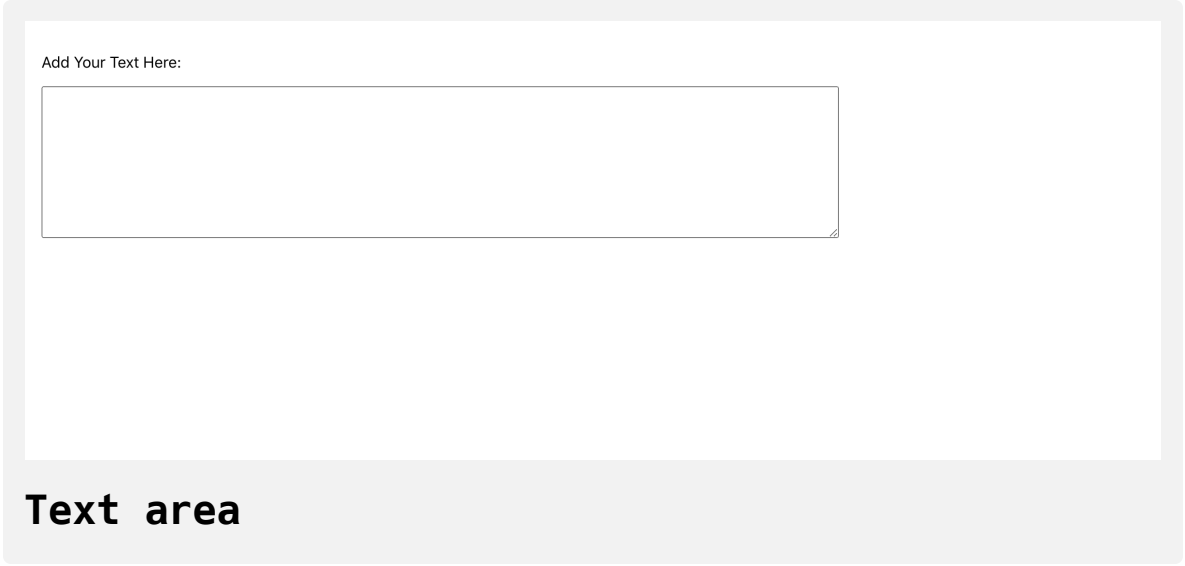
Add some styling to the application by replacing the contents with the following:

debug-tutorial/src/components/App.App.css

```
.wrapper {
    padding: 20px;
}

.wrapper button {
    background: none;
    border: black solid 1px;
    cursor: pointer;
    margin-right: 10px;
}

.wrapper div {
    margin: 20px 0;
}
```

Here you add some padding to the `wrapper` class, then simplify child `<button>` elements by removing the background color and adding some margin. Finally, you add a small margin to child `<div>` elements. These styles will apply to components you will build to display information about the text.

Save the file. When you do, the browser will refresh and you'll see the input:



Open `App.js`:

```
nano src/components/App/App.js
```

Next, create a context to hold the value from the `<textarea>` element. Capture the data using the `useState` Hook:

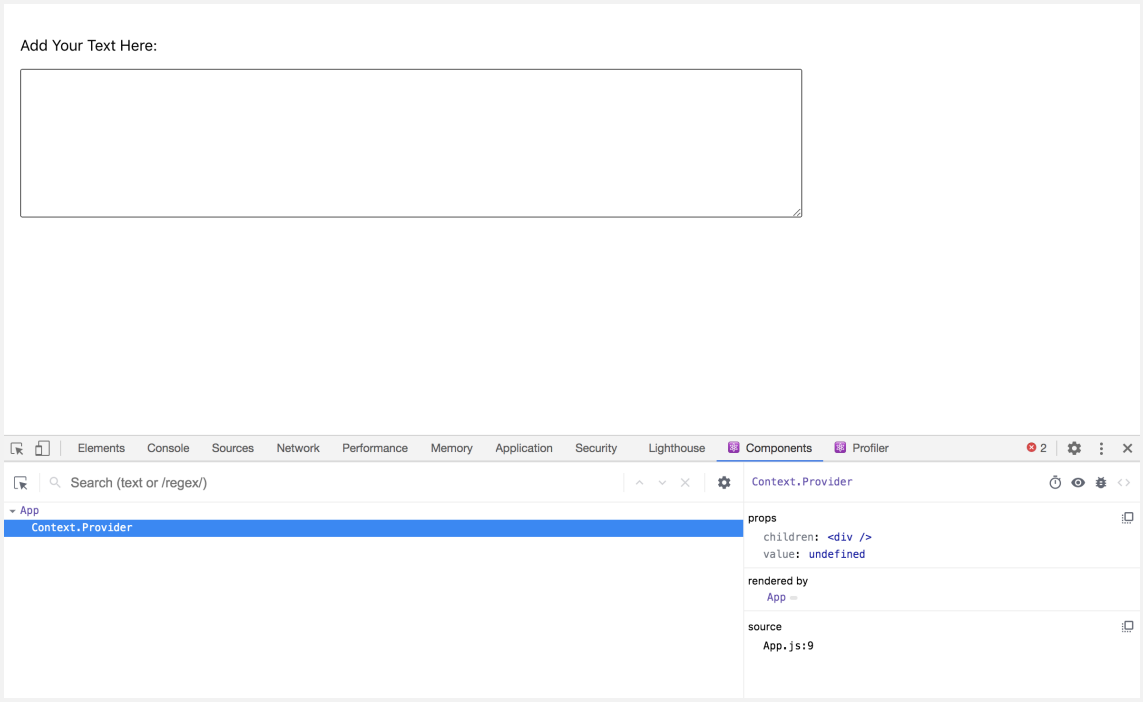
debug-tutorial/src/components/App/App.js

```
import React, { createContext, useState } from 'react';
import './App.css';

export const TextContext = createContext();

function App() {
  const [text, setText] = useState('');

  return(
    <TextContext.Provider value={text}>
      <div className="wrapper">
        <label htmlFor="text">
          Add Your Text Here:
          <br>
          <textarea
            id="text"
            name="text"
            rows="10"
            cols="100"
            onChange={e => setText(e.target.value)}
          >
          </textarea>
        </label>
      </div>
    </TextContext.Provider>
```

```
  )
}

export default App;
```

Be sure to export `TextContext`, then wrap the whole component with the `TextContext.Provider`. Capture the data by adding an `onChange` prop to your `<textarea>` element.

Save the file. The browser will reload. Be sure that you have React Developer Tools open and notice that `App` component now shows the `Context.Provider` as a child component.



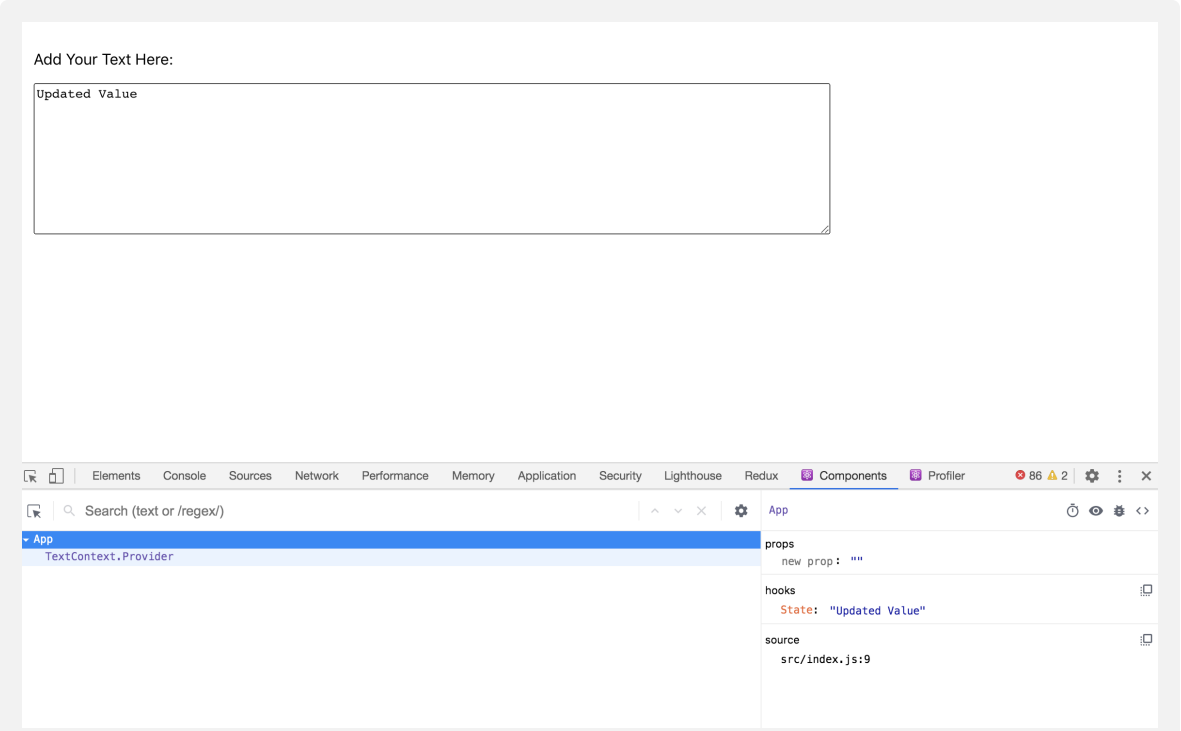
Component context in React Developer Tools

The component by default has a generic name—`Context`—but you can change that by adding a `displayName` property to the generated context. Inside `App.js`, add a line where you set the `displayName` to `TextContext`:

debug-tutorial/src/components/App/App.js

```
import React, { createContext, useState } from 'react';
import './App.css';

export const TextContext = createContext();
TextContext.displayName = 'TextContext';

function App() {
    ...
}

export default App;
```

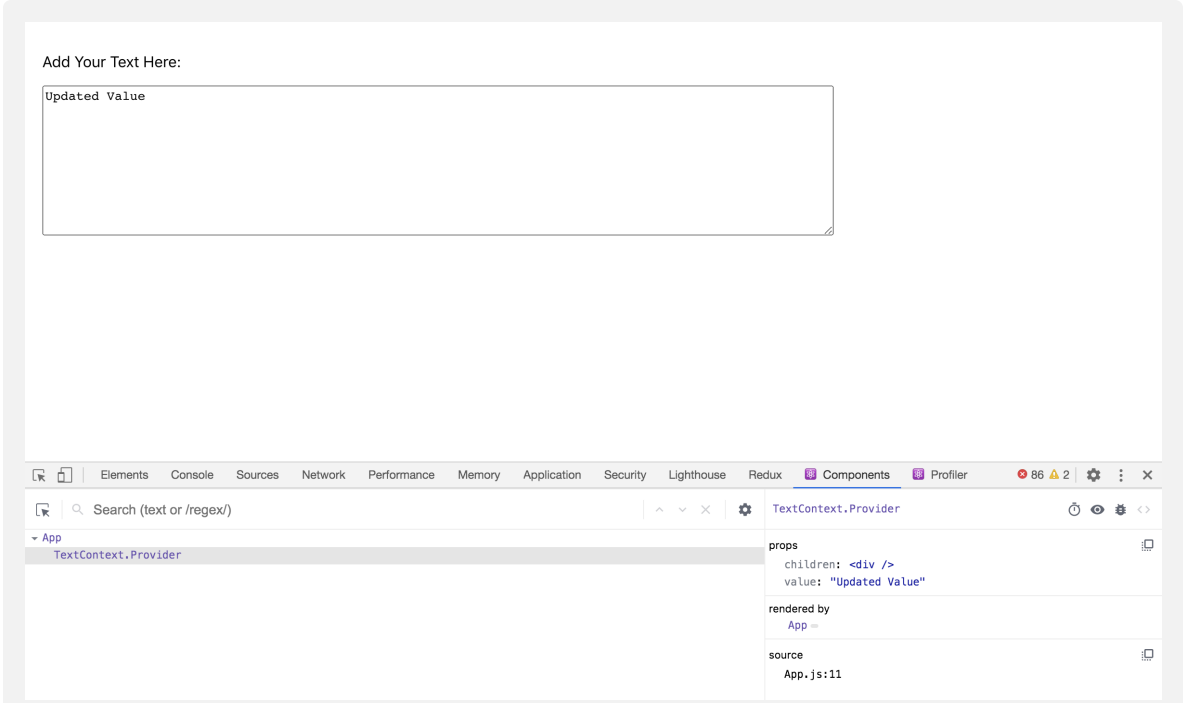
It is not necessary to add a `displayName`, but it does help to navigate components when analyzing the component tree in the console. You will also see the value of the `useState` Hook in the side bar. Type some text in the input and you'll see the updated value in React Developer Tools under the **hooks** section on the `App` component.



Update Value in Developer Tools

The Hook also has a generic name of `State`, but this is not as easy to update as the context. There is a `useDebugValue` Hook, but it only works on custom Hooks and is not recommended for all custom Hooks.

In this case, the state for the `App` component is the prop to `TextContext.Provider`. Click on the `TextContext.Provider` in the React Developer Tools and you'll see that the `value` also reflects the input value that you set with the state:



Updated value for the context

React Developer Tools is showing you real time prop and context information, and the value will grow as you add components.

Next, add a component called `TextInformation`. This component will be a container for the components with specific data analysis, such as the word count.

First, make the directory:

```
mkdir src/components/TextInformation
```

Then open `TextInformation.js` in your text editor.

```
nano src/components/TextInformation/TextInformation.js
```

Inside the component, you will have three separate components: `CharacterCount`, `WordCount`, and `CharacterMap`. You'll make these components in just a moment.

The `TextInformation` component will use the `useReducer` Hook to toggle the display of each component. Create a `reducer` function that toggles the display value of each component and a button to toggle each component with an `onClick` action:

debug-tutorial/src/components/TextInformation/TextInformation.js

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


const reducer = (state, action) => {
  return {
    ...state,
    [action]: !state[action]
  }
}
export default function TextInformation() {
  const [tabs, toggleTabs] = useReducer(reducer, {
    characterCount: true,
    wordCount: true,
    characterMap: true
  });


  return(
    <div>
      <button onClick={() => toggleTabs('characterCount')}>Char
      <button onClick={() => toggleTabs('wordCount')}>Word Coun
      <button onClick={() => toggleTabs('characterMap')}>Charac
    </div>
  )
}
```

Notice that your `useReducer` Hook starts with an object that maps each key to a boolean. The reducer function uses the spread operator to preserve the previous value while setting a new value using the `action` parameter.

Save and close the file. Then open `App.js`:

```
nano src/components/App/App.js
```

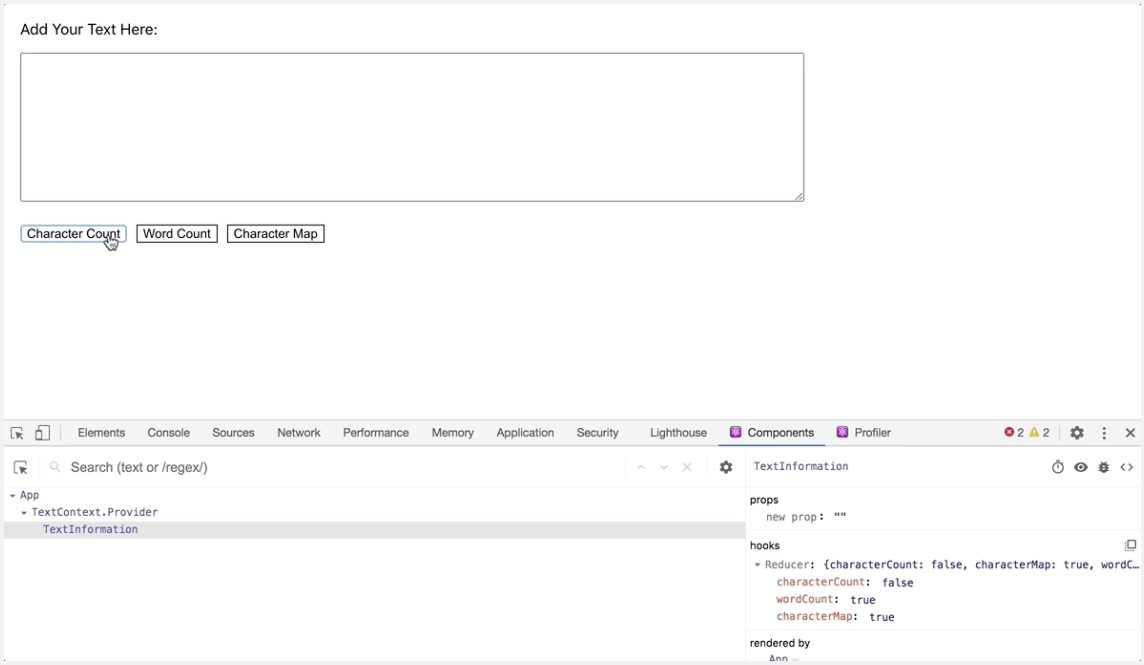
Add in your new component:

debug-tutorial/src/components/App/App.js

```
import React, { createContext, useState } from 'react';
import './App.css';
import TextInformation from '../TextInformation/TextInformation


...

function App() {
  const [text, setText] = useState('');

  return(
    <TextContext.Provider value={text}>
      <div className="wrapper">
        <label htmlFor="text">
          Add Your Text Here:
          <br>
          <textarea
            id="text"
            name="text"
            rows="10"
            cols="100"
            onChange={e => setText(e.target.value)}
          >
          </textarea>
        </label>
        <TextInformation />
```

```
      </div>
    </TextContext.Provider>
  )
}

export default App;
```

Save and close the file. When you do, the browser will reload, and you'll see the updated component. If you click on `TextInformation` in React Developer Tools, you'll see the value update on each button click:



Update Reducer on Click

Now that you have the container component, you'll need to create each informational component. Each component will take a prop called `show`. If `show` is falsy, the component will return `null`. The components will consume the `TextContext`, analyze the data, and display the result.

To start, create the `CharacterCount` component.

First, make a new directory:

```
mkdir src/components/CharacterCount
```

Then, open `CharacterCount.js` in your text editor:

```
nano src/components/CharacterCount/CharacterCount.js
```

Inside the component, create a function that uses the `show` prop and displays `null` if `show` is falsy:

debug-tutorial/src/components/CharacterCount/CharacterCount.js

```
import React, { useContext } from 'react';
import PropTypes from 'prop-types';
import { TextContext } from '../App/App';

export default function CharacterCount({ show }) {
  const text = useContext(TextContext);

  if(!show) {
    return null;
  }

  return(
    <div>
      Character Count: {text.length}
    </div>
  )
}

CharacterCount.proTypes = {
  show: PropTypes.bool.isRequired
}
```

Inside the `CharacterCount` function, you assign the value of `TextContext` to a variable using the `useContext` Hook. You then return a `<div>` that shows the character count using the `length` method. Finally, `PropTypes` adds a weak typing system to provide some enforcement to make sure the wrong prop type is not passed.

Save and close the file. Open `TextInformation.js`:

```
nano src/components/TextInformation/TextInformation.js
```

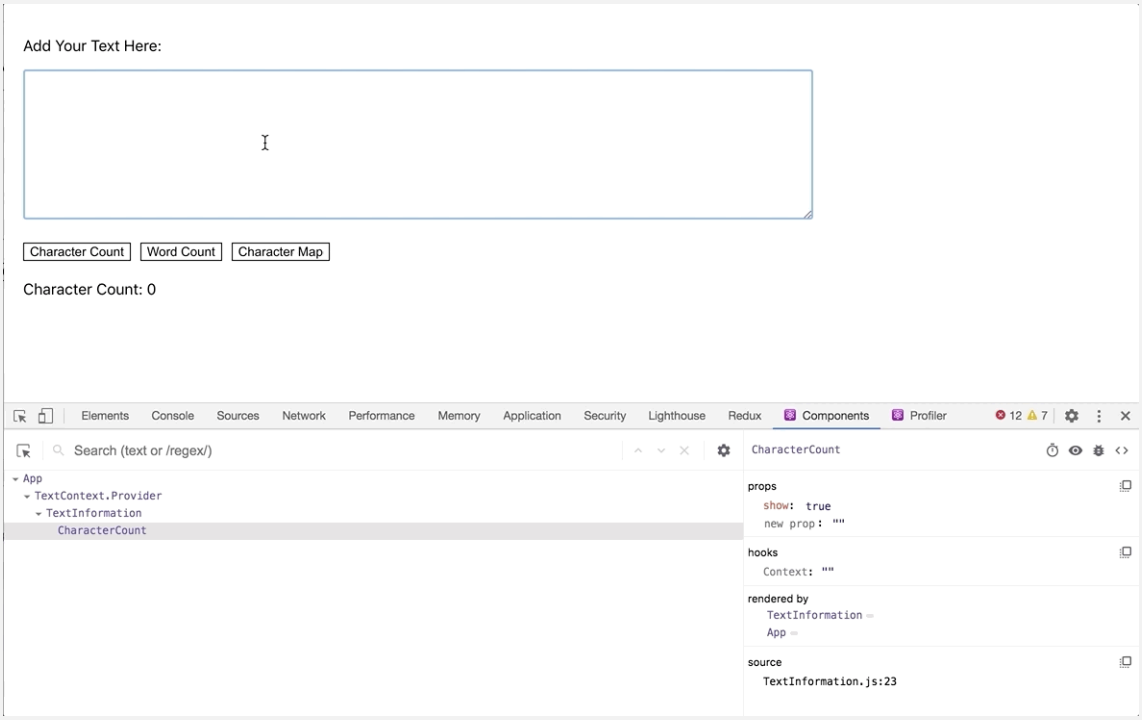
Import `CharacterCount` and add the component after the buttons, passing `tabs.characterCount` as the `show` prop:

debug-tutorial/src/components/TextInformation/TextInformation.js

```
import React, { useReducer } from 'react';
import CharacterCount from '../CharacterCount/CharacterCount';

const reducer = (state, action) => {
  return {
    ...state,
    [action]: !state[action]
  }
}

export default function TextInformation() {
  const [tabs, toggleTabs] = useReducer(reducer, {
    characterCount: true,
    wordCount: true,
    characterMap: true
  });

  return(
    <div>
      <button onClick={() => toggleTabs('characterCount')}>Char
      <button onClick={() => toggleTabs('wordCount')}>Word Coun
      <button onClick={() => toggleTabs('characterMap')}>Charac
      <CharacterCount show={tabs.characterCount} />
    </div>
```

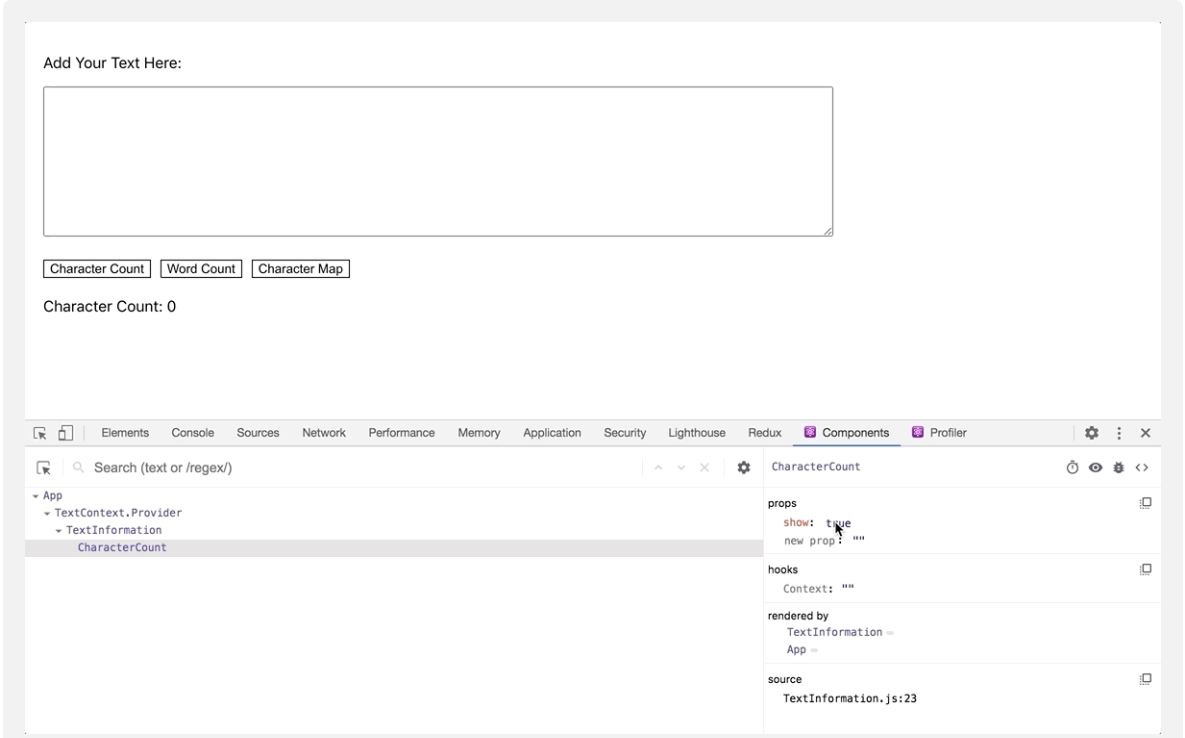
```
  )
}
```

Save the file. The browser will reload and you'll see the component in the React Developer Tools. Notice that as you add words in the input, the context will update. If you toggle the component, you'll see the props update after each click:



Adding text and toggling

You can also manually add or change a prop by clicking on the property and updating the value:



Manually Changing Props

Next, add a `WordCount` component.

Create the directory:

```
mkdir src/components/WordCount
```

Open the file in a text editor:

```
nano src/components/WordCount/WordCount.js
```

Make a component that is similar to `CharacterCount`, but use the `split` method on spaces to create an array of words before showing the length:

debug-tutorial/src/components/WordCount/WordCount.js

```
import React, { useContext } from 'react';
import PropTypes from 'prop-types';
import { TextContext } from '../App/App';

export default function WordCount({ show }) {
  const text = useContext(TextContext);

  if(!show) {
    return null;
  }

  return(
    <div>
      Word Count: {text.split(' ').length}
    </div>
  )
}

WordCount.proTypes = {
  show: PropTypes.bool.isRequired
}
```

Save and close the file.

Finally, create a `CharacterMap` component. This component will show how often a specific character is used in a block of text. It will then sort the characters by frequency in the passage and display the results.

First, make the directory:

```
mkdir src/components/CharacterMap
```

Next, open `CharacterMap.js` in a text editor:

```
nano src/components/CharacterMap/CharacterMap.js
```

Import and use the `TextContext` component and use the `show` prop to display results as you did in the previous components:

debug-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React, { useContext } from 'react';
import PropTypes from 'prop-types';
import { TextContext } from '../App/App';

export default function CharacterMap({ show }) {
  const text = useContext(TextContext);

  if(!show) {
    return null;
  }

  return(
    <div>
      Character Map: {text.length}
    </div>
  )
}

CharacterMap.proTypes = {
  show: PropTypes.bool.isRequired
}
```

This component will need a slightly more complicated function to create the frequency map for each letter. You'll need to go through each character and increment a value anytime there is a repeat. Then you'll need to take that data and sort it so that the most frequent letters are at the top of the list.

To do this, add the following highlighted code:

debug-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React, { useContext } from 'react';
import PropTypes from 'prop-types';
import { TextContext } from '../App/App';

function itemize(text){
 const letters = text.split('')
   .filter(l => l !== ' ')
   .reduce((collection, item) => {
     const letter = item.toLowerCase();
     return {
       ...collection,
       [letter]: (collection[letter] || 0) + 1
     }
   }, {})
 return Object.entries(letters)
   .sort((a, b) => b[1] - a[1]);
}

export default function CharacterMap({ show }) {
  const text = useContext(TextContext);

  if(!show) {
    return null;
```

```
  }


  return(
    <div>
     Character Map:
     {itemize(text).map(character => (
       <div key={character[0]}>
         {character[0]}: {character[1]}
       </div>
     ))}
    </div>
  )
}


CharacterMap.proTypes = {
  show: PropTypes.bool.isRequired
}
```

In this code, you create a function called `itemize` that splits the text into an array of characters using the `split()` string method. Then you `reduce` the array to an object by adding the character and then incrementing the count for each subsequent character. Finally, you convert the object to an array of pairs using `Object.entries` and `sort` to put the most used characters at the top.

After you create the function, you pass the text to the function in the `render` method and `map` over the results to display the character—array value `[0]`—and the count—array value `[1]`—inside a `<div>`.

Save and close the file. This function will give you an opportunity to explore some performance features of the React Developer Tools in the next section.

Next, add the new components to `TextInformation` and look at the values in React Developer Tools.

Open `TextInformation.js`:

```
nano src/components/TextInformation/TextInformation.js
```

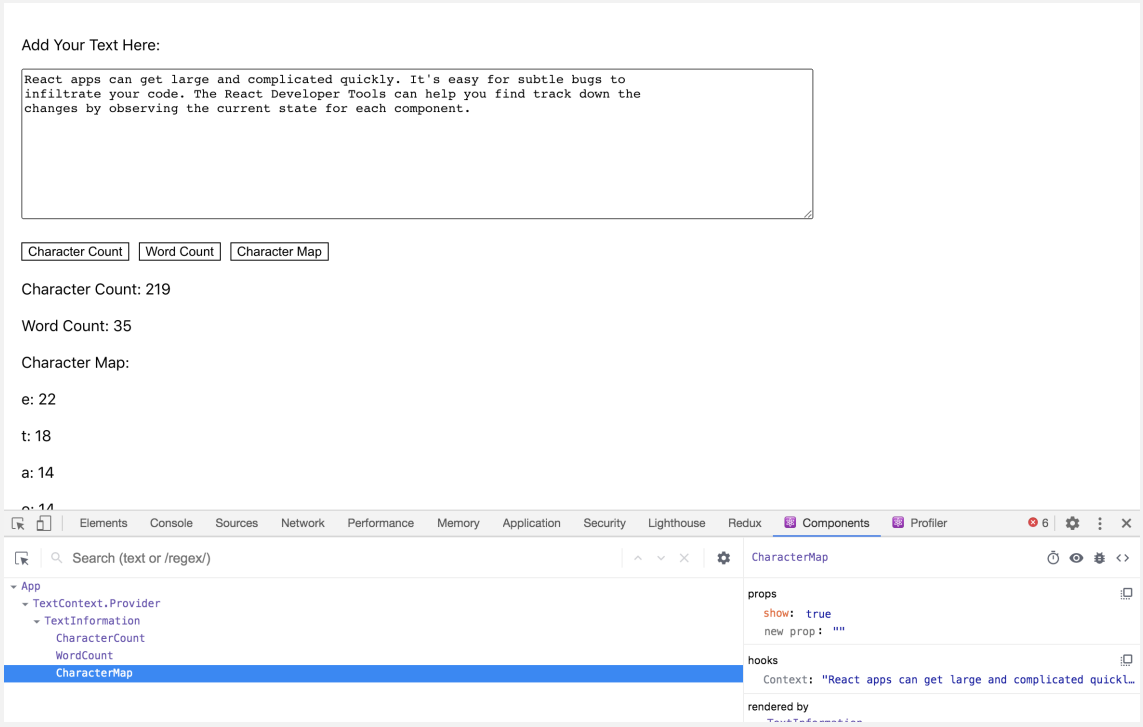
Import and render the new components:

debug-tutorial/src/components/TextInformation/TextInformation.js

```
import React, { useReducer } from 'react';
import CharacterCount from '../CharacterCount/CharacterCount';
import CharacterMap from '../CharacterMap/CharacterMap';
import WordCount from '../WordCount/WordCount';

const reducer = (state, action) => {
  return {
    ...state,
    [action]: !state[action]
  }
}

export default function TextInformation() {
  const [tabs, toggleTabs] = useReducer(reducer, {
    characterCount: true,
    wordCount: true,
    characterMap: true
  });

  return(
    <div>
      <button onClick={() => toggleTabs('characterCount')}>Char
      <button onClick={() => toggleTabs('wordCount')}>Word Coun
      <button onClick={() => toggleTabs('characterMap')}>Charac
```

```
        <CharacterCount show={tabs.characterCount} />
        <WordCount show={tabs.wordCount} />
        <CharacterMap show={tabs.characterMap} />
      </div>
    )
}
```

Save and close the file. When you do, the browser will refresh, and if you add in some data, you'll find the character frequency analysis in the new components:



CharacterMap Component in React Developer Tools

In this section, you used React Developer Tools to explore the component tree. You also learned how to see the real-time props for each component and how to manually change the props using the developer tools. Finally, you viewed the context for the component change with input.

In the next section, you'll use the React Developer Tools **Profiler** tab to identify components that have long render times.

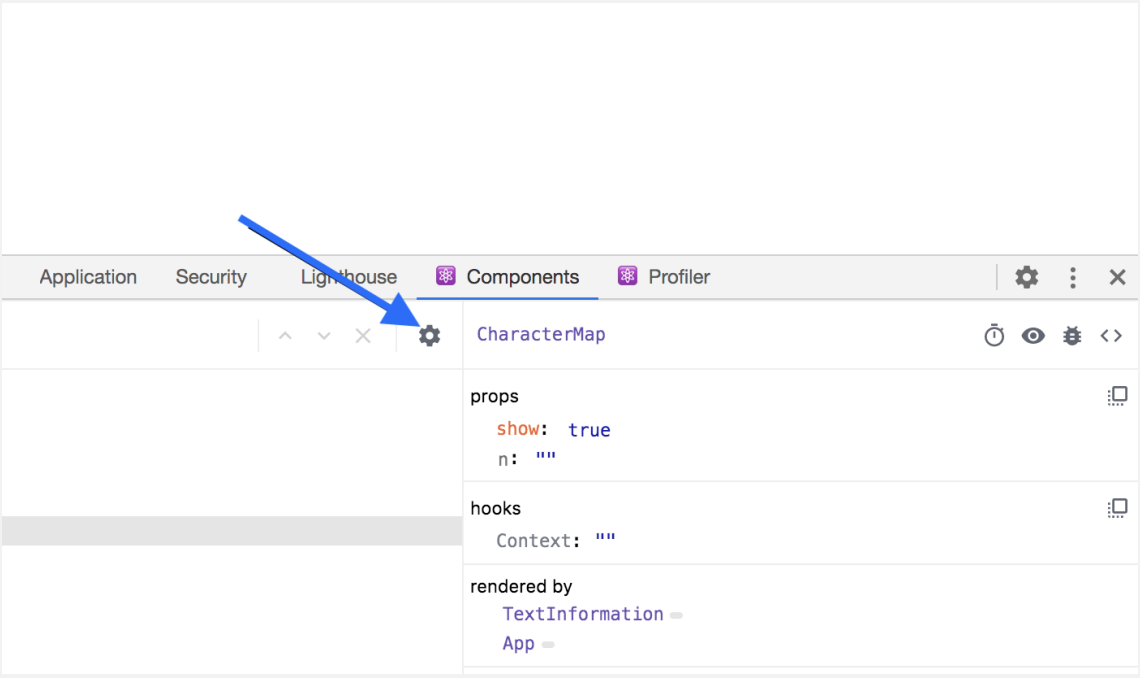
Step 3 — Tracking Component Rendering Across Interactions

In this step, you'll use the React Developer Tools profiler to track component rendering and re-rendering as you use the sample application. You'll navigate flamegraphs, or visualizations of your app's relevant optimization metrics, and use the information to identify inefficient components, reduce rendering time, and increase application speed.

By the end of this step, you'll know how to identify components that render during user interactions and how to compose components to reduce inefficient rendering.

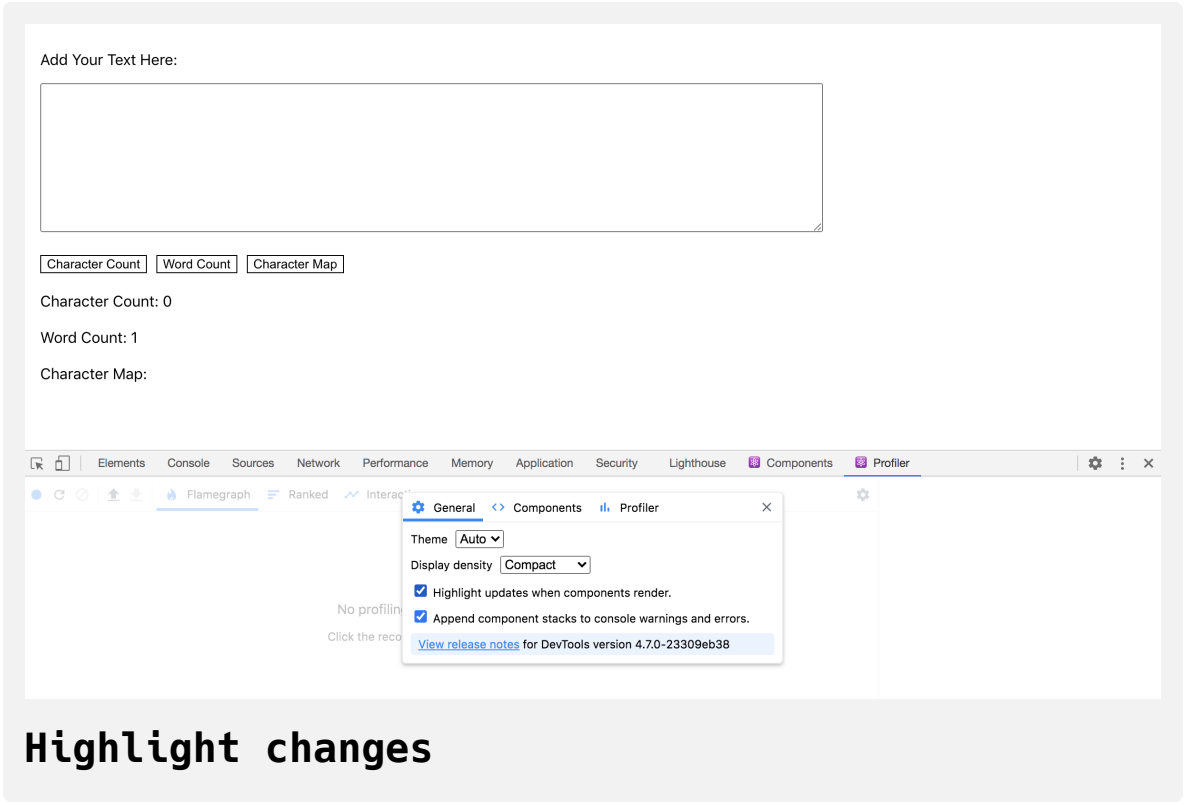
A quick way to see how components change each other is to enable highlighting when a component is re-rendered. This will give you a visual overview of how the components respond to changing data.

In React Developer Tools, click on the **Settings** icon. It will look like a gear:



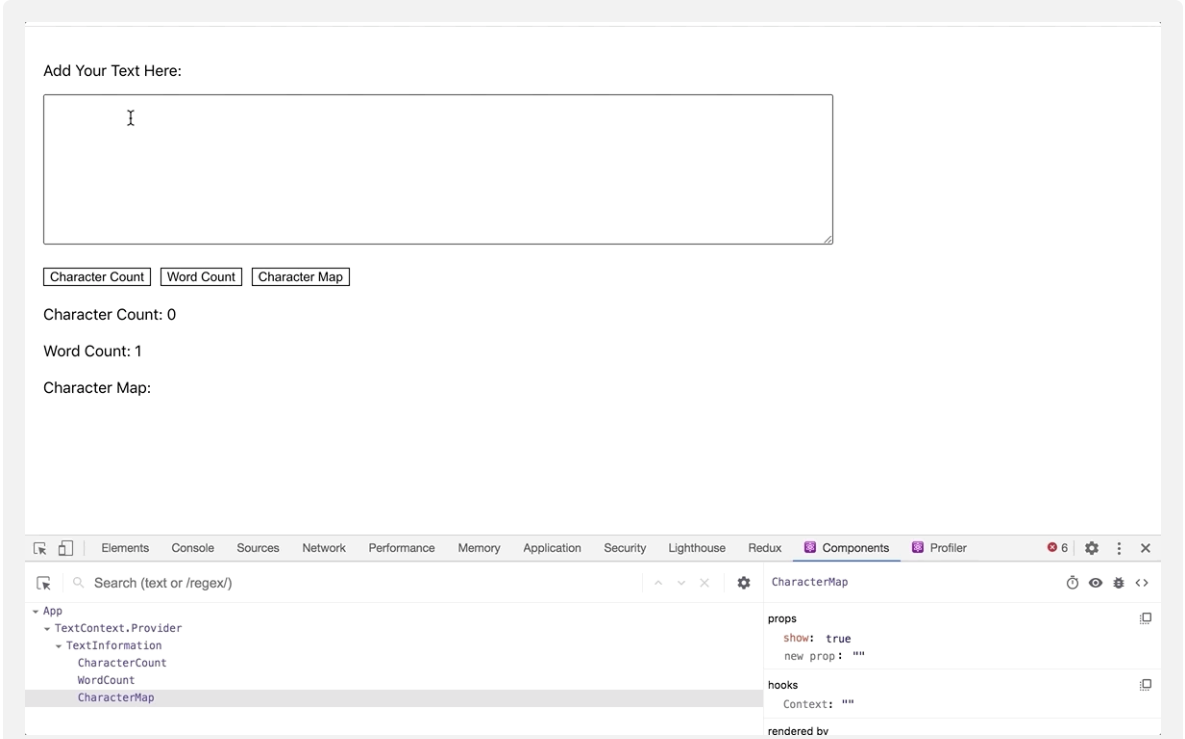
Settings icon

Then select the option under **General** that says **Highlight updates when components render**.



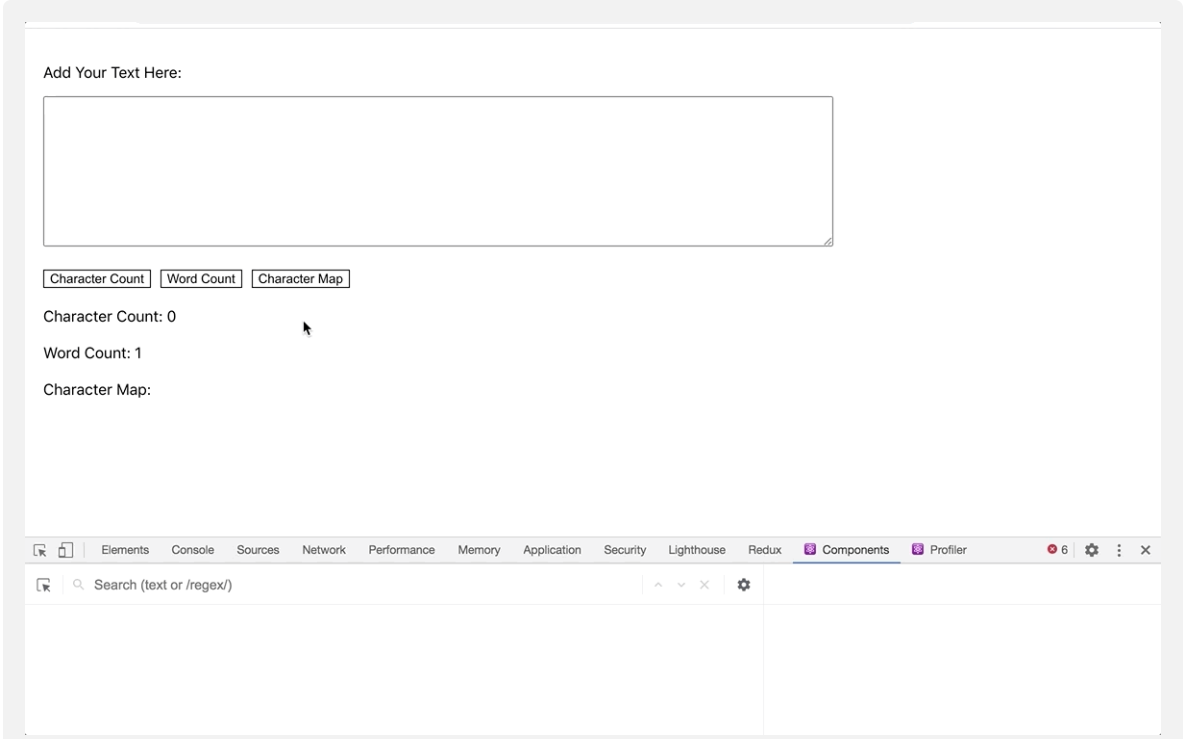
When you make any changes, React Developer Tools will highlight components that re-render. For example, when you change input, every component re-renders because the data is stored on a Hook at the root level and every change will re-render the whole component tree.

Notice the highlighting around the components, including the top of the screen around the root component:



Highlighting Text

Compare that to how the components re-render when you click on one of the buttons to toggle the data. If you click one of the buttons, the components under `TextInformation` will re-render, but not the root component:



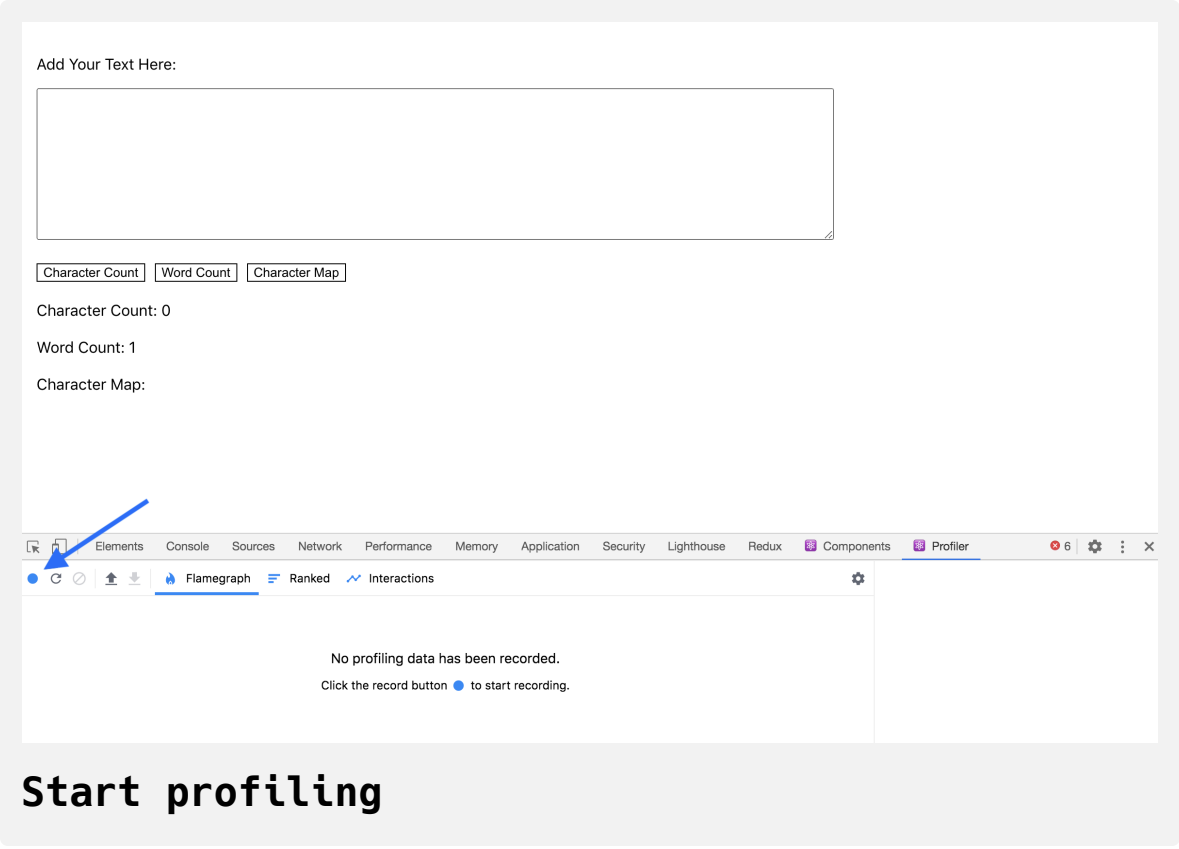
Rerendering lower components only

Showing the re-renders will give you a quick idea of how the components are related, but it doesn't give you a lot of data to analyze specific components. To gain more insight, let's look at the profiler tools.

The profiler tools are designed to help you measure precisely how long each component takes to render. This can help you identify components that may be slow or process intense.

Re-open the settings and uncheck the box for **Highlight updates when components render**. Then click on the **Profiler** tab in the console.

To use the profiler, click the blue circle on the left of the screen to begin recording and click it again when you are finished:

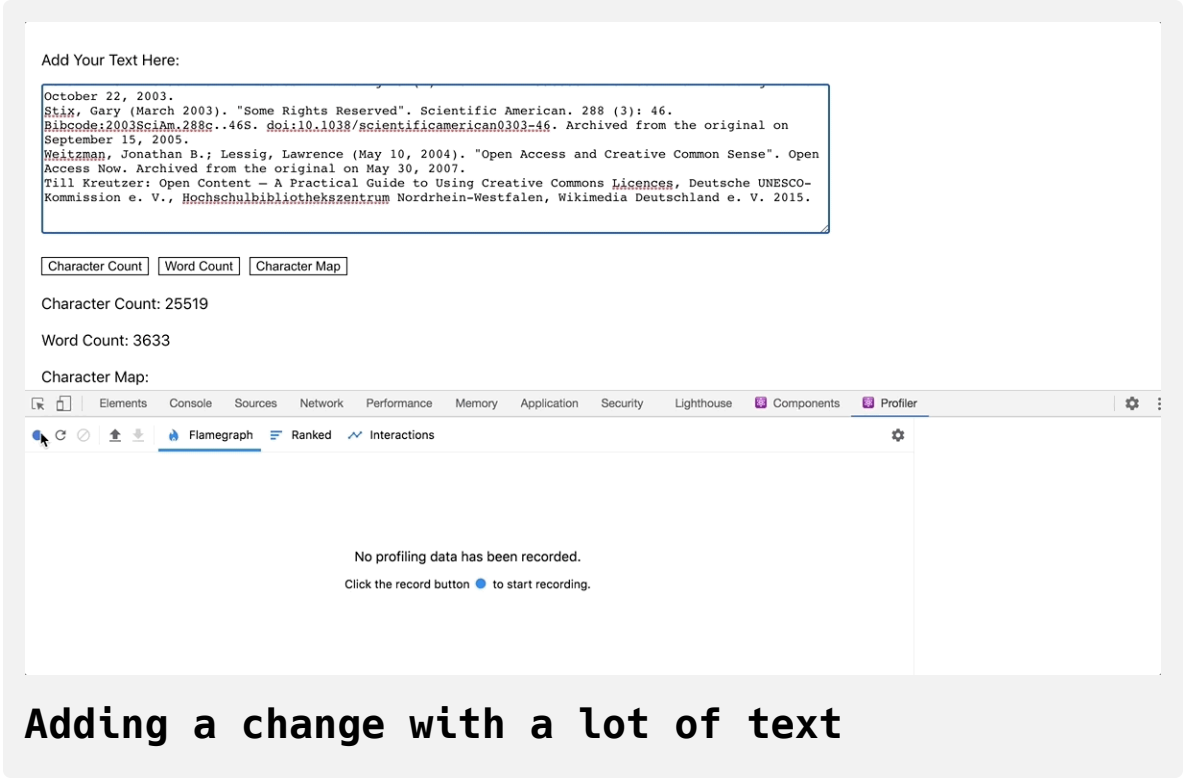


Start profiling

When you stop recording, you'll find a graph of the component changes including, how long each item took to render.

To get a good sense of the relative efficiency of the components, paste in the Wikipedia page for Creative Commons. This text is long enough to give interesting results, but not so big that it will crash the application.

After pasting in the text, start the profiler, then make a small change to the input. Stop profiling after the component has finished re-rendering. There will be a long pause, because the application is handling a long re-rendering:

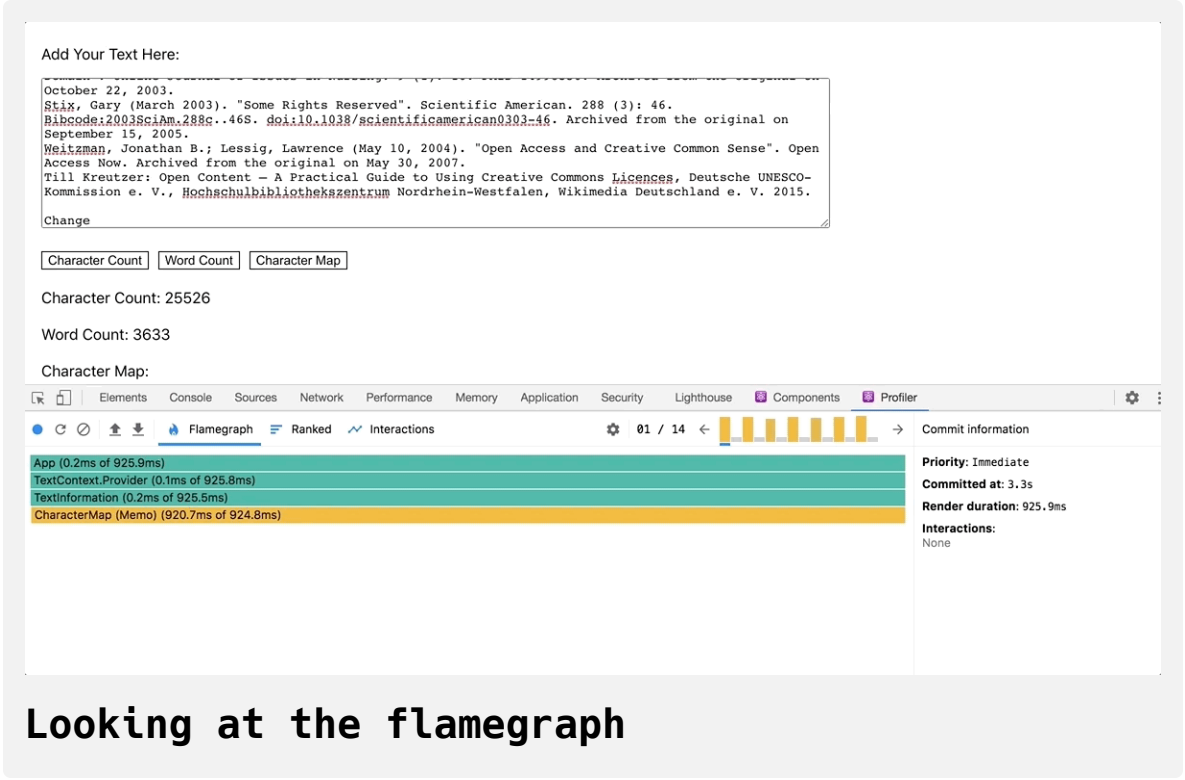


Adding a change with a lot of text

When you end the recording, React Developer Tools will create a flamegraph that shows every component that re-rendered and how long it took to re-render each component.

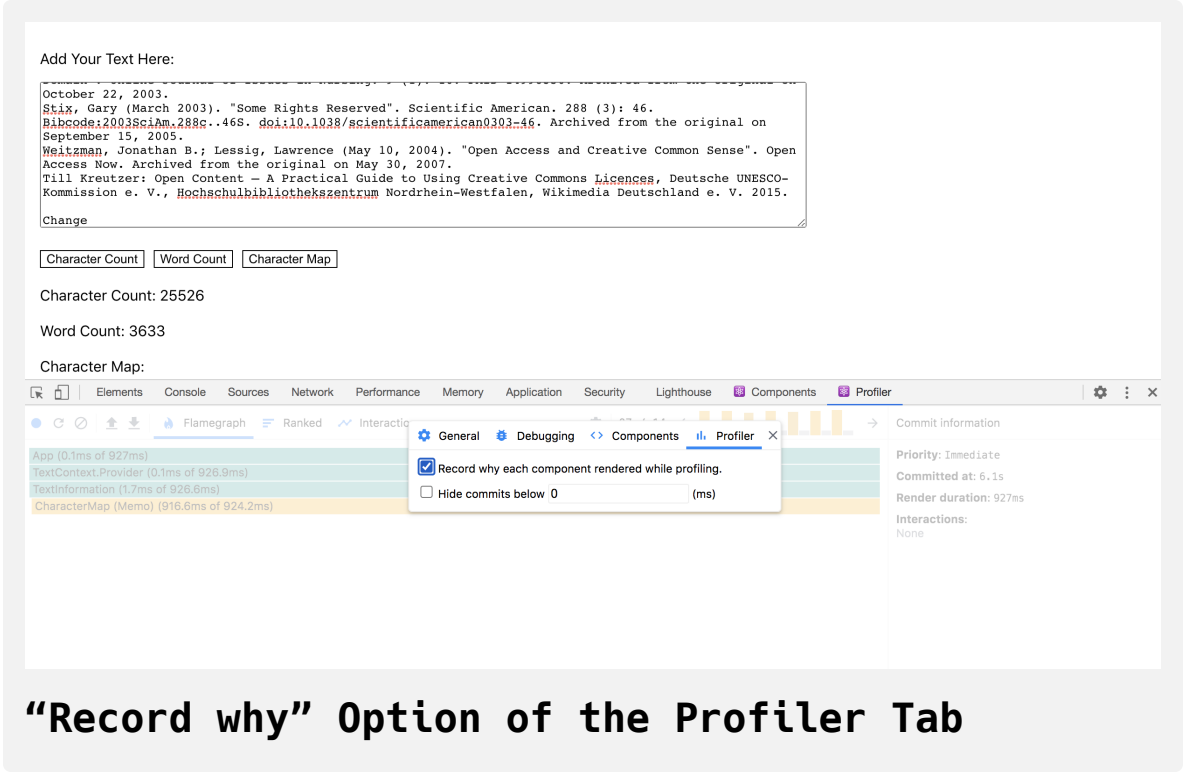
In this case, every keypress from the word “Change” causes a re-render. More importantly, it shows how long each render takes and why there was a long delay. The components `App`, `TextContext.Provider`, and `TextInformation` take about .2 milliseconds to rerender. But the `CharacterMap` component takes around 1 second per keystroke to re-render because of the complicated data parsing in the `itemize` function.

In the display, each yellow bar is a new keystroke. You can replay the sequence one at a time by clicking on each bar. Notice that there is slight variation in the render times, but the `CharacterMap` is consistently slow:



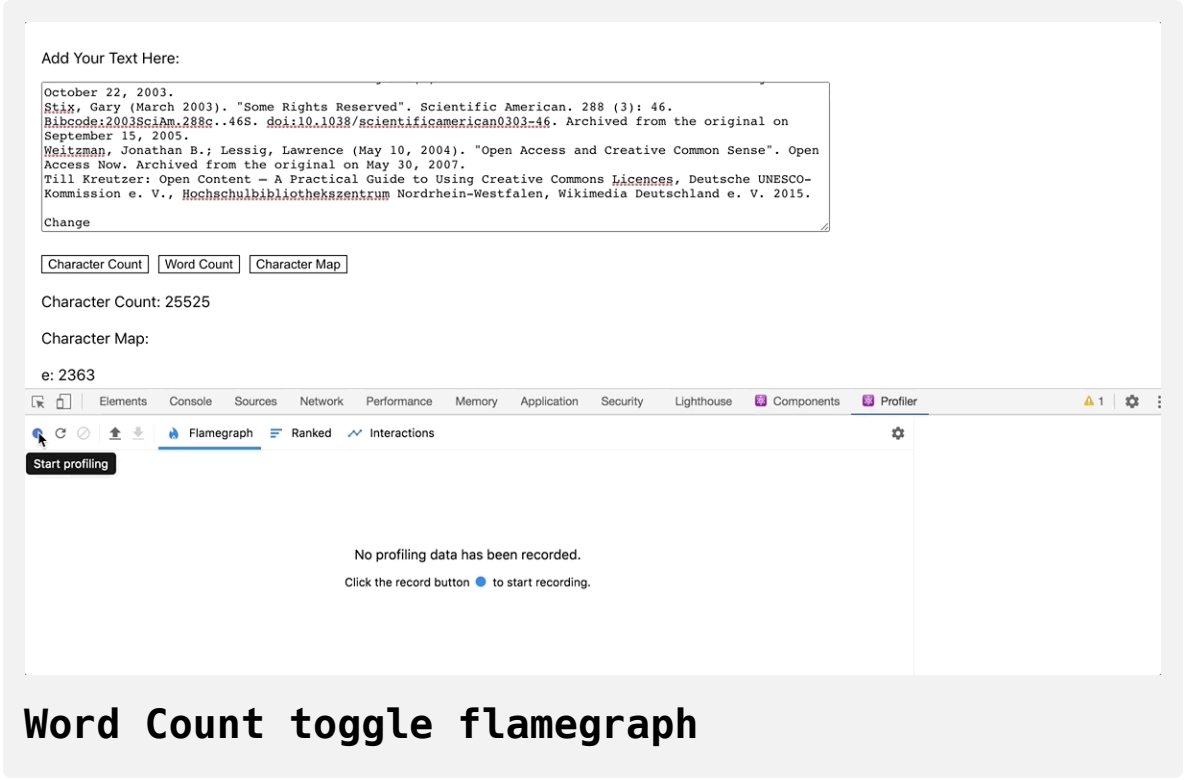
Looking at the flamegraph

You can get more information by selecting the option **Record why each component rendered while profiling.** under the **Profiler** section of the settings.



“Record why” Option of the Profiler Tab

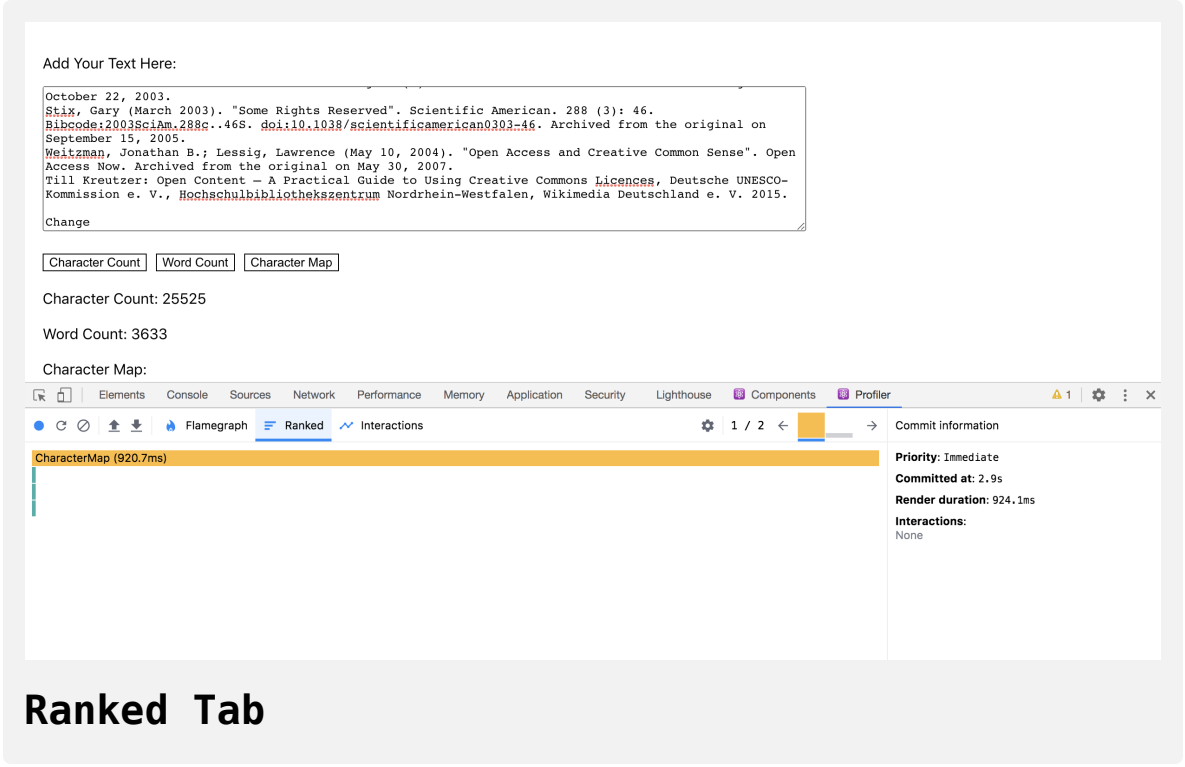
Try toggling the **Word Count** component and notice how long the changes take. The application still lags even though you haven’t changed the text content:



Word Count toggle flamegraph

Now when you hover your cursor over a component, you'll find that it includes a reason the component re-rendered. In this case, the reason the component changed is **The parent component rendered**. That's a problem for the `CharacterMap` component. `CharacterMap` is doing an expensive calculation every time the parent changes, even if the props and the context do not change. That is, it's recalculating data even though the data is identical to the previous render.

Click on the **Ranked** tab and you'll find how much more time `CharacterMap` takes when compared to all other components:



Ranked Tab

React Developer Tools have helped isolate a problem: the `CharacterMap` component re-renders and performs an expensive calculation anytime any parent component changes.

There are multiple ways to solve the problem, but they all involve some sort of caching via memoization, a process by which already calculated data is remembered rather than recalculated. You can either use a library like lodash/memoize or memoize-one to cache the results of the `itemize` function, or you can use the built in React `memo` function to memoize the whole component.

If you use the React `memo`, the function will only re-render if the props or context change. In this case, you'll use React `memo`. In general, you should memoize the data itself first since it's a more isolated case, but there are

some interesting changes in the React Developer Tools if you memoize the whole component, so you'll use that approach in this tutorial.

Open `CharacterMap.js`:

```
nano src/components/CharacterMap/CharacterMap.js
```

Import `memo` from React, then pass the entire function into the `memo` function:

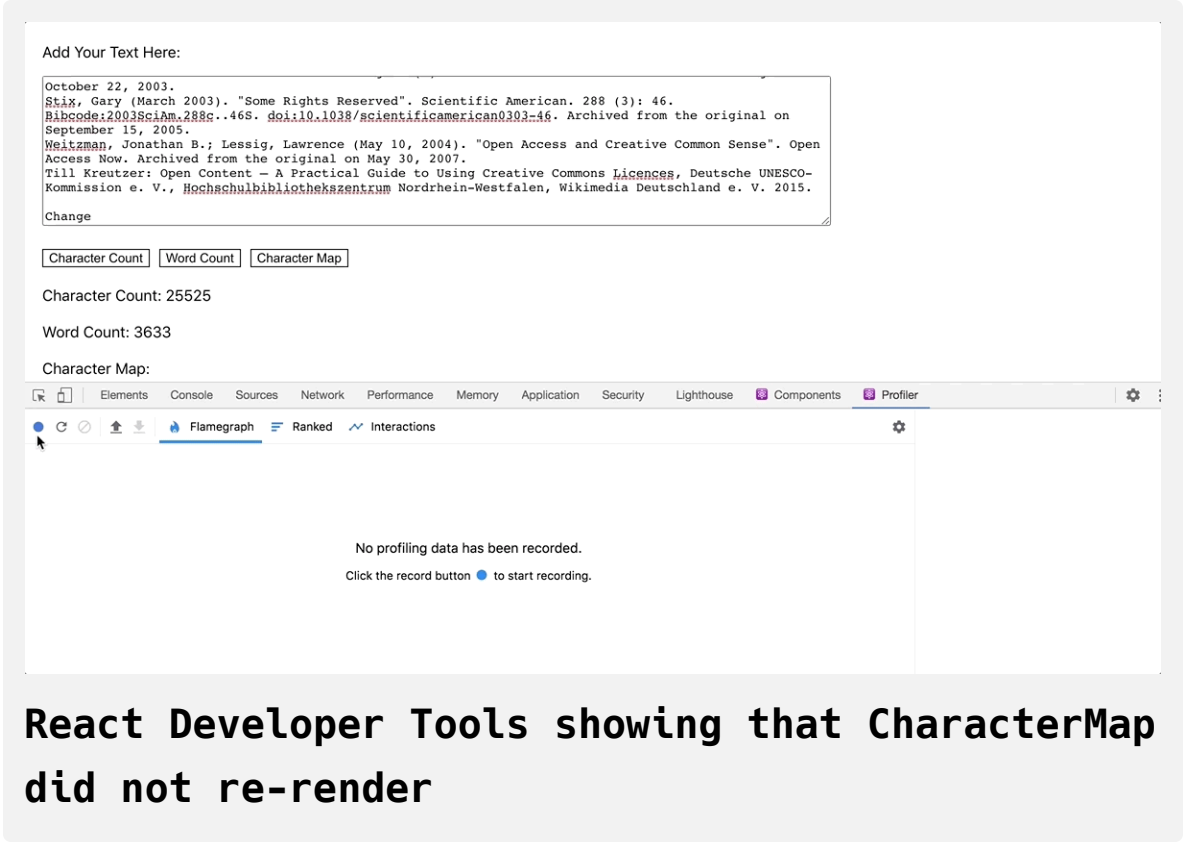
debug-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React, { memo, useContext } from 'react';
import PropTypes from 'prop-types';
import { TextContext } from '../App/App';


...


function CharacterMap({ show }) {
  const text = useContext(TextContext);

  if(!show) {
    return null;
  }

  return(
    <div>
     Character Map:
     {itemize(text).map(character => (
       <div key={character[0]}>
         {character[0]}: {character[1]}
       </div>
     ))}
    </div>
  )
}
```

```
CharacterMap.proTypes = {
  show: PropTypes.bool.isRequired
}

export default memo(CharacterMap);
```

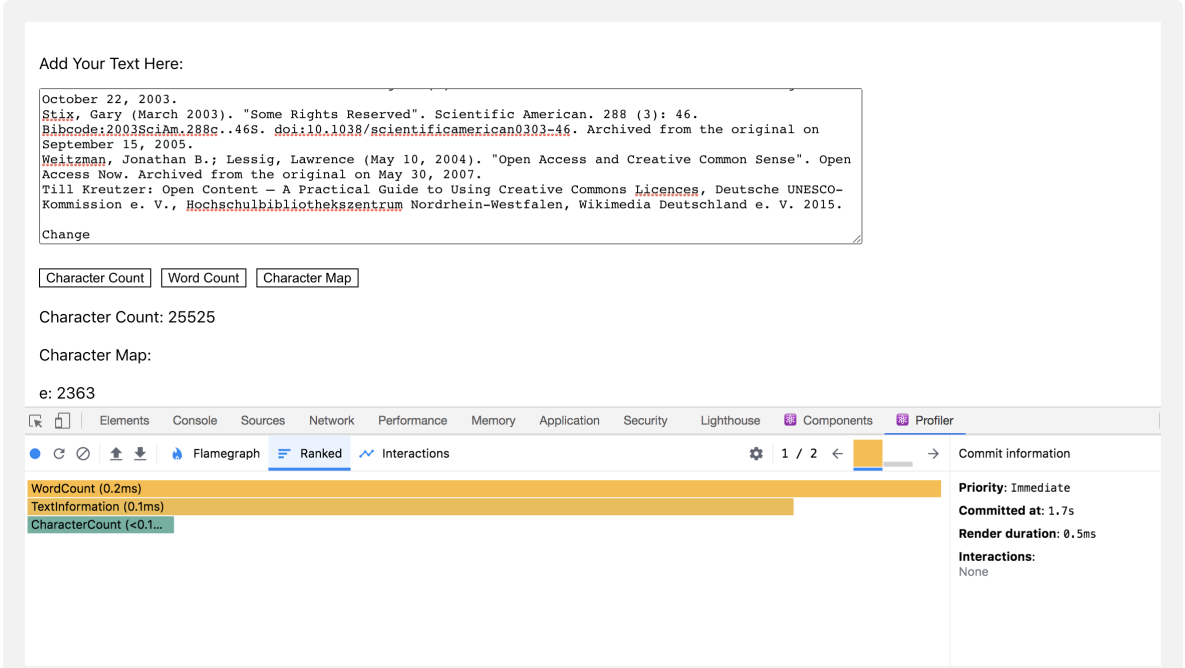
You move the `export default` line to the end of the code in order to pass the component to `memo` right before exporting. After that, React will compare the props before re-rendering.

Save and close the file. The browser will reload, and when you toggle the `WordCount`, the component will update much faster. This time, `CharacterMap` does not re-render. Instead, in React Developer Tools, you'll see a gray rectangle showing re-rendering was prevented.



React Developer Tools showing that CharacterMap did not re-render

If you look at the **Ranked** tab, you'll find that both the `CharacterCount` and the `WordCount` re-rendered, but for different reasons. Since `CharacterCount` is not memoized, it re-rendered because the parent changed. The `WordCount` re-rendered because the props changed. Even if it was wrapped in `memo`, it would still rerender.



Ranked view of memoized app

Note: Memoizing is helpful, but you should only use it when you have a clear performance problem, as you did in this case. Otherwise, it can create a performance problem: React will have to check props every time it re-renders, which can cause a delay on smaller components.

In this step, you used the profiler to identify re-renders and componenent re-rendering. You also used flamegraphs and ranked graphs to identify components that are slow to re-render and then used the `memo` function to prevent re-rendering when there are no changes to the props or context.

Conclusion

The React Developer Tools browser extension gives you a powerful set of utilities to explore your components in their applications. With these tools, you'll be able to explore a component's state and identify bugs using real data without console statements or debuggers. You can also use the profiler to explore how components interact with each other, allowing you to identify and optimize components that have slow rendering in your full application. These tools are a crucial part of the development process and give you an opportunity to explore the components as part of an application and not just as static code.

If you would like to learn more about debugging JavaScript, see our article on How To Debug Node.js with the Built-In Debugger and Chrome DevTools. For more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Handle DOM and Window Events with React

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

In web development, events represent actions that happen in the web browser. By responding to events with event handlers, you can create dynamic JavaScript applications that respond to any user action, including clicking with a mouse, scrolling along a webpage, touching a touch screen, and more.

In React apps, you can use event handlers to update state data, trigger prop changes, or prevent default browser actions. To do this, React uses a `SyntheticEvent` wrapper instead of the native `Event` interface. `SyntheticEvent` closely emulates the standard browser event, but provides more consistent behavior for different web browsers. React also gives you tools to safely add and remove a `Window` event listener when a component mounts and unmounts from the Document Object Model (DOM), giving you control over `Window` events while preventing memory leaks from improperly removed listeners.

In this tutorial, you'll learn how to handle events in React. You'll build several sample components that handle user events, including a self-validating input component and an informative tooltip for the input form. Throughout the tutorial, you'll learn how to add event handlers to

components, pull information from the `SyntheticEvent`, and add and remove `Window` event listeners. By the end of this tutorial, you'll be able to work with a variety of event handlers and apply the catalog of events supported by React.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.22.0 and npm version 6.14.6. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- A React development environment set up with Create React App, with the non-essential boilerplate removed. To set this up, follow **Step 1 — Creating an Empty Project** of the How To Manage State on React Class Components tutorial. This tutorial will use `events-tutorial` as the project name.

- You will also need a basic knowledge of JavaScript and HTML, which you can find in our How To Build a Website with HTML series and in How To Code in JavaScript. Basic knowledge of CSS would also be useful, which you can find at the Mozilla Developer Network.

- You will be using React components, the `useState` Hook, and the `useReducer` Hook, which you can learn about in our tutorials How To

Create Custom Components in React and How To Manage State with Hooks on React Components.

Step 1 — Extracting Event Data with `SyntheticEvent`

In this step, you'll create a validating component using an `<input>` HTML element and the `onChange` event handler. This component will accept input and validate it, or make sure that the content adheres to a specific text pattern. You'll use the `SyntheticEvent` wrapper to pass event data into the callback function and update the component using the data from the `<input>`. You will also call functions from the `SyntheticEvent`, such as `preventDefault` to prevent standard browser actions.

In React, you don't need to select elements before adding event listeners. Instead, you add event handlers directly to your JSX using props. There are a large number of supported events in React, including common events such as `onClick` or `onChange` and less common events such as `onWheel`.

Unlike native DOM `onevent` handlers, React passes a special wrapper called `SyntheticEvent` to the event handler rather than the native browser `Event`. The abstraction helps reduce cross-browser inconsistencies and gives your components a standard interface for working with events. The API for `SyntheticEvent` is similar to the native `Event`, so most tasks are accomplished in the same manner.

To demonstrate this, you will start by making your validating input. First, you will create a component called `FileNamer`. This will be a `<form>`

element with an input for naming a file. As you fill in the input, you'll see the information update a preview box above the component. The component will also include a submit button to run the validation, but for this example the form will not actually submit anything.

First, create the directory:

```
mkdir src/components/FileNamer
```

Then open `FileNamer.js` in your text editor:

```
nano src/components/FileNamer/FileNamer.js
```

Inside `FileNamer.js`, create a wrapper `<div>`, then add another `<div>` with a class name of `preview` and a `<form>` element inside the wrapper by writing the following lines of code:

events-tutorial/src/components/FileNamer/FileNamer.js

```
import React from 'react';


export default function FileNamer() {
  return(
    <div className="wrapper">
      <div className="preview">
      </div>
      <form>
      </form>
    </div>
  )
}
```

Next, add an input element for the name to display in the preview box and a **Save** button. Add the following highlighted lines:

events-tutorial/src/components/FileNamer/FileNamer.js

```
import React from 'react';


export default function FileNamer() {
  return(
    <div className="wrapper">
      <div className="preview">
        <h2>Preview:</h2>
      </div>
      <form>
        <label>
          <p>Name:</p>
          <input name="name"/>
        </label>
        <div>
          <button>Save</button>
        </div>
      </form>
    </div>
  )
}
```

In the `preview` `<div>`, you added an `<h2>` element with the text `Preview`. This will be your preview box. Inside your form, you added an `<input>` surrounded by a `<label>` element with `Name:` as its text. Then you added a `button` called **Save** directly before the closing `<form>` tag.

Save and close the file.

Next, open `App.js`:

```
nano src/components/App/App.js
```

Import `FileNamer`, then render inside the `App` function by adding the following highlighted lines:

events-tutorial/src/components/App/App.js

```
import React from 'react';
import FileNamer from '../FileNamer/FileNamer';

function App() {
    return <FileNamer />
}

export default App;
```

Save and close the file. When you do the browser will refresh and you'll see your component.

```
Name element
```

Next, add some light styling to help define the sections and to add some padding and margins to the elements.

Open `FileNamer.css` in your text editor:

```
nano src/components/FileNamer/FileNamer.css
```

Give the `.preview` class a gray border and padding, then give the `.wrapper` class a small amount of padding. Display the items in a column using `flex` and `flex-direction`, and make all the text align left. Finally, remove the default button styles by removing the border and adding a black border:

events-tutorial/src/components/FileNamer/FileNamer.css

```
.preview {
    border: 1px darkgray solid;
    padding: 10px;
}

.wrapper {
    display: flex;
    flex-direction: column;
    padding: 20px;
    text-align: left;
}

.wrapper button {
    background: none;
    border: 1px black solid;
    margin-top: 10px;
}
```

Save and close the file. Then open `FileNamer.js`:

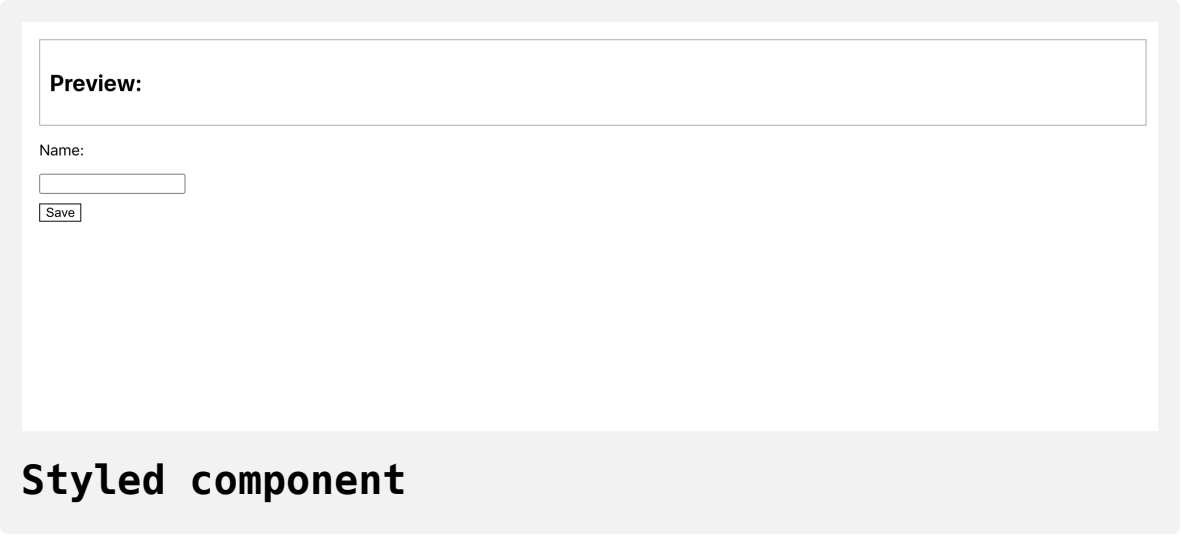
```
nano src/components/FileNamer/FileNamer.js
```

Import the styles to apply them to your component:

events-tutorial/src/components/FileNamer/FileNamer.js

```
import React from 'react';
import'./FileNamer.css';


export default function FileNamer( ) {
  return(
    <div className="wrapper">
      <div className="preview">
        <h2>Preview:</h2>
      </div>
      <form>
        <label>
          <p>Name:</p>
          <input name="name" />
        </label>
        <div>
          <button>Save</button>
        </div>
      </form>
    </div>
  )
}
```

Save the file. When you do, the browser will refresh and you'll find the component has the new styles.



Now that you have a basic component, you can add event handlers to the `<input>` element. But first, you'll need a place to store the data in the input field. Add the `useState` Hook to hold the input:

events-tutorial/src/components/FileNamer/FileNamer.js

```
import React, { useState } from 'react';
import './FileNamer.css';

export default function FileNamer() {
  const [name, setName] =useState('');
  return(
    <div className="wrapper">
      <div className="preview">
        <h2>Preview: {name}.js</h2>
      </div>
      <form>
        <label>
          <p>Name:</p>
          <input name="name" />
        </label>
        <div>
          <button>Save</button>
        </div>
      </form>
    </div>
  )
}
```

In this code, you destructured `useState` into a variable `name` to hold the input and a function called `setName` to update the data. Then you displayed the `name` in the preview section followed by the `.js` extension, as if the user were naming a file.

Now that you can store the input data, you can add an event handler to the `<input>` component. There are often several different event handlers you can use for a given task. In this case, your app needs to capture the data the user types into the element. The most common handler for this situation is `onChange`, which fires every time the component changes. However, you could also use keyboard events, such as `onKeyDown`, `onKeyPress`, and `onKeyUp`. The difference primarily has to do with when the event fires and the information passed to the `SyntheticEvent` object. For example, `onBlur`, an event for when an element becomes unfocused, fires before `onClick`. If you want to handle user information before another event fires, you can pick an earlier event.

Your choice of event is also determined by the type of data you want to pass to the `SyntheticEvent`. The `onKeyPress` event, for example, will include the `charCode` of the key that the user pressed, while `onChange` will not include the specific character code, but will include the full input. This is important if you want to perform different actions depending on which key the user pressed.

For this tutorial, use `onChange` to capture the entire input value and not just the most recent key. This will save you the effort of storing and concatenating the value on every change.

Create a function that takes the `event` as an argument and pass it to the `<input>` element using the `onChange` prop:

events-tutorial/src/components/FileNamer/FileNamer.js

```
import React, { useState } from 'react';
import './FileNamer.css';

export default function FileNamer() {
  const [name, setName] = useState('');
  return(
    <div className="wrapper">
      <div className="preview">
        <h2>Preview: {name}.js</h2>
      </div>
      <form>
        <label>
          <p>Name:</p>
          <input name="name" onChange={event=> {}}/>
        </label>
        <div>
          <button>Save</button>
        </div>
      </form>
    </div>
  )
}
```

As mentioned earlier, the `event` here is not the native browser event. It's the `SyntheticEvent` provided by React, which is often treated the same. In the rare case you need the native event, you can use the `nativeEvent` attribute on the `SyntheticEvent`.

Now that you have the event, pull out the current value from the `target.value` property of the event. Pass the value to `setName` to update the preview:

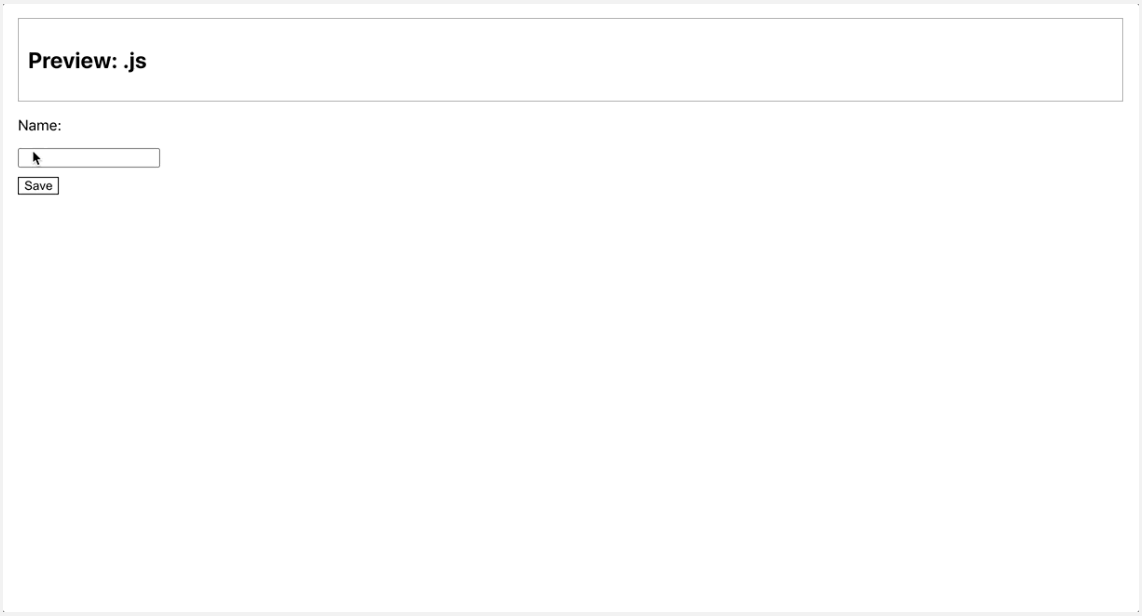
events-tutorial/src/components/FileNamer/FileNamer.js

```
import React, { useState } from 'react';
import './FileNamer.css';

export default function FileNamer() {
  const [name, setName] = useState('');
  return(
    <div className="wrapper">
      <div className="preview">
        <h2>Preview: {name}.js</h2>
      </div>
      <form>
        <label>
          <p>Name:</p>
          <input
           autoComplete="off"
           name="name"
           onChange={event => setName(event.target.value) }
         />
        </label>
        <div>
          <button>Save</button>
        </div>
      </form>
    </div>
```

```
  )
}
```

In addition, you set the attribute `autoComplete` to `"off"` to turn off browser suggestions.

Save the file. When you do, the page will reload, and when you type in the `<input>` you'll see an update in the preview.



Typing into the input element

Note: You could also access the name of the input using `event.target.name`. This would be useful if you were using the same event handler across multiple inputs, since the `name` would automatically match the `name` attribute of the component.

At this point, you have a working event handler. You are taking the user information, saving it to state, and updating another component with the data. But in addition to pulling information from an event, there are situations where you'll need to halt an event, such as if you wanted to prevent a form submission or prevent a keypress action.

To stop an event, call the `preventDefault` action on the event. This will stop the browser from performing the default behavior.

In the case of the `FileNamer` component, there are certain characters that could break the process of choosing a file that your app should forbid. For example, you wouldn't want a user to add a `*` to a filename since it conflicts with the wildcard character, which could be interpreted to refer to a different set of files. Before a user can submit the form, you'll want to check to make sure there are no invalid characters. If there is an invalid character, you'll stop the browser from submitting the form and display a message for the user.

First, create a Hook that will generate an `alert` boolean and a `setAlert` function. Then add a `<div>` to display the message if `alert` is true:

events-tutorial/src/components/FileNamer/FileNamer.js

```
import React, { useState } from 'react';
import './FileNamer.css';

export default function FileNamer() {
  const [name, setName] = useState('');
  const [alert, setAlert] =useState(false);

  return(
    <div className="wrapper">
      <div className="preview">
        <h2>Preview: {name}.js</h2>
      </div>
      <form>
        <label>
          <p>Name:</p>
          <input
            autoComplete="off"
            name="name"
            onChange={event => setName(event.target.value) }
          />
        </label>
        {alert >&&<div> Forbidden Character:*</div>}
        <div>
          <button>Save</button>
```

```
        </div>
      </form>
    </div>
  )
}
```

In this code, you used the `&&` operator to only show the new `<div>` if `alert` is set equal to `true` first. The message in the `<div>` will tell the user that the `*` character is not allowed in the input.

Next, create a function called `validate`. Use the regular expression `.test` method to find out if the string contains a `*`. If it does, you will prevent the form submission:

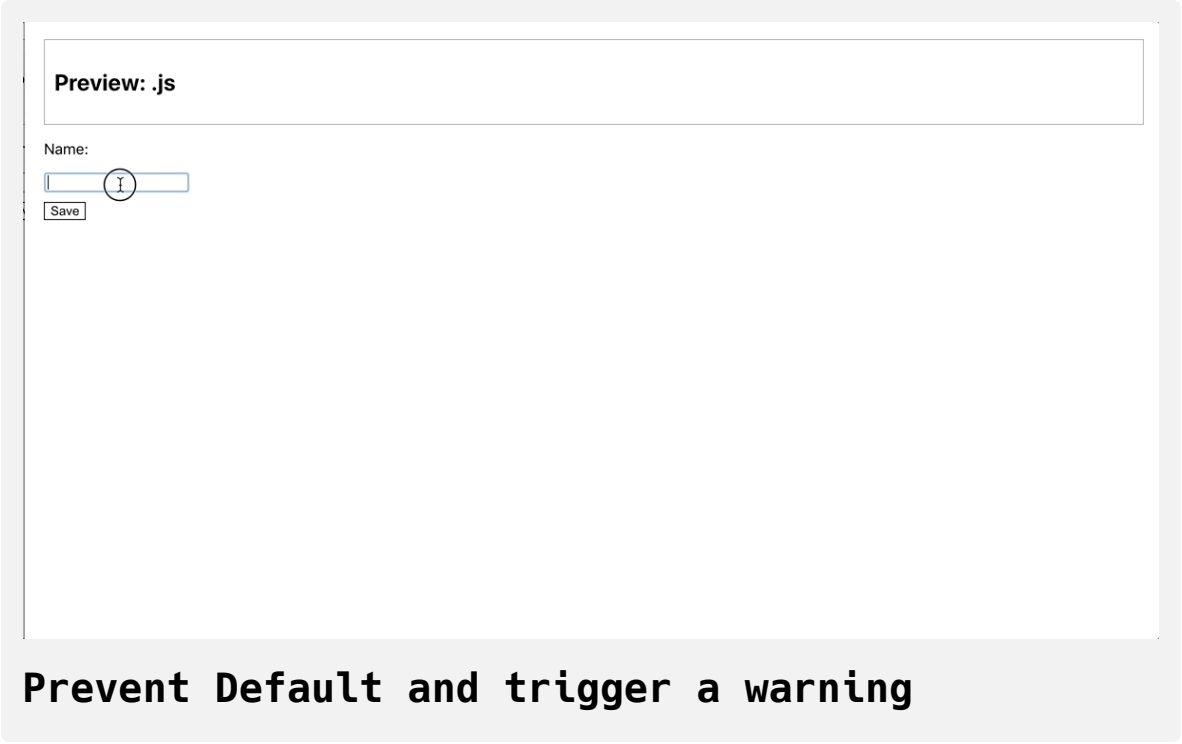
events-tutorial/src/components/FileNamer/FileNamer.js

```
import React, { useState } from 'react';
import './FileNamer.css';

export default function FileNamer( ) {
  const [name, setName] = useState('');
  const [alert, setAlert] = useState(false);
  const validate =event=> {
    if(/\*/.test(name)) {
      >event.preventDefault( );
      setAlert(true);
      return;
    }
      setAlert(false);
 };

  return(
    <div className="wrapper">
      <div className="preview">
        <h2>Preview: {name}.js</h2>
      </div>
      <form>
        <label>
          <p>Name:</p>
          <input
```

```
            autoComplete="off"
            name="name"
            onChange={event => setName(event.target.value) }
          />
        </label>
        {alert && <div> Forbidden Character: *</div>}
        <div>
          <button onClick={validate}>Save</button>
        </div>
      </form>
    </div>
  )
}
```

When the `validate` function is called and the test returns `true`, it will use `event.preventDefault` then call `setAlert(true)`. Otherwise, it will call `setAlert(false)`. In the last part of the code, you added the event handler to the `<button>` element with `onClick`.

Save the file. As before, you could have also used `onMouseDown`, but `onClick` is more common and thus allows you to avoid any unexpected side effects. This form doesn't have any submit actions, but by preventing the default action, you prevent the page from reloading:



Prevent Default and trigger a warning

Now you have a form that uses two event handlers: `onChange` and `onClick`. You are using the event handlers to connect user actions to the component and the application, making it interactive. In doing so, you learned to add events to DOM elements, and how there are several events that fire on the same action, but that provide different information in the `SyntheticEvent`. You also learned how to extract information from the `SyntheticEvent`, update other components by saving that data to state, and halt an event using `preventDefault`.

In the next step, you'll add multiple events to a single DOM element to handle a variety of user actions.

Step 2 — Adding Multiple Event Handlers to the Same Element

There are situations when a single component will fire multiple events, and you'll need to be able to connect to the different events on a single component. For example, in this step you'll use the `onFocus` and `onBlur` event handlers to give the user just-in-time information about the component. By the end of this step, you'll know more about the different supported events in React and how to add them to your components.

The `validate` function is helpful for preventing your form from submitting bad data, but it's not very helpful for user experience: The user only receives information about the valid characters after they've filled out the entire form. If there were multiple fields, it wouldn't give the user any feedback until the last step. To make this component more user friendly, display the allowed and disallowed characters when the user enters the field by adding an `onFocus` event handler.

First, update the `alert` `<div>` to include information about what characters are allowed. Tell the user alphanumeric characters are allowed and the `*` is not allowed:

events-tutorial/src/components/FileNamer/FileNamer.js

```
import React, { useState } from 'react';
import './FileNamer.css';

export default function FileNamer() {
...
  return(
    <div className="wrapper">
      <div className="preview">
        <h2>Preview: {name}.js</h2>
      </div>
      <form>
        <label>
          <p>Name:</p>
          <input
            autocomplete="off"
            name="name"
            onChange={event => setName(event.target.value) }
          />
        </label>
        {alert &&
         <div>
           <span role="img" aria-label="allowed">✅</span> Alpha
           <br />
           <span role="img" aria-label="not allowed">⛔</span>*
```

```
          </div>
        }
          <div>
            <button onClick={validate}>Save</button>
          </div>
        </form>
      </div>
    )
}
```

In this code, you used Accessible Rich Internet Applications (ARIA) standards to make the component more accessible to screen readers.

Next, add another event handler to the `<input>` element. You will alert the user about the allowed and disallowed characters when they activate the component by either clicking or tabbing into the input. Add in the following highlighted line:

events-tutorial/src/components/FileNamer/FileNamer.js

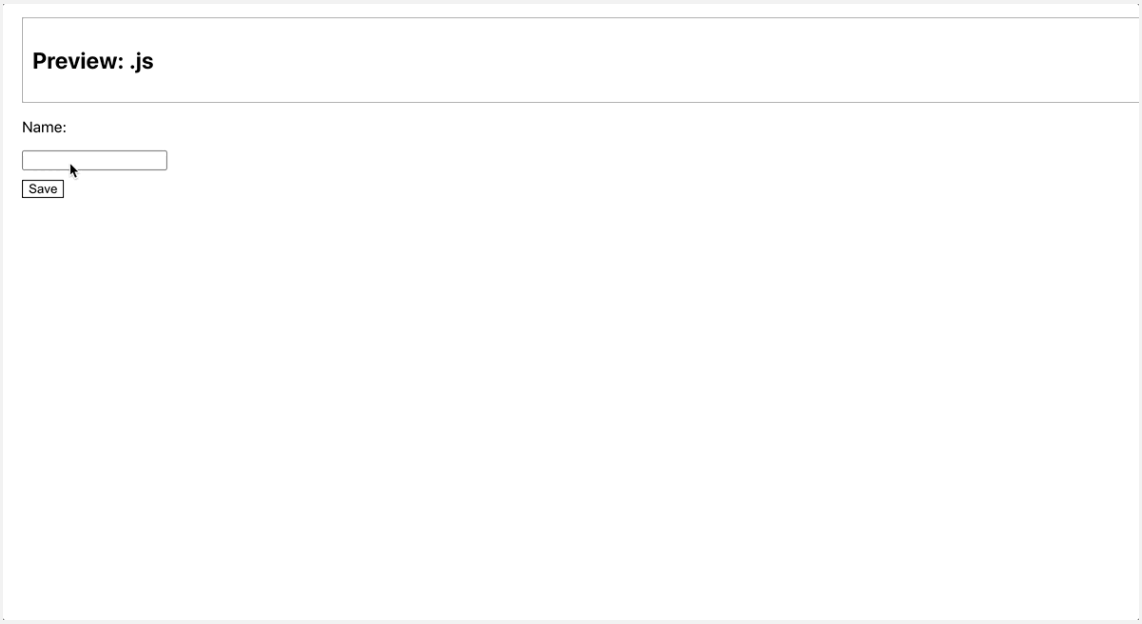
```
import React, { useState } from 'react';
import './FileNamer.css';

export default function FileNamer( ) {
...
  return(
    <div className="wrapper">
      <div className="preview">
        <h2>Preview: {name}.js</h2>
      </div>
      <form>
        <label>
          <p>Name:</p>
          <input
            autocomplete="off"
            name="name"
            onChange={event => setName(event.target.value) }
            onFocus={( ) =>setAlert(true)}
          />
        </label>
        {alert &&
          <div>
            <span role="img" aria-label="allowed">✔</span> Alph
            <br />
```

```
            <span role="img" aria-label="not allowed">⛔</span>
          </div>
        }
        <div>
          <button onClick={validate}>Save</button>
        </div>
      </form>
    </div>
  )
}
```

You added the `onFocus` event handler to the `<input>` element. This event triggers when the user selects the field. After adding the event handler, you passed an anonymous function to `onFocus` that will call `setAlert(true)` and display the data. In this case, you don't need any information from the `SyntheticEvent`; you only need to trigger an event when the user acts. React is still sending the `SyntheticEvent` to the function, but in the current situation you don't need to use the information in it.

Note: You could trigger the data display with `onClick` or even `onMouseDown`, but that wouldn't be accessible for users that use the keyboard to tab into the form fields. In this case, the `onFocus` event will handle both cases.

Save the file. When you do, the browser will refresh and the information will remain hidden until the user clicks on the input.



Trigger the event when clicking on the input

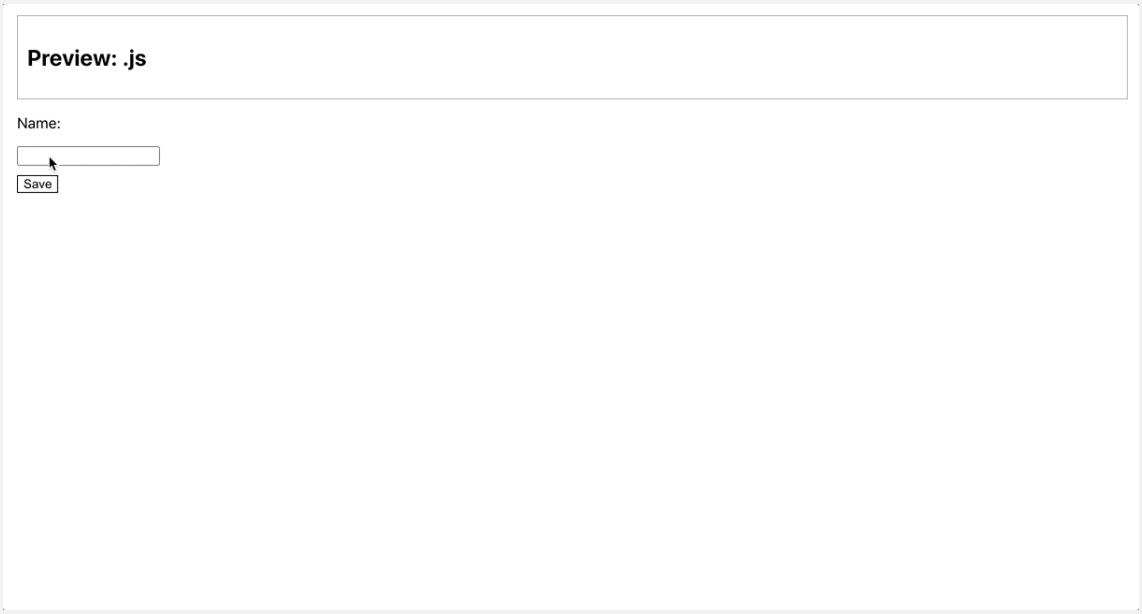
The user information now appears when the field is focused, but now the data is present for the duration of the component. There's no way to make it go away. Fortunately, there's another event called `onBlur` that fires when the user leaves an input. Add the `onBlur` event handler with an anonymous function that will set the `alert` to `false`. Like `onFocus`, this will work both when a user clicks away or when a user tabs away:

events-tutorial/src/components/FileNamer/FileNamer.js

```
import React, { useState } from 'react';
import './FileNamer.css';

export default function FileNamer() {
...
  return(
    <div className="wrapper">
      <div className="preview">
        <h2>Preview: {name}.js</h2>
      </div>
      <form>
        <label>
          <p>Name:</p>
          <input
            autocomplete="off"
            name="name"
            onBlur={() =>setAlert(false)}
            onChange={event => setName(event.target.value) }
            onFocus={() => setAlert(true)}
          />
        </label>
        {alert &&
          <div>
            <span role="img" aria-label="allowed">✔</span> Alph
```

```
            <br />
            <span role="img" aria-label="not allowed">⛔</span>
          </div>
        }
        <div>
          <button onClick={validate}>Save</button>
        </div>
      </form>
    </div>
  )
}
```

Save the file. When you do, the browser will refresh and the information will display when the user clicks on the element and disappear when the user clicks away:




```
Show information on focus and hide on blur
```

You can add as many event handlers as you need to an element. If you have an idea of an event you need, but aren't sure of the name, scroll through the supported events and you may find what you need.

In this step you added multiple event handlers to a single DOM element. You learned how different event handlers can handle a broad range of events—such as both click and tab—or a narrow range of events.

In the next step, you'll add global event listeners to the `Window` object to capture events that occur outside the immediate component.

Step 3 — Adding Window Events

In this step, you'll put the user information in a pop-up component that will activate when the user focuses an input and will close when the user clicks

anywhere else on the page. To achieve this effect, you'll add a global event listener to the `Window` object using the `useEffect` Hook. You'll also remove the event listener when the component unmounts to prevent memory leaks, when your app take up more memory than it needs to.

By the end of this step, you'll be able to safely add and remove event listeners on individual components. You'll also learn how to use the `useEffect` Hook to perform actions when a component mounts and unmounts.

In most cases, you'll add event handlers directly to DOM elements in your JSX. This keeps your code focused and prevents confusing situations where a component is controlling another component's behavior through the `Window` object. But there are times in which you'll need to add global event listeners. For example, you may want a scroll listener to load new content, or you may want to capture click events outside of a component.

In this tutorial, you only want to show the user the information about the input if they specifically ask for it. After you display the information, you'll want to hide it whenever the user clicks the page outside the component.

To start, move the `alert` display into a new `<div>` with a `className` of `information-wrapper`. Then add a new button with a `className` of `information` and an `onClick` event that will call `setAlert(true)`:

events-tutorial/src/components/FileNamer/FileNamer.js

```
import React, { useState } from 'react';
import './FileNamer.css';

export default function FileNamer() {
...
  return(
    <div className="wrapper">
      <div className="preview">
        <h2>Preview: {name}.js</h2>
      </div>
      <form>
        <label>
          <p>Name:</p>
          <input
            autocomplete="off"
            name="name"
            onChange={event => setName(event.target.value) }
          />
        </label>
        <div className="information-wrapper">
          <button
            className="information"
            onClick={() =>setAlert(true)}
            type="button"
```

```
          >
            more information
          </button>
          {alert &&
            <div className="popup">
              <span role="img" aria-label="allowed">✔</span> Alp
              <br />
              <span role="img" aria-label="not allowed">⛔</span>
            </div>
          }
        </div>
        <div>
          <button onClick={validate}>Save</button>
        </div>
      </form>
    </div>
  )
}
```

You also removed the `onFocus` and `onBlur` handlers from the `<input>` element to remove the behavior from the last step.

Save and close the file. Then open `FileNamer.css`:

```
nano src/components/FileNamer/FileNamer.css
```

Add some styling to absolutely position the `popup` information above the button. Then change the `<button>` with a class of `information` to be blue

with no border.

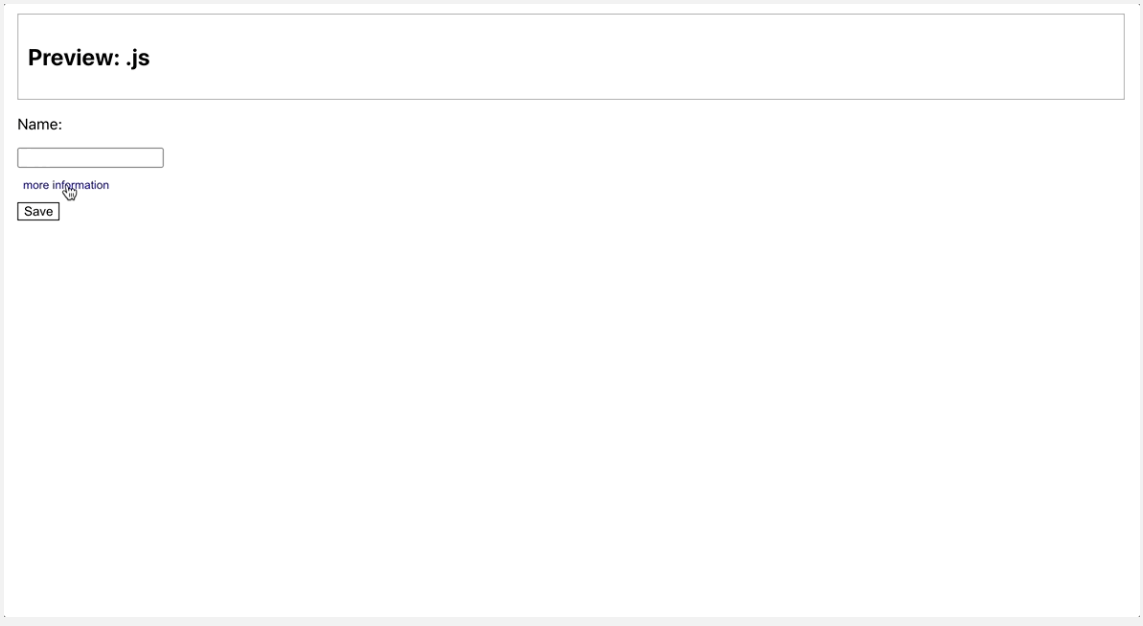
events-tutorial/src/components/FileNamer/FileNamer.css

```
.information {
   font-size: .75em;
   color: blue;
   cursor: pointer;
}

.wrapper button.information {
    border: none;
}

.information-wrapper {
   position: relative;
}

.popup {
    position: absolute;
    background: white;
    border: 1px darkgray solid;
    padding: 10px;
    top: -70px;
    left: 0;
}
```

```
.preview {

    border: 1px darkgray solid;

    padding: 10px;

}


.wrapper {

    display: flex;

    flex-direction: column;

    padding: 20px;
```

```
    text-align: left;
}

.wrapper button {
    background: none;
    border: 1px black solid;
    margin-top: 10px;
}
```

Save and close the file. When you do, the browser will reload, and when you click on `more information`, the information about the component will appear:



Trigger information pop-up

Now you can trigger the pop-up, but there's no way to clear it. To fix that problem, add a global event listener that calls `setAlert(false)` on any click outside of the pop-up.

The event listener would look something like this:

```
window.addEventListener('click', () => setAlert(false))
```

However, you have to be mindful about when you set the event listener in your code. You can't, for example, add an event listener at the top of your component code, because then every time something changed, the component would re-render and add a new event listener. Since your component will likely re-render many times, that would create a lot of unused event listeners that take up memory.

To solve this, React has a special Hook called `useEffect` that will run only when specific properties change. The basic structure is this:

```
useEffect(() => {
 // run code when anything in the array changes
}, [someProp, someOtherProp])
```

In the simplified example, React will run the code in the anonymous function whenever `someProp` or `someOtherProp` changes. The items in the array are called dependencies. This Hook listens for changes to the dependencies and then runs the function after the change.

Now you have the tools to add and remove a global event listener safely by using `useEffect` to add the event listener whenever `alert` is `true` and remove it whenever `alert` is `false`.

There is one more step. When the component unmounts, it will run any function that you return from inside of your `useEffect` Hook. Because of this, you'll also need to return a function that removes the event listener when the component unmounts.

The basic structure would be like this:

```
useEffect(() => {
 // run code when anything in the array changes
  return () => {} // run code when the component unmounts
}, [someProp, someOtherProp])
```

Now that you know the shape of your `useEffect` Hook, use it in your application. Open up `FileNamer.js`:

```
nano src/components/FileNamer/FileNamer.js
```

Inside, import `useEffect`, then add an empty anonymous function with a dependency of `alert` and `setAlert` in the array after the function:

events-tutorial/src/components/FileNamer/FileNamer.js

```
import React, { useEffect, useState } from 'react';
import './FileNamer.css';

export default function FileNamer() {
  const [name, setName] = useState('');
  const [alert, setAlert] = useState(false);

  useEffect(() => {
  }, [alert, setAlert]);
...
```

In this code, you added both `alert` and `setAlert`. To be complete, React recommends you add all external dependencies to the `useEffect` function. Since you will be calling the `setAlert` function, it can be considered a dependency. `setAlert` will not change after the first render, but it's a good practice to include anything that could be considered a dependency.

Next, inside the anonymous function, create a new function called `handleWindowClick` that calls `setAlert(false)`:

events-tutorial/src/components/FileNamer/FileNamer.js

```
import React, { useEffect, useState } from 'react';
import './FileNamer.css';

export default function FileNamer() {
  const [name, setName] = useState('');
  const [alert, setAlert] = useState(false);

  useEffect(() => {
    const handleWindowClick = () =>setAlert(false)
  }, [alert, setAlert]);
  ...
}
```

Then add a conditional that will call `window.addEventListener('click', handleWindowClick)` when `alert` is `true` and will call `window.removeEventListener('click', handleWindowClick)` when `alert` is `false`. This will add the event listener every time you trigger the pop-up and remove it everytime the pop-up is closed:

events-tutorial/src/components/FileNamer/FileNamer.js

```
import React, { useEffect, useState } from 'react';
import './FileNamer.css';

export default function FileNamer() {
  const [name, setName] = useState('');
  const [alert, setAlert] = useState(false);

  useEffect(() => {
    const handleWindowClick = () => setAlert(false)
    if(alert) {
      window.addEventListener('click', handleWindowClick);
    } else {
      window.removeEventListener('click', handleWindowClick);
    }
  }, [alert, setAlert]);
  ...
}
```

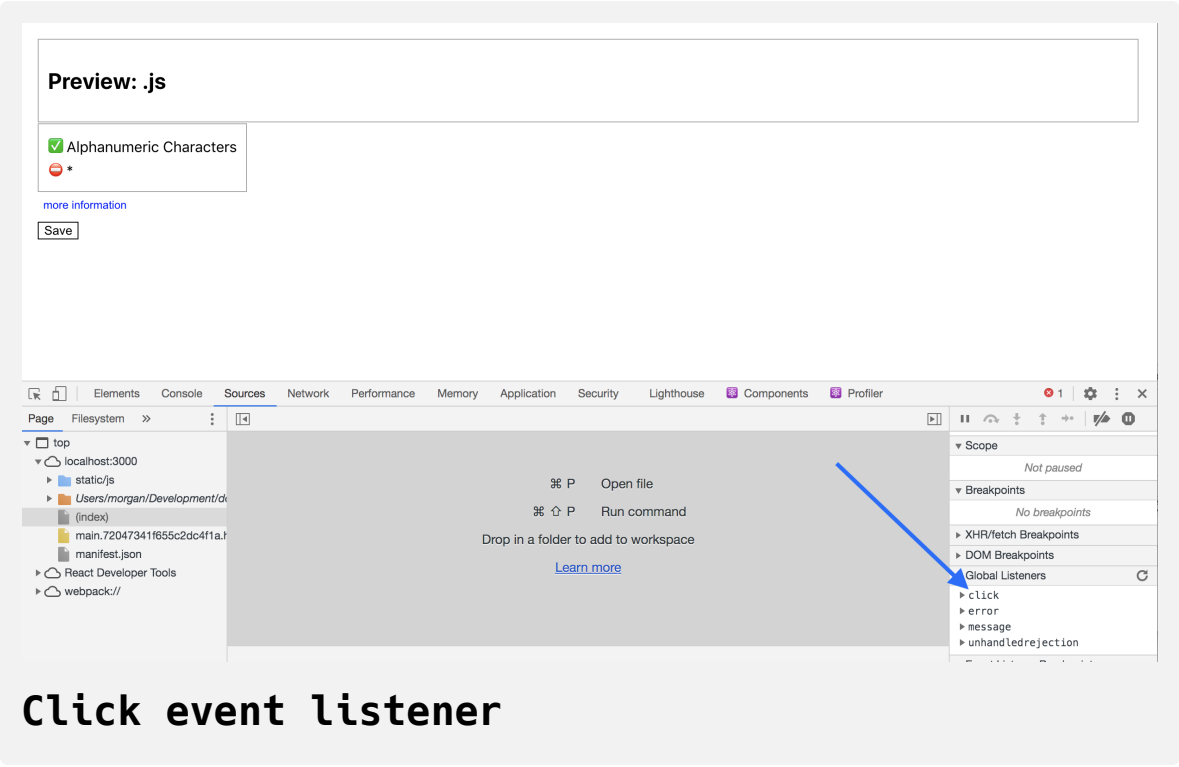
Finally, return a function that will remove the event listener. Once again, this will run when the component unmounts. There may not be a live event listener, but it's still worth cleaning up in situations where the listener still exists:

events-tutorial/src/components/FileNamer/FileNamer.js

```
import React, { useEffect, useState } from 'react';
import './FileNamer.css';

export default function FileNamer() {
  const [name, setName] = useState('');
  const [alert, setAlert] = useState(false);

  useEffect(() => {
    const handleWindowClick = () => setAlert(false)
    if(alert) {
      window.addEventListener('click', handleWindowClick);
    } else {
      window.removeEventListener('click', handleWindowClick)
    }
    return () =>window.removeEventListener('click', handleWindo
  }, [alert, setAlert]);
  ...
}
```

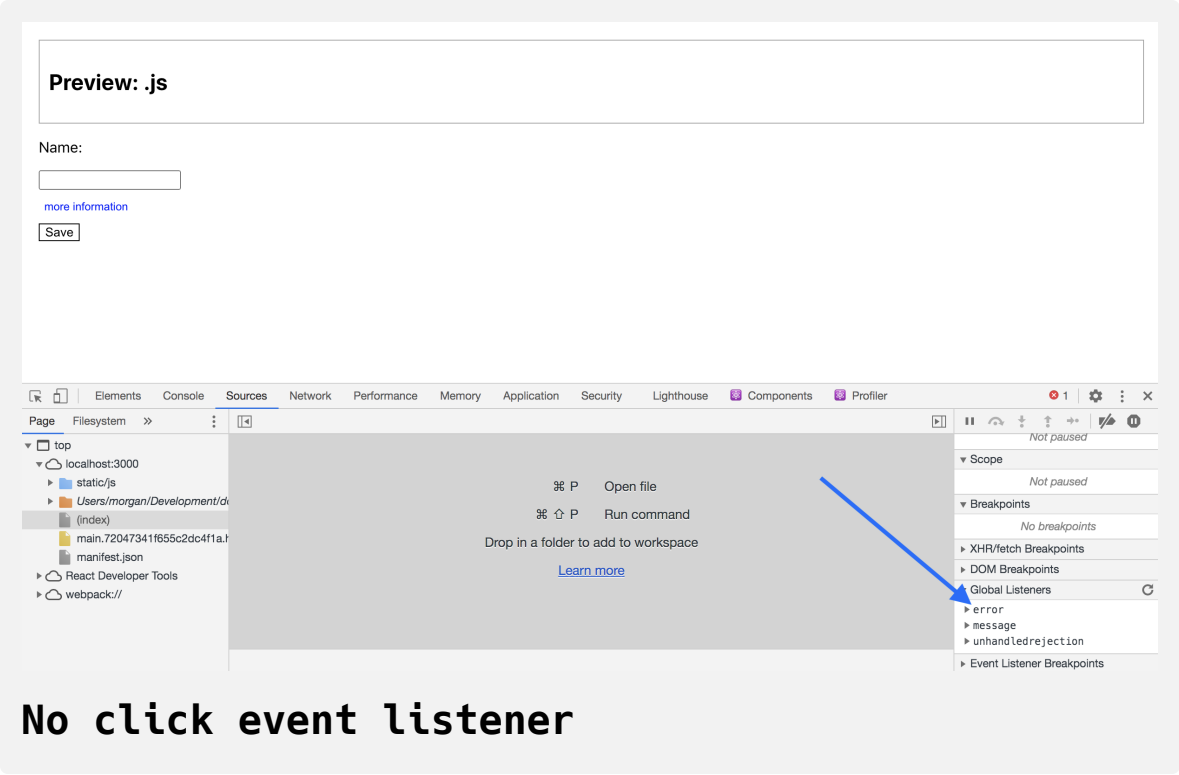
Save the file. When you do, the browser will refresh. If you click on the **more information** button, the message will appear. If you look at the global

event listeners in the developer tools, you'll see there is a `click` listener:



`Click event listener`

Click anywhere outside the component. The message will disappear and you'll no longer see the global click event listener.



Your `useEffect` Hook successfully added and removed a global event listener based on a user interaction. It wasn't tied to a specific DOM element, but was instead triggered by a change in the component state.

Note: From an accessibility standpoint, this component is not complete. If a user is not able to use the mouse, they will be stuck with an open pop-up because they would never be able to click outside the component. The solution would be to add another event listener for `keydown` that would also remove the message. The code would be nearly identical except the method would be `keydown` instead of `click`.

In this step you added global event listeners inside a component. You also learned how to use the `useEffect` Hook to properly add and remove the event listener as the state changes and how to clean up event listeners when the component unmounts.

Conclusion

Event handlers give you the opportunity to align your components with user actions. These will give your applications a rich experience and will increase the interactive possibilities of your app. They will also give you the ability to capture and respond to user actions.

React's event handlers let you keep your event callbacks integrated with the HTML so that you can share functionality and design across an application. In most cases, you should focus on adding event handlers directly to DOM elements, but in situations where you need to capture events outside of the component, you can add event listeners and clean them up when they are no longer in use to prevent memory leaks and create performative applications.

If you would like to look at more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page. To learn more about events in JavaScript, read our Understanding Events in JavaScript and Using Event Emitters in Node.js tutorials.

How To Build Forms in React

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

Forms are a crucial component of React web applications. They allow users to directly input and submit data in components ranging from a login screen to a checkout page. Since most React applications are single page applications (SPAs), or web applications that load a single page through which new data is displayed dynamically, you won't submit the information directly from the form to a server. Instead, you'll capture the form information on the client-side and send or display it using additional JavaScript code.

React forms present a unique challenge because you can either allow the browser to handle most of the form elements and collect data through React change events, or you can use React to fully control the element by setting and updating the input value directly. The first approach is called an uncontrolled component because React is not setting the value. The second approach is called a controlled component because React is actively updating the input.

In this tutorial, you'll build forms using React and handle form submissions with an example app that submits requests to buy apples. You'll also learn the advantages and disadvantages of controlled and uncontrolled components. Finally, you'll dynamically set form properties to enable and

disable fields depending on the form state. By the end of this tutorial, you'll be able to make a variety of forms using text inputs, checkboxes, select lists, and more.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.20.1 and npm version 6.14.4. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- A React development environment set up with Create React App, with the non-essential boilerplate removed. To set this up, follow **Step 1 — Creating an Empty Project** of the How To Manage State on React Class Components tutorial. This tutorial will use `form-tutorial` as the project name.

- You will be using React events and Hooks, including the `useState` and the `useReducer` Hooks. You can learn about events in our How To Handle DOM and Window Events with React tutorial, and Hooks at How to Manage State with Hooks on React Components.

- You will also need a basic knowledge of JavaScript and HTML, which you can find in our How To Build a Website with HTML series and in How To Code in JavaScript. Basic knowledge of CSS would also be useful, which you can find at the Mozilla Developer Network.

Step 1 — Creating a Basic Form with JSX

In this step, you'll create an empty form with a single element and a submit button using JSX. You'll handle the form submit event and pass the data to another service. By the end of this step, you'll have a basic form that will submit data to an asynchronous function.

To begin, open `App.js`:

```
nano src/components/App/App.js
```

You are going to build a form for purchasing apples. Create a `<div>` with a `className` of `<wrapper>`. Then add an `<h1>` tag with the text "How About Them Apples" and an empty `form` element by adding the following highlighted code:

form-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';

function App() {
  return (
    <div className="wrapper">
      <h1>How About Them Apples</h1>
      <form>
      </form>
    </div>
  )
}

export default App;
```

Next, inside the `<form>` tag, add a `<fieldset>` element with an `<input>` element surrounded by a `<label>` tag. By wrapping the `<input>` element with a `<label>` tag, you are aiding screen readers by associating the label with the input. This will increase the accessibility of your application.

Finally, add a submit `<button>` at the bottom of the form:

form-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';

function App() {
  return(
    <div className="wrapper">
      <h1>How About Them Apples</h1>
      <form>
      <fieldset>
         <label>
           <p>Name</p>
           <input name="name" />
         </label>
       </fieldset>
       <button type="submit">Submit</button>
      </form>
    </div>
  )
}

export default App;
```

Save and close the file. Then open `App.css` to set the styling:

```
nano src/components/App/App.css
```

Add `padding` to the `.wrapper` and `margin` to the `fieldset` to give some space between elements:

form-tutorial/src/components/App/App.css

```
.wrapper {
    padding: 5px 20px;
}

.wrapper fieldset {
    margin: 20px 0;
}
```

Save and close the file. When you do, the browser will reload and you'll see a basic form.

Basic form with a field for “name” and a submit button

If you click on the **Submit** button, the page will reload. Since you are building a single page application, you will prevent this standard behavior for a button with a `type="submit"`. Instead, you’ll handle the `submit` event inside the component.

Open `App.js`:

```
nano src/components/App/App.js
```

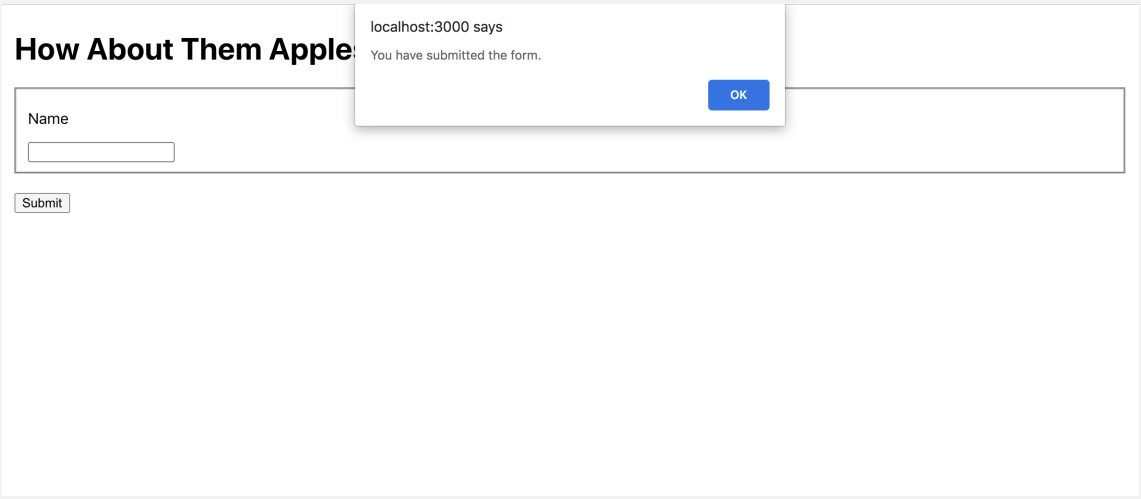
To handle the event, you’ll add an event handler to the `<form>` element, not the `<button>`. Create a function called `handleSubmit` that will take the `SyntheticEvent` as an argument. The `SyntheticEvent` is a wrapper around the standard `Event` object and contains the same interface. Call `.preventDefault` to stop the page from submitting the form then trigger an `alert` to show that the form was submitted:

form-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';

function App() {
  const handleSubmit = event => {
   event.preventDefault();
   alert('You have submitted the form.')
 }

  return(
    <div className="wrapper">
      <h1>How About Them Apples</h1>
      <form onSubmit={handleSubmit}>
        <fieldset>
          <label>
            <p>Name</p>
            <input name="name" />
          </label>
        </fieldset>
        <button type="submit">Submit</button>
      </form>
    </div>
  )
}
```

```
export default App;
```

Save the file. When you do the browser will reload. If you click the submit button, the alert will pop up, but the window will not reload.



Form submit alert

In many React applications, you'll send the data to an external service, like a Web API. When the service resolves, you'll often show a success message, redirect the user, or do both.

To simulate an API, add a `setTimeout` function in the `handleSubmit` function. This will create an asynchronous operation that waits a certain amount of time before completing, which behaves similarly to a request for external data. Then use the `useState` Hook to create a `submitting` variable

and a `setSubmitting` function. Call `setSubmitting(true)` when the data is submitted and call `setSubmitting(false)` when the timeout is resolved:

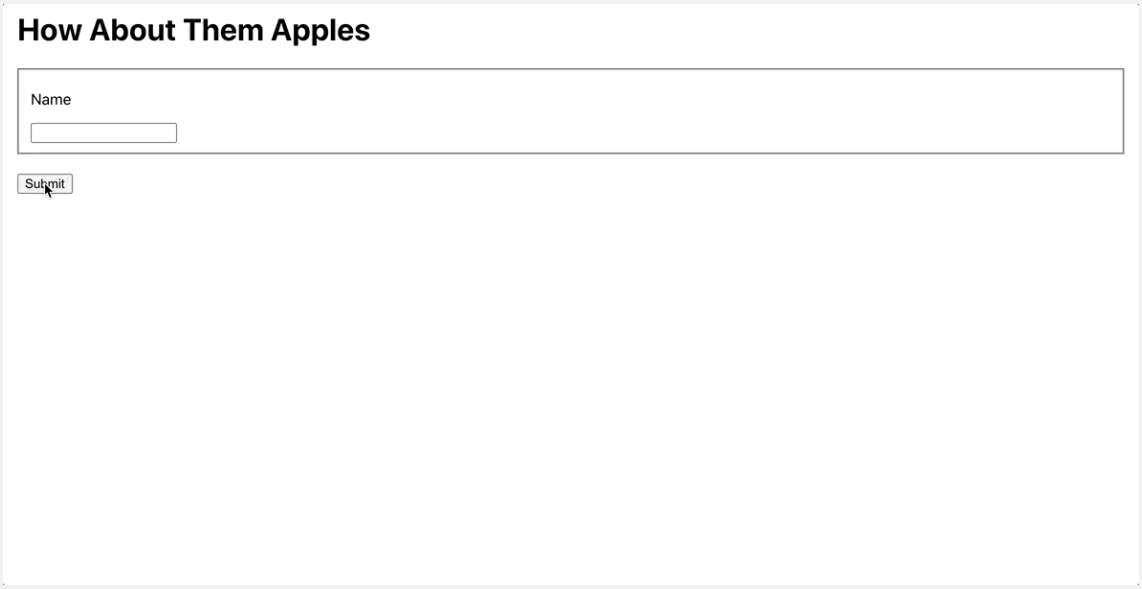
form-tutorial/src/components/App/App.js

```
import React, { useState } from 'react';
import './App.css';

function App() {
  const [submitting, setSubmitting] = useState(false);
  const handleSubmit = event => {
    event.preventDefault();
   setSubmitting(true);

   setTimeout(() => {
     setSubmitting(false);
   },3000)
 }

  return(
    <div className="wrapper">
      <h1>How About Them Apples</h1>
      {submitting &&
       <div>Submtting Form...</div>
     }
      <form onSubmit={handleSubmit}>
        <fieldset>
          <label>
            <p>Name</p>
```

```
            <input name="name" />
          </label>
        </fieldset>
        <button type="submit">Submit</button>
      </form>
    </div>
  )
}

export default App;
```

In addition, you will alert the user that their form is submitting by displaying a short message in the HTML that will display when `submitting` is `true`.

Save the file. When you do, the browser will reload and you'll receive the message on submit:



Form submitting shows message for 3 seconds

Now you have a basic form that handles the submit event inside the React component. You've connected it to your JSX using the `onSubmit` event handler and you are using Hooks to conditionally display an alert while the `handleSubmit` event is running.

In the next step, you'll add more user inputs and save the data to state as the user fills out the form.

Step 2 — Collecting Form Data Using Uncontrolled Components

In this step, you'll collect form data using uncontrolled components. An uncontrolled component is a component that does not have a `value` set by React. Instead of setting the data on the component, you'll connect to the `onChange` event to collect the user input. As you build the components, you'll

learn how React handles different input types and how to create a reusable function to collect form data into a single object.

By the end of this step, you'll be able to build a form using different form elements, including dropdowns and checkboxes. You'll also be able to collect, submit, and display form data.

> **Note:** In most cases, you'll use controlled components for your React application. But it's a good idea to start with uncontrolled components so that you can avoid subtle bugs or accidental loops that you might introduce when incorrectly setting a value.

Currently, you have a form that can submit information, but there is nothing to submit. The form has a single `<input>` element, but you are not collecting or storing the data anywhere in the component. In order to be able to store and process the data when the user submits a form, you'll need to create a way to manage state. You'll then need to connect to each input using an event handler.

Inside `App.js`, use the `useReducer` Hook to create a `formData` object and a `setFormData` function. For the reducer function, pull the `name` and `value` from the `event.target` object and update the `state` by spreading the current state while adding the `name` and `value` at the end. This will create a state object that preserves the current state while overwriting specific values as they change:

form-tutorial/src/components/App/App.js

```
import React, { useReducer, useState } from 'react';
import './App.css';

const formReducer = (state, event) => {
 return {
   ...state,
   [event.target.name]: event.target.value
 }
}

function App() {
  const [formData, setFormData]= useReducer(formReducer, {});
  const [submitting, setSubmitting] = useState(false);

  const handleSubmit = event => {
    event.preventDefault();
    setSubmitting(true);

    setTimeout(() => {
      setSubmitting(false);
    }, 3000)
  }

  return(
    <div className="wrapper">
```

```
      <h1>How About Them Apples</h1>
      {submitting &&
        <div>Submtting Form...</div>
      }
      <form onSubmit={handleSubmit}>
        <fieldset>
          <label>
            <p>Name</p>
            <input name="name" onChange={setFormData}/>
          </label>
        </fieldset>
        <button type="submit">Submit</button>
      </form>
    </div>
  )
}

export default App;
```

After making the reducer, add `setFormData` to the `onChange` event handler on the input. Save the file. When you do, the browser will reload. However, if you try and type in the input, you'll get an error:

Error with SyntheticEvent

The problem is that the `SyntheticEvent` is reused and cannot be passed to an asynchronous function. In other words, you can't pass the event directly. To fix this, you'll need to pull out the data you need before calling the reducer function.

Update the reducer function to take an object with a property of `name` and `value`. Then create a function called `handleChange` that pulls the data from the `event.target` and passes the object to `setFormData`. Finally, update the `onChange` event handler to use the new function:

form-tutorial/src/components/App/App.js

```
import React, { useReducer, useState } from 'react';
import './App.css';
```

```
const formReducer = (state, event) => {
 return {
   ...state,
   [event.name]: event.value
 }
}

function App() {
  const [formData, setFormData] = useReducer(formReducer, {});
  const [submitting, setSubmitting] = useState(false);

  const handleSubmit = event => {
    event.preventDefault();
    setSubmitting(true);

    setTimeout(() => {
      setSubmitting(false);
    }, 3000);
  }

  const handleChange = event => {
    setFormData({
      name: event.target.name,
     value: event.target.value,
    });
  }

  return(
    <div className="wrapper">
      <h1>How About Them Apples</h1>
      {submitting &&
        <div>Submtting Form...</div>
      }
      <form onSubmit={handleSubmit}>
        <fieldset>
          <label>
            <p>Name</p>
            <input name="name" onChange={handleChange}/>
          </label>
        </fieldset>
        <button type="submit">Submit</button>
      </form>
    </div>
  )
}

export default App;
```

Save the file. When you do the page will refresh and you'll be able to enter data.

Now that you are collecting the form state, update the user display message to show the data in an unordered list (`<ul>`) element.

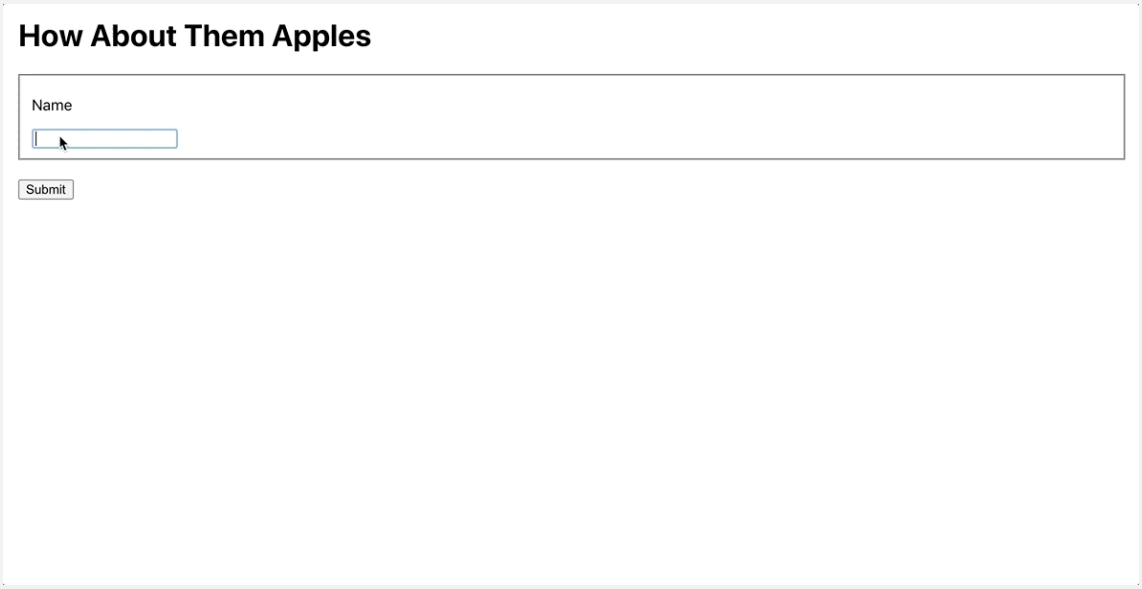
Convert the data to an array using `Object.entries`, then map over the data converting each member of the array to an `<li>` element with the name and the value. Be sure to use the `name` as the `key` prop for the element:

form-tutorial/src/components/App/App.js

```
...
  return(
    <div className="wrapper">
      <h1>How About Them Apples</h1>
      {submitting &&
       <div>
         You are submitting the following:
         <ul>
           {Object.entries(formData).map(([name, value]) => (
             <li key={name}><strong>{name}</strong>:{value.toSt
           ))}
         </ul>
       </div>
      }
      <form onSubmit={handleSubmit}>
        <fieldset>
          <label>
            <p>Name</p>
            <input name="name" onChange={handleChange}/>
          </label>
        </fieldset>
        <button type="submit">Submit</button>
      </form>
    </div>
  )
```

```
}


export default App;
```

Save the file. When you do the page will reload and you'll be able to enter and submit data:



Fill out the form and submit

Now that you have a basic form, you can add more elements. Create another `<fieldset>` element and add in a `<select>` element with different apple varieties for each `<option>`, an `<input>` with a `type="number"` and a `step="1"` to get a count that increments by 1, and an `<input>` with a `type="checkbox"` for a gift wrapping option.

For each element, add the `handleChange` function to the `onChange` event handler:

form-tutorial/src/components/App/App.js

```
...
  return(
    <div className="wrapper">
      <h1>How About Them Apples</h1>
      {submitting &&
        <div>
          You are submitting the following:
          <ul>
            {Object.entries(formData).map(([name, value]) => (
              <li key={name}><strong>{name}</strong>: {value.to
            ))}
          </ul>
        </div>
      }
      <form onSubmit={handleSubmit}>
        <fieldset>
          <label>
            <p>Name</p>
            <input name="name" onChange={handleChange}/>
          </label>
        </fieldset>
```

```
          <fieldset>
           <label>
             <p>Apples</p>
             <select name="apple" onChange={handleChange}>
                 <option value="">--Please choose an option--</op
                 <option value="fuji">Fuji</option>
                 <option value="jonathan">Jonathan</option>
                 <option value="honey-crisp">Honey Crisp</option>
             </select>
           </label>

           <label>

             <p>Count</p>

             <input type="number" name="count" onChange={handleCh

           </label>

           <label>

             <p>Gift Wrap</p>

             <input type="checkbox" name="gift-wrap" onChange={ha

           </label>

         </fieldset>

          <button type="submit">Submit</button>

       </form>

     </div>

   )

}


export default App;
```

Save the file. When you do, the page will reload and you'll have a variety of input types for your form:

Form with all input types

There is one special case here to take into consideration. The `value` for the gift wrapping checkbox will always be `"on"`, regardless of whether the item is checked or not. Instead of using the event's `value`, you'll need to use the `checked` property.

Update the `handleChange` function to see if the `event.target.type` is `checkbox`. If it is, pass the `event.target.checked` property as the `value` instead of `event.target.value`:

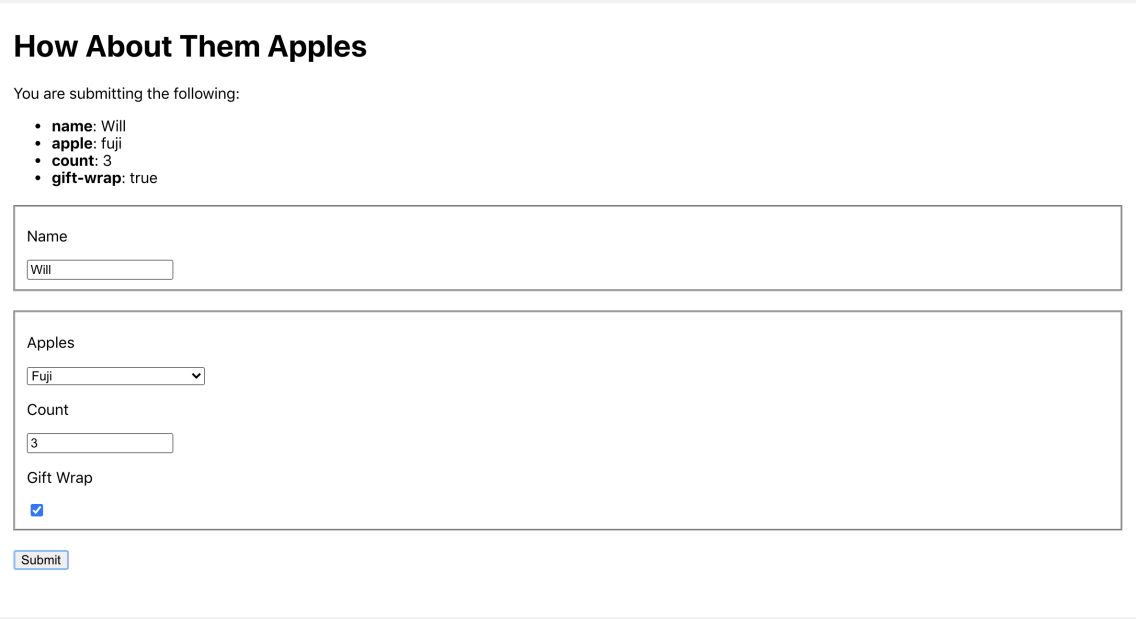
form-tutorial/src/components/App/App.js

```
import React, { useReducer, useState } from 'react';
import './App.css';


...


function App() {
  const [formData, setFormData] = useReducer(formReducer, {});
  const [submitting, setSubmitting] = useState(false);

  const handleSubmit = event => {
    event.preventDefault();
    setSubmitting(true);

    setTimeout(() => {
      setSubmitting(false);
    }, 3000);
  }

  const handleChange = event => {
   const isCheckbox = event.target.type === 'checkbox';
   setFormData({
     name: event.target.name,
     value: isCheckbox ? event.target.checked : event.target.va
   })
```

```
}
...
```

In this code, you use the ? ternary operator to make the conditional statement.

Save the file. After the browser refreshes, fill out the form and click submit. You'll find that the alert matches the data in the form:



Form elements submitting correct data

In this step, you learned how to create uncontrolled form components. You saved the form data to a state using the `useReducer` Hook and reused that data in different components. You also added different types of form

components and adjusted your function to save the correct data depending on the element type.

In the next step, you'll convert the components to controlled components by dynamically setting the component value.

Step 3 — Updating Form Data Using Controlled Components

In this step, you'll dynamically set and update data using controlled components. You'll add a `value` prop to each component to set or update the form data. You'll also reset the form data on submit.

By the end of this step, you'll be able to dynamically control form data using React state and props.

With uncontrolled components, you don't have to worry about synchronizing data. Your application will always hold on to the most recent changes. But there are many situations where you'll need to both read from and write to an input component. To do this, you'll need the component's value to be dynamic.

In the previous step, you submitted a form. But after the form submission was successful, the form still contained the old stale data. To erase the data from each input, you'll need to change the components from uncontrolled components to controlled components.

A controlled component is similar to an uncontrolled component, but React updates the `value` prop. The downside is that if you are not careful and do

not properly update the `value` prop the component will appear broken and won’t seem to update.

In this form, you are already storing the data, so to convert the components, you’ll update the `value` prop with data from the `formData` state. There is one problem, though: the `value` cannot be `undefined`. If your value is `undefined`, you’ll receive an error in your console.

Since your initial state is an empty object, you’ll need to set the value to be either the value from `formData` or a default value such as an empty string. For example, the value for the name would be `formData.name || ''`:

form-tutorial/src/components/App/App.js

```
...
  return(
    <div className="wrapper">
      <h1>How About Them Apples</h1>
      {submitting &&
        <div>
          You are submitting the following:
          <ul>
            {Object.entries(formData).map(([name, value]) => (
              <li key={name}><strong>{name}</strong>: {value.to
            ))}
          </ul>
        </div>
      }
      <form onSubmit={handleSubmit}>
        <fieldset>
          <label>
            <p>Name</p>
            <input name="name" onChange={handleChange} value={f
          </label>
        </fieldset>
        <fieldset>
          <label>
            <p>Apples</p>
            <select name="apple" onChange={handleChange} value=
```

```
                <option value="">--Please choose an option--</o
                <option value="fuji">Fuji</option>
                <option value="jonathan">Jonathan</option>
                <option value="honey-crisp">Honey Crisp</option
              </select>
            </label>
            <label>
              <p>Count</p>
              <input type="number" name="count" onChange={handleC
                value={formData.count || ''}/>
            </label>
            <label>
              <p>Gift Wrap</p>
              <input type="checkbox" name="gift-wrap" onChange={h
                checked={formData['gift-wrap'] || false}/>
            </label>
          </fieldset>
          <button type="submit">Submit</button>
        </form>
      </div>
    )
}


export default App;
```

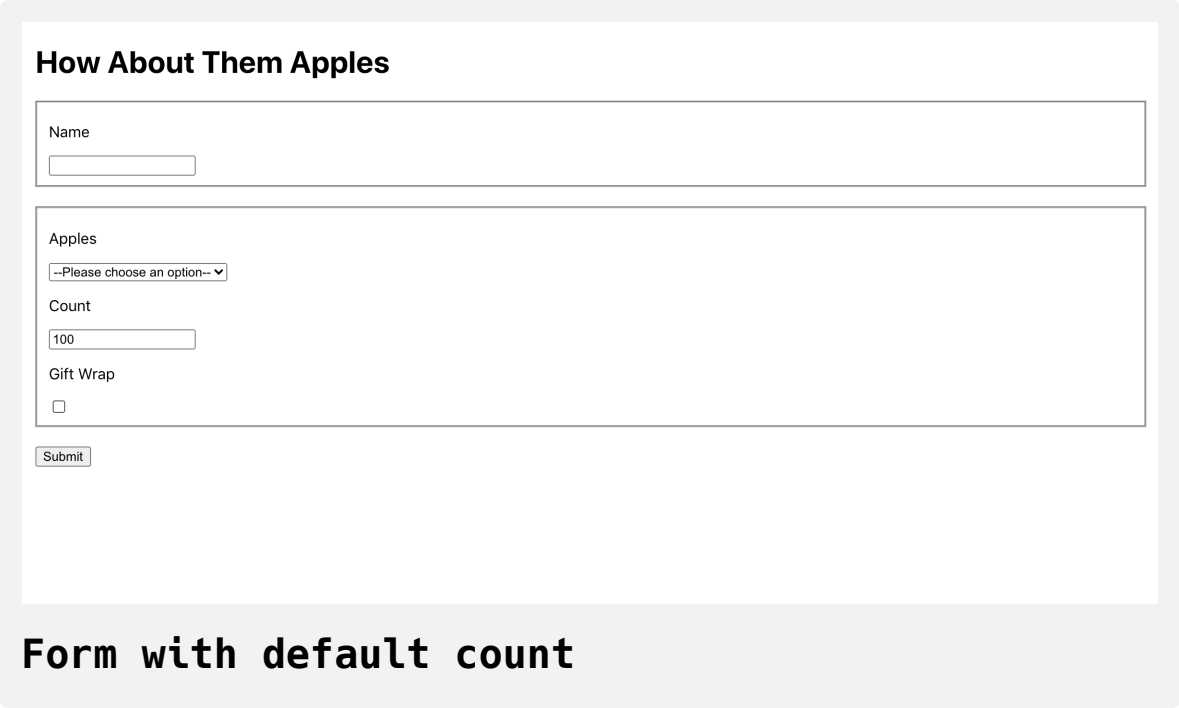
As before, the checkbox is a little different. Instead of setting a value, you'll need to set the `checked` attribute. If the attribute is truthy, the browser will show the box as checked. Set the initial `checked` attribute to false with `formData['gift-wrap'] || false`.

If you want to pre-fill the form, add some default data to the `formData` state. Set a default value for the `count` by giving `formState` a default value of `{ count: 100 }`. You could also set the default values in the initial object, but you'd need to filter out the falsy values before displaying the form information:

form-tutorial/src/components/App/App.js

```
...

function App() {
  const [formData, setFormData] = useReducer(formReducer, {
   count: 100,
 });
  const [submitting, setSubmitting] = useState(false);
...
```

Save the file. When you do, the browser will reload and you'll see the input with the default data:



Form with default count

Note: The `value` attribute is different from the `placeholder` attribute, which is native on browsers. The `placeholder` attribute shows information but will disappear as soon as the user makes a change; it is not stored on the component. You can actively edit the `value`, but a `placeholder` is just a guide for users.

Now that you have active components, you can clear the data on submit. To do so, add a new condition in your `formReducer`. If `event.reset` is truthy, return an object with empty values for each form element. Be sure to add a value for each input. If you return an empty object or an incomplete object, the components will not update since the value is `undefined`.

After you add the new event condition in the `formReducer`, update your submit function to reset the state when the function resolves:

form-tutorial/src/components/App/App.js

```
import React, { useReducer, useState } from 'react';
import './App.css';

const formReducer = (state, event) => {
  if(event.reset) {
   return {
     apple: '',
     count: 0,
     name: '',
     'gift-wrap': false,
   }
 }
  return {
    ...state,
    [event.name]: event.value
  }
}

function App() {
  const [formData, setFormData] = useReducer(formReducer, {
    count: 100
  });
  const [submitting, setSubmitting] = useState(false);

  const handleSubmit = event => {
```

```
    event.preventDefault();
    setSubmitting(true);

    setTimeout(() => {
      setSubmitting(false);
      setFormData({
       reset: true
     })
    }, 3000);
  }

...
```

Save the file. When you do, the browser will reload and the form will clear on submit.

`Save the form and then clear the data`

In this step, you converted your uncontrolled components to controlled components by setting the `value` or the `checked` attributes dynamically. You also learned how to refill data by setting a default state and how to clear the data by updating the form reducer to return default values.

In this next step, you'll set form component properties dynamically and disable a form while it is submitting.

Step 4 — Dynamically Updating Form Properties

In this step, you'll dynamically update form element properties. You'll set properties based on previous choices and disable your form during submit to prevent accidental multiple submissions.

Currently, each component is static. They do not change as the form changes. In most applications, forms are dynamic. Fields will change based

on the previous data. They'll validate and show errors. They may disappear or expand as you fill in other components.

Like most React components, you can dynamically set properties and attributes on components and they will re-render as the data changes.

Try setting an input to be `disabled` until a condition is met by another input. Update the **gift wrapping** checkbox to be disabled unless the user selects the `fuji` option.

Inside `App.js`, add the `disabled` attribute to the checkbox. Make the property truthy if the `formData.apple` is `fuji`:

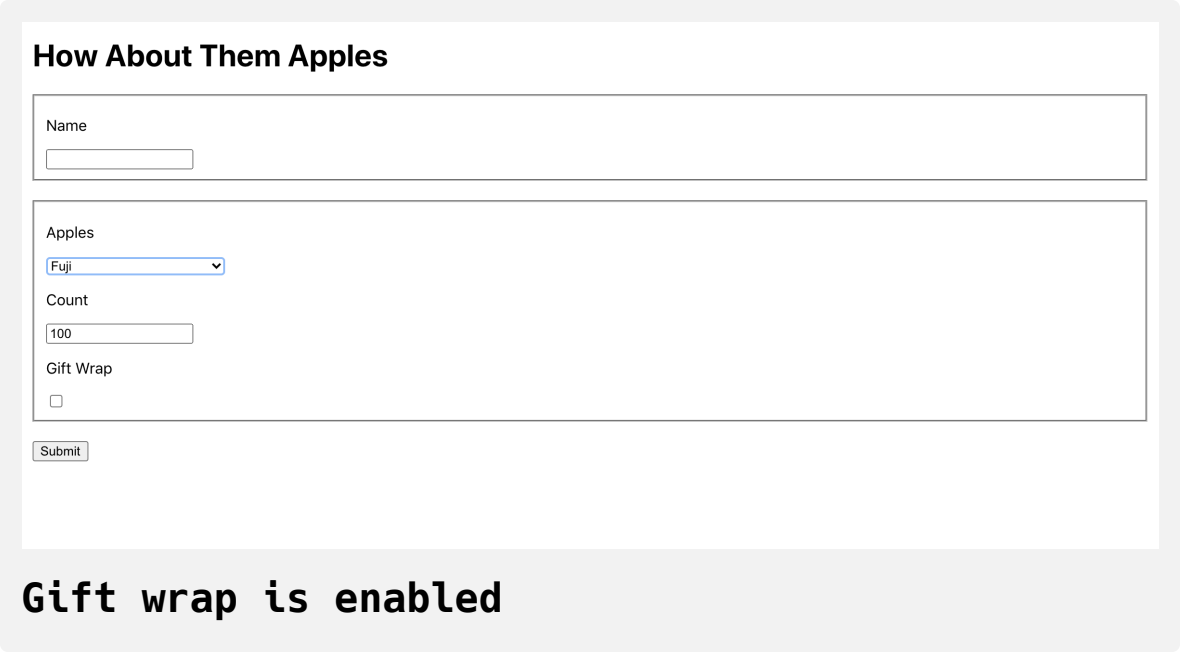
form-tutorial/src/components/App/App.js

```
...
        <fieldset>
          <label>
            <p>Apples</p>
            <select name="apple" onChange={handleChange} value=
                <option value="">--Please choose an option--</o
                <option value="fuji">Fuji</option>
                <option value="jonathan">Jonathan</option>
                <option value="honey-crisp">Honey Crisp</option
            </select>
          </label>
          <label>
            <p>Count</p>
            <input type="number" name="count" onChange={handleC
              value={formData.count || ''}/>
          </label>
          <label>
            <p>Gift Wrap</p>
            <input
              checked={formData['gift-wrap'] || false}
              disabled={formData.apple !== 'fuji'}
              name="gift-wrap"
              onChange={handleChange}
              type="checkbox"
            />
```

```
          </label>
        </fieldset>
        <button type="submit">Submit</button>
      </form>
    </div>
  )
}

export default App;
```

Save the file. When you do, the browser will reload and the checkbox will be disabled by default:

If you select the apple type of **Fuji**, the element will be enabled:



In addition to changing properties on individual components, you can modify entire groups of components by updating the `fieldset` component.

As an example, you can disable the form while the form is actively submitting. This will prevent double submissions and prevent the user from changing fields before the `handleSubmit` function fully resolves.

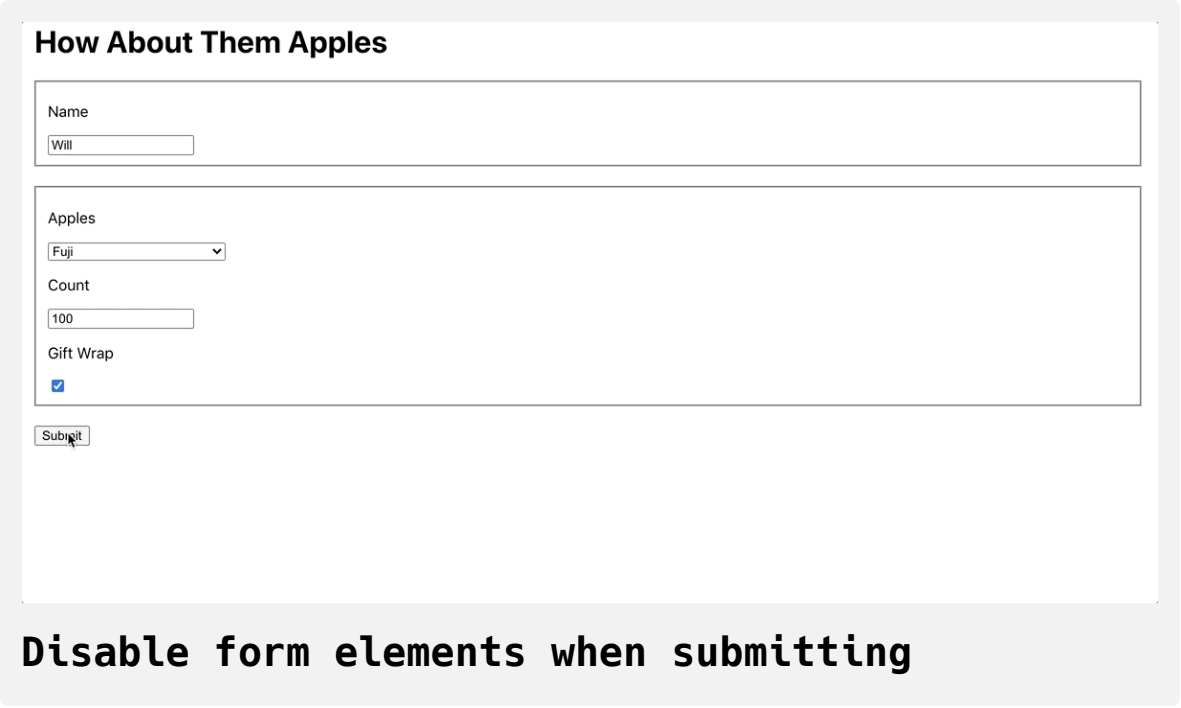
Add `disabled={submitting}` to each `<fieldset>` element and the `<button>` element:

form-tutorial/src/components/App/App.js

```
...
      <form onSubmit={handleSubmit}>
        <fieldset disabled={submitting}>
          <label>
            <p>Name</p>
            <input name="name" onChange={handleChange} value={f
          </label>
        </fieldset>
        <fieldset disabled={submitting}>
          <label>
            <p>Apples</p>
            <select name="apple" onChange={handleChange} value=
                <option value="">--Please choose an option--</o
                <option value="fuji">Fuji</option>
                <option value="jonathan">Jonathan</option>
                <option value="honey-crisp">Honey Crisp</option
            </select>
          </label>
          <label>
            <p>Count</p>
            <input type="number" name="count" onChange={handleC
              value={formData.count || ''}/>
          </label>
          <label>
            <p>Gift Wrap</p>
```

```
            <input
              checked={formData['gift-wrap'] || false}
              disabled={formData.apple !== 'fuji'}
              name="gift-wrap"
              onChange={handleChange}
              type="checkbox"
            />
          </label>
        </fieldset>
        <button type="submit" disabled={submitting}>Submit</but
      </form>
    </div>
  )
}

export default App;
```

Save the file, and the browser will refresh. When you submit the form, the fields will be disabled until the submitting function resolves:



`Disable form elements when submitting`

You can update any attribute on an input component. This is helpful if you need to change the `maxvalue` for a number input or if you need to add a dynamic `pattern` attribute for validation.

In this step, you dynamically set attributes on form components. You added a property to dynamically enable or disable a component based on the input from another component and you disabled entire sections using the `<fieldset>` component.

Conclusion

Forms are key to rich web applications. In React, you have different options for connecting and controlling forms and elements. Like other components, you can dynamically update properties including the `value` input elements. Uncontrolled components are best for simplicity, but might not fit situations

when a component needs to be cleared or pre-populated with data. Controlled components give you more opportunities to update the data, but can add another level of abstraction that may cause unintentional bugs or re-renders.

Regardless of your approach, React gives you the ability to dynamically update and adapt your forms to the needs of your application and your users.

If you would like to read more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Handle Async Data Loading, Lazy Loading, and Code Splitting with React

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

As a JavaScript web developer, asynchronous code gives you the ability to run some parts of your code while other parts are still waiting for data or resolving. This means that important parts of your app will not have to wait for less important parts before they render. With asynchronous code you can also update your application by requesting and displaying new information, giving users a smooth experience even when long functions and requests are processing in the background.

In React development, asynchronous programming presents unique problems. When you use React functional components for example, asynchronous functions can create infinite loops. When a component loads, it can start an asynchronous function, and when the asynchronous function resolves it can trigger a re-render that will cause the component to recall the asynchronous function. This tutorial will explain how to avoid this with a special Hook called `useEffect`, which will run functions only when specific data changes. This will let you run your asynchronous code deliberately instead of on each render cycle.

Asynchronous code is not just limited to requests for new data. React has a built-in system for lazy loading components, or loading them only when the user needs them. When combined with the default webpack configuration in Create React App, you can split up your code, reducing a large application into smaller pieces that can be loaded as needed. React has a special component called `Suspense` that will display placeholders while the browser is loading your new component. In future versions of React, you'll be able to use `Suspense` to load data in nested components without render blocking.

In this tutorial, you'll handle asynchronous data in React by creating an app that displays information on rivers and simulates requests to Web APIs with `setTimeout`. By the end of this tutorial, you'll be able to load asynchronous data using the `useEffect` Hook. You'll also be able to safely update the page without creating errors if the component unmounts before data resolution. Finally, you'll split a large application into smaller parts using code splitting.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.20.1 and npm version 6.14.4. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- A React development environment set up with Create React App, with the non-essential boilerplate removed. To set this up, follow **Step 1 — Creating an Empty Project** of the How To Manage State on React Class Components tutorial. This tutorial will use `async-tutorial` as the project name.

- You will be using React events and Hooks, including the `useState` and the `useReducer` Hooks. You can learn about events in our How To Handle DOM and Window Events with React tutorial, and Hooks at How to Manage State with Hooks on React Components.

- You will also need a basic knowledge of JavaScript and HTML, which you can find in our How To Build a Website with HTML series and in How To Code in JavaScript. Basic knowledge of CSS would also be useful, which you can find at the Mozilla Developer Network.

Step 1 — Loading Asynchronous Data with `useEffect`

In this step, you'll use the `useEffect` Hook to load asynchronous data into a sample application. You'll use the Hook to prevent unnecessary data fetching, add placeholders while the data is loading, and update the component when the data resolves. By the end of this step, you'll be able to load data with `useEffect` and set data using the `useState` Hook when it resolves.

To explore the topic, you are going to create an application to display information about the longest rivers in the world. You'll load data using an asynchronous function that simulates a request to an external data source.

First, create a component called `RiverInformation`. Make the directory:

```
mkdir src/components/RiverInformation
```

Open `RiverInformation.js` in a text editor:

```
nano src/components/RiverInformation/RiverInformation.js
```

Then add some placeholder content:

async-tutorial/src/components/RiverInformation/RiverInformation.js

```
import React from 'react';


export default function RiverInformation() {
  return(
    <div>
      <h2>River Information</h2>
    </div>
  )
}
```

Save and close the file. Now you need to import and render the new component to your root component. Open `App.js`:

```
nano src/components/App/App.js
```

Import and render the component by adding in the highlighted code:

async-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';
import RiverInformation from '../RiverInformation/RiverInformat

function App() {
  return (
    <div className="wrapper">
      <h1>World's Longest Rivers</h1>
      <RiverInformation />
    </div>
  );
}

export default App;
```

Save and close the file.

Finally, in order to make the app easier to read, add some styling. Open `App.css`:

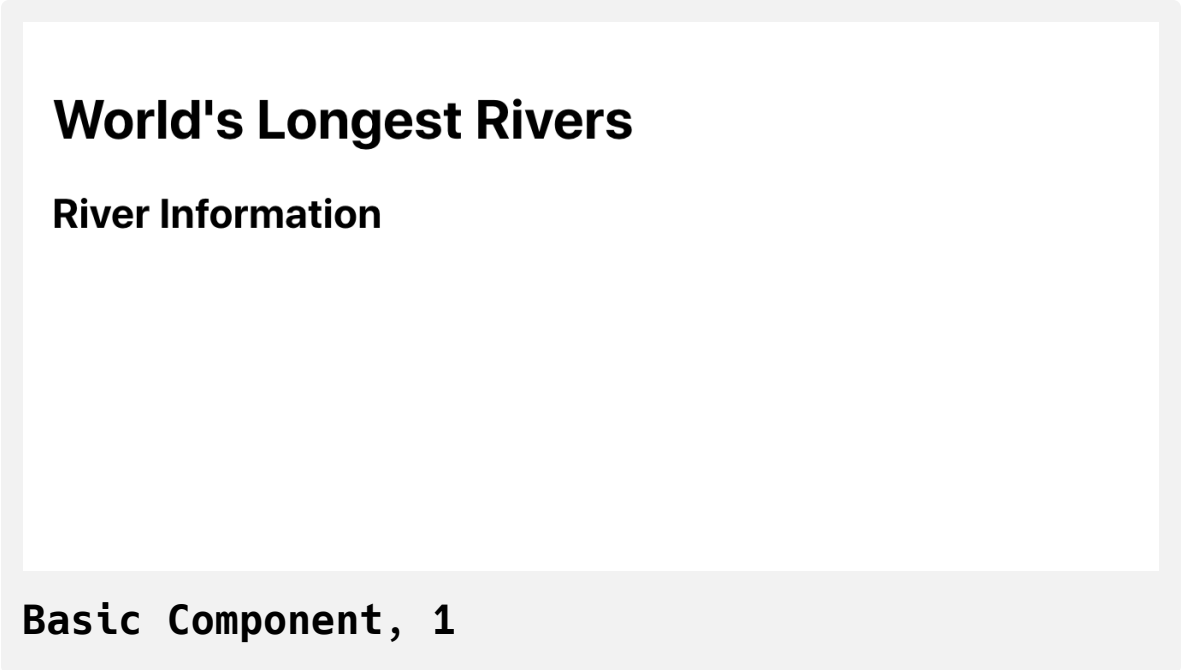
```
nano src/components/App/App.css
```

Add some padding to the `wrapper` class by replacing the CSS with the following:

async-tutorial/src/components/App/App.css

```
.wrapper {
    padding: 20px
}
```

Save and close the file. When you do, the browser will refresh and render the basic components.



In this tutorial, you'll make generic services for returning data. A service refers to any code that can be reused to accomplish a specific task. Your component doesn't need to know how the service gets its information. All it needs to know is that the service will return a Promise. In this case, the data request will be simulated with `setTimeout`, which will wait for a specified amount of time before providing data.

Create a new directory called `services` under the `src/` directory:

```
mkdir src/services
```

This directory will hold your asynchronous functions. Open a file called `rivers.js`:

```
nano src/services/rivers.js
```

Inside the file, export a function called `getRiverInformation` that returns a promise. Inside the promise, add a `setTimeout` function that will resolve the promise after `1500` milliseconds. This will give you some time to see how the component will render while waiting for data to resolve:

async-tutorial/src/services/rivers.js

```
export function getRiverInformation() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve({
        continent: 'Africa',
        length: '6,650 km',
        outflow: 'Mediterranean'
      })
    }, 1500)
  })
}
```

In this snippet, you are hard-coding the river information, but this function will be similar to any asynchronous functions you may use, such as an API call. The important part is that the code returns a promise.

Save and close the file.

Now that you have a service that returns the data, you need to add it to your component. This can sometimes lead to a problem. Suppose you called the asynchronous function inside of your component and then set the data to a variable using the `useState` Hook. The code will be like this:

```
import React, { useState } from 'react';
import { getRiverInformation } from '../../services/rivers';

export default function RiverInformation() {
  const [riverInformation, setRiverInformation] = useState({});

  getRiverInformation()
  .then(d => {
    setRiverInformation(d)
  })

  return(
    ...
  )
}
```

When you set the data, the Hook change will trigger a components re-render. When the component re-renders, the `getRiverInformation` function will run again, and when it resolves it will set the state, which will trigger another re-render. The loop will continue forever.

To solve this problem, React has a special Hook called `useEffect` that will only run when specific data changes.

The `useEffect` Hook accepts a function as the first argument and an array of triggers as the second argument. The function will run on the first render

after the layout and paint. After that, it will only run if one of the triggers changes. If you supply an empty array, it will only run one time. If you do not include an array of triggers, it will run after every render.

Open `RiverInformation.js`:

```
nano src/components/RiverInformation/RiverInformation.js
```

Use the `useState` Hook to create a variable called `riverInformation` and a function called `setRiverInformation`. You'll update the component by setting the `riverInformation` when the asynchronous function resolves. Then wrap the `getRiverInformation` function with `useEffect`. Be sure to pass an empty array as a second argument. When the promise resolves, update the `riverInformation` with the `setRiverInformation` function:

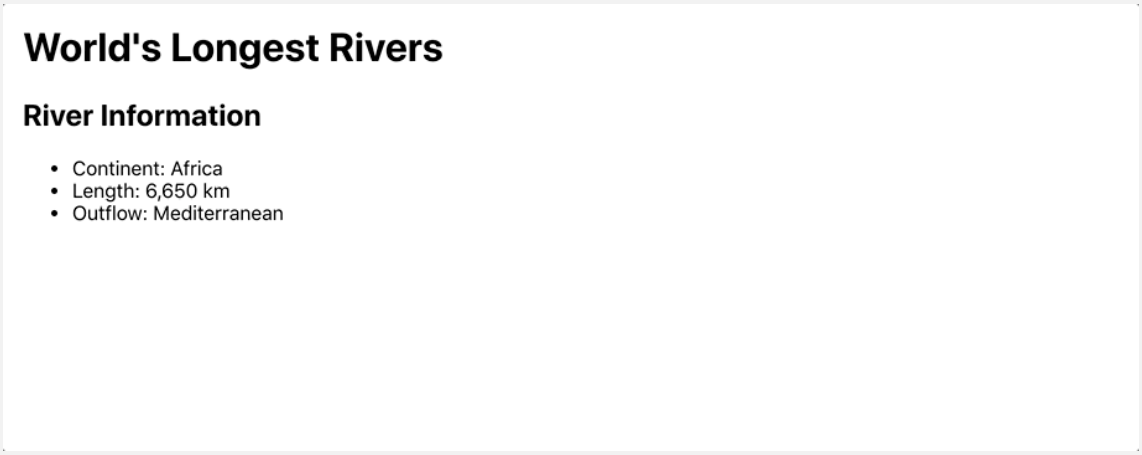
async-tutorial/src/components/RiverInformation/RiverInformation.js

```
import React, { useEffect, useState } from 'react';
import { getRiverInformation } from '../../services/rivers';

export default function RiverInformation() {
  const [riverInformation, setRiverInformation] = useState({});

  useEffect(() => {
   getRiverInformation()
   .then(data =>
     setRiverInformation(data)
   );
  }, [])


  return(
    <div>
      <h2>River Information</h2>
      <ul>
        <li>Continent: {riverInformation.continent}</li>
        <li>Length: {riverInformation.length}</li>
        <li>Outflow: {riverInformation.outflow}</li>
      </ul>
    </div>
```

```
  )
}
```

After the asynchronous function resolves, update an unordered list with the new information.

Save and close the file. When you do the browser will refresh and you'll find the data after the function resolves:



River Information Updating After Load, 2

Notice that the component renders before the data is loaded. The advantage with asynchronous code is that it won't block the initial render. In this case, you have a component that shows the list without any data, but you could also render a spinner or a scalable vector graphic (SVG) placeholder.

There are times when you'll only need to load data once, such as if you are getting user information or a list of resources that never change. But many

times your asynchronous function will require some arguments. In those cases, you'll need to trigger your use `useEffect` Hook whenever the data changes.

To simulate this, add some more data to your service. Open `rivers.js`:

```
nano src/services/rivers.js
```

Then add an object that contains data for a few more rivers. Select the data based on a `name` argument:

async-tutorial/src/services/rivers.js

```
const rivers = {
 nile: {
   continent: 'Africa',
   length: '6,650 km',
   outflow: 'Mediterranean'
 },
 amazon: {
   continent: 'South America',
   length: '6,575 km',
   outflow: 'Atlantic Ocean'
 },
 yangtze: {
   continent: 'Asia',
   length: '6,300 km',
   outflow: 'East China Sea'
 },
 mississippi: {
   continent: 'North America',
   length: '6,275 km',
   outflow: 'Gulf of Mexico'
 }
}


export function getRiverInformation(name) {
  return new Promise((resolve) => {
```

```
    setTimeout(() => {
      resolve(
        rivers[name]
      )
    }, 1500)
  })
}
```

Save and close the file. Next, open `App.js` so you can add more options:

```
nano src/components/App/App.js
```

Inside `App.js`, create a stateful variable and function to hold the selected river with the `useState` Hook. Then add a button for each river with an `onClick` handler to update the selected river. Pass the `river` to `RiverInformation` using a prop called `name`:

async-tutorial/src/components/App/App.js

```
import React, { useState } from 'react';
import './App.css';
import RiverInformation from '../RiverInformation/RiverInformat

function App() {
  const [river, setRiver] = useState('nile');
  return (
    <div className="wrapper">
      <h1>World's Longest Rivers</h1>
      <button onClick={() => setRiver('nile')}>Nile</button>
      <button onClick={() => setRiver('amazon')}>Amazon</button
      <button onClick={() => setRiver('yangtze')}>Yangtze</butt
      <button onClick={() => setRiver('mississippi')}>Mississip
      <RiverInformation name={river} />
    </div>
  );
}

export default App;
```

Save and close the file. Next, open `RiverInformation.js`:

```
nano src/components/RiverInformation/RiverInformation.js
```

Pull in the `name` as a prop and pass it to the `getRiverInformation` function. Be sure to add `name` to the array for `useEffect`, otherwise it will not rerun:

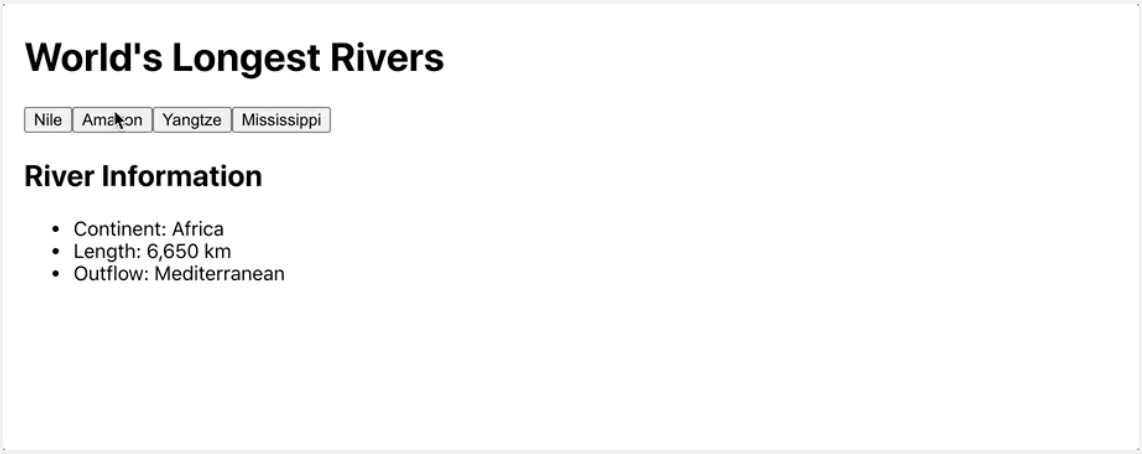
async-tutorial/src/components/RiverInformation/RiverInformation.js

```
import React, { useEffect, useState } from 'react';
import PropTypes from 'prop-types';
import { getRiverInformation } from '../../services/rivers';

export default function RiverInformation({ name }) {
  const [riverInformation, setRiverInformation] = useState({});

  useEffect(() => {
    getRiverInformation(name)
    .then(data =>
      setRiverInformation(data)
    );
  }, [name])


  return(
    <div>
      <h2>River Information</h2>
      <ul>
        <li>Continent: {riverInformation.continent}</li>
        <li>Length: {riverInformation.length}</li>
        <li>Outflow: {riverInformation.outflow}</li>
      </ul>
    </div>
```

```
  )
}

RiverInformation.propTypes = {
 name: PropTypes.string.isRequired
}
```

In this code, you also added a weak typing system with `PropTypes`, which will make sure that the prop is a string.

Save the file. When you do, the browser will refresh and you can select different rivers. Notice the delay between when you click and when the data renders:



Update river information, 3

If you had left out the `name` prop from the `useEffect` array, you would receive a build error in the browser console. It would be something like

this:

```
Error
Compiled with warnings.


./src/components/RiverInformation/RiverInformation.js
  Line 13:6: React Hook useEffect has a missing dependency: 'n
ame'. Either include it or remove the dependency array react-h
ooks/exhaustive-deps

Search for the keywords to learn more about each warning.
To ignore, add // eslint-disable-next-line to the line before.
```

This error tells you that the function in your effect has dependencies that you are not explicitly setting. In this situation, it's clear that the effect wouldn't work, but there are times when you may be comparing prop data to stateful data inside the component, which makes it possible to lose track of items in the array.

The last thing to do is to add some defensive programming to your component. This is a design principle that emphasizes high availability for your application. You want to ensure that your component will render even if the data is not in the correct shape or if you do not get any data at all from an API request.

As your app is now, the effect will update the `riverInformation` with any type of data it receives. This will usually be an object, but in cases where

it’s not, you can use optional chaining to ensure that you will not throw an error.

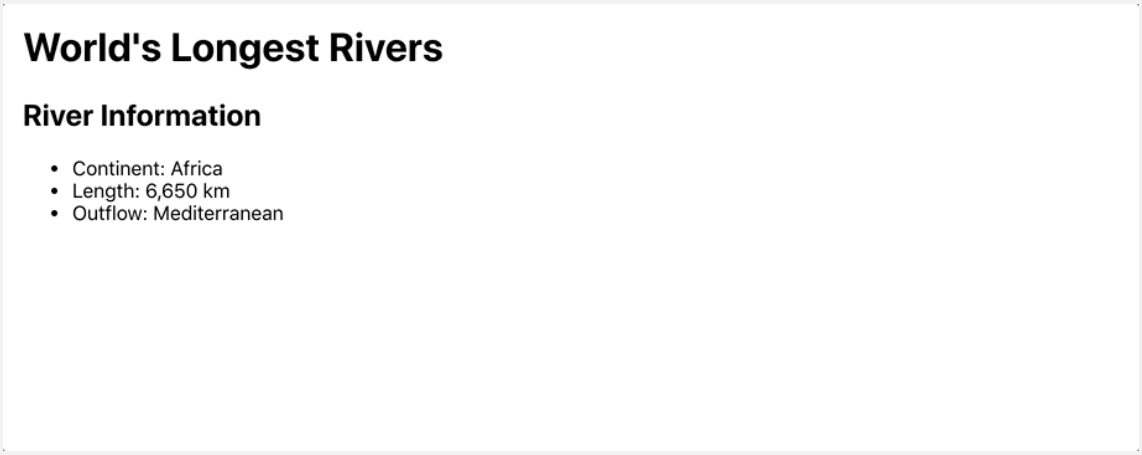
Inside `RiverInformation.js`, replace the instance of an object dot chaining with optional chaining. To test if it works, remove the default object `{}` from the `useState` function:

async-tutorial/src/components/RiverInformation/RiverInformation.js

```
import React, { useEffect, useState } from 'react';
import PropTypes from 'prop-types';
import { getRiverInformation } from '../../services/rivers';

export default function RiverInformation({ name }) {
  const [riverInformation, setRiverInformation] = useState();

  useEffect(() => {
    getRiverInformation(name)
    .then(data =>
      setRiverInformation(data)
    );
  }, [name])

  return(
    <div>
      <h2>River Information</h2>
      <ul>
        <li>Continent: {riverInformation?.continent}</li>
        <li>Length: {riverInformation?.length}</li>
        <li>Outflow: {riverInformation?.outflow}</li>
      </ul>
    </div>
  )
```

```
}


RiverInformation.propTypes = {
  name: PropTypes.string.isRequired
}
```

Save and close the file. When you do, the file will still load even though the code is referencing properties on `undefined` instead of an object:



River Information Updating After Load, 4

Defensive programming is usually considered a best practice, but it’s especially important on asynchronous functions such as API calls when you can’t guarantee a response.

In this step, you called asynchronous functions in React. You used the `useEffect` Hook to fetch information without triggering re-renders and triggered a new update by adding conditions to the `useEffect` array.

In the next step, you'll make some changes to your app so that it updates components only when they are mounted. This will help your app avoid memory leaks.

Step 2 — Preventing Errors on Unmounted Components

In this step, you'll prevent data updates on unmounted components. Since you can never be sure when data will resolve with asynchronous programming, there's always a risk that the data will resolve after the component has been removed. Updating data on an unmounted component is inefficient and can introduce memory leaks in which your app is using more memory than it needs to.

By the end of this step, you'll know how to prevent memory leaks by adding guards in your `useEffect` Hook to update data only when the component is mounted.

The current component will always be mounted, so there's no chance that the code will try and update the component after it is removed from the DOM, but most components aren't so reliable. They will be added and removed from the page as the user interacts with the application. If a component is removed from a page before the asynchronous function resolves, you can have a memory leak.

To test out the problem, update `App.js` to be able to add and remove the river details.

Open `App.js`:

```
nano src/components/App/App.js
```

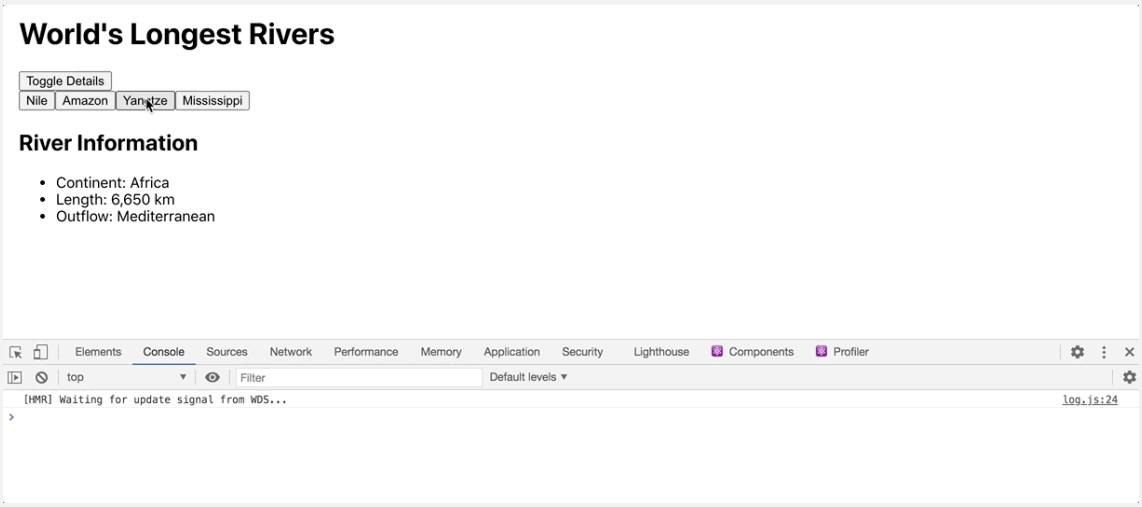
Add a button to toggle the river details. Use the `useReducer` Hook to create a function to toggle the details and a variable to store the toggled state:

async-tutorial/src/components/App/App.js

```
import React, { useReducer, useState } from 'react';
import './App.css';
import RiverInformation from '../RiverInformation/RiverInformat

function App() {
  const [river, setRiver] = useState('nile');
  const [show, toggle] = useReducer(state => !state, true);
  return (
    <div className="wrapper">
      <h1>World's Longest Rivers</h1>
      <div><button onClick={toggle}>Toggle Details</button></di
      <button onClick={() => setRiver('nile')}>Nile</button>
      <button onClick={() => setRiver('amazon')}>Amazon</button
      <button onClick={() => setRiver('yangtze')}>Yangtze</butt
      <button onClick={() => setRiver('mississippi')}>Mississip
      {show && <RiverInformation name={river} />}
    </div>
  );
}

export default App;
```

Save the file. When you do the browse will reload and you'll be able to toggle the details.

Click on a river, then immediately click on the **Toggle Details** button to hide details. React will generate an error warning that there is a potential memory leak.



To fix the problem you need to either cancel or ignore the asynchronous function inside `useEffect`. If you are using a library such as RxJS, you can cancel an asynchronous action when the component unmounts by returning a function in your `useEffect` Hook. In other cases, you'll need a variable to store the mounted state.

Open `RiverInformation.js`:

```
nano src/components/RiverInformation/RiverInformation.js
```

Inside the `useEffect` function, create a variable called `mounted` and set it to `true`. Inside the `.then` callback, use a conditional to set the data if `mounted` is true:

async-tutorial/src/components/RiverInformation/RiverInformation.js

```
import React, { useEffect, useState } from 'react';
import PropTypes from 'prop-types';
import { getRiverInformation } from '../../services/rivers';

export default function RiverInformation({ name }) {
  const [riverInformation, setRiverInformation] = useState();

  useEffect(() => {
    let mounted = true;
    getRiverInformation(name)
    .then(data => {
      if(mounted) {
        setRiverInformation(data)
      }
    });
  }, [name])


  return(
    <div>
      <h2>River Information</h2>
      <ul>
        <li>Continent: {riverInformation?.continent}</li>
```

```
        <li>Length: {riverInformation?.length}</li>
        <li>Outflow: {riverInformation?.outflow}</li>
      </ul>
    </div>
  )
}

RiverInformation.propTypes = {
  name: PropTypes.string.isRequired
}
```

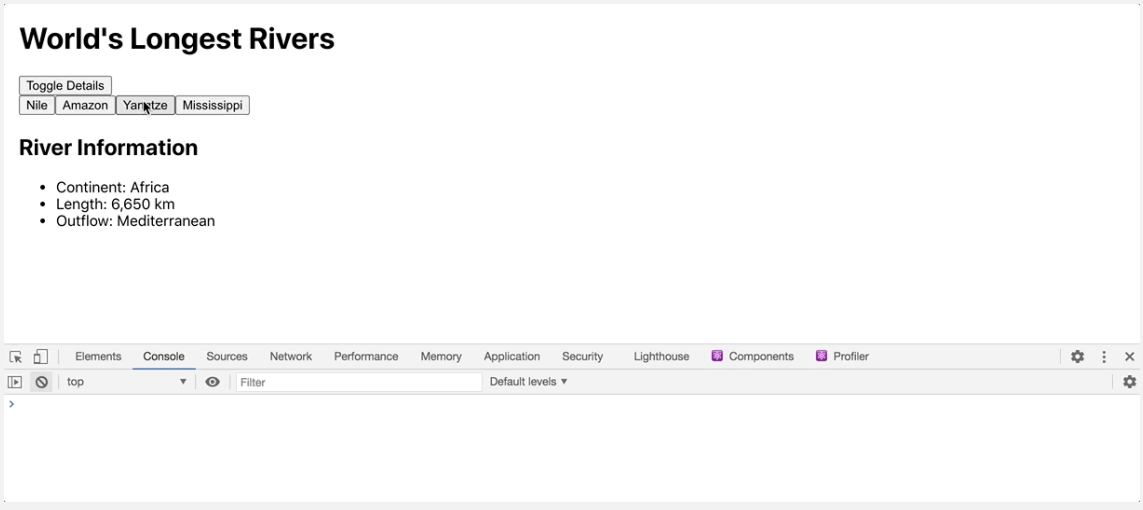
Now that you have the variable, you need to be able to flip it when the component unmounts. With the `useEffect` Hook, you can return a function that will run when the component unmounts. Return a function that sets `mounted` to `false`:

async-tutorial/src/components/RiverInformation/RiverInformation.js

```
import React, { useEffect, useState } from 'react';
import PropTypes from 'prop-types';
import { getRiverInformation } from '../../services/rivers';

export default function RiverInformation({ name }) {
  const [riverInformation, setRiverInformation] = useState();

  useEffect(() => {
    let mounted = true;
    getRiverInformation(name)
    .then(data => {
      if(mounted) {
        setRiverInformation(data)
      }
    });
    return () => {
     mounted = false;
   }
  }, [name])

  return(
    <div>
      <h2>River Information</h2>
```

```
      <ul>
        <li>Continent: {riverInformation?.continent}</li>
        <li>Length: {riverInformation?.length}</li>
        <li>Outflow: {riverInformation?.outflow}</li>
      </ul>
    </div>
  )
}

RiverInformation.propTypes = {
  name: PropTypes.string.isRequired
}
```

Save the file. When you do, you'll be able to toggle the details without an error.



No warning when toggling, 6

When you unmount, the component `useEffect` updates the variable. The asynchronous function will still resolve, but it won't make any changes to unmounted components. This will prevent memory leaks.

In this step, you made your app update state only when a component is mounted. You updated the `useEffect` Hook to track if the component is mounted and returned a function to update the value when the component unmounts.

In the next step, you'll asynchronously load components to split code into smaller bundles that a user will load as needed.

Step 3 — Lazy Loading a Component with `Suspense` and `lazy`

In this step, you'll split your code with React `Suspense` and `lazy`. As applications grow, the size of the final build grows with it. Rather than forcing users to download the whole application, you can split the code into smaller chunks. React `Suspense` and `lazy` work with webpack and other build systems to split your code into smaller pieces that a user will be able to load on demand. In the future, you will be able to use `Suspense` to load a variety of data, including API requests.

By the end of this step, you'll be able to load components asynchronously, breaking large applications into smaller, more focused chunks.

So far you've only worked with asynchronously loading data, but you can also asynchronously load components. This process, often called code

splitting, helps reduce the size of your code bundles so your users don't have to download the full application if they are only using a portion of it.

Most of the time, you import code statically, but you can import code dynamically by calling `import` as a function instead of a statement. The code would be something like this:

```
import('my-library')
.then(library => library.action())
```

React gives you an additional set of tools called `lazy` and `Suspense`. React `Suspense` will eventually expand to handle data loading, but for now you can use it to load components.

Open `App.js`:

```
nano src/components/App/App.js
```

Then import `lazy` and `Suspense` from `react`:

async-tutorial/src/components/App/App.js

```
import React, { lazy, Suspense, useReducer, useState } from 're
import './App.css';
import RiverInformation from '../RiverInformation/RiverInformat

function App() {
  const [river, setRiver] = useState('nile');
  const [show, toggle] = useReducer(state => !state, true);
  return (
    <div className="wrapper">
      <h1>World's Longest Rivers</h1>
      <div><button onClick={toggle}>Toggle Details</button></di
      <button onClick={() => setRiver('nile')}>Nile</button>
      <button onClick={() => setRiver('amazon')}>Amazon</button
      <button onClick={() => setRiver('yangtze')}>Yangtze</butt
      <button onClick={() => setRiver('mississippi')}>Mississip
      {show && <RiverInformation name={river} />}
    </div>
  );
}

export default App;
```

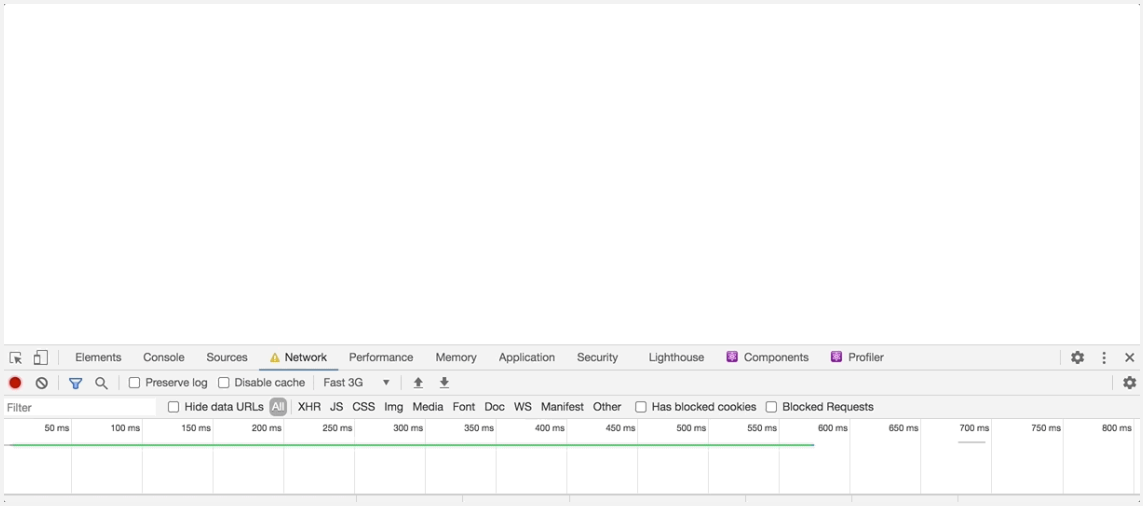
`lazy` and `Suspsense` have two distinct jobs. You use the `lazy` function to dynamically import the component and set it to a variable. `Suspense` is a built-in component you use to display a fallback message while the code is loading.

Replace `import RiverInformation from '../RiverInformation/RiverInformation';` with a call to `lazy`. Assign the result to a variable called `RiverInformation`. Then wrap `{show && <RiverInformation name={river} />}` with the `Suspense` component and a `<div>` with a message of `Loading Component` to the `fallback` prop:

async-tutorial/src/components/App/App.js

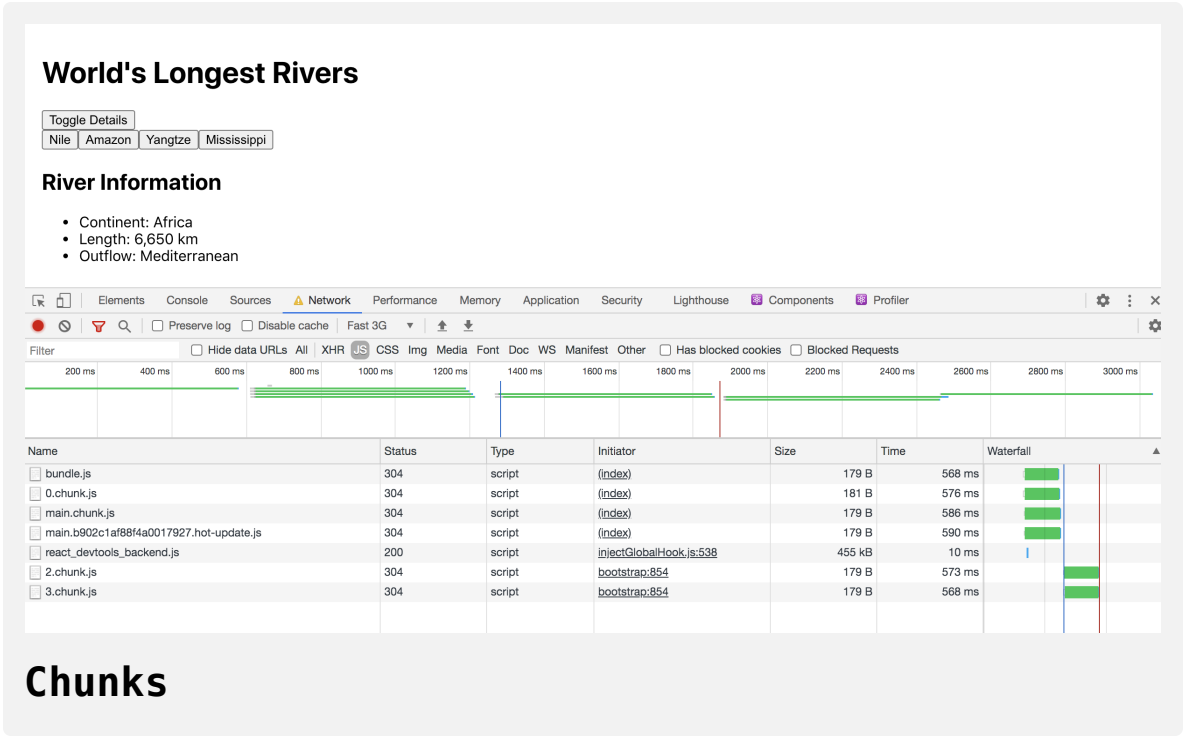
```
import React, { lazy, Suspense, useReducer, useState } from 're
import './App.css';
const RiverInformation = lazy(() => import('../RiverInformation

function App() {
  const [river, setRiver] = useState('nile');
  const [show, toggle] = useReducer(state => !state, true);
  return (
    <div className="wrapper">
      <h1>World's Longest Rivers</h1>
      <div><button onClick={toggle}>Toggle Details</button></di
      <button onClick={() => setRiver('nile')}>Nile</button>
      <button onClick={() => setRiver('amazon')}>Amazon</button
      <button onClick={() => setRiver('yangtze')}>Yangtze</butt
      <button onClick={() => setRiver('mississippi')}>Mississip
      <Suspense fallback={<div>Loading Component</div>}>
        {show && <RiverInformation name={river} />}
      </Suspense>
    </div>
  );
}

export default App;
```

Save the file. When you do, reload the page and you'll find that the component is dynamically loaded. If you want to see the loading message, you can throttle the response in the Chrome web browser.



Component Loading

If you navigate to the **Network** tab in Chrome or Firefox, you'll find that the code is broken into different chunks.



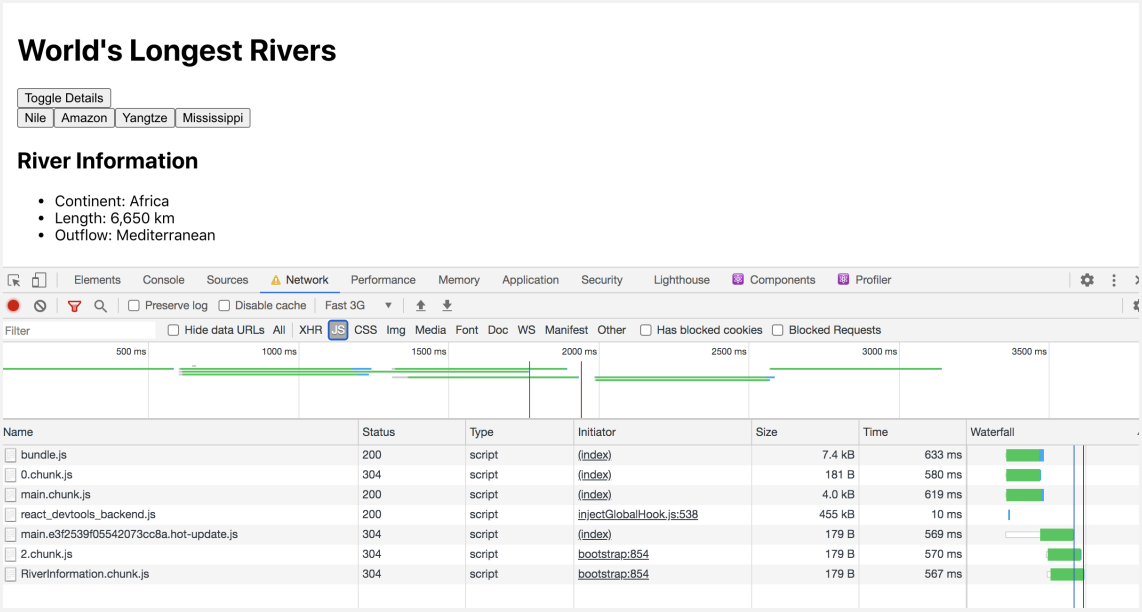
Each chunk gets a number by default, but with Create React App combined with webpack, you can set the chunk name by adding a comment by the dynamic import.

In `App.js`, add a comment of `/* webpackChunkName: "RiverInformation" */` inside the `import` function:

async-tutorial/src/components/App/App.js

```
import React, { lazy, Suspense, useReducer, useState } from 're
import './App.css';
const RiverInformation = lazy(() => import(/* webpackChunkName:

function App() {
  const [river, setRiver] = useState('nile');
  const [show, toggle] = useReducer(state => !state, true);
  return (
    <div className="wrapper">
      <h1>World's Longest Rivers</h1>
      <div><button onClick={toggle}>Toggle Details</button></di
      <button onClick={() => setRiver('nile')}>Nile</button>
      <button onClick={() => setRiver('amazon')}>Amazon</button
      <button onClick={() => setRiver('yangtze')}>Yangtze</butt
      <button onClick={() => setRiver('mississippi')}>Mississip
      <Suspense fallback={<div>Loading Component</div>}>
        {show && <RiverInformation name={river} />}
      </Suspense>
    </div>
  );
}

export default App;
```

Save and close the file. When you do, the browser will refresh and the `RiverInformation` chunk will have a unique name.



`River Information Chunk`

In this step, you asynchronously loaded components. You used `lazy` and `Suspense` to dynamically import components and to show a loading message while the component loads. You also gave custom names to webpack chunks to improve readability and debugging.

Conclusion

Asynchronous functions create efficient user-friendly applications. However, their advantages come with some subtle costs that can evolve into bugs in your program. You now have tools that will let you split large applications into smaller pieces and load asynchronous data while still giving the user a visible application. You can use the knowledge to

incorporate API requests and asynchronous data manipulations into your applications creating fast and reliable user experiences.

If you would like to read more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Call Web APIs with the useEffect Hook in React

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

In React development, web application programming interfaces (APIs) are an integral part of single-page application (SPA) designs. APIs are the primary way for applications to programmatically communicate with servers to provide users with real-time data and to save user changes. In React applications, you will use APIs to load user preferences, display user information, fetch configuration or security information, and save application state changes.

In this tutorial, you'll use the `useEffect` and `useState` Hooks to fetch and display information in a sample application, using JSON server as a local API for testing purposes. You'll load information when a component first mounts and save customer inputs with an API. You'll also refresh data when a user makes a change and learn how to ignore API requests when a component unmounts. By the end of this tutorial, you'll be able to connect your React applications to a variety of APIs and you'll be able to send and receive real-time data.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.22.0 and npm version 6.14.6. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- A React development environment set up with Create React App, with the non-essential boilerplate removed. To set this up, follow **Step 1 — Creating an Empty Project** of the How To Manage State on React Class Components tutorial. This tutorial will use `api-tutorial` as the project name.

- You will be using React components and Hooks in this tutorial, including the `useState` and `useEffect` Hooks. You can learn about components and Hooks in our tutorials How To Manage State with Hooks on React Components and How To Handle Async Data Loading, Lazy Loading, and Code Splitting with React.

- You will also need a basic knowledge of JavaScript and HTML, which you can find in our How To Build a Website with HTML series and in How To Code in JavaScript. Basic knowledge of CSS would also be useful, which you can find at the Mozilla Developer Network.

Step 1 — Creating a Project and a Local API

In this step, you'll create a local REST API using JSON server, which you will use as a test data source. Later, you'll build an application to display a

grocery list and to add items to the list. JSON server will be your local API and will give you a live URL to make `GET` and `POST` requests. With a local API, you have the opportunity to prototype and test components while you or another team develops live APIs.

By the end of this step, you'll be able to create local mock APIs that you can connect to with your React applications.

On many agile teams, front-end and API teams work on a problem in parallel. In order to develop a front-end application while a remote API is still in development, you can create a local version that you can use while waiting for a complete remote API.

There are many ways to make a mock local API. You can create a simple server using Node or another language, but the quickest way is to use the JSON server Node package. This project creates a local REST API from a JSON file.

To begin, install `json-server`:

```
npm install --save-dev json-server
```

When the installation is finished, you'll receive a success message:

Output

```
+ json-server@0.16.1
added 108 packages from 40 contributors and audited 1723 packa
ges in 14.505s

73 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

`json-server` creates an API based on a JavaScript object. The keys are the URL paths and the values are returned as a response. You store the JavaScript object locally and commit it to your source control.

Open a file called `db.json` in the root of your application. This will be the JSON that stores the information you request from the API:

```
nano db.json
```

Add an object with the key of `list` and an array of values with an `id` and a key of `item`. This will list the item for the grocery list. The key `list` will eventually give you a URL with an endpoint of `/list`:

api-tutorial/db.json

```
{
  "list": [
    { "id": 1, "item": "bread" },
    { "id": 2, "item": "grapes" }
  ]
}
```

In this snippet, you have hard-coded `bread` and `grapes` as a starting point for your grocery list.

Save and close the file. To run the API server, you will use `json-server` from the command line with an argument point to the API configuration file. Add it as a script in your `package.json`.

Open `package.json`:

```
nano package.json
```

Then add a script to run the API. In addition, add a `delay` property. This will throttle the response, creating a delay between your API request and the API response. This will give you some insights into how the application will behave when waiting for a server response. Add a `delay` of `1500` milliseconds. Finally, run the API on port `3333` using the `-p` option so it won't conflict with the `create-react-app` run script:

api-tutorial/package.json

```
{
  "name": "do-14-api",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^4.2.4",
    "@testing-library/react": "^9.3.2",
    "@testing-library/user-event": "^7.1.2",
    "react": "^16.13.1",
    "react-dom": "^16.13.1",
    "react-scripts": "3.4.3"
  },
  "scripts": {
    "api": "json-server db.json -p 3333 --delay 1500",
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": {
    "production": [
      ">0.2%",
```

```
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  },
  "devDependencies": {
    "json-server": "^0.16.1"
  }
}
```

Save and close the file. In a new terminal or tab, start the API server with the following command:

```
npm run api
```

Keep this running during the rest of the tutorial.

When you run the command, you will receive an output that lists the API resources:

Output

```
> json-server db.json -p 3333



  \{^_^}/ hi!

  Loading db.json
  Done

  Resources
  http://localhost:3333/list

  Home
  http://localhost:3333

  Type s + enter at any time to create a snapshot of the datab
ase
```

Open `http://localhost:3333/list` and you'll find the live API:

```
[
  {
    "id": 1,
    "item": "bread"
  },
  {
    "id": 2,
    "item": "grapes"
  }
]
```


API results, 1

When you open an endpoint in your browser, you are using the `GET` method. But `json-server` is not limited to the `GET` method. You can perform many other REST methods as well. For example, you can `POST` new items. In a new terminal window or tab, use `curl` to `POST` a new item with a type of `application/json`:

```
curl -d '{"item":"rice"}' -H 'Content-Type: application/json'
 -X POST http://localhost:3333/list
```

Note that you must stringify the content before you send it. After running the `curl` command, you'll receive a success message:

Output

```
{
  "item": "rice",
  "id": 3
}
```

If you refresh the browser, the new item will appear:

```
[
  {
    "id": 1,
    "item": "bread"
  },
  {
    "id": 2,
    "item": "grapes"
  },
  {
    "item": "rice",
    "id": 3
  }
]
```

Updated content, 2

The `POST` request will also update the `db.json` file. Be mindful of the changes, since there are no barriers to accidentally saving unstructured or unhelpful content as you work on your application. Be sure to check any changes before committing into version control.

In this step, you created a local API. You learned how to create a static file with default values and how to fetch or update those values using RESTful actions such as `GET` and `POST`. In the next step, you'll create services to fetch data from the API and to display in your application.

Step 2 — Fetching Data from an API with `useEffect`

In this step, you'll fetch a list of groceries using the `useEffect` Hook. You'll create a service to consume APIs in separate directories and call that service in your React components. After you call the service, you'll save the data with the `useState` Hook and display the results in your component.

By the end of this step, you'll be able to call web APIs using the Fetch method and the `useEffect` Hook. You'll also be able to save and display the results.

Now that you have a working API, you need a service to fetch the data and components to display the information. Start by creating a service. You can fetch data directly inside any React component, but your projects will be easier to browse and update if you keep your data retrieval functions separate from your display components. This will allow you to reuse methods across components, mock in tests, and update URLs when endpoints change.

Create a directory called `services` inside the `src` directory:

```
mkdir src/services
```

Then open a file called `list.js` in your text editor:

```
nano src/services/list.js
```

You'll use this file for any actions on the `/list` endpoint. Add a function to retrieve the data using the `fetch` function:

api-tutorial/src/services/list

```
export function getList() {
  return fetch('http://localhost:3333/list')
    .then(data => data.json())
}
```

The only goal of this function is to access the data and to convert the response into JSON using the `data.json()` method. `GET` is the default action, so you don't need any other parameters.

In addition to `fetch`, there are other popular libraries such as Axios that can give you an intuitive interface and will allow you to add default headers or perform other actions on the service. But `fetch` will work for most requests.

Save and close the file. Next, open `App.css` and add some minimal styling:

```
nano src/components/App/App.css
```

Add a class of `wrapper` with a small amount of padding by replacing the CSS with the following:

api-tutorial/src/components/App/App.css

```
.wrapper {
    padding: 15px;
}
```

Save and close the file. Now you need to add in code to retrieve the data and display it in your application.

Open `App.js`:

```
nano src/components/App/App.js
```

In functional components, you use the `useEffect` Hook to fetch data when the component loads or some information changes. For more information on the `useEffect` Hook, check out How To Handle Async Data Loading, Lazy Loading, and Code Splitting with React. You'll also need to save the results with the `useState` Hook.

Import `useEffect` and `useState`, then create a variable called `list` and a setter called `setList` to hold the data you fetch from the service using the `useState` Hook:

api-tutorial/src/components/App/App.js

```
import React, { useEffect, useState } from 'react';
import './App.css';

function App() {
  const [list, setList] = useState([]);
  return(
    <>
    </>
  )
}

export default App;
```

Next, import the service, then call the service inside your `useEffect` Hook. Update the `list` with `setList` if the component is mounted. To understand why you should check if the component is mounted before setting the data, see **Step 2 — Preventing Errors on Unmounted Components** in How To Handle Async Data Loading, Lazy Loading, and Code Splitting with React.

Currently you are only running the effect once when the page loads, so the dependency array will be empty. In the next step, you'll trigger the effect based on different page actions to ensure that you always have the most up-to-date information.

Add the following highlighted code:

api-tutorial/src/components/App/App.js

```
import React, { useEffect, useState } from 'react';
import './App.css';
import { getList } from '../../services/list';

function App() {
  const [list, setList] = useState([]);

  useEffect(() => {
   let mounted = true;
   getList()
     .then(items => {
       if(mounted) {
         setList(items)
       }
     })
   return () => mounted = false;
 }, [])

  return(
    <>
    </>
  )
}
```

```
export default App;
```

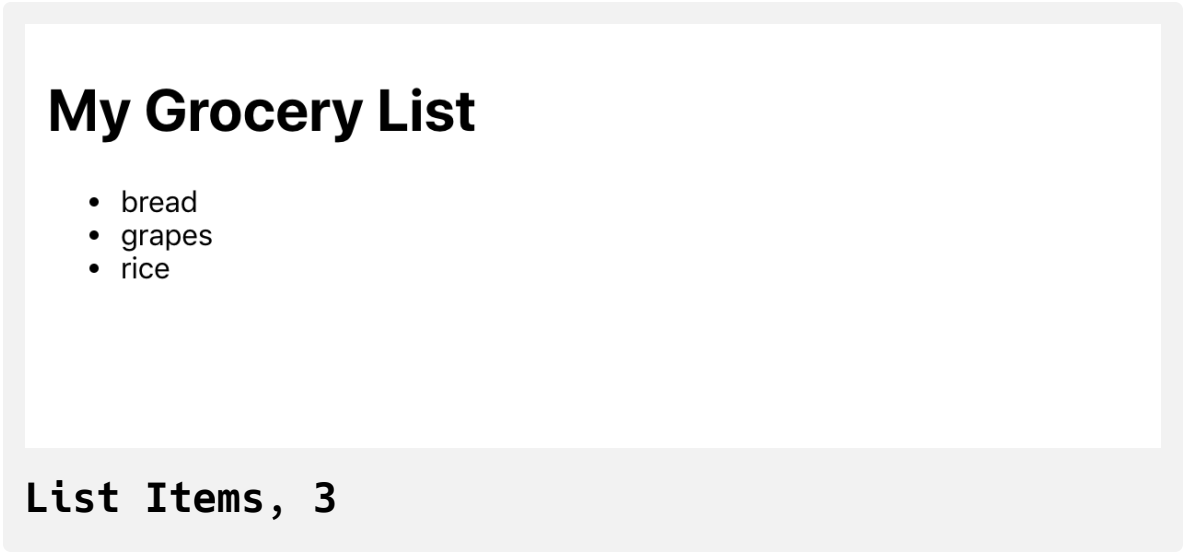
Finally, loop over the items with `.map` and display them in a list:

api-tutorial/src/components/App/App

```
import React, { useEffect, useState } from 'react';
import './App.css';
import { getList } from '../../services/list';

function App() {
  const [list, setList] = useState([]);

  useEffect(() => {
    let mounted = true;
    getList()
      .then(items => {
        if(mounted) {
          setList(items)
        }
      })
    return () => mounted = false;
  }, [])

  return(
    <div className="wrapper">
     <h1>My Grocery List</h1>
     <ul>
       {list.map(item => <li key={item.item}>{item.item}</li>)}
     </ul>
   </div>
```

```
  )
}


export default App;
```

Save and close the file. When you do, the browser will refresh and you'll find a list of items:



In this step, you set up a service to retrieve data from an API. You learned how to call the service using the `useEffect` Hook and how to set the data on the page. You also displayed the data inside your JSX.

In the next step, you'll submit data to the API using `POST` and use the response to alert your users that an actions was successful.

Step 3 — Sending Data to an API

In this step, you'll send data back to an API using the Fetch API and the `POST` method. You'll create a component that will use a web form to send the data with the `onSubmit` event handler and will display a success message when the action is complete.

By the end of this step, you'll be able to send information to an API and you'll be able to alert the user when the request resolves.

Sending Data to a Service

You have an application that will display a list of grocery items, but it's not a very useful grocery app unless you can save content as well. You need to create a service that will `POST` a new item to the API.

Open up `src/services/list.js`:

```
nano src/services/list.js
```

Inside the file, add a function that will take an `item` as an argument and will send the data using the `POST` method to the API. As before, you can use the Fetch API. This time, you'll need more information. Add an object of options as the second argument. Include the method—`POST`—along with headers to set the `Content-Type` to `application/json`. Finally, send the new object in the `body`. Be sure to convert the object to a string using `JSON.stringify`.

When you receive a response, convert the value to JSON:

tutorial/src/services/list.js

```
export function getList() {
  return fetch('http://localhost:3333/list')
    .then(data => data.json())
}


export function setItem(item) {
 return fetch('http://localhost:3333/list', {
   method: 'POST',
   headers: {
     'Content-Type': 'application/json'
   },
   body: JSON.stringify({ item })
 })
   .then(data => data.json())
}
```

Save and close the file.

Note: In production applications, you'll need to add error handling and checking. For example, if you misspelled the endpoint, you'd still receive a `404` response and the `data.json()` method would return an empty object. To solve the issue, instead of converting the response to JSON, you could check the `data.ok` property. If it is falsy, you could throw an error and then use the `.catch` method in your component to display a failure message to the users.

Now that you have created a service, you need to consume the service inside your component.

Open `App.js`:

```
nano src/components/App/App.js
```

Add a `form` element surrounding an `input` and a submit `button`:

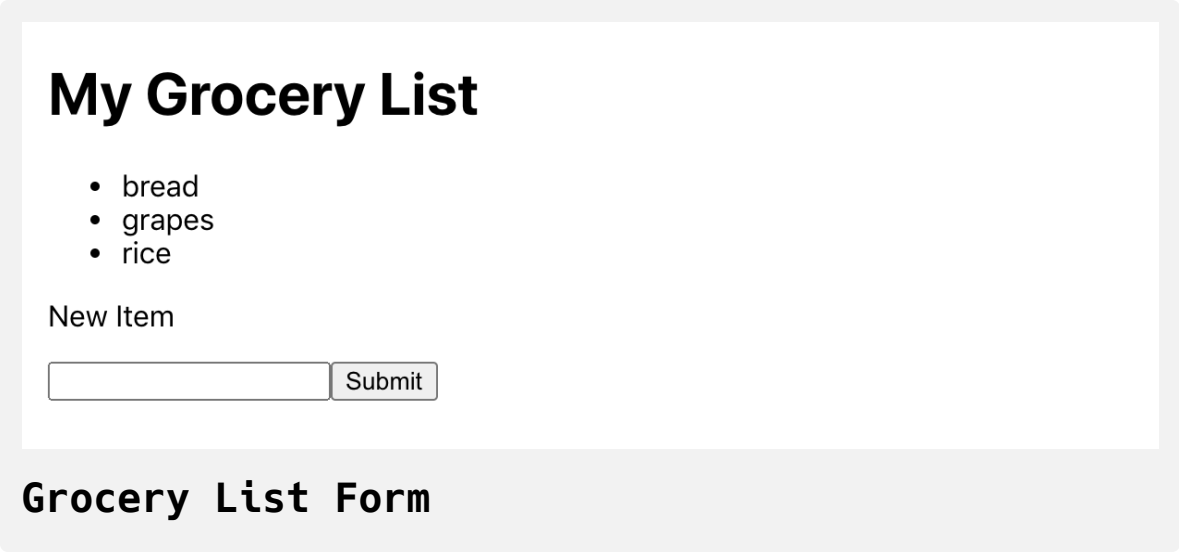
api-tutorial/src/components/App/App.js

```
import React, { useEffect, useState } from 'react';
import './App.css';
import { getList } from '../../services/list';

function App() {
  const [list, setList] = useState([]);

  useEffect(() => {
    let mounted = true;
    getList()
      .then(items => {
        if(mounted) {
          setList(items)
        }
      })
    return () => mounted = false;
  }, [])

  return(
    <div className="wrapper">
      <h1>My Grocery List</h1>
      <ul>
        {list.map(item => <li key={item.item}>{item.item}</li>)
      </ul>
```

```
      <form>
        <label>
          <p>New Item</p>
          <input type="text" />
        </label>
        <button type="submit">Submit</button>
      </form>
    </div>
  )
}

export default App;
```

Be sure to surround the `input` with a `label` so that the form is accessible by a screen reader. It's also a good practice to add a `type="submit"` to the `button` so that it's clear the role is to submit the form.

Save the file. When you do, the browser will refresh and you'll find your form.



`Grocery List Form`

Next, convert the `input` to a controlled component. You'll need a controlled component so that you can clear the field after the user successfully submits a new list item.

First, create a new state handler to hold and set the input information using the `useState` Hook:

api-tutorial/src/components/App/App.js

```
import React, { useEffect, useState } from 'react';
import './App.css';
import { getList } from '../../services/list';

function App() {
  const [itemInput, setItemInput] = useState('');
  const [list, setList] = useState([]);

  useEffect(() => {
    let mounted = true;
    getList()
      .then(items => {
        if(mounted) {
          setList(items)
        }
      })
    return () => mounted = false;
  }, [])

  return(
    <div className="wrapper">
      <h1>My Grocery List</h1>
      <ul>
        {list.map(item => <li key={item.item}>{item.item}</li>)
      </ul>
```

```
      <form>
        <label>
          <p>New Item</p>
          <input type="text" onChange={event => setItemInput(ev
            value={itemInput} />
        </label>
        <button type="submit">Submit</button>
      </form>
    </div>
  )
}


export default App;
```

After creating the state handlers, set the value of the `input` to `itemInput` and update the value by passing the `event.target.value` to the `setItemInput` function using the `onChange` event handler.

Now your users can fill out a form with new list items. Next you will connect the form to your service.

Create a function called `handleSubmit`. `handleSubmit` will take an event as an argument and will call `event.preventDefault()` to stop the form from refreshing the browser.

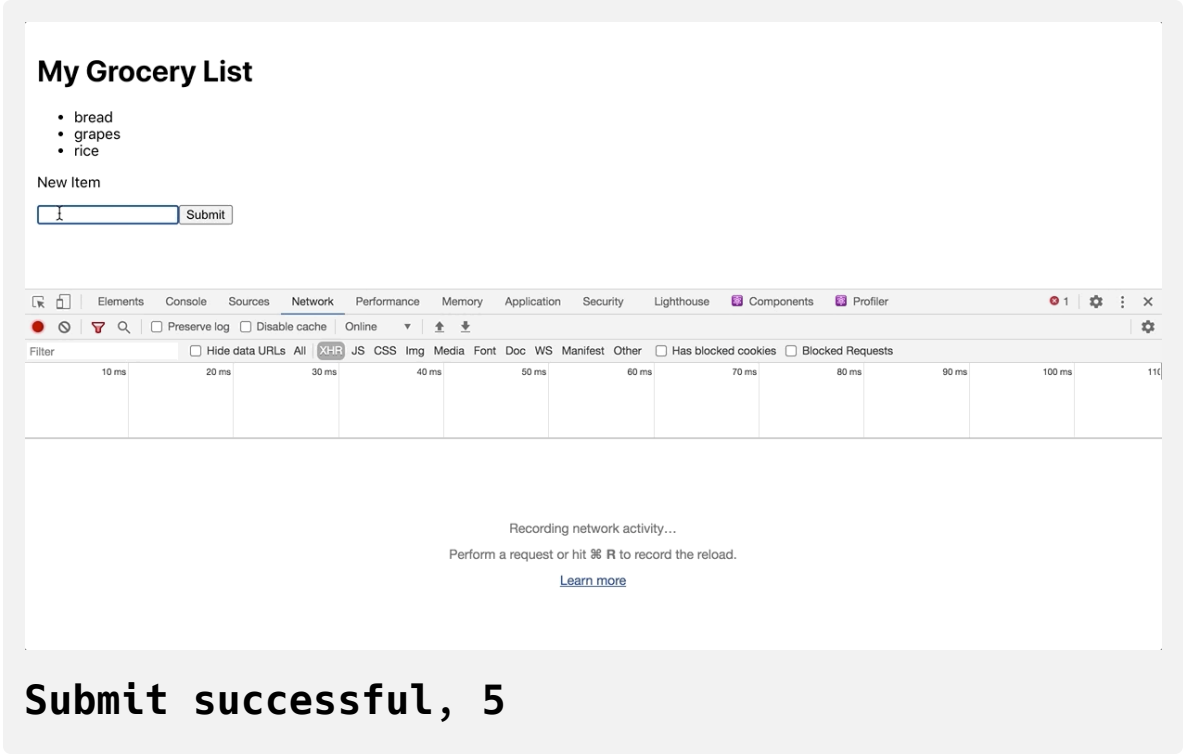
Import `setItem` from the service, then call `setItem` with the `itemInput` value inside the `handleSubmit` function. Connect `handleSubmit` to the form by passing it to the `onSubmit` event handler:

api-tutorial/src/components/App/App.js

```
import React, { useEffect, useState } from 'react';
import './App.css';
import { getList, setItem } from '../../services/list';

function App() {
  const [itemInput, setItemInput] = useState('');
  const [list, setList] = useState([]);

  useEffect(() => {
    let mounted = true;
    getList()
      .then(items => {
        if(mounted) {
          setList(items)
        }
      })
    return () => mounted = false;
  }, [])

  const handleSubmit = (e) => {
    e.preventDefault();
    setItem(itemInput)
  };

  return(
```

```
    <div className="wrapper">
      <h1>My Grocery List</h1>
      <ul>
        {list.map(item => <li key={item.item}>{item.item}</li>)
      </ul>
      <form onSubmit={handleSubmit}>
        <label>
          <p>New Item</p>
          <input type="text" onChange={event => setItemInput(ev
            value={itemInput} />
        </label>
        <button type="submit">Submit</button>
      </form>
    </div>
  )
}


export default App;
```

Save the file. When you do, you'll be able to submit values. Notice that you'll receive a successful response in the network tab. But the list doesn't update and the input doesn't clear.




```
Submit successful, 5
```

Showing a Success Message

It’s always a good practice to give the user some indication that their action was successful. Otherwise a user may try and resubmit a value multiple times or may think their action failed and will leave the application.

To do this, create a stateful variable and setter function with `useState` to indicate whether to show a user an alert message. If `alert` is true, display an `<h2>` tag with the message **Submit Successful**.

When the `setItem` promise resolves, clear the input and set the alert message:

api-tutorial/src/components/App/App.js

```
import React, { useEffect, useState } from 'react';
import './App.css';
import { getList, setItem } from '../../services/list';

function App() {
  const [alert, setAlert] = useState(false);
  const [itemInput, setItemInput] = useState('');
  const [list, setList] = useState([]);

  useEffect(() => {
    let mounted = true;
    getList()
      .then(items => {
        if(mounted) {
          setList(items)
        }
      })
    return () => mounted = false;
  }, [])

  const handleSubmit = (e) => {
    e.preventDefault();
    setItem(itemInput)
      .then(() => {
        setItemInput('');
```

```
        setAlert(true);
      })
  };


  return(
    <div className="wrapper">
      <h1>My Grocery List</h1>
      <ul>
        {list.map(item => <li key={item.item}>{item.item}</li>)
      </ul>
      {alert && <h2> Submit Successful</h2>}
      <form onSubmit={handleSubmit}>
        <label>
          <p>New Item</p>
          <input type="text" onChange={event => setItemInput(ev
            value={itemInput} />
        </label>
        <button type="submit">Submit</button>
      </form>
    </div>
  )
}


export default App;
```

Save the file. When you do, the page will refresh and you'll see a success message after the API request resolves.

There are many other optimizations you can add. For example, you may want to disable form inputs while there is an active request. You can learn more about disabling form elements in How To Build Forms in React.

Now you have alerted a user that the result was successful, but the alert message doesn't go away and the list doesn't update. To fix this, start by hiding the alert. In this case, you'd want to hide the information after a brief period, such as one second. You can use the `setTimeout` function to call `setAlert(false)`, but you'll need to wrap it in `useEffect` to ensure that it does not run on every component render.

Inside of `App.js` create a new effect and pass the `alert` to the array of triggers. This will cause the effect to run any time `alert` changes. Notice that this will run if `alert` changes from `false` to `true`, but it will also run when `alert` changes from `true` to `false`. Since you only want to hide the alert if it is displayed, add a condition inside the effect to only run `setTimeout` if `alert` is `true`:

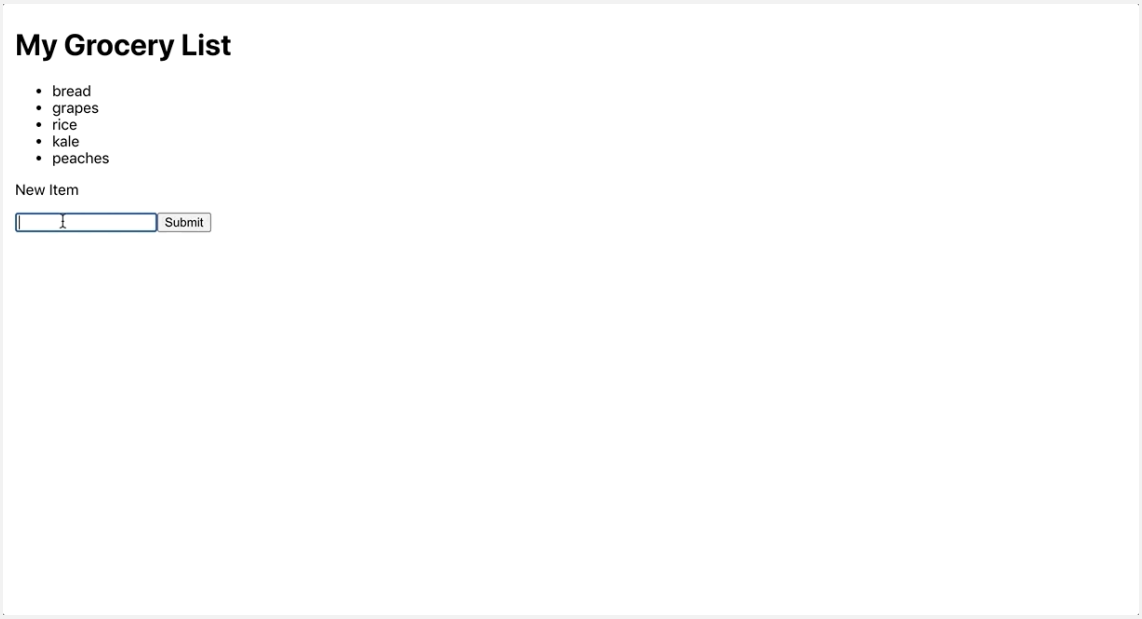
api-tutorial/src/components/App/App.js

```
import React, { useEffect, useState } from 'react';
import './App.css';
import { getList, setItem } from '../../services/list';

function App() {
  const [alert, setAlert] = useState(false);
  const [itemInput, setItemInput] = useState('');
  const [list, setList] = useState([]);
    ...

  useEffect(() => {
    if(alert) {
      setTimeout(() => {
        setAlert(false);
      }, 1000)
    }
  }, [alert])

  const handleSubmit = (e) => {
    e.preventDefault();
    setItem(itemInput)
      .then(() => {
        setItemInput('');
        setAlert(true);
      })
```

```
  };

  return(
    <div className="wrapper">
      ...
    </div>
  )
}

export default App;
```

Run the `setTimeout` function after `1000` milliseconds to ensure the user has time to read the change.

Save the file. Now you have an effect that will run whenever `alert` changes. If there is an active alert, it will start a timeout function that will close the alert after one second.




```
Hide alert, 7
```

Refreshing Fetched Data

Now you need a way to refresh the stale list of data. To do this, you can add a new trigger to the `useEffect` Hook to rerun the `getList` request. To ensure you have the most relevant data, you need a trigger that will update anytime there is a change to the remote data. Fortunately, you can reuse the `alert` state to trigger another data refresh since you know it will run any time a user updates the data. As before, you have to plan for the fact that the effect will run every time `alert` changes including when the alert message disappears.

This time, the effect also needs to fetch data when the page loads. Create a conditional that will exit the function before the data fetch if `list.length` is truthy—indicating you have already fetched the data—and `alert` is `fals`

`e`—indicating you have already refreshed the data. Be sure to add `alert` and `list` to the array of triggers:

```
import React, { useEffect, useState } from 'react';
import './App.css';
import { getList, setItem } from '../../services/list';

function App() {
  const [alert, setAlert] = useState(false);
  const [itemInput, setItemInput] = useState('');
  const [list, setList] = useState([]);

  useEffect(() => {
    let mounted = true;
    if(list.length && !alert) {
      return;
    }
    getList()
      .then(items => {
        if(mounted) {
          setList(items)
        }
      })
    return () => mounted = false;
  }, [alert, list])

  ...

  return(
```

```
    <div className="wrapper">
      ...
    </div>
  )
}

export default App;
```

Save the file. When you do, the data will refresh after you submit a new item:

List Refresh, 8

In this case, `alert` is not directly related to the `list` state. However, it does occur at the same time as an event that will invalidate the old data, so you

can use it to refresh the data.

Preventing Updates on Unmounted Components

The last problem you need to account for is making sure you do not set state on an unmounted component. You have a solution to the problem with `let mounted = true` in your effect to fetch data, but the solution will not work for `handleSubmit`, since it is not an effect. You can't return a function to set the value to false when it is unmounted. Further, it would be inefficient to add the same check to every function.

To solve this problem, you can make a shared variable that is used by multiple functions by lifting `mounted` out of the `useEffect` Hook and holding it on the level of the component. You'll still use the function to set the value to `false` at the end of the `useEffect`.

Inside `App.js`, declare `mounted` at the start of the function. Then check if the component is mounted before setting data in the other asynchronous functions. Make sure to remove the `mounted` declaration inside the `useEffect` function:

api-tutorial/src/components/App/App.js

```
import React, { useEffect, useState } from 'react';
import './App.css';
import { getList, setItem } from '../../services/list';

function App() {
  const [alert, setAlert] = useState(false);
  const [itemInput, setItemInput] = useState('');
  const [list, setList] = useState([]);
  let mounted = true;

  useEffect(() => {
    if(list.length && !alert) {
      return;
    }
    getList()
      .then(items => {
        if(mounted) {
          setList(items)
        }
      })
    return () => mounted = false;
  }, [alert, list])

  useEffect(() => {
    if(alert) {
```

```
      setTimeout(() => {
        if(mounted) {
          setAlert(false);
        }
      }, 1000)
    }
  }, [alert])

  const handleSubmit = (e) => {
    e.preventDefault();
    setItem(itemInput)
      .then(() => {
        if(mounted) {
          setItemInput('');
          setAlert(true);
        }
      })
  };

  return(
    <div className="wrapper">
      ...
    </div>
  )
}

export default App;
```

When you make the change, you'll receive an error in the terminal where you are running your React app:

```
Error
Assignments to the 'mounted' variable from inside React Hook useEffect will be lost after each render. To preserve the value over time, store it in a useRef Hook and keep the mutable value in the '.current' property. Otherwise, you can move this variable directly inside useEffect  react-hooks/exhaustive-deps
```

React is alerting you that variables are not stable. Whenever there is a re-render, it will recalculate the variable. Normally, this will ensure up-to-date information. In this case, you are relying on that variable to persist.

The solution is another Hook called `useRef`. The `useRef` Hook will preserve a variable for the lifetime of the component. The only trick is to get the value you need to use the `.current` property.

Inside `App.js`, convert `mounted` to a reference using the `useRef` Hook. Then convert each usage of `mounted` to `mounted.current`:

api-tutorial/src/components/App/App.js

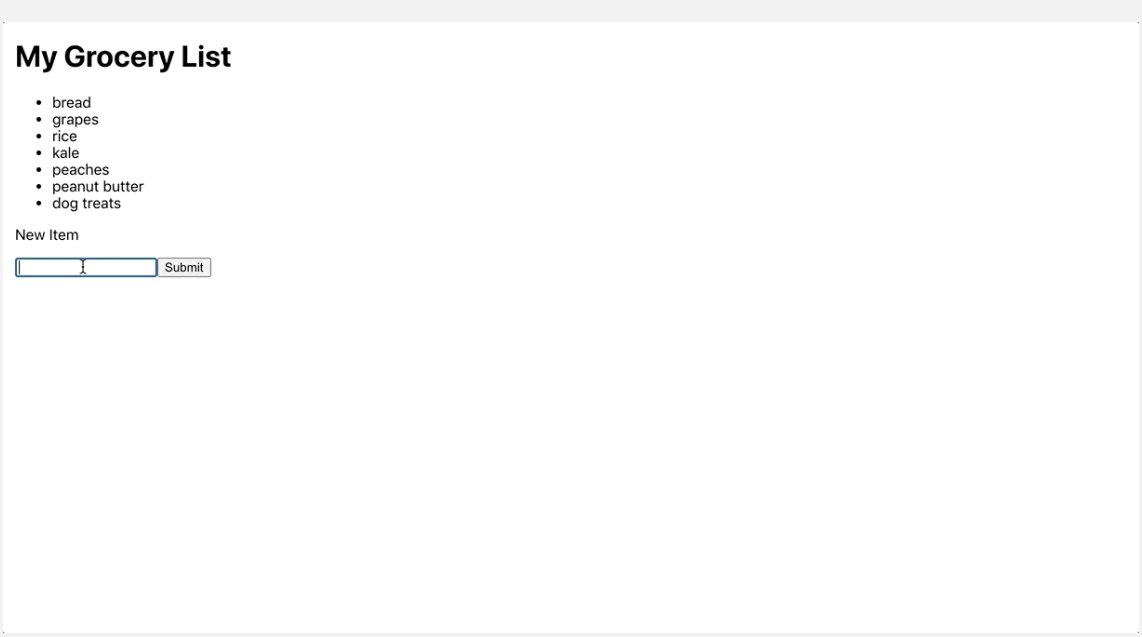
```
import React, { useEffect, useRef, useState } from 'react';
import './App.css';
import { getList, setItem } from '../../services/list';

function App() {
  const [alert, setAlert] = useState(false);
  const [itemInput, setItemInput] = useState('');
  const [list, setList] = useState([]);
  const mounted = useRef(true);

  useEffect(() => {
    mounted.current = true;
    if(list.length && !alert) {
      return;
    }
    getList()
      .then(items => {
        if(mounted.current) {
          setList(items)
        }
      })
    return () => mounted.current = false;
  }, [alert, list])

  useEffect(() => {
```

```
    if(alert) {
      setTimeout(() => {
        if(mounted.current) {
          setAlert(false);
        }
      }, 1000)
    }
  }, [alert])

  const handleSubmit = (e) => {
    e.preventDefault();
    setItem(itemInput)
      .then(() => {
        if(mounted.current) {
          setItemInput('');
          setAlert(true);
        }
      })
  };

  return(
    <div className="wrapper">
       ...
    </div>
  )
}
```

```
export default App;
```

In addition, be cautious about setting the variable in the cleanup function for `useEffect`. The cleanup function will always run before the effect reruns. That means that the cleanup function `() => mounted.current = false` will run every time the `alert` or `list` change. To avoid any false results, be sure to update the `mounted.current` to `true` at the start of the effect. Then you can be sure it will only be set to `false` when the component is unmounted.

Save and close the file. When the browser refreshes, you'll be able to save new list items:




```
Saving again, 9
```

Note: It is a common problem to accidentally rerun an API multiple times. Every time a component is removed and then remounted, you will rerun all the original data fetching. To avoid this, consider a caching method for APIs that are particularly data heavy or slow. You can use anything from memoizing the service calls, to caching with service workers, to a custom Hook. There are a few popular custom Hooks for caching service calls, including useSWR and react query.

No matter which approach you use, be sure to consider how you will invalidate the cache because there are times where you'll want to fetch the newest data.

In this step, you sent data to an API. You learned how to update the user when the data is submitted and how to trigger a refresh on your list data. You also avoided setting data on unmounted components by using the `useRef` Hook to store the status of the component so that it can be used by multiple services.

Conclusion

APIs give you the ability to connect to many useful services. They allow you to store and retrieve data even after a user closes their browser or stops using an application. With well organized code, you can isolate your services from your components so that your components can focus on rendering data without knowing where the data is originating. Web APIs extend your application far beyond the capabilities of a browser session or

storage. They open your application to the whole world of web technologies.

If you would like to read more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Manage State in React with Redux

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

Redux is a popular data store for JavaScript and React applications. It follows a central principle that data binding should flow in one direction and should be stored as a single source of truth. Redux gained popularity because of the simplicity of the design concept and the relatively small implementation.

Redux operates according to a few concepts. First, the store is a single object with fields for each selection of data. You update the data by dispatching an action that says how the data should change. You then interpret actions and update the data using reducers. Reducers are functions that apply actions to data and return a new state, instead of mutating the previous state.

In small applications, you may not need a global data store. You can use a mix of local state and context to manage state. But as your application scales, you may encounter situations where it would be valuable to store information centrally so that it will persist across routes and components. In that situation, Redux will give you a standard way to store and retrieve data in an organized manner.

In this tutorial, you'll use Redux in a React application by building a bird watching test application. Users will be able to add birds they have seen and increment a bird each time they see it again. You'll build a single data store, and you'll create actions and reducers to update the store. You'll then pull data into your components and dispatch new changes to update the data.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.22.0 and npm version 6.14.6. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- A React development environment set up with Create React App, with the non-essential boilerplate removed. To set this up, follow **Step 1 — Creating an Empty Project** of the How To Manage State on React Class Components tutorial. This tutorial will use `redux-tutorial` as the project name.

- You will be using React components, Hooks, and forms in this tutorial, including the `useState` Hook and custom Hooks. You can learn about components and Hooks in our tutorials How To Manage State with Hooks on React Components and How To Build Forms in React.

- You will also need a basic knowledge of JavaScript, HTML, and CSS, which you can find in our How To Build a Website With HTML series,

How To Build a Website With CSS series, and in How To Code in JavaScript.

Step 1 — Setting Up a Store

In this step, you'll install Redux and connect it to your root component. You'll then create a base store and show the information in your component. By the end of this step, you'll have a working instance of Redux with information displaying in your components.

To start, install `redux` and `react-redux`. The package `redux` is framework agnostic and will connect your actions and reducers. The package `react-redux` contains the bindings to run a Redux store in a React project. You'll use code from `react-redux` to send actions from your components and to pull data from the store into your components.

Use `npm` to install the two packages with the following command:

```
npm install --save redux react-redux
```

When the component is finished installing, you'll receive output like this. Your output may be slightly different:

Output

```
...
+ redux@4.0.5
+ react-redux@7.2.1
added 2 packages from 1 contributor, updated 1 package and aud
ited 1639 packages in 20.573s
```

Now that you have the packages installed, you need to connect Redux to your project. To use Redux, you'll need to wrap your root components with a `Provider` to ensure that the store is available to all child components in the tree. This is similar to how you would add a `Provider` using React's native `context`.

Open `src/index.js`:

```
nano src/index.js
```

Import the `Provider` component from the `react-redux` package. Add the `Provider` to your root component around any other components by making the following highlighted changes to your code:

redux-tutorial/src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './components/App/App';
import * as serviceWorker from './serviceWorker';
import { Provider } from 'react-redux';

ReactDOM.render(
  <React.StrictMode>
    <Provider>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want your app to work offline and load faster, you ca
// unregister() to register() below. Note this comes with some
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Now that you have wrapped your components, it's time to add a `store`. The `store` is your central collection of data. In the next step, you'll learn to

create `reducers` that will set the default values and update your store, but for now you will hard-code the data.

Import the `createStore` function from `redux`, then pass a function that returns an object. In this case, return an object with a field called `birds` that points to an array of individual birds. Each bird will have a `name` and a `views` count. Save the output of the function to a value called `store`, then pass the `store` to a prop called `store` in the `Provider`:

redux-tutorial/src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './components/App/App';
import * as serviceWorker from './serviceWorker';
import { Provider } from 'react-redux';
import { createStore } from 'redux';

const store = createStore(() => ({
  birds: [
    {
      name: 'robin',
      views: 1
    }
  ]
}));

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
);
```

```
// If you want your app to work offline and load faster, you ca
// unregister() to register() below. Note this comes with some
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Save and close the file. Now that you have some data, you need to be able to display it. Open `src/components/App/App.js`:

```
nano src/components/App/App.js
```

Like with `context`, every child component will be able to access the store without any additional props. To access items in your Redux store, use a Hook called `useSelector` from the `react-redux` package. The `useSelector` Hook takes a selector function as an argument. The selector function will receive the state of your store as an argument that you will use to return the field you want:

redux-tutorial/src/components/App/App.js

```
import React from 'react';
import { useSelector } from 'react-redux';
import './App.css';

function App() {
  const birds = useSelector(state => state.birds);

  return <></>
}

export default App;
```

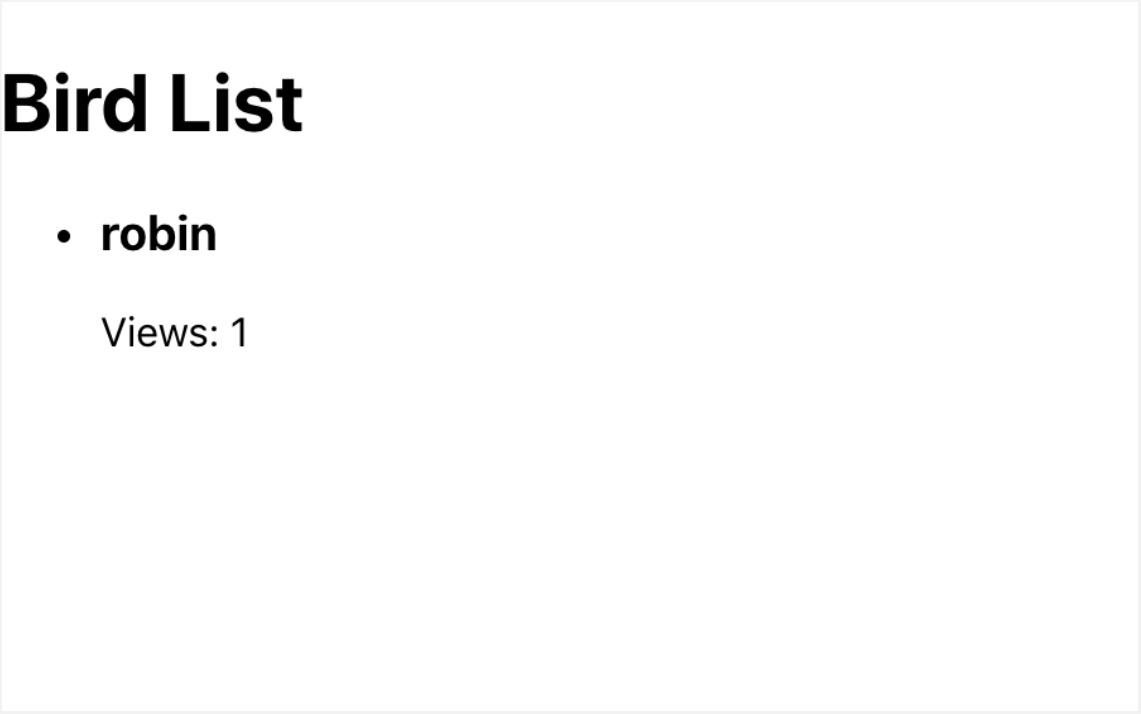
Since `useSelector` is a custom Hook, the component will re-render whenever the Hook is called. That means that the data—`birds`—will always be up to date.

Now that you have the data, you can display it in an unordered list. Create a surrounding `<div>` with a `className` of `wrapper`. Inside, add a `<ul>` element and loop over the `birds` array with `map()`, returning a new `<li>` item for each. Be sure to use the `bird.name` as a `key`:

redux-tutorial/src/components/App/App.js

```
import React from 'react';
import { useSelector } from 'react-redux'
import './App.css';

function App() {
  const birds = useSelector(state => state.birds);

  return (
    <div className="wrapper">
      <h1>Bird List</h1>
      <ul>
        {birds.map(bird => (
          <li key={bird.name}>
            <h3>{bird.name}</h3>
            <div>
              Views: {bird.views}
            </div>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

Save the file. Once the file is saved, the browser will reload and you'll find your bird list::




```
List of birds
```

Now that you have a basic list, add in the rest of the components you'll need for your bird watching app. First, add a button to increment the views after the list of views:

redux-tutorial/src/components/App/App.js

```
import React from 'react';
import { useSelector } from 'react-redux'
import './App.css';

function App() {
  const birds = useSelector(state => state.birds);

  return (
    <div className="wrapper">
      <h1>Bird List</h1>
      <ul>
        {birds.map(bird => (
          <li key={bird.name}>
            <h3>{bird.name}</h3>
            <div>
              Views: {bird.views}
              <button><span role="img" aria-label="add">+</spar
            </div>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

```
export default App;
```

Next, create a `<form>` with a single `<input>` before the bird list so a user can add in a new bird. Be sure to surround the `<input>` with a `<label>` and to add a `type` of `submit` to the add button to make sure everything is accessible:

redux-tutorial/src/components/App/App.js

```
import React from 'react';
import { useSelector } from 'react-redux'
import './App.css';

function App() {
  const birds = useSelector(state => state.birds);

  return (
    <div className="wrapper">
      <h1>Bird List</h1>
      <form>
        <label>
          <p>
            Add Bird
          </p>
          <input type="text" />
        </label>
        <div>
          <button type="submit">Add</button>
        </div>
      </form>
      <ul>
        {birds.map(bird => (
          <li key={bird.name}>
            <h3>{bird.name}</h3>
```

```
            <div>
              Views: {bird.views}
              <button><span role="img" aria-label="add">+</spar
            </div>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

Save and close the file. Next, open up `App.css` to add some styling:

```
nano src/components/App/App.css
```

Add some `padding` to the `wrapper` class. Then capitalize the `h3` element, which holds the bird name. Finally, style the buttons. Remove the default button styles on the add `<button>` and then add a margin to the form `<butt on>`.

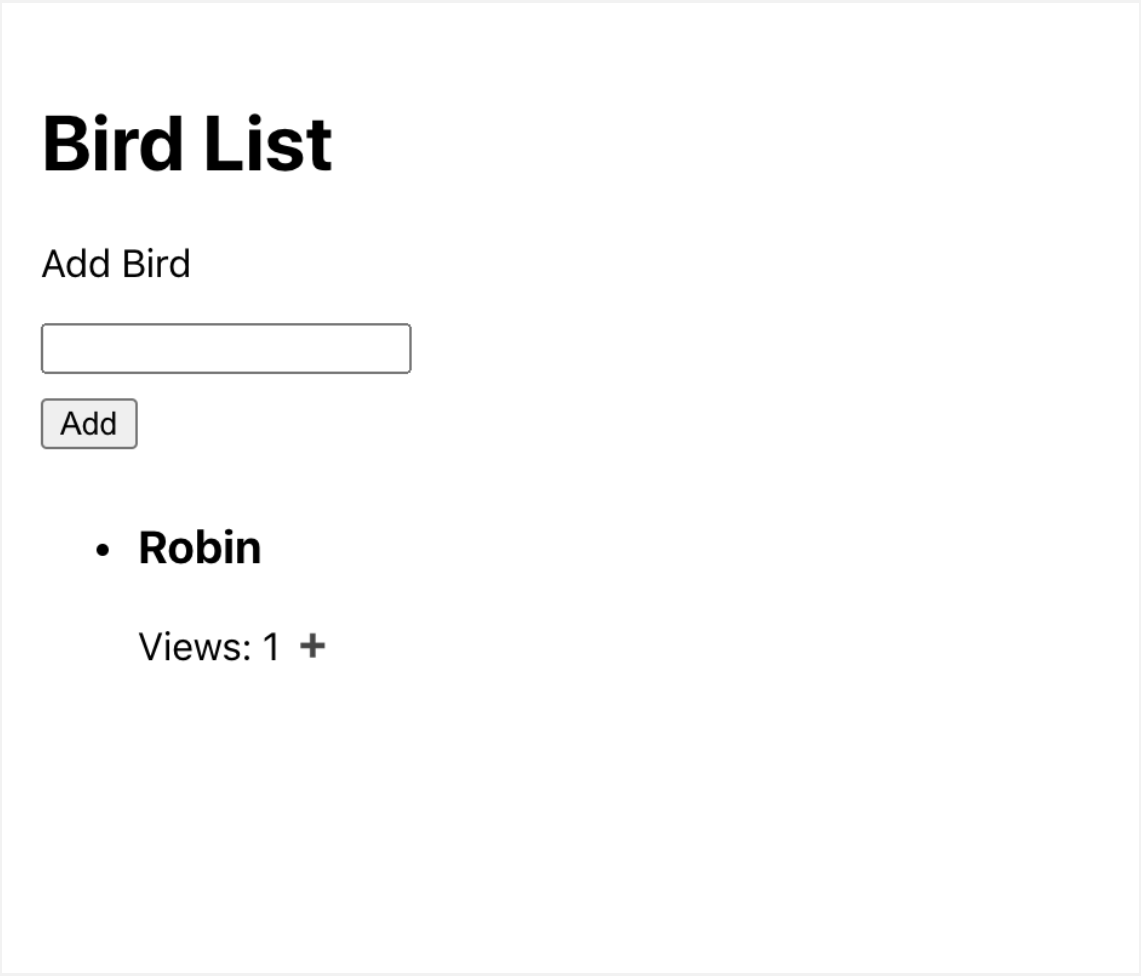
Replace the file's contents with the following:

redux-tutorial/src/components/App/App.css

```
.wrapper {
    padding: 20px;
}

.wrapper h3 {
    text-transform: capitalize;
}

.wrapper form button {
    margin: 10px 0;
    cursor: pointer;
}

.wrapper ul button {
    background: none;
    border: none;
    cursor: pointer;
}
```

Additionally, give each button a `cursor` of `pointer`, which will change the cursor when hovering over the button to indicate to the user that the button is clickable.

Save and close the file. When you do the browser will refresh with your components:



Bird watching app with form

The buttons and form are not connected to any actions yet, and so can not interact with the Redux store. You'll add the actions in Step 2 and connect them in Step 3.

In this step, you installed Redux and created a new store for your application. You connected the store to your application using `Provider`

and accessed the elements inside your components using the `useSelector` Hook.

In the next step, you'll create actions and reducers to update your store with new information.

Step 2 — Creating Actions and Reducers

Next, you'll create actions to add a bird and to increment a view. You'll then make a reducer that will update the information depending on the action type. Finally, you'll use the reducers to create a default store using `combineReducers`.

Actions are the message you send to the data store with the intended change. Reducers take those messages and update the shared store by applying the changes depending on the action type. Your components will send the actions they want your store to use, and your reducers will use actions to update the data in the store. You never call reducers directly, and there are cases where one action may impact several reducers.

There are many different options for organizing your actions and reducers. In this tutorial, you'll organize by domain. That means your actions and reducers will be defined by the type of feature they will impact.

Create a directory called `store`:

```
mkdir src/store
```

This directory will contain all of your actions and reducers. Some patterns store them alongside components, but the advantage here is that you have a separate point of reference for the shape of the whole store. When a new developer enters the project, they will be able to read the structure of the store at a glance.

Make a directory called `birds` inside the `store` directory. This will contain the actions and reducers specifically for updating your bird data:

```
mkdir src/store/birds
```

Next, open up a file called `birds.js` so that you can start to add actions and reducers. If you have a large number of actions and reducers you may want to split them into separate files, such as `birds.actions.js` and `birds.reducers.js`, but when there are only a few it can be easier to read when they are in the same location:

```
nano src/store/birds/birds.js
```

First, you are going to create actions. Actions are the messages that you send from a component to your store using a method called `dispatch`, which you'll use in the next step.

An action must return an object with a `type` field. Otherwise, the return object can include any additional information you want to send.

Create a function called `addBirds` that takes a `bird` as an argument and returns an object containing a `type` of `'ADD_BIRD'` and the `bird` as a field:

redux-tutorial/src/store/birds/birds.js

```
export function addBird(bird) {
  return {
    type: 'ADD_BIRD',
    bird,
  }
}
```

Notice that you are exporting the function so that you can later import and dispatch it from your component.

The `type` field is important for communicating with reducers, so by convention most Redux stores will save the type to a variable to protect against misspelling.

Create a `const` called `ADD_BIRD` that saves the string `'ADD_BIRD'`. Then update the action:

redux-tutorial/src/store/birds/birds.js

```
const ADD_BIRD = 'ADD_BIRD';

export function addBird(bird) {
  return {
    type: ADD_BIRD,
    bird,
  }
}
```

Now that you have an action, create a reducer that will respond to the action.

Reducers are functions that will determine how a state should change based on actions. The actions don’t make changes themselves; the reducers will take the state and make changes based on actions.

A reducer receives two arguments: the current state and the action. The current state refers to the state for a particular section of the store. Generally, the name of the reducer will match with a field in the store. For example, suppose you had a store shaped like this:

```
{
  birds: [
    // collection of bird objects
  ],
  gear: {
    // gear information
  }
}
```

You would create two reducers: `birds` and `gear`. The `state` for the `birds` reducer will be the array of birds. The `state` for the `gear` reducer would be the object containing the gear information.

Inside `birds.js` create a reducer called `birds` that takes `state` and `action` and returns the `state` without any changes:

redux-tutorial/src/store/birds/birds.js

```
const ADD_BIRD = 'ADD_BIRD';

export function addBird(bird) {
  return {
    type: ADD_BIRD,
    bird,
  }
}

function birds(state, action) {
  return state;
}
```

Notice that you are not exporting the reducer. You will not use the reducer directly and instead will combine them into a usable collection that you will export and use to create your base store in `index.js`. Notice also that you need to return the `state` if there are no changes. Redux will run all the reducers anytime you dispatch an action, so if you don't return state you risk losing your changes.

Finally, since Redux returns the state if there are no changes, add a default state using default parameters.

Create a `defaultBirds` array that will have the placeholder bird information. Then update the `state` to include `defaultBirds` as the default parameter:

redux-tutorial/src/store/birds/birds

```
const ADD_BIRD = 'ADD_BIRD';

export function addBird(bird) {
  return {
    type: ADD_BIRD,
    bird,
  }
}

const defaultBirds = [
  {
    name: 'robin',
    views: 1,
  }
];

function birds(state=defaultBirds, action) {
  return state;
}
```

Now that you have a reducer returning your state, you can use the action to apply the changes. The most common pattern is to use a `switch` on the `action.type` to apply changes.

Create a `switch` statement that will look at the `action.type`. If the case is `ADD_BIRD`, spread out the current state into a new array and add the bird with a single view:

redux-tutorial/src/store/birds/birds.js

```
const ADD_BIRD = 'ADD_BIRD';

export function addBird(bird) {
  return {
    type: ADD_BIRD,
    bird,
  }
}

const defaultBirds = [
  {
    name: 'robin',
    views: 1,
  }
];

function birds(state=defaultBirds, action) {
  switch (action.type) {
    case ADD_BIRD:
      return [
        ...state,
        {
          name: action.bird,
          views: 1
        }
```

```
      ];
    default:
      return state;
  }
}
```

Notice that you are returning the `state` as the `default` value. More importantly, you are not mutating `state` directly. Instead, you are creating a new array by spreading the old array and adding a new value.

Now that you have one action, you can create an action for incrementing a view.

Create an action called `incrementBird`. Like the `addBird` action, this will take a bird as an argument and return an object with a `type` and a `bird`. The only difference is the type will be `'INCREMENT_BIRD'`:

redux-tutorial/src/store/birds/birds.js

```
const ADD_BIRD = 'ADD_BIRD';
const INCREMENT_BIRD = 'INCREMENT_BIRD';

export function addBird(bird) {
  return {
    type: ADD_BIRD,
    bird,
  }
}

export function incrementBird(bird) {
  return {
    type: INCREMENT_BIRD,
    bird
  }
}

const defaultBirds = [
  {
    name: 'robin',
    views: 1,
  }
];

function birds(state=defaultBirds, action) {
```

```
  switch (action.type) {
    case ADD_BIRD:
      return [
        ...state,
        {
          name: action.bird,
          views: 1
        }
      ];
    default:
      return state;
  }
}
```

This action is separate, but you will use the same reducer. Remember, the actions convey the change you want to make on the data and the reducer applies those changes to return a new state.

Incrementing a bird involves a bit more than adding a new bird. Inside of `birds` add a new case for `INCREMENT_BIRD`. Then pull the bird you need to increment out of the array using `find()` to compare each `name` with the `action.bird`:

redux-tutorial/src/store/bird/birds.js

```
const ADD_BIRD = 'ADD_BIRD';
...
function birds(state=defaultBirds, action) {
  switch (action.type) {
    case ADD_BIRD:
      return [
        ...state,
        {
          name: action.bird,
          views: 1
        }
      ];
    case INCREMENT_BIRD:
      const bird = state.find(b => action.bird === b.name);
      return state;
    default:
      return state;
  }
}
```

You have the bird you need to change, but you need to return a new state containing all the unchanged birds as well as the bird you're updating. Select all remaining birds with `state.filter` by selecting all birds with a `n`

`ame` that does not equal `action.name`. Then return a new array by spreading the `birds` array and adding the `bird` at the end:

redux-tutorial/src/store/bird/birds.js

```
const ADD_BIRD = 'ADD_BIRD';
...

function birds(state=defaultBirds, action) {
  switch (action.type) {
    case ADD_BIRD:
      return [
        ...state,
        {
          name: action.bird,
          views: 1
        }
      ];
    case INCREMENT_BIRD:
      const bird = state.find(b => action.bird === b.name);
      const birds = state.filter(b => action.bird !== b.name);
      return [
        ...birds,
        bird,
      ];
    default:
      return state;
  }
}
```

Finally, update the `bird` by creating a new object with an incremented `view`:

redux-tutorial/src/store/bird/birds.js

```
const ADD_BIRD = 'ADD_BIRD';
...
function birds(state=defaultBirds, action) {
  switch (action.type) {
    case ADD_BIRD:
      return [
        ...state,
        {
          name: action.bird,
          views: 1
        }
      ];
    case INCREMENT_BIRD:
      const bird = state.find(b => action.bird === b.name);
      const birds = state.filter(b => action.bird !== b.name);
      return [
        ...birds,
        {
          ...bird,
          views: bird.views + 1
        }
      ];
    default:
      return state;
```

```
  }
}
```

Notice that you are not using the reducers to sort the data. Sorting could be considered a view concern since the view displays the information to a user. You could have one view that sorts by name and one view that sorts by view count, so it's better to let individual components handle the sorting. Instead, keep reducers focused on updating the data, and the component focused on converting the data to a usable view for a user.

This reducer is also imperfect since you could add birds with the same name. In a production app you would need to either validate before adding or give birds a unique `id` so that you could select the bird by `id` instead of `name`.

Now you have two complete actions and a reducer. The final step is to export the reducer so that it can initialize the store. In the first step, you created the store by passing a function that returns an object. You will do the same thing in this case. The function will take the `store` and the `action` and then pass the specific slice of the `store` to the reducers along with the action. It would look something like this:

```
export function birdApp(store={}, action) {
    return {
        birds: birds(store.birds, action)
```

```
    }
}
```

To simplify things, Redux has a helper function called `combineReducers` that combines the reducers for you.

Inside of `birds.js`, import `combineReducers` from `redux`. Then call the function with `birds` and export the result:

redux-tutorial/src/store/bird/birds.js

```
import { combineReducers } from 'redux';
const ADD_BIRD = 'ADD_BIRD';
const INCREMENT_BIRD = 'INCREMENT_BIRD';

export function addBird(bird) {
  return {
    type: ADD_BIRD,
    bird,
  }
}

export function incrementBird(bird) {
  return {
    type: INCREMENT_BIRD,
    bird
  }
}

const defaultBirds = [
  {
    name: 'robin',
    views: 1,
  }
];
```

```
function birds(state=defaultBirds, action) {
  switch (action.type) {
    case ADD_BIRD:
      return [
        ...state,
        {
          name: action.bird,
          views: 1
        }
      ];
    case INCREMENT_BIRD:
      const bird = state.find(b => action.bird === b.name);
      const birds = state.filter(b => action.bird !== b.name);
      return [
        ...birds,
        {
          ...bird,
          views: bird.views + 1
        }
      ];
    default:
      return state;
  }
}

const birdApp = combineReducers({
  birds
```

```
});


export default birdApp;
```

Save and close the file.

Your actions and reducers are all set up. The final step is to initialize your store using the combined reducers instead of a placeholder function.

Open `src/index.js`:

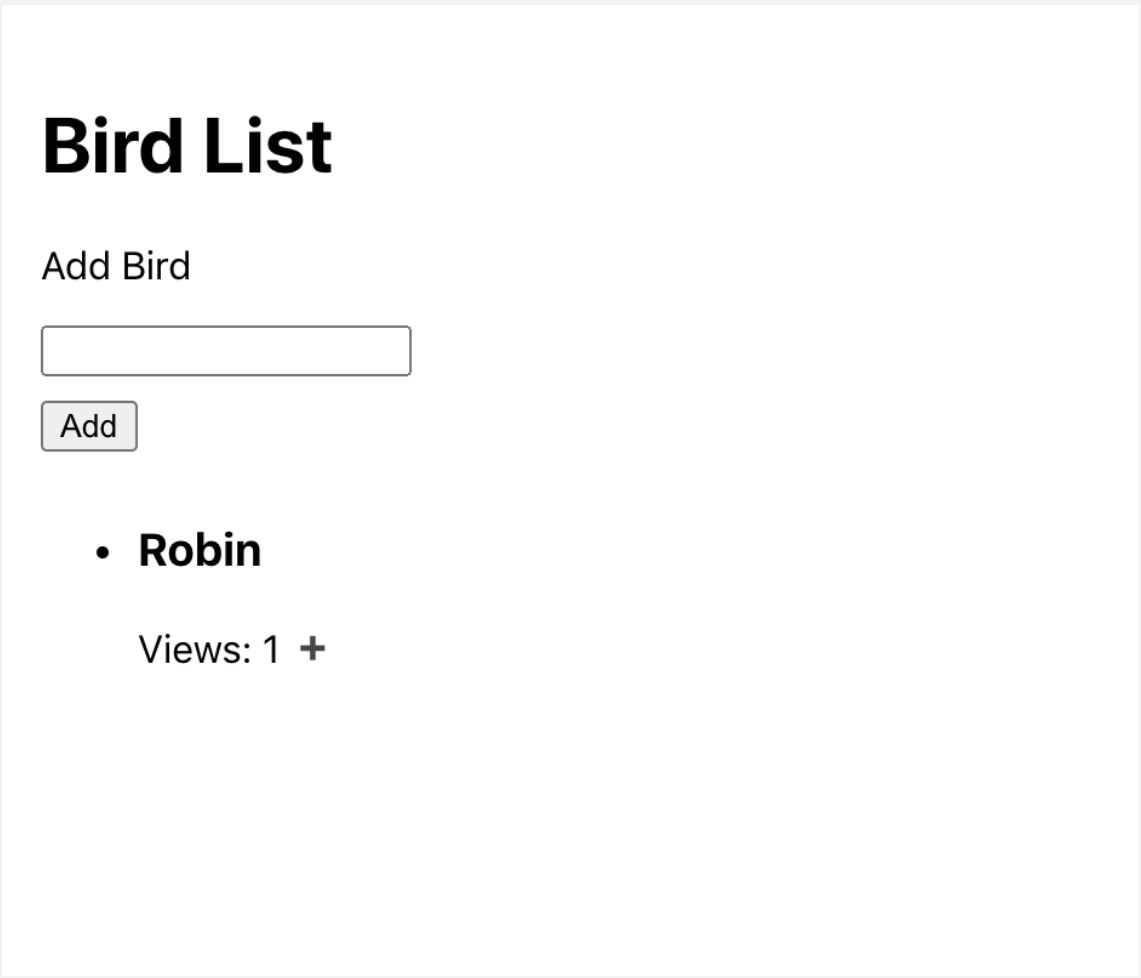
```
nano src/index.js
```

Import the `birdApp` from `birds.js`. Then initialize the `store` using `birdAp p`:

redux-tutorial/src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './components/App/App';
import * as serviceWorker from './serviceWorker';
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import birdApp from './store/birds/birds';

const store = createStore(birdApp);

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want your app to work offline and load faster, you ca
// unregister() to register() below. Note this comes with some
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Save and close the file. When you do the browser will refresh with your application:



Bird watching app with form

In this step you created actions and reducers. You learned how to create actions that return a `type` and how to build reducers that use the action to

build and return a new state based on the action. Finally, you combined the reducers into a function that you used to initialize the store.

Your Redux store is now all set up and ready for changes. In the next step you'll dispatch actions from a component to update the data.

Step 3 — Dispatching Changes in a Component

In this step, you'll import and call your actions from your component. You'll use a method called `dispatch` to send the action and you'll dispatch the actions inside of event handlers for the `form` and the `button`.

By the end of this step, you'll have a working application that combines a Redux store and your custom components. You'll be able to update the Redux store in real time and will be able to display the information in your component as it changes.

Now that you have working actions, you need to connect them to your events so that you can update the store. The method you will use is called `dispatch` and it sends a particular action to the Redux store. When Redux receives an action you have dispatched, it will pass the action to the reducers and they will update the data.

Open `App.js`:

```
nano src/components/App/App.js
```

Inside of `App.js` import the Hook `useDispath` from `react-redux`. Then call the function to create a new `dispatch` function:

redux-tutorial/src/components/App/App.js

```
import React from 'react';
import { useDispatch, useSelector } from 'react-redux'
import './App.css';

function App() {
  ...
}

export default App;
```

Next you'll need to import your actions. Remember, actions are functions that return an object. The object is what you will ultimately pass into the `dispatch` function.

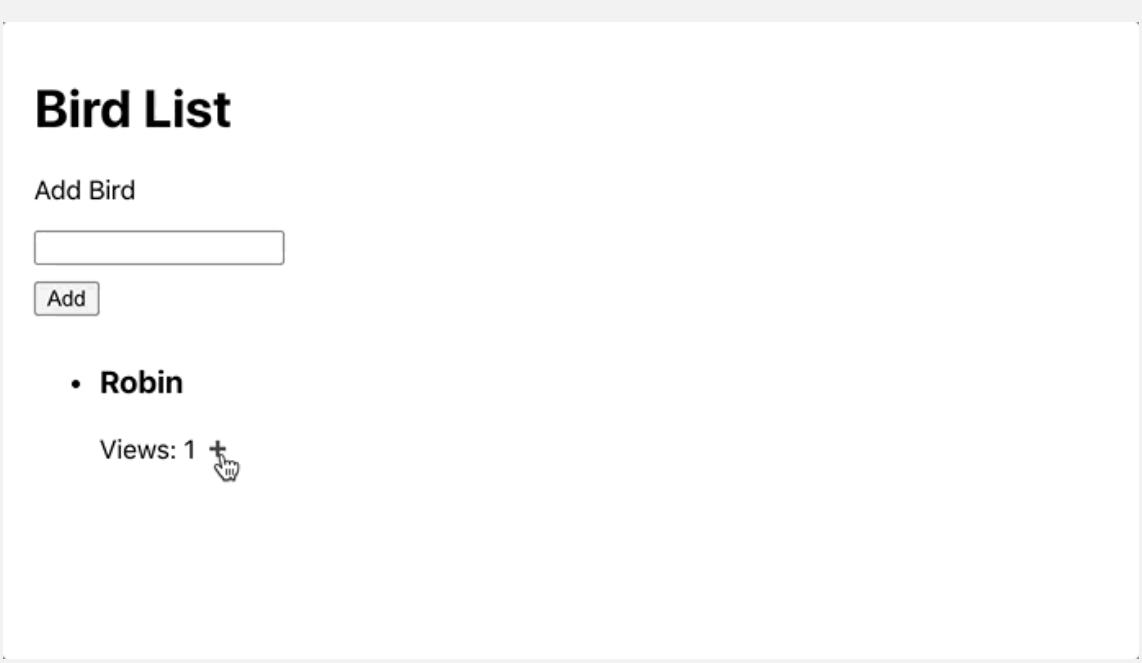
Import `incrementBird` from the store. Then create an `onClick` event on the button. When the user clicks on the button, call `incrementBird` with `bird.name` and pass the result to `dispatch`. To make things more readable, call the `incrementBird` function inside of `dispatch`:

redux-tutorial/src/components/App/App.js

```
import React from 'react';
import { useDispatch, useSelector } from 'react-redux'
import { incrementBird } from '../../store/birds/birds';
import './App.css';

function App() {
  const birds = useSelector(state => state.birds);
  const dispatch = useDispatch();

  return (
    <div className="wrapper">
      <h1>Bird List</h1>
      <form>
        <label>
          <p>
            Add Bird
          </p>
          <input type="text" />
        </label>
        <div>
          <button type="submit">Add</button>
        </div>
      </form>
      <ul>
        {birds.map(bird => (
```

```
          <li key={bird.name}>
            <h3>{bird.name}</h3>
            <div>
              Views: {bird.views}
              <button onClick={() => dispatch(incrementBird(bir
                <span role="img" aria-label="add">+</span></but
            </div>
          </li>
        ))}
      </ul>
    </div>
  );
}


export default App;
```

Save the file. When you do, you'll be able to increment the robin count:



Increment a bird

Next, you need to dispatch the `addBird` action. This will take two steps: saving the input to an internal state and triggering the dispatch with `onSubmit`.

Use the `useState` Hook to save the input value. Be sure to convert the input to a controlled component by setting the `value` on the input. Check out the tutorial How To Build Forms in React for a more in-depth look at controlled components.

Make the following changes to your code:

redux-tutorial/src/components/App/App.js

```
import React, { useState } from 'react';
import { useDispatch, useSelector } from 'react-redux'
import { incrementBird } from '../../store/birds/birds';
import './App.css';

function App() {
  const [birdName, setBird] = useState('');
  const birds = useSelector(state => state.birds);
  const dispatch = useDispatch();

  return (
    <div className="wrapper">
      <h1>Bird List</h1>
      <form>
        <label>
          <p>
            Add Bird
          </p>
          <input
            type="text"
            onChange={e => setBird(e.target.value)}
            value={birdName}
          />
        </label>
        <div>
```

```
          <button type="submit">Add</button>
        </div>
      </form>
      <ul>
        ...
      </ul>
    </div>
  );
}

export default App;
```

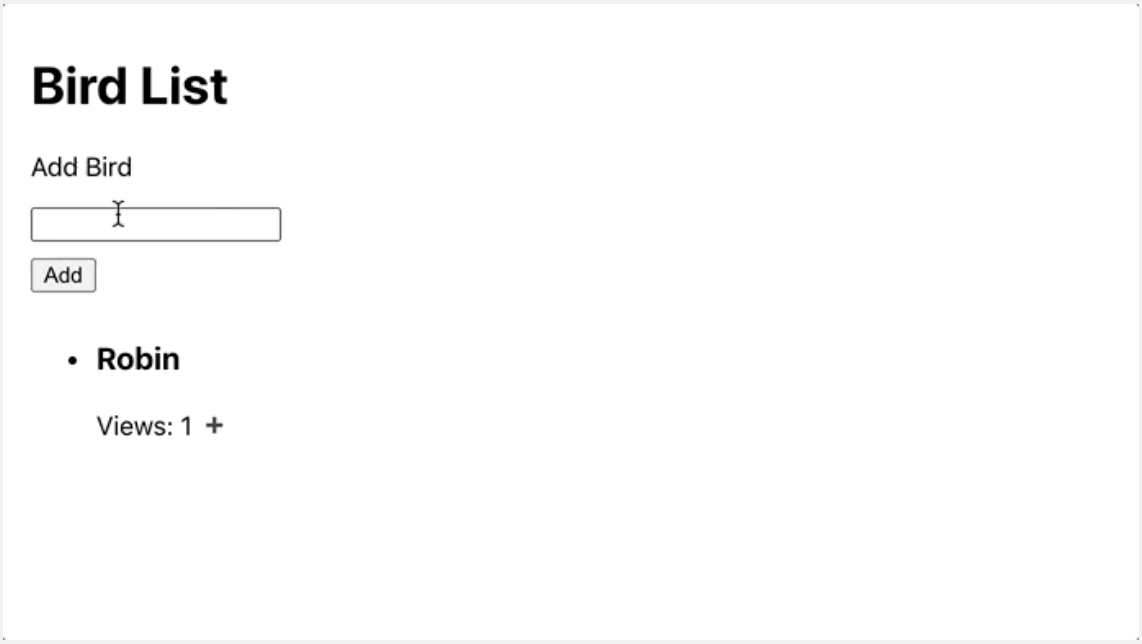
Next, import `addBird` from `birds.js`, then create a function called `handleSubmit`. Inside the `handleSubmit` function, prevent the page form submission with `event.preventDefault`, then dispatch the `addBird` action with the `birdName` as an argument. After dispatching the action, call `setBird('')` to clear the input. Finally, pass `handleSubmit` to the `onSubmit` event handler on the `form`:

redux-tutorial/src/components/App/App.js

```
import React, { useState } from 'react';
import { useDispatch, useSelector } from 'react-redux'
import { addBird, incrementBird } from '../../store/birds/birds
import './App.css';

function App() {
  const [birdName, setBird] = useState('');
  const birds = useSelector(state => state.birds);
  const dispatch = useDispatch();

  const handleSubmit = event => {
    event.preventDefault();
    dispatch(addBird(birdName))
    setBird('');
  };

  return (
    <div className="wrapper">
      <h1>Bird List</h1>
      <form onSubmit={handleSubmit}>
        <label>
          <p>
            Add Bird
          </p>
          <input
```

```
            type="text"
            onChange={e => setBird(e.target.value)}
            value={birdName}
          />
        </label>
        <div>
          <button type="submit">Add</button>
        </div>
      </form>
      <ul>
        {birds.map(bird => (
          <li key={bird.name}>
            <h3>{bird.name}</h3>
            <div>
              Views: {bird.views}
              <button onClick={() => dispatch(incrementBird(bir
                <span role="img" aria-label="add">+</span></but
            </div>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

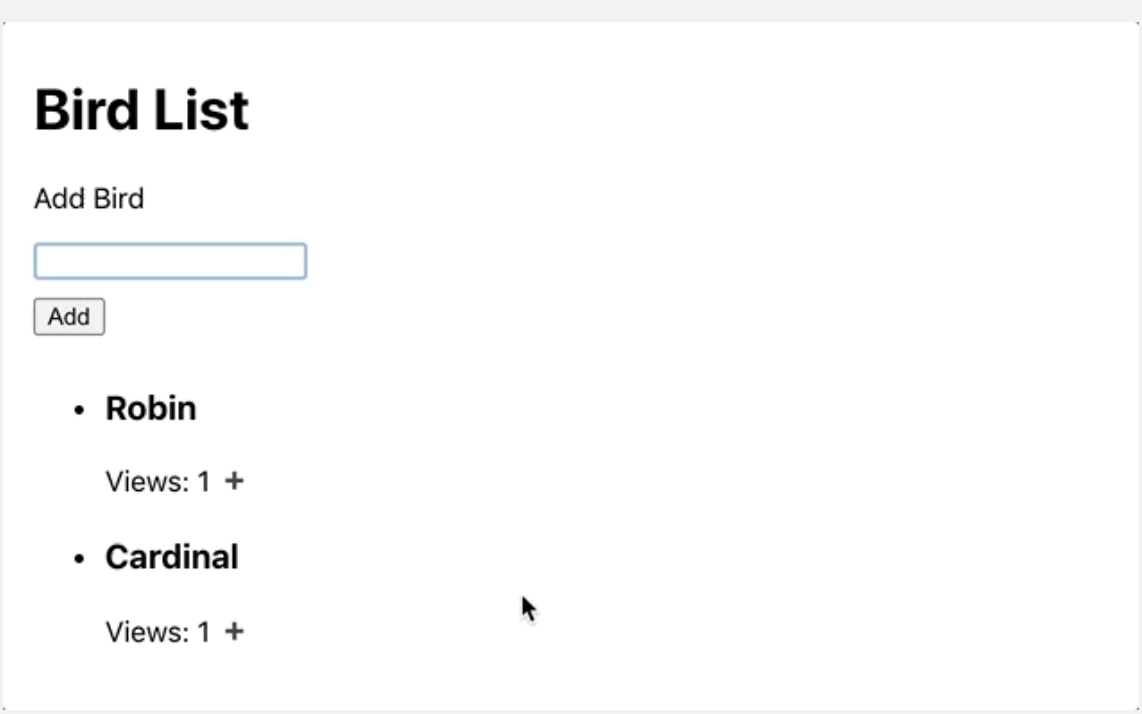
Save the file. When you do, the browser will reload and you'll be able to add a bird:



`Save new bird`

You are now calling your actions and updating your birds list in the store. Notice that when your application refreshed you lost the previous information. The store is all contained in memory and so a page refresh will wipe the data.

This list order will also change if you increment a bird higher in the list.



Robin goes to the bottom on reorder

As you saw in Step 2, your reducer is not concerned with sorting the data. To prevent an unexpected change in the components, you can sort the data in your component. Add a `sort()` function to the `birds` array. Remember that sorting will mutate the array and you never want to mutate the store. Be sure to create a new array by spreading the data before sorting:

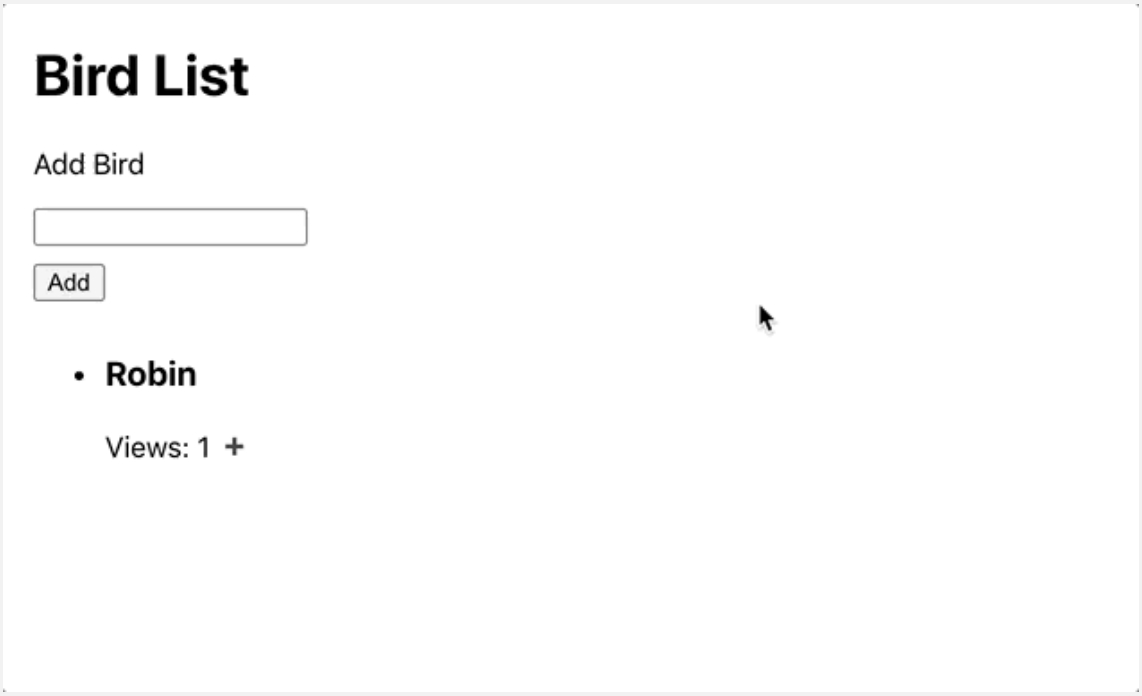
redux-tutorial/src/components/App/App.js

```
import React, { useState } from 'react';
import { useDispatch, useSelector } from 'react-redux'
import { addBird, incrementBird } from '../../store/birds/birds
import './App.css';

function App() {
  const [birdName, setBird] = useState('');
  const birds = [...useSelector(state => state.birds)].sort((a,
    return a.name.toLowerCase() > b.name.toLowerCase() ? 1 : -1
  });
  const dispatch = useDispatch();

  const handleSubmit = event => {
    event.preventDefault();
    dispatch(addBird(birdName))
    setBird('');
  };

  return (
    <div className="wrapper">
      <h1>Bird List</h1>
      <form onSubmit={handleSubmit}>
        <label>
          <p>
            Add Bird
```

```
          </p>
          <input
            type="text"
            onChange={e => setBird(e.target.value)}
            value={birdName}
          />
        </label>
        <div>
          <button type="submit">Add</button>
        </div>
      </form>
      <ul>
        {birds.map(bird => (
          <li key={bird.name}>
            <h3>{bird.name}</h3>
            <div>
              Views: {bird.views}
              <button onClick={() => dispatch(incrementBird(bir
                <span role="img" aria-label="add">+</span></but
            </div>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

```
export default App;
```

Save the file. When you do, the components will stay in alphabetical order as you increment birds.



Cardinal stays on top

It’s important to not try and do too much in your Redux store. Keep the reducers focused on maintaining up-to-date information then pull and manipulate the data for your users inside the component.

Note: In this tutorial, notice that there is a fair amount of code for each action and reducer. Fortunately, there is an officially supported project called Redux Toolkit that can help you reduce the amount of boilerplate code. The Redux Toolkit provides an opinionated set of utilities to quickly create actions and reducers, and will also let you create and configure your store with less code.

In this step, you dispatched your actions from a component. You learned how to call actions and how to send the result to a dispatch function, and you connected them to event handlers on your components to create a fully interactive store. Finally, you learned how to maintain a consistent user experience by sorting the data without directly mutating the store.

Conclusion

Redux is a popular single store. It can be advantageous when working with components that need a common source of information. However, it is not always the right choice in all projects. Smaller projects or projects with isolated components will be able to use built-in state management and context. But as your applications grow in complexity, you may find that central storage is critical to maintaining data integrity. In such cases, Redux is an excellent tool to create a single unified data store that you can use across your components with minimal effort.

If you would like to read more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Handle Routing in React Apps with React Router

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

In React, routers help create and navigate between the different URLs that make up your web application. They allow your user to move between the components of your app while preserving user state, and can provide unique URLs for these components to make them more shareable. With routers, you can improve your app's user experience by simplifying site navigation.

React Router is one of the most popular routing frameworks for React. The library is designed with intuitive components to let you build a declarative routing system for your application. This means that you can declare exactly which of your components has a certain route. With declarative routing, you can create intuitive routes that are human-readable, making it easier to manage your application architecture.

In this tutorial, you'll install and configure React Router, build a set of routes, and connect to them using the `<Link>` component. You'll also build dynamic routes that collect data from a URL that you can access in your component. Finally, you'll use Hooks to access data and other routing information and create nested routes that live inside components that are rendered by parent routes.

By the end of this tutorial, you'll be able to add routes to any React project and read information from your routes so that you can create flexible components that respond to URL data.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.22.0 and npm version 6.14.6. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- A React development environment set up with Create React App, with the non-essential boilerplate removed. To set this up, follow **Step 1 — Creating an Empty Project** of the How To Manage State on React Class Components tutorial. This tutorial will use `router-tutorial` as the project name.

- You will be using React components and custom Hooks throughout the tutorial. You can learn about components in How To Create Custom Components in React and Hooks in How To Manage State with Hooks on React Components.

- You will also need a basic knowledge of JavaScript, HTML, and CSS, which you can find in our How To Build a Website With HTML series, How To Build a Website With CSS series, and in How To Code in JavaScript.

Step 1 — Installing React Router

In this step, you'll install React Router into your base project. In this project, you are going to make a small website about marine mammals. Each mammal will need a separate component that you'll render with the router. After installing the library, you'll create a series of components for each mammal. By the end of this step, you'll have a foundation for rendering different mammals based on route.

To start, install the React Router package. There are two different versions: a web version and a native version for use with React Native. You will install the web version.

In your terminal, use `npm` to install the package:

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

The package will install and you'll receive a message such as this one when the installation is complete. Your message may vary slightly:

Output

```
...
+ react-router-dom@5.2.0
added 11 packages from 6 contributors and audited 1981 package
s in 24.897s

114 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

You now have the package installed. For the remainder of this step, you'll create a series of components that will each have a unique route.

To start, make a directory for three different mammals: manatees, narwhals, and whales. Run the following commands:

```
mkdir src/components/Manatee
mkdir src/components/Narwhal
mkdir src/components/Whale
```

Next, create a component for each animal. Add an `<h2>` tag for each mammal. In a full application, the child components can be as complex as you want. They can even import and render their own child components. For this tutorial, you'll render only the `<h2>` tag.

Begin with the manatee component. Open `Manatee.js` in your text editor:

```
nano src/components/Manatee/Manatee.js
```

Then add the basic component:

router-tutorial/src/components/Manatee/Manatee.js

```
import React from 'react';


export default function Manatee() {
  return <h2>Manatee</h2>;
}
```

Save and close the file.

Next, create a component for the narwhal:

```
nano src/components/Narwhal/Narwhal.js
```

Add the same basic component, changing the `<h2>` to `Narwhal`:

router-tutorial/src/components/Narwhal/Narwhal.js

```
import React from 'react';

export default function Narwhal() {
  return <h2>Narwhal</h2>;
}
```

Save and close the file.

Finally, create a file for `Whale`:

```
nano src/components/Whale/Whale.js
```

Add the same basic component, changing the `<h2>` to `Whale`:

router-tutorial/src/components/Whale/Whale.js

```
import React from 'react';

export default function Whale() {
  return <h2>Whale</h2>;
}
```

Save and close the file. In the next step, you'll start connecting routes; for now, render the basic component in your application.

Open `App.js`:

```
nano src/components/App/App.js
```

Add an `<h1>` tag with the name of the website (`Marine Mammals`) inside of a `<div>` with a `className` of `wrapper`. This will serve as a template. The wrapper and `<h1>` tag will render on every page. In full applications, you might add a navigation bar or a header component that you'd want on every page.

Add the following highlighted lines to the file:

router-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';
function App() {
  return (
    <div className="wrapper">
      <h1>Marine Mammals</h1>
    </div>
  );
}

export default App;
```

Next, import `Manatee` and render inside the `<div>`. This will serve as a placeholder until you add more routes:

router-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';

import Manatee from '../Manatee/Manatee';

function App() {
  return (
    <div className="wrapper">
      <h1>Marine Mammals</h1>
      <Manatee />
    </div>
  );
}

export default App;
```

Save and close the file.

Now that you have all of the components, add some padding to give the application a little space.

Open `App.css`:

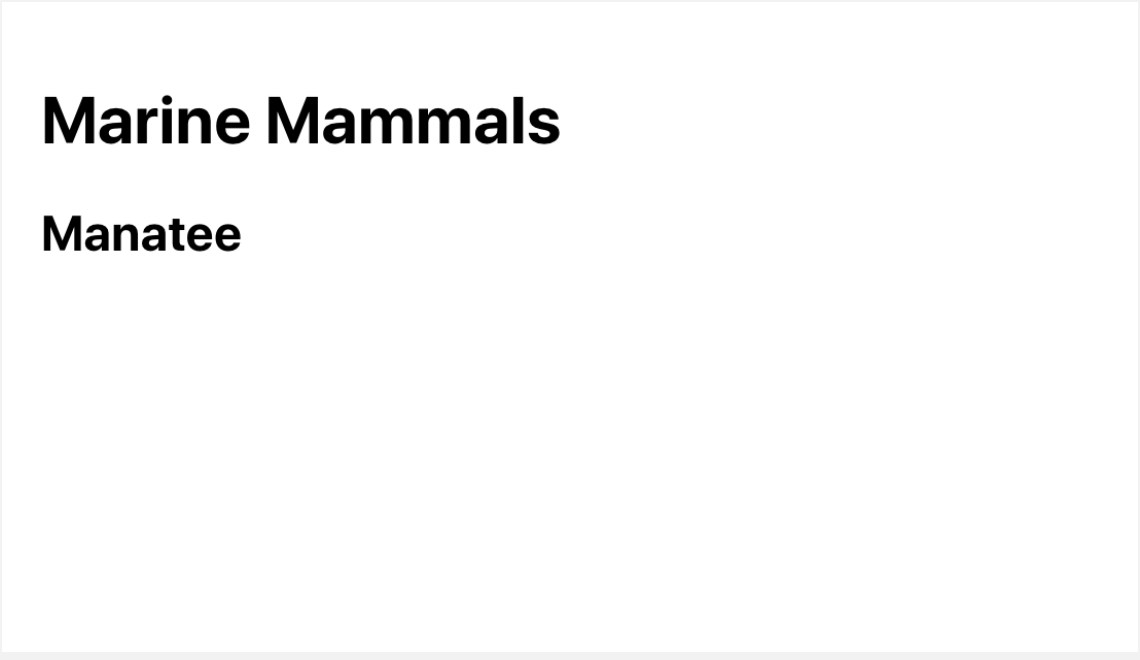
```
nano src/components/App/App.css
```

Then replace the contents with the following code that adds a `padding` of `20px` to the `.wrapper` class:

router-tutorial/src/components/App/App.css

```
.wrapper {
    padding: 20px;
}
```

Save and close the file. When you do, the browser will refresh to show your basic component:



Marine Mammals

Now you have a basic root component that you will use to display other components. If you didn't have a router, you could conditionally display components using the `useState` Hook. But this wouldn't offer a great experience for your users. Anytime a user refreshes the page, the user's selection would disappear. Further, they wouldn't be able to bookmark or share specific states of the application. A router will solve all these problems. The router will preserve the user state and will give the user a clear URL that they can save or send to others.

In this step, you installed React Router and created basic components. The components are going to be individual pages that you'll display by route. In the next step, you'll add routes and use the `<Link>` component to create performant hyperlinks.

Step 2 — Adding Routes

In this step, you'll create a base router with individual routes for each page. You'll order your routes to ensure that components are rendered correctly and you'll use the `<Link>` component to add hyperlinks to your project that won't trigger a page refresh.

By the end of this step, you'll have an application with a navigation that will display your components by route.

React Router is a declarative routing framework. That means that you will configure the routes using standard React components. There are a few advantages to this approach. First, it follows the standard declaractive nature of React code. You don't need to add a lot of code in `componentDidM`

`ount` methods or inside a `useEffect` Hook; your routes are components. Second, you can intuitively place routes inside of a component with other components serving as a template. As you read the code, you'll find exactly where the dynamic components will fit in relation to the global views such as navigation or footers.

To start adding routes, open `App.js`:

```
nano src/components/App/App.js
```

The `<h1>` tag is going to serve as a global page title. Since you want it to appear on every page, configure the router after the tag.

Import `BrowserRouter`, `Route`, and `Switch` from `react-router-dom`. `BrowserRouter` will be the base configuration. `Switch` will wrap the dynamic routes and the `Route` component will configure specific routes and wrap the component that should render:

router-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import './App.css';

import Manatee from '../Manatee/Manatee';

function App() {
  return (
    <div className="wrapper">
      <h1>Marine Mammals</h1>
      <Manatee />
    </div>
  );
}

export default App;
```

Add the `BrowserRouter` component to create a base router. Anything outside of this component will render on every page, so place it after your `<h1>` tag. In addition, if you have site-wide `context` that you want to use, or some other store such as Redux, place those components outside the router. This will make them available to all components on any route:

router-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import './App.css';

import Manatee from '../Manatee/Manatee';

function App() {
  return (
    <div className="wrapper">
      <h1>Marine Mammals</h1>
      <BrowserRouter>
        <Manatee />
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Next, add the `Switch` component inside `BrowserRouter`. This component will activate the correct route, much like the JavaScript `switch` statement. Inside of `Switch`, add a `Route` component for each route. In this case, you'll want the following routes: `/manataee`, `/narwhal`, and `/whale`. The `R`

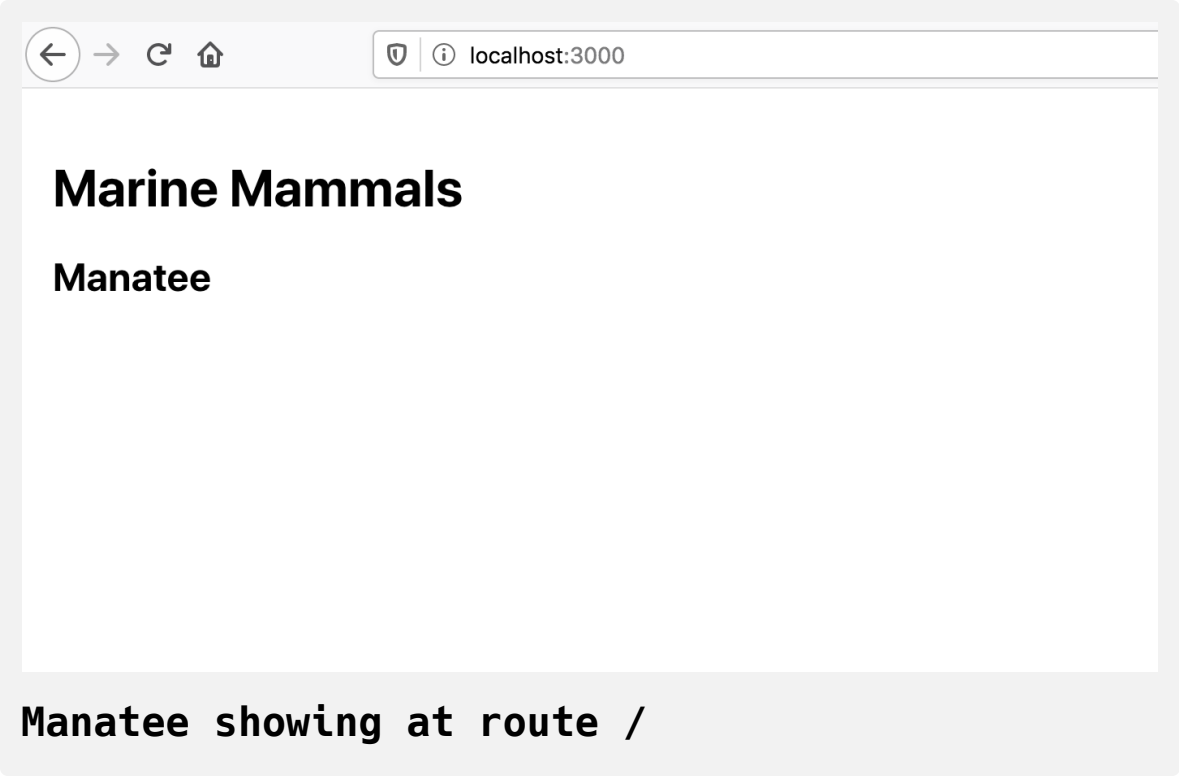
`oute` component will take a `path` as a parameter and surround a child component. The child component will display when the route is active.

Create a route for the path `/` and render the `Manatee` component:

router-tutorial/src/components/App/App.js

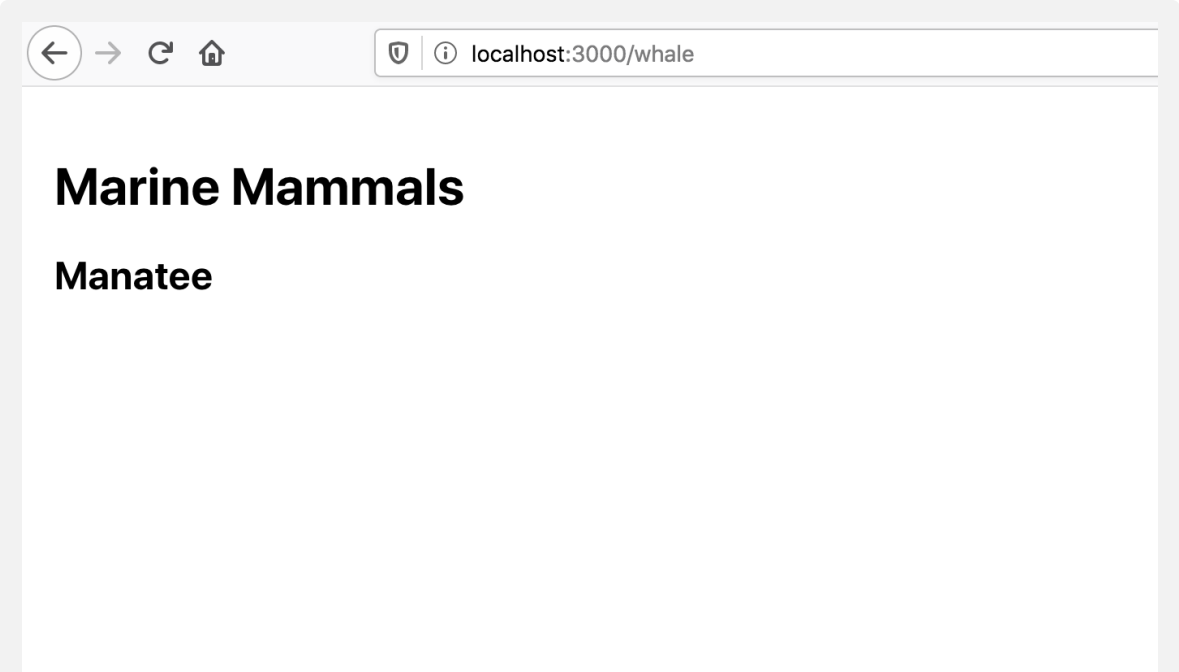
```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import './App.css';

import Manatee from '../Manatee/Manatee';

function App() {
  return (
    <div className="wrapper">
      <h1>Marine Mammals</h1>
      <BrowserRouter>
        <Switch>
          <Route path="/">
            <Manatee />
          </Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Save the file. When you do the browser will reload and you'll find the information for the manatee component:




```
Manatee showing at route /
```

If you try a different route such as `http://localhost:3000/whale`, you'll still find the manatee component.



Manatee on /whale route

The `Switch` component will render the first route that matches that pattern. Any route will match `/`, so it will render on every page. That also means that order is important. Since the router will exit as soon as it finds a match, always put a more specific route before a less specific route. In other words, `/whale` would go before `/` and `/whale/beluga` would go before `/whale`.

If you want the route to match only the route as written and not any child routes, you can add the `exact` prop. For example, `<Route exact path="/manatee">` would match `/manatee`, but not `/manatee/african`.

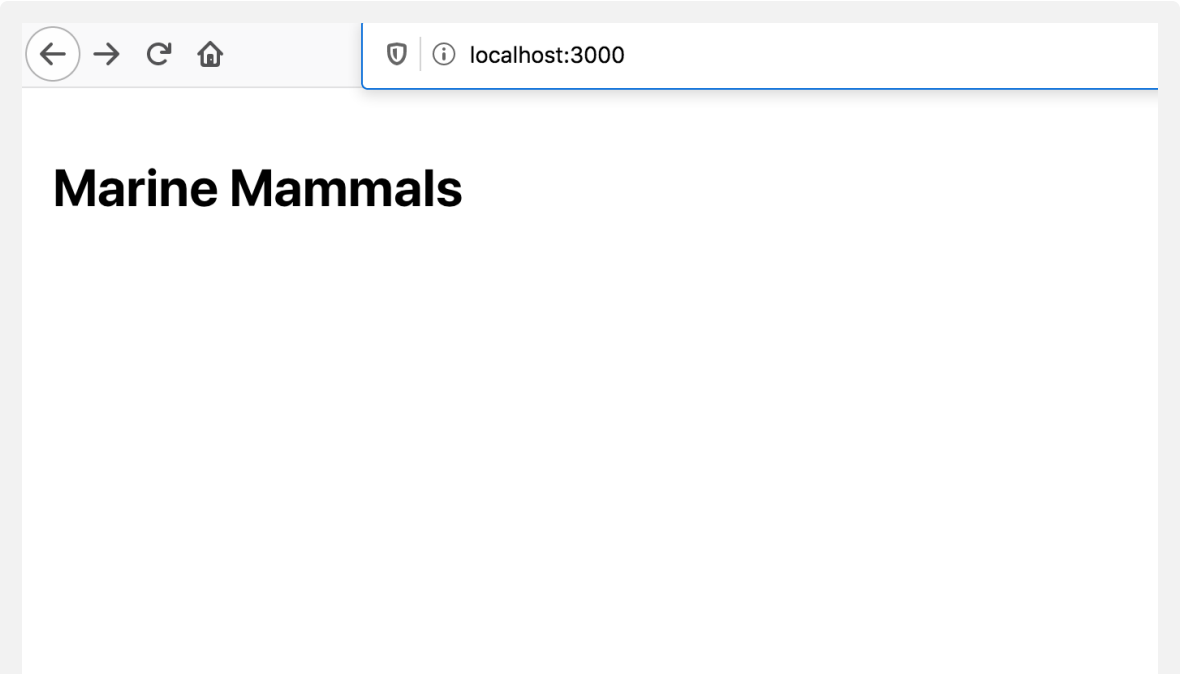
Update the route for the `Manatee` component to `/manatee`, then import the remaining components and create a route for each:

router-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import './App.css';

import Manatee from '../Manatee/Manatee';
import Narwhal from '../Narwhal/Narwhal';
import Whale from '../Whale/Whale';

function App() {
  return (
    <div className="wrapper">
      <h1>Marine Mammals</h1>
      <BrowserRouter>
        <Switch>
          <Route path="/manatee">
            <Manatee />
          </Route>
          <Route path="/narwhal">
            <Narwhal />
          </Route>
          <Route path="/whale">
            <Whale />
          </Route>
        </Switch>
      </BrowserRouter>
```

```
    </div>
  );
}

export default App;
```

Save the file. When you do, the browser will refresh. If you visit `http://localhost:3000/`, only the `<h1>` tag will render, because no routes match any of the `Route` components:



If you visit `http://localhost:3000/whale`, you'll find the `Whale` component:

Whale on /whale route

Now that you have some components, create navigation for a user to move between pages.

Use the `<nav>` element to denote that you are creating a navigation portion of the page. Then add an unordered list (`<ul>`) with a list item (`<li>`) and a hyperlink (`<a>`) for each mammal:

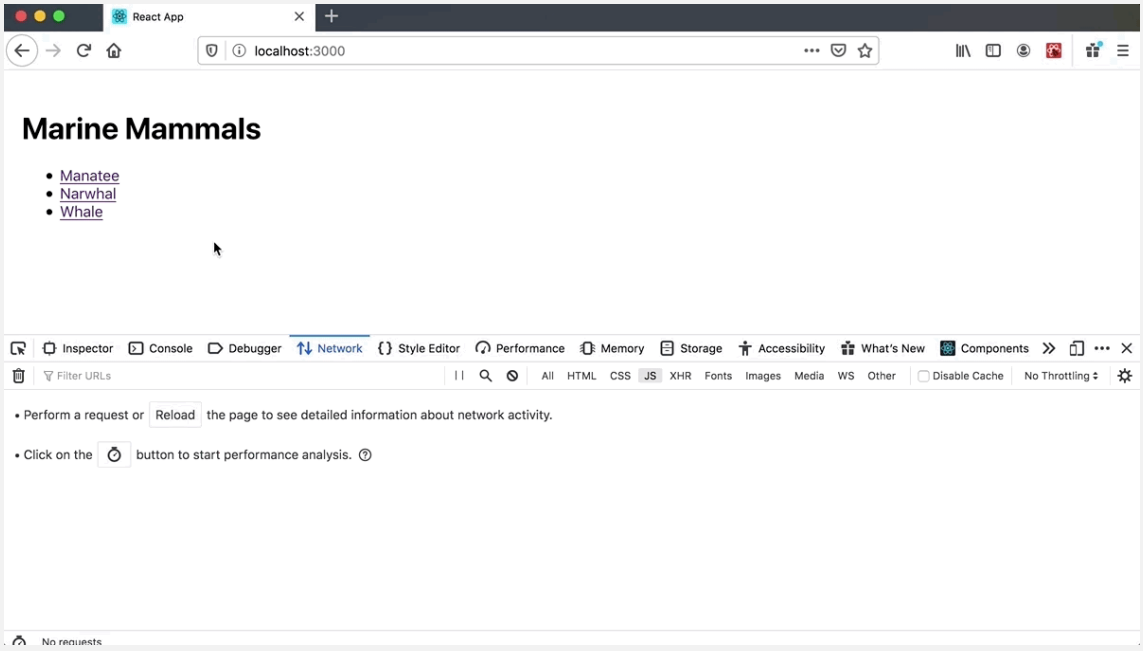
router-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import './App.css';

import Manatee from '../Manatee/Manatee';
import Narwhal from '../Narwhal/Narwhal';
import Whale from '../Whale/Whale';

function App() {
  return (
    <div className="wrapper">
      <h1>Marine Mammals</h1>
      <nav>
        <ul>
          <li><a href="/manatee">Manatee</a></li>
          <li><a href="/narwhal">Narwhal</a></li>
          <li><a href="/whale">Whale</a></li>
        </ul>
      </nav>
      <BrowserRouter>
        ...
      </BrowserRouter>
    </div>
  );
}
```

```
export default App;
```

Save the file. When you do, the browser will refresh, but there will be a problem. Since you are using the native browser links—`<a>` tags—you will get the default browser behavior any time you click on a link. That means any time you click on a link, you'll trigger a full page refresh.

Notice that the network will reload all of the JavaScript files when you click a link. That's a big performance cost for your users.



Browser refresh on link click

At this point, you could add a `click` event handler on each link and prevent the default action. That would be a lot of work. Instead, React Router has a special component called `Link` that will handle the work for you. It will create a link tag, but prevent the default brower behavior while pushing the new location.

In `App.js`, import `Link` from `react-router-dom`. Then replace each `<a>` with a `Link`. You'll also need to change the `href` attribute to the `to` prop.

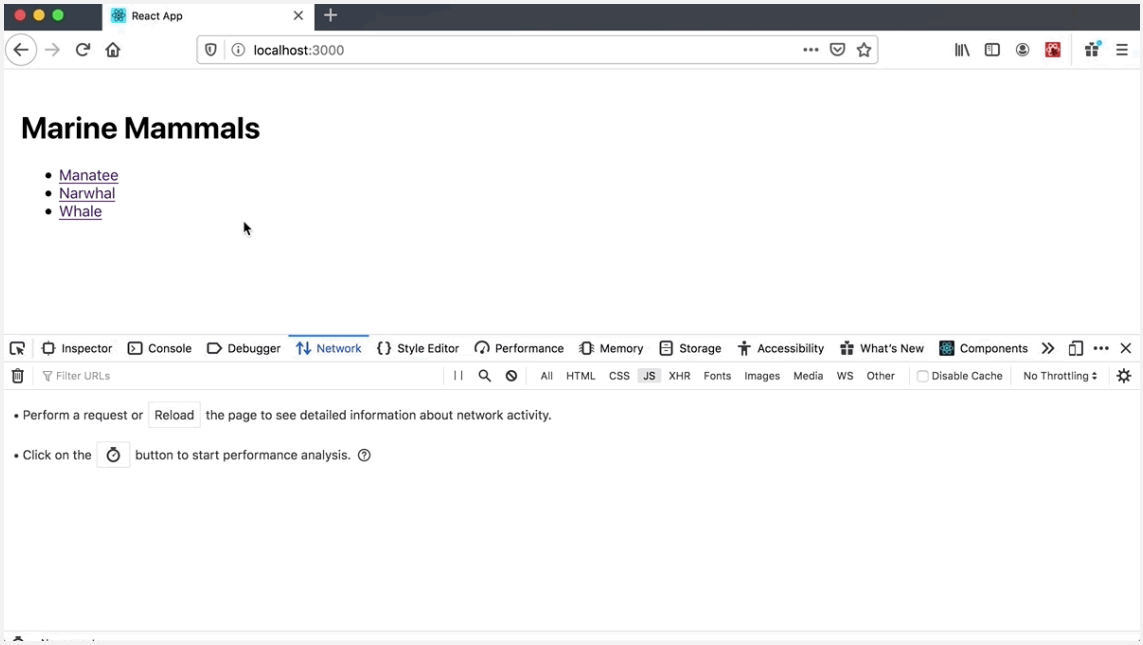
Finally, move the `<nav>` component inside of the `BrowserRouter`. This ensures that the `Link` component is controlled by `react-router`:

router-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Link, Route, Switch } from 'react-route
import './App.css';

import Manatee from '../Manatee/Manatee';
import Narwhal from '../Narwhal/Narwhal';
import Whale from '../Whale/Whale';

function App() {
  return (
    <div className="wrapper">
      <h1>Marine Mammals</h1>
      <BrowserRouter>
        <nav>
          <ul>
            <li><Link to="/manatee">Manatee</Link></li>
            <li><Link to="/narwhal">Narwhal</Link></li>
            <li><Link to="/whale">Whale</Link></li>
          </ul>
        </nav>
        <Switch>
          <Route path="/manatee">
            <Manatee />
          </Route>
```

```
          <Route path="/narwhal">
            <Narwhal />
          </Route>
          <Route path="/whale">
            <Whale />
          </Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Save the file. When you do, the browser will refresh. When you click links, the page will not refresh and the browser will not reload the JavaScript code:



No refresh on link click

In this step you added React Router to your current project. You created a route for each component and you added a navigation using the `Link` component to switch between routes without a page refresh.

In the next step, you'll add more complex routes that render different components using URL parameters.

Step 3 — Accessing Route Data with Hooks

In this step, you'll use URL queries and parameters to create dynamic routes. You'll learn how to pull information from search parameters with the `useLocation` Hook and how to read information from dynamic URLs using the `useParams` Hook.

By the end of this step, you'll know how to access route information inside of your components and how you can use that information to dynamically load components.

Suppose you wanted to add another level to your marine mammal application. There are many types of whales, and you could display information about each one. You have two choices of how to accomplish this: You could use the current route and add a specific whale type with search parameters, such as `?type=beluga`. You could also create a new route that includes the specific name after the base URL, such as `/whale/beluga`. This tutorial will start with search parameters, since they are flexible and can handle multiple, different queries.

First, make new components for different whale species.

Open a new file `Beluga.js` in your text editor:

```
nano src/components/Whale/Beluga.js
```

Add an `<h3>` tag with the name `Beluga`:

router-tutorial/src/components/Whale/Beluga.js

```
import React from 'react';


export default function Beluga( ) {
  return(
    <h3>Beluga</h3>
  );
}
```

Do the same thing for a blue whale. Open a new file `Blue.js` in your text editor:

```
nano src/components/Whale/Blue.js
```

Add an `<h3>` tag with the name `Blue`:

router-tutorial/src/components/Whale/Blue.js

```
import React from 'react';


export default function Blue() {
  return(
    <h3>Blue</h3>
  );
}
```

Save and close the file.

Passing Additional Information with Search Parameters

Next, you are going to pass the whale information as a search parameter. This will let you pass information without needing to create a new URL.

Open `App.js` so you can add new links:

```
nano src/components/App/App.js
```

Add two new links, one to `/whale?type=beluga` and one for `/whale?type=blue`:

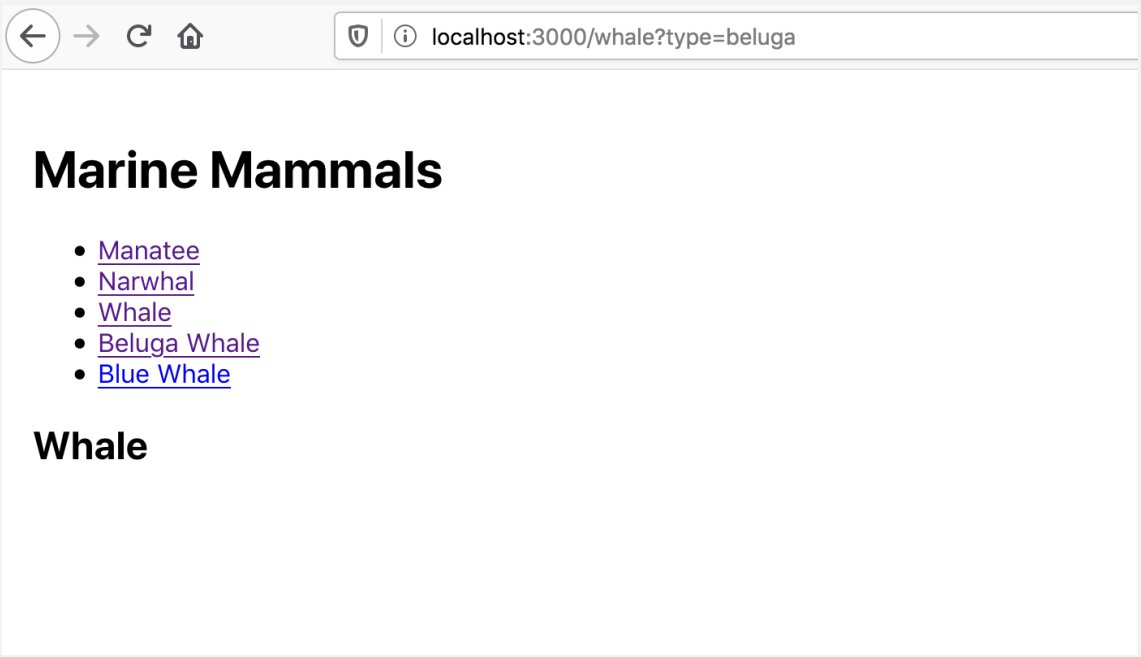
router-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Link, Route, Switch } from 'react-route
import './App.css';

import Manatee from '../Manatee/Manatee';
import Narwhal from '../Narwhal/Narwhal';
import Whale from '../Whale/Whale';

function App() {
  return (
    <div className="wrapper">
      <h1>Marine Mammals</h1>
      <BrowserRouter>
        <nav>
          <ul>
            <li><Link to="/manatee">Manatee</Link></li>
            <li><Link to="/narwhal">Narwhal</Link></li>
            <li><Link to="/whale">Whale</Link></li>
            <li><Link to="/whale?type=beluga">Beluga Whale</Lin
            <li><Link to="/whale?type=blue">Blue Whale</Link></
          </ul>
        </nav>
        <Switch>
          ...
        </Switch>
```

```
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Save and close the file.

If you click on the links, you'll still see the regular whale page. This shows that the standard route is still working correctly:



Beluga router with whale page

Since you are correctly rendering the `Whale` component, you'll need to update the component to pull the search query out of the URL and use it to render the correct child component.

Open `Whale.js`:

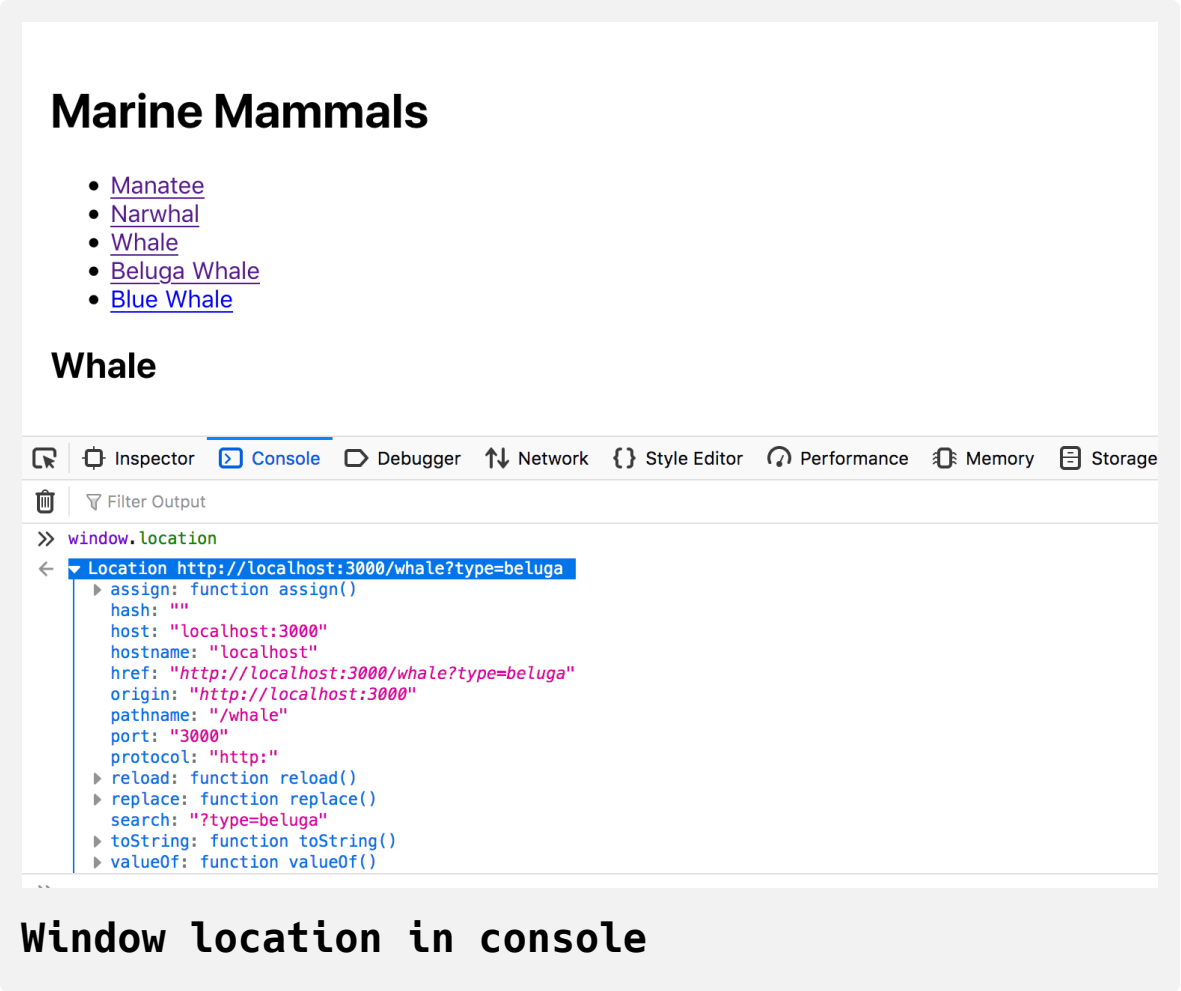
```
nano src/components/Whale/Whale.js
```

First, import the `Beluga` and `Blue` components. Next, import a Hook called `useLocation` from `react-router-dom`:

router-tutorial/src/components/Whale/Whale.js

```
import React from 'react';
import { useLocation } from 'react-router-dom';
import Beluga from './Beluga';
import Blue from './Blue';

export default function Whale() {
  return <h2>Whale</h2>;
}
```

The `useLocation` Hook pulls the location information from your page. This is not unique to React Router. The `location` object is a standard object on all browsers. If you open your browser console and type `window.location`, you'll get an object with information about your URL.



Window location in console

Notice that the location information includes `search`, but also includes other information, such as the `pathname` and the full `href`. The `useLocation` Hook will provide this information for you. Inside of `Whale.js`, call the `useLocation` Hook. Destructure the result to pull out the `search` field. This will be a parameter string, such as `?type=beluga`:

router-tutorial/src/components/Whale/Whale.js

```
import React from 'react';
import { useLocation } from 'react-router-dom';
import Beluga from './Beluga';
import Blue from './Blue';

export default function Whale() {
  const { search } = useLocation();
  return <h2>Whale</h2>;
}
```

There are a number of libraries, such as `query-string`, that can parse the search for you and convert it into an object that is easier to read and update. In this example, you can use a regular expression to pull out the information about the whale `type`.

Use the `.match` method on the search string to pull out the `type`: `search.match(/type=(.*)/)`. The parentheses inside the regular expression will capture the match into a results array. The first item in the array is the full match: `type=beluga`. The second item is the information from the parentheses: `beluga`.

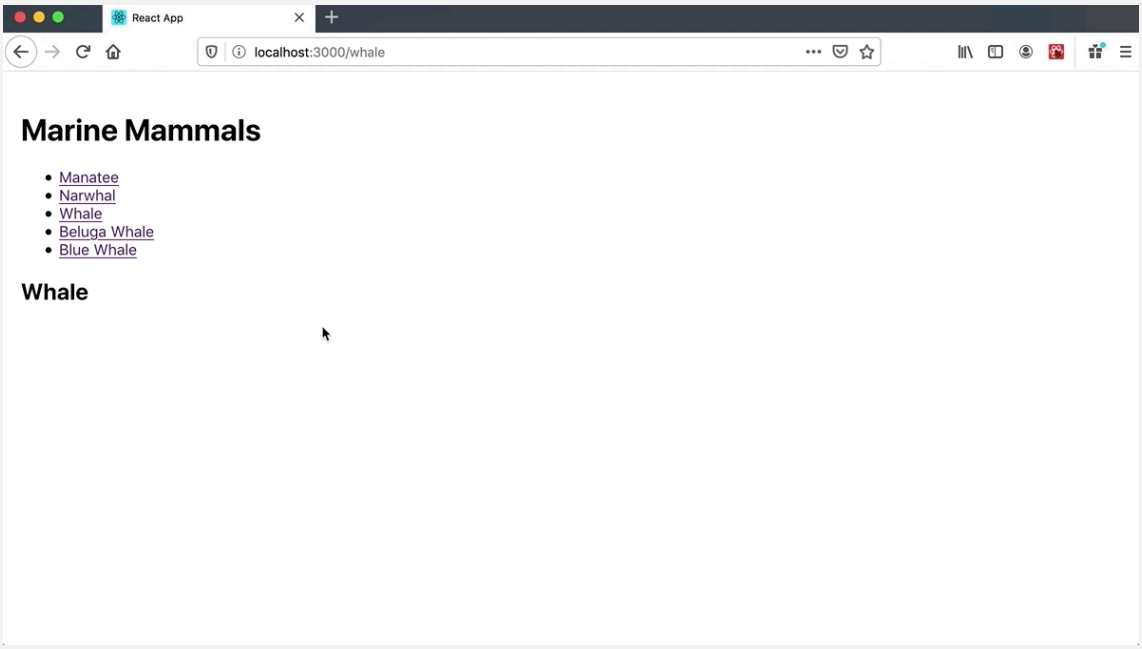
Use the data from the `.match` method to render the correct child component:

router-tutorial/src/components/Whale/Whale.js

```
import React from 'react';
import { useLocation } from 'react-router-dom';
import Beluga from './Beluga';
import Blue from './Blue';

export default function Whale() {
  const { search } = useLocation();
  const match = search.match(/type=(.*)/);
  const type = match?.[1];

  return (
    <>
      <h2>Whale</h2>
      {type === 'beluga' && <Beluga />}
      {type === 'blue' && <Blue />}
    </>
  );
}
```

The symbol `?.` is called optional chaining. If the value exists, it returns the value. Otherwise, it will return `undefined`. This will protect your component in instances where the search parameter is empty.

Save the file. When you do, the browser will refresh and will render different whales:



Different whales with search params

Accessing URL Parameters

Search parameters work, but they aren't the best solution in this case. Generally, you'd use search parameters to refine a page: toggling information or loading specific data. In this case, you are not refining a page; you are creating a new static page. Fortunately, React Router provides a way to create dynamic URLs that preserve variable data called URL Parameters.

Open `App.js`:

```
nano src/components/App/App.js
```

Instead of passing the whale information as a search, you will add it directly to the URL itself. That means that you will move the seach into the URL instead of adding it after a `?`. For example, the query `/whale?type=blue` will now be `/whale/blue`:

router-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Link, Route, Switch } from 'react-route
import './App.css';

import Manatee from '../Manatee/Manatee';
import Narwhal from '../Narwhal/Narwhal';
import Whale from '../Whale/Whale';

function App() {
  return (
    <div className="wrapper">
      <h1>Marine Mammals</h1>
      <BrowserRouter>
        <nav>
          <ul>
            <li><Link to="/manatee">Manatee</Link></li>
            <li><Link to="/narwhal">Narwhal</Link></li>
            <li><Link to="/whale">Whale</Link></li>
            <li><Link to="/whale/beluga">Beluga Whale</Link></l
            <li><Link to="/whale/blue">Blue Whale</Link></li>
          </ul>
        </nav>
        <Switch>
          <Route path="/manatee">
            <Manatee />
```

```
          </Route>
          <Route path="/narwhal">
            <Narwhal />
          </Route>
          <Route path="/whale">
            <Whale />
          </Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Now you need to create a new route that can capture both `/whale/beluga` and `/whale/blue`. You could add them by hand, but this wouldn't work in situations where you don't know all the possibilities ahead of time, such as when you have a list of users or other dynamic data.

Instead of making a route for each one, add a URL param to the current path. The URL param is a keyword prefaced with a colon. React Router will use the parameter as a wildcard and will match any route that contains that pattern.

In this case, create a keyword of `:type`. The full `path` will be `/whale/:type`. This will match any route that starts with `/whale` and it will save the variable information inside a parameter variable called `type`. This route will not match `/whale`, since that does not contain an additional parameter.

You can either add `/whale` as a route after the new route or you can add it before the route of `/whale/:type` with the `exact` keyword.

Add a new route of `/whale/:type` and add an `exact` property to the current route:

router-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Link, Route, Switch } from 'react-route
import './App.css';

import Manatee from '../Manatee/Manatee';
import Narwhal from '../Narwhal/Narwhal';
import Whale from '../Whale/Whale';

function App() {
  return (
    <div className="wrapper">
      <h1>Marine Mammals</h1>
      <BrowserRouter>
        <nav>
          <ul>
            <li><Link to="/manatee">Manatee</Link></li>
            <li><Link to="/narwhal">Narwhal</Link></li>
            <li><Link to="/whale">Whale</Link></li>
            <li><Link to="/whale/beluga">Beluga Whale</Link></l
            <li><Link to="/whale/blue">Blue Whale</Link></li>
          </ul>
        </nav>
        <Switch>
          <Route path="/manatee">
            <Manatee />
```

```
          </Route>
          <Route path="/narwhal">
            <Narwhal />
          </Route>
          <Route exact path="/whale">
            <Whale />
          </Route>
          <Route path="/whale/:type">
            <Whale />
          </Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Save and close the file. Now that you are passing new information, you need to access it and use the information to render dynamic components.

Open `Whale.js`:

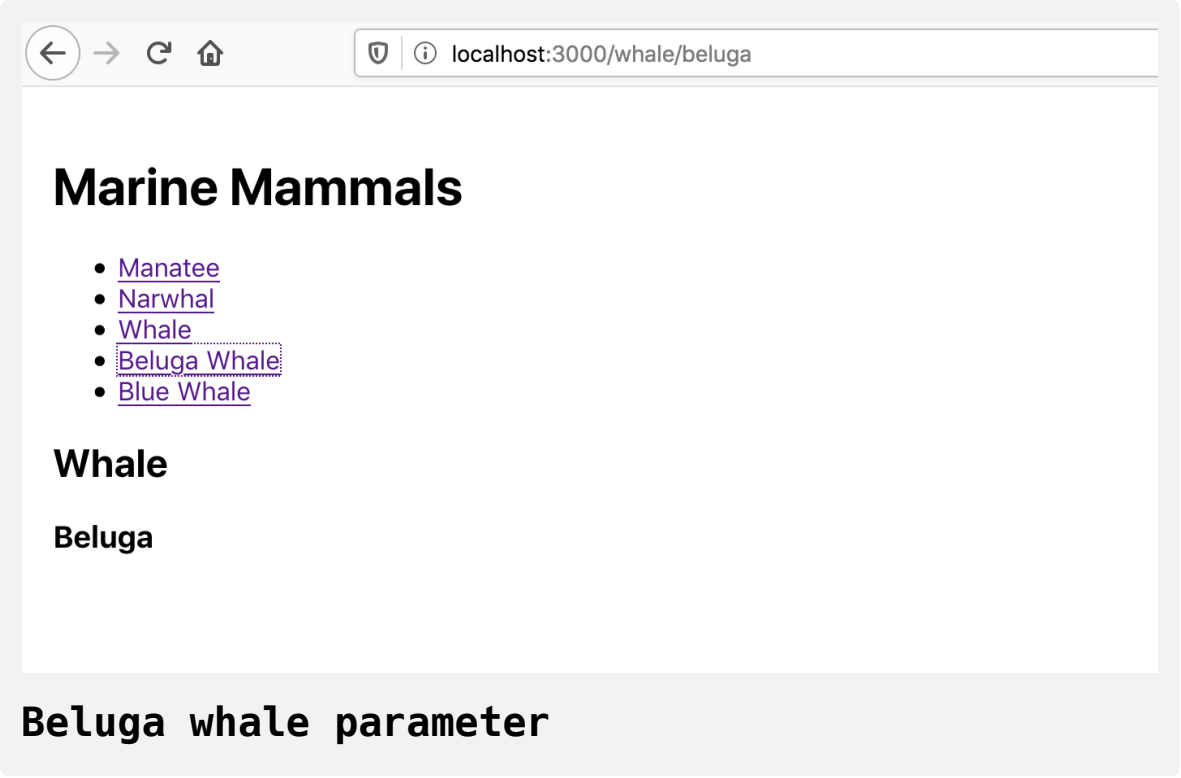
```
nano src/components/Whale/Whale.js
```

Import the `useParams` Hook. This will connect to your router and pull out any URL parameters into an object. Destructure the object to pull out the `type` field. Remove the code for parsing the `search` and use the parameter to conditionally render child components:

router-tutorial/src/components/Whale/Whale.js

```
import React from 'react';
import { useParams } from 'react-router-dom';
import Beluga from './Beluga';
import Blue from './Blue';

export default function Whale() {
  const { type } = useParams();

  return (
    <>
      <h2>Whale</h2>
      {type === 'beluga' && <Beluga />}
      {type === 'blue' && <Blue />}
    </>
  );
}
```

Save and close the file. When you do, the browser will refresh and you'll be able to use the new URLs, such as `http://localhost:3000/whale/beluga`:



Beluga whale parameter

URL parameters are a clear way to pass conditional data. They are not as flexible as search parameters, which can be combined or reordered, but they are more clear and easier for search engines to index.

In this step you passed variable data using search parameters and URL parameters. You also used the `useLocation` and `useParams` Hooks to pull information out and to render conditional components.

But there is one problem: The list of routes is getting long and you are starting to get near duplicates with the `/whale` and `/whale/:type` routes. React Router lets you split out child routes directly in the component, which

means you don’t need to have the whole list in a single component. In the next step, you’ll render routes directly inside of child components.

Step 4 — Nesting Routes

Routes can grow and become more complex. React Router uses nested routes to render more specific routing information inside of child components. In this step, you’ll use nested routes and add routes in different components. By the end of this step, you’ll have different options for rendering your information.

In the last step, you added routes inside of `App.js`. This has some advantages: It keeps all routes in one place, essentially creating a site map for your application. But it can easily grow and be difficult to read and maintain. Nested routes group your routing information directly in the components that will render other components, giving you the ability to create mini-templates throughout your application.

Open `App.js`:

```
nano src/components/App/App.js
```

Remove the `/whale/:type` route and remove the `exact` prop so you only have a whale route:

router-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Link, Route, Switch } from 'react-route
import './App.css';

import Manatee from '../Manatee/Manatee';
import Narwhal from '../Narwhal/Narwhal';
import Whale from '../Whale/Whale';

function App() {
  return (
    <div className="wrapper">
      <h1>Marine Mammals</h1>
      <BrowserRouter>
        <nav>
          <ul>
            <li><Link to="/manatee">Manatee</Link></li>
            <li><Link to="/narwhal">Narwhal</Link></li>
            <li><Link to="/whale">Whale</Link></li>
            <li><Link to="/whale/beluga">Beluga Whale</Link></l
            <li><Link to="/whale/blue">Blue Whale</Link></li>
          </ul>
        </nav>
        <Switch>
          <Route path="/manatee">
            <Manatee />
```

```
          </Route>
          <Route path="/narwhal">
            <Narwhal />
          </Route>
          <Route path="/whale">
            <Whale />
          </Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Save and close the file.

Next, open `Whale.js`. This is where you will add the nested route.

```
nano src/components/Whale/Whale.js
```

You will need to do two things. First, get the current path with the `useRouteMatch` Hook. Next, render the new `<Switch>` and `<Route>` components to display the correct components.

Import `useRouteMatch`. This will return an object that contains the `path` and the `url`. Destructure the object to get the `path`. You'll use this as the basis for your new routes:

router-tutorial/src/components/Whale/Whale.js

```
import React from 'react';
import { useRouteMatch } from 'react-router-dom';
import Beluga from './Beluga';
import Blue from './Blue';

export default function Whale() {
  const { path } = useRouteMatch();

  return (
    <>
      <h2>Whale</h2>
      {type === 'beluga' && <Beluga />}
      {type === 'blue' && <Blue />}
    </>
  );
}
```

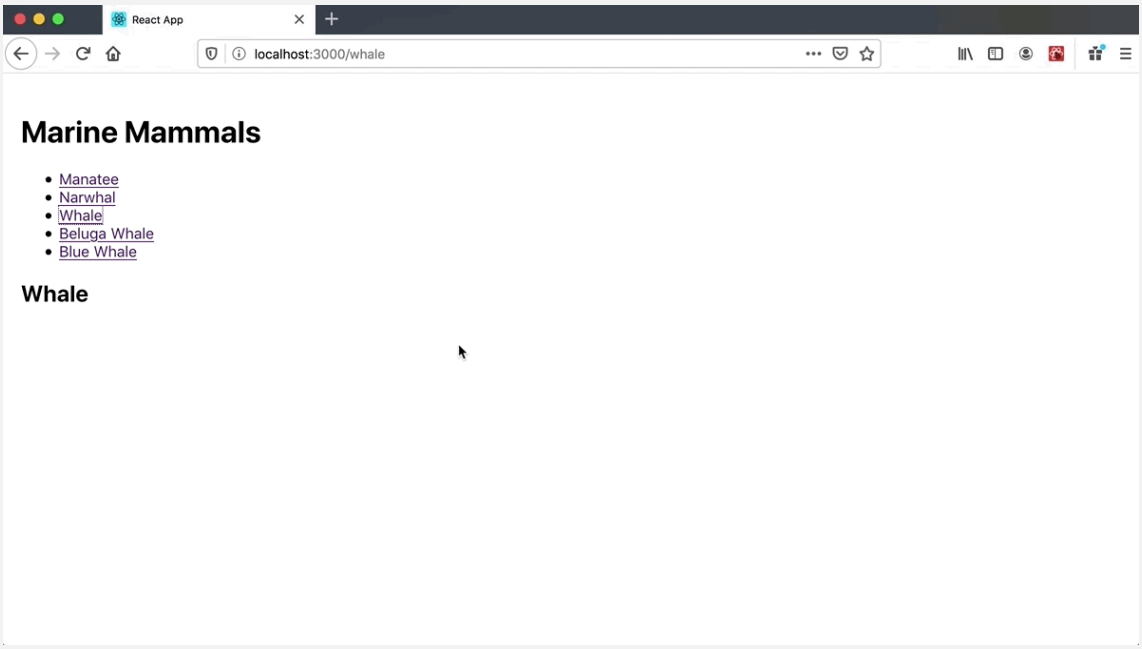
Next, import `Switch` and `Route` so you can add in new routes. Your new routes will be the same as you made in `App.js`, but you do not need to wrap

them with `BrowserRouter`. Add the new routes, but prefix the route with the `path`. The new component will render exactly where you place them, so add the new routes after the `<h2>`:

router-tutorial/src/components/Whale/Whale.js

```
import React from 'react';
import { Switch, Route, useRouteMatch } from 'react-router-dom'
import Beluga from './Beluga';
import Blue from './Blue';

export default function Whale() {
  const { path } = useRouteMatch();
  return (
    <>
      <h2>Whale</h2>
      <Switch>
        <Route path={`${path}/beluga`}>
          <Beluga />
        </Route>
        <Route path={`${path}/blue`}>
          <Blue />
        </Route>
      </Switch>
    </>
  );
}
```

Save the file. When you do, the browser will refresh and you'll be able to visit the child routes.



Visiting child routes

This is a little extra code, but it keeps the child routes situated with their parent. Not all projects use nested routes: some prefer having an explicit list. It is a matter of team preference and consistency. Choose the option that is best for your project, and you can always refactor later.

In this step, you added nested routes to your project. You pulled out the current path with the `useRouteMatch` Hook and added new routes in a component to render the new components inside of a base component.

Conclusion

React Router is an important part of any React project. When you build single page applications, you'll use routes to separate your application into usable pieces that users can access easily and consistently.

As you start to separate your components into routes, you'll be able to take advantage of code splitting, preserving state via query parameters, and other tools to improve the user experience.

If you would like to read more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Add Login Authentication to React Applications

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

Many web applications are a mix of public and private pages. Public pages are available to anyone, while a private page requires a user login. You can use authentication to manage which users have access to which pages. Your React application will need to handle situations where a user tries to access a private page before they are logged in, and you will need to save the login information once they have successfully authenticated.

In this tutorial, you'll create a React application using a token-based authentication system. You'll create a mock API that will return a user token, build a login page that will fetch the token, and check for authentication without rerouting a user. If a user is not authenticated, you'll provide an opportunity for them to log in and then allow them to continue without navigating to a dedicated login page. As you build the application, you'll explore different methods for storing tokens and will learn the security and experience trade-offs for each approach. This tutorial will focus on storing tokens in `localStorage` and `sessionStorage`.

By the end of this tutorial, you'll be able to add authentication to a React application and integrate the login and token storage strategies into a complete user workflow.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.22.0 and npm version 6.14.6. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- A React development environment set up with Create React App, with the non-essential boilerplate removed. To set this up, follow **Step 1 — Creating an Empty Project** of the How To Manage State on React Class Components tutorial. This tutorial will use `auth-tutorial` as the project name.

- You will be fetching data from APIs using React. You can learn about working with APIs in How To Call Web APIs with the useEffect Hook in React.

- You will also need a basic knowledge of JavaScript, HTML, and CSS, which you can find in our How To Build a Website With HTML series, How To Style HTML with CSS, and in How To Code in JavaScript.

Step 1 — Building a Login Page

In this step, you'll create a login page for your application. You'll start by installing React Router and creating components to represent a full application. Then you'll render the login page on any route so that your users can login to the application without being redirected to a new page.

By the end of this step, you'll have a basic application that will render a login page when a user is not logged into the application.

To begin, install react router with `npm`. There are two different versions: a web version and a native version for use with React Native. Install the web version:

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

The package will install and you'll receive a message when the installation is complete. Your message may vary slightly:

Output

```
...
+ react-router-dom@5.2.0
added 11 packages from 6 contributors, removed 10 packages and
audited 1945 packages in 12.794s
...
```

Next, create two components called `Dashboard` and `Preferences` to act as private pages. These will represent components that a user should not see until they have successfully logged into the application.

First, create the directories:

```
mkdir src/components/Dashboard
mkdir src/components/Preferences
```

Then open `Dashboard.js` in a text editor. This tutorial will use nano:

```
nano src/components/Dashboard/Dashboard.js
```

Inside of `Dashboard.js`, add an `<h2>` tag with the content of `Dashboard`:

auth-tutorial/src/components/Dashboard/Dashboard.js

```
import React from 'react';

export default function Dashboard() {
  return(
    <h2>Dashboard</h2>
  );
}
```

Save and close the file.

Repeat the same steps for `Preferences`. Open the component:

```
nano src/components/Preferences/Preferences.js
```

Add the content:

auth-tutorial/src/components/Preferences/Preferences.js

```
import React from 'react';


export default function Preferences() {
  return(
    <h2>Preferences</h2>
  );
}
```

Save and close the file.

Now that you have some components, you need to import the components and create routes inside of `App.js`. Check out the tutorial How To Handle Routing in React Apps with React Router for a full introduction to routing in React applications.

To begin, open `App.js`:

```
nano src/components/App/App.js
```

Then import `Dashboard` and `Preferences` by adding the following highlighted code:

auth-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Preferences from '../Preferences/Preferences';

function App() {
  return (
    <></>
  );
}

export default App;
```

Next, import `BrowserRouter`, `Switch`, and `Route` from `react-router-dom`:

auth-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import Dashboard from '../Dashboard/Dashboard';
import Preferences from '../Preferences/Preferences';

function App() {
  return (
    <></>
  );
}

export default App;
```

Add a surrounding `<div>` with a `className` of `wrapper` and an `<h1>` tag to serve as a template for the application. Be sure that you are importing `App.css` so that you can apply the styles.

Next, create routes for the `Dashboard` and `Preferences` components. Add `BrowserRouter`, then add a `Switch` component as a child. Inside of the `Switch`, add a `Route` with a `path` for each component:

tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import Dashboard from '../Dashboard/Dashboard';
import Preferences from '../Preferences/Preferences';

function App() {
  return (
    <div className="wrapper">
      <h1>Application</h1>
      <BrowserRouter>
        <Switch>
          <Route path="/dashboard">
            <Dashboard />
          </Route>
          <Route path="/preferences">
            <Preferences />
          </Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Save and close the file.

The final step is to add some padding to the main `<div>` so your component is not directly at the edge of the browser. To do this, you will change the CSS.

Open `App.css`:

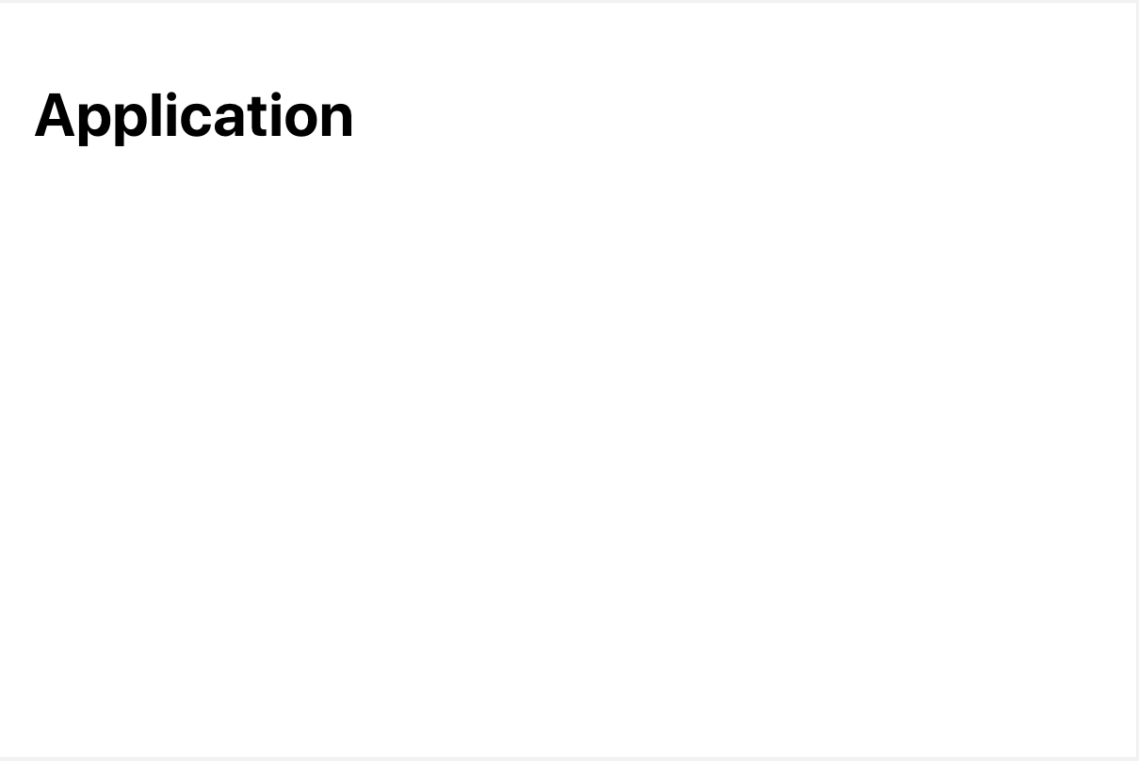
```
nano src/components/App/App.css
```

Replace the contents with a class of `.wrapper` with `padding` of `20px`:

auth-tutorial/src/components/App/App.css

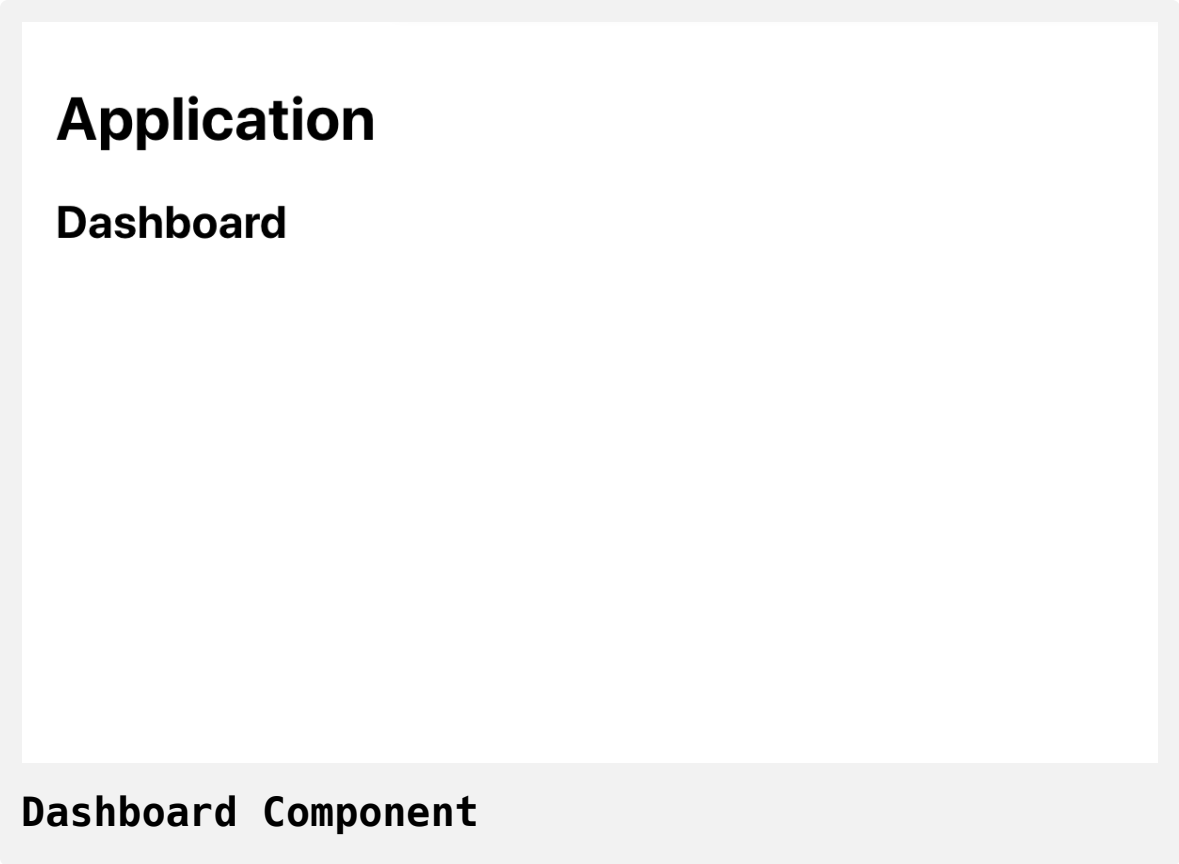
```
.wrapper {
    padding: 20px;
}
```

Save and close the file. When you do, the browser will reload and you'll find your basic components:



Basic component

Check each of the routes. If you visit `http://localhost:3000/dashboard`, you'll find the dashboard page:



Dashboard Component

Your routes are working as expected, but there is a slight problem. The route `/dashboard` should be a protected page and should not be viewable by an unauthenticated user. There are different ways to handle a private page. For example, you can create a new route for a login page and use React Router to redirect if the user is not logged in. This is a fine approach, but the user would lose their route and have to navigate back to the page they originally wanted to view.

A less intrusive option is to generate the login page regardless of the route. With this approach, you'll render a login page if there is not a stored user token and when the user logs in, they'll be on the same route that they

initially visited. That means if a user visits `/dashboard`, they will still be on the `/dashboard` route after login.

To begin, make a new directory for the `Login` component:

```
mkdir src/components/Login
```

Next, open `Login.js` in a text editor:

```
nano src/components/Login/Login.js
```

Create a basic form with a submit `<button>` and an `<input>` for the username and the password. Be sure to set the input type for the password to `password`:

auth-tutorial/src/components/Login/Login.js

```
import React from 'react';


export default function Login() {
  return(
    <form>
      <label>
        <p>Username</p>
        <input type="text" />
      </label>
      <label>
        <p>Password</p>
        <input type="password" />
      </label>
      <div>
        <button type="submit">Submit</button>
      </div>
    </form>
  )
}
```

For more on forms in React, check out the tutorial How To Build Forms in React.

Next, add an `<h1>` tag asking the user to log in. Wrap the `<form>` and the `<h1>` in a `<div>` with a `className` of `login-wrapper`. Finally, import `Login.css`:

auth-tutorial/src/components/Login/Login.js

```
import React from 'react';
import './Login.css';

export default function Login() {
  return(
    <div className="login-wrapper">
      <h1>Please Log In</h1>
      <form>
        <label>
          <p>Username</p>
          <input type="text" />
        </label>
        <label>
          <p>Password</p>
          <input type="password" />
        </label>
        <div>
          <button type="submit">Submit</button>
        </div>
      </form>
    </div>
  )
}
```

Save and close the file.

Now that you have a basic `Login` component, you'll need to add some styling. Open `Login.css`:

```
nano src/components/Login/Login.css
```

Center the component on the page by adding a `display` of `flex`, then setting the `flex-direction` to `column` to align the elements vertically and adding `align-items` to `center` to make the component centered in the browser:

auth-tutorial/src/components/Login/Login.css

```
.login-wrapper {
    display: flex;
    flex-direction: column;
    align-items: center;
}
```

For more information on using Flexbox, see our CSS Flexbox Cheatsheet

Save and close the file.

Finally, you'll need to render it inside of `App.js` if there is no user token. Open `App.js`:

```
nano src/components/App/App.js
```

In **Step 3**, you'll explore options for storing the token. For now, you can store the token in memory using the `useState` Hook.

Import `useState` from `react`, then call `useState` and set return values to `token` and `setToken`:

auth-tutorial/src/components/App/App.js

```
import React, { useState } from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Preferences from '../Preferences/Preferences';

function App() {
  const [token, setToken] = useState();
  return (
    <div className="wrapper">
      <h1>Application</h1>
      <BrowserRouter>
        <Switch>
          <Route path="/dashboard">
            <Dashboard />
          </Route>
          <Route path="/preferences">
            <Preferences />
          </Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}
```

```
export default App;
```

Import the `Login` component. Add a conditional statement to display `Login` if the `token` is falsy.

Pass the `setToken` function to the `Login` component:

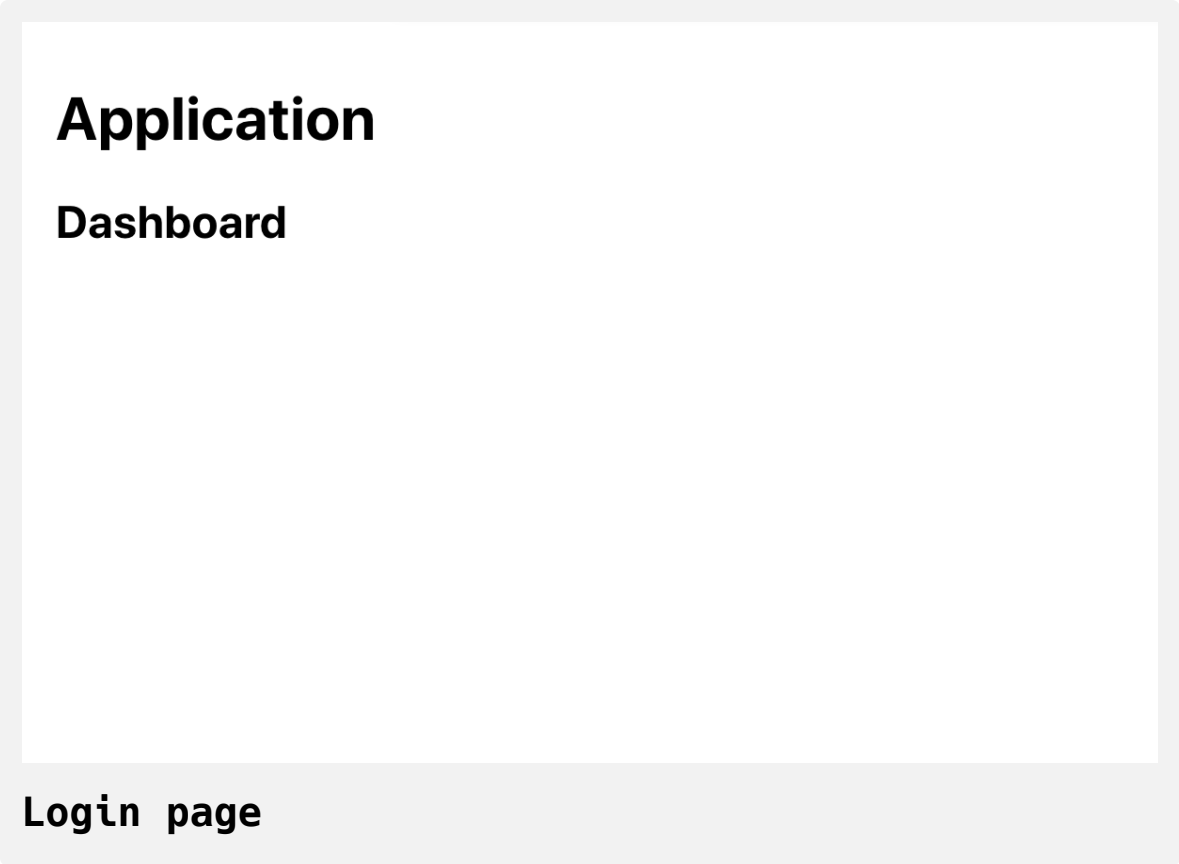
auth-tutorial/src/components/App/App.js

```
import React, { useState } from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom'

import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Login from '../Login/Login';
import Preferences from '../Preferences/Preferences';

function App() {
  const [token, setToken] = useState();

  if(!token) {
    return <Login setToken={setToken} />
  }

  return (
    <div className="wrapper">
      <h1>Application</h1>
      <BrowserRouter>
        <Switch>
          <Route path="/dashboard">
            <Dashboard />
          </Route>
          <Route path="/preferences">
            <Preferences />
```

```
          </Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}

export default App;
```

For now, there is no token; in the next step, you'll call an API and set the token with the return value.

Save and close the file. When you do, the browser will reload and you'll see the login page. Notice that if you visit `http://localhost:3000/dashboard`, you'll still find the login page since the token has not yet been set:



In this step, you created an application with private components and a login component that will display until you set a token. You also configured routes to display the pages and added a check to display the `Login` component on every route if the user is not yet logged into the application.

In the next step, you'll create a local API that will return a user token. You'll call the API from the `Login` component and save the token to memory on success.

Step 2 — Creating a Token API

In this step, you'll create a local API to fetch a user token. You'll build a mock API using Node.js that will return a user token. You'll then call that

API from your login page and render the component after you successfully retrieve the token. By the end of this step, you'll have an application with a working login page and protected pages that will only be accessible after login.

You are going to need a server to act as a backend that will return the token. You can create a server quickly using Node.js and the Express web framework. For a detailed introduction to creating an Express server, see the tutorial Basic Express Server in Node.js.

To start, install `express`. Since the server is not a requirement of the final build, be sure to install as a `devDependency`.

You'll also need to install `cors`. This library will enable cross origin resource sharing for all routes.

Warning: Do not enable CORS for all routes in a production application. This can lead to security vulnerabilities.

```
npm install --save-dev express cors
```

When the installation is complete, you'll receive a success message:

Output

```
...
+ cors@2.8.5
+ express@4.17.1
removed 10 packages, updated 2 packages and audited 2059 packa
ges in 12.597s
...
```

Next, open a new file called `server.js` in the root of your application. Do not add this file to the `/src` directory since you do not want it to be part of the final build.

```
nano server.js
```

Import `express`, then initialize a new app by calling `express()` and saving the result to a variable called `app`:

auth-tutorial/server.js

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

After creating the `app`, add `cors` as a middleware. First, import `cors`, then add it to the application by calling the `use` method on `app`:

auth-tutorial/server.js

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


app.use(cors());
```

Next, listen to a specific route with `app.use`. The first argument is the path the application will listen to and the second argument is a callback function that will run when the application serves the path. The callback takes a `req` argument, which contains the request data and a `res` argument that handles the result.

Add in a handler for the `/login` path. Call `res.send` with a JavaScript object containing a token:

auth-tutorial/server.js

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

app.use(cors());

app.use('/login', (req, res) => {
  res.send({
    token: 'test123'
  });
});
```

Finally, run the server on port `8080` using `app.listen`:

auth-tutorial/server.js

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

app.use(cors());

app.use('/login', (req, res) => {
  res.send({
    token: 'test123'
  });
});

app.listen(8080, () => console.log('API is running on http://lo
```

Save and close the file. In a new terminal window or tab, start the server:

```
node server.js
```

You will receive a response indicating that the server is starting:

Output

```
API is running on http://localhost:8080/login
```

Visit `http://localhost:8080/login` and you'll find your JSON object.

```
{"token":"test123"}
```

Token response

When you fetch the token in your browser, you are making a `GET` request, but when you submit the login form you will be making a `POST` request. That's not a problem. When you set up your route with `app.use`, Express will handle all requests the same. In a production application, you should be more specific and only allow certain request methods for each route.

Now that you have a running API server, you need to make a request from your login page. Open `Login.js`:

```
nano src/components/Login/Login.js
```

In the previous step, you passed a new prop called `setToken` to the `Login` component. Add in the `PropType` from the new prop and destructure the props object to pull out the `setToken` prop.

auth-tutorial/src/components/Login/Login.js

```
import React from 'react';
import PropTypes from 'prop-types';

import './Login.css';

export default function Login({ setToken }) {
  return(
    <div className="login-wrapper">
      <h1>Please Log In</h1>
      <form>
        <label>
          <p>Username</p>
          <input type="text" />
        </label>
        <label>
          <p>Password</p>
          <input type="password" />
        </label>
        <div>
          <button type="submit">Submit</button>
        </div>
      </form>
    </div>
  )
```

```
}


Login.propTypes = {
  setToken: PropTypes.func.isRequired
}
```

Next, create a local state to capture the `Username` and `Password`. Since you do not need to manually set data, make the `<inputs>` uncontrolled components. You can find detailed information about uncontrolled components in How To Build Forms in React.

auth-tutorial/src/components/Login/Login.js

```
import React, { useState } from 'react';
import PropTypes from 'prop-types';

import './Login.css';

export default function Login({ setToken }) {
  const [username, setUserName] = useState();
  const [password, setPassword] = useState();
  return(
    <div className="login-wrapper">
      <h1>Please Log In</h1>
      <form>
        <label>
          <p>Username</p>
          <input type="text" onChange={e => setUserName(e.targe
        </label>
        <label>
          <p>Password</p>
          <input type="password" onChange={e => setPassword(e.t
        </label>
        <div>
          <button type="submit">Submit</button>
        </div>
      </form>
    </div>
```

```
  )
}


Login.propTypes = {
  setToken: PropTypes.func.isRequired
};
```

Next, create a function to make a `POST` request to the server. In a large application, you would add these to a separate directory. In this example, you'll add the service directly to the component. Check out the tutorial How To Call Web APIs with the useEffect Hook in React for a detailed look at calling APIs in React components.

Create an `async` function called `loginUser`. The function will take `credentials` as an argument, then it will call the `fetch` method using the `POST` option:

auth-tutorial/src/components/Login/Login.js

```
import React, { useState } from 'react';
import PropTypes from 'prop-types';
import './Login.css';

async function loginUser(credentials) {
 return fetch('http://localhost:8080/login', {
   method: 'POST',
   headers: {
     'Content-Type': 'application/json'
   },
   body: JSON.stringify(credentials)
 })
   .then(data => data.json())
}

export default function Login({ setToken }) {
...
```

Finally, create a form submit handler called `handleSubmit` that will call `loginUser` with the `username` and `password`. Call `setToken` with a successful result. Call `handleSubmit` using the `onSubmit` event handler on the `<form>`:

auth-tutorial/src/components/Login/Login.js

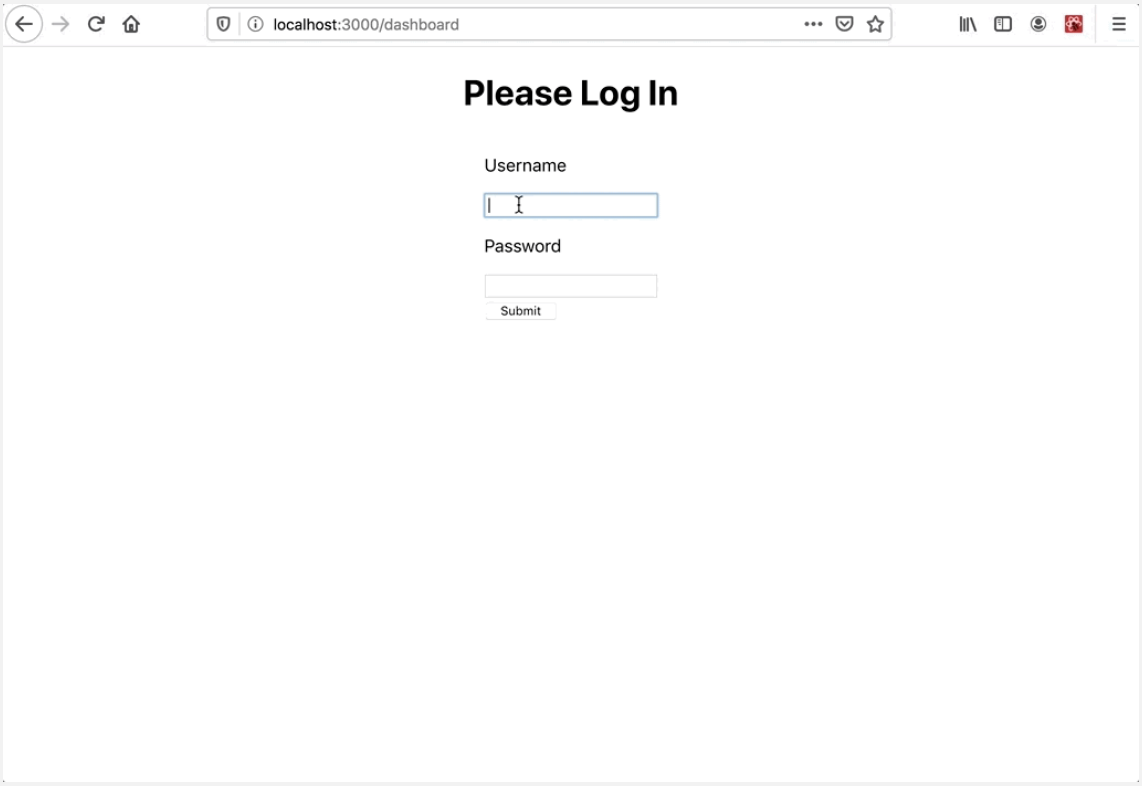
```
import React, { useState } from 'react';
import PropTypes from 'prop-types';
import './Login.css';

async function loginUser(credentials) {
 return fetch('http://localhost:8080/login', {
   method: 'POST',
   headers: {
     'Content-Type': 'application/json'
   },
   body: JSON.stringify(credentials)
 })
   .then(data => data.json())
}

export default function Login({ setToken }) {
  const [username, setUserName] = useState();
  const [password, setPassword] = useState();

  const handleSubmit = async e => {
    e.preventDefault();
    const token = await loginUser({
      username,
      password
    });
```

```
    setToken(token);
  }

  return(
    <div className="login-wrapper">
      <h1>Please Log In</h1>
      <form onSubmit={handleSubmit}>
        <label>
          <p>Username</p>
          <input type="text" onChange={e => setUserName(e.targe
        </label>
        <label>
          <p>Password</p>
          <input type="password" onChange={e => setPassword(e.t
        </label>
        <div>
          <button type="submit">Submit</button>
        </div>
      </form>
    </div>
  )
}

Login.propTypes = {
  setToken: PropTypes.func.isRequired
};
```

Note: In a full application, you'll need to handle situations where the component unmounts before a Promise resolves. Check out the tutorial How To Call Web APIs with the useEffect Hook in React for more information.

Save and close the file. Make sure that your local API is still running, then open a browser to `http://localhost:3000/dashboard`.

You will see the login page instead of the dashboard. Fill out and submit the form and you will receive a web token then redirect to the page for the dashboard.



Login page

You now have a working local API and an application that requests a token using a username and password. But there is still a problem. The token is currently stored using a local state, which means that it is stored in JavaScript memory. If you open a new window, tab, or even just refresh the page, you will lose the token and the user will need to login again. This will be addressed in the next step.

In this step you created a local API and a login page for your application. You learned how to create a Node server to send a token and how to call the server and store the token from a login component. In the next step, you'll learn how to store the user token so that a session will persist across page refreshes or tabs.

Step 3 — Storing a User Token with `sessionStorage` and `localStorage`

In this step, you'll store the user token. You'll implement different token storage options and learn the security implications of each approach. Finally, you'll learn how different approaches will change the user experience as the user opens new tabs or closes a session.

By the end of this step, you'll be able to choose a storage approach based on the goals for your application.

There are several options for storing tokens. Every option has costs and benefits. In brief the options are: storing in JavaScript memory, storing in `sessionStorage`, storing in `localStorage`, and storing in a cookie. The primary trade-off is security. Any information that is stored outside of the memory of the current application is vulnerable to Cross-Site Scripting (XSS) attacks. The danger is that if a malicious user is able to load code into your application, it can access `localStorage`, `sessionStorage`, and any cookie that is also accessible to your application. The benefit of the non-memory storage methods is that you can reduce the number of times a user will need to log in to create a better user experience.

This tutorial will cover `sessionStorage` and `localStorage`, since these are more modern than using cookies.

Session Storage

To test the benefits of storing outside of memory, convert the in-memory storage to `sessionStorage`. Open `App.js`:

```
nano src/components/App/App.js
```

Remove the call to `useState` and create two new functions called `setToken` and `getToken`. Then call `getToken` and assign the results to a variable called `token`:

auth-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom'

import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Login from '../Login/Login';
import Preferences from '../Preferences/Preferences';

function setToken(userToken) {
}

function getToken() {
}

function App() {
  const token = getToken();

  if(!token) {
    return <Login setToken={setToken} />
  }

  return (
    <div className="wrapper">
     ...
    </div>
```

```
  );
}


export default App;
```

Since you are using the same function and variable names, you will not need to change any code in the `Login` component or the rest of the `App` component.

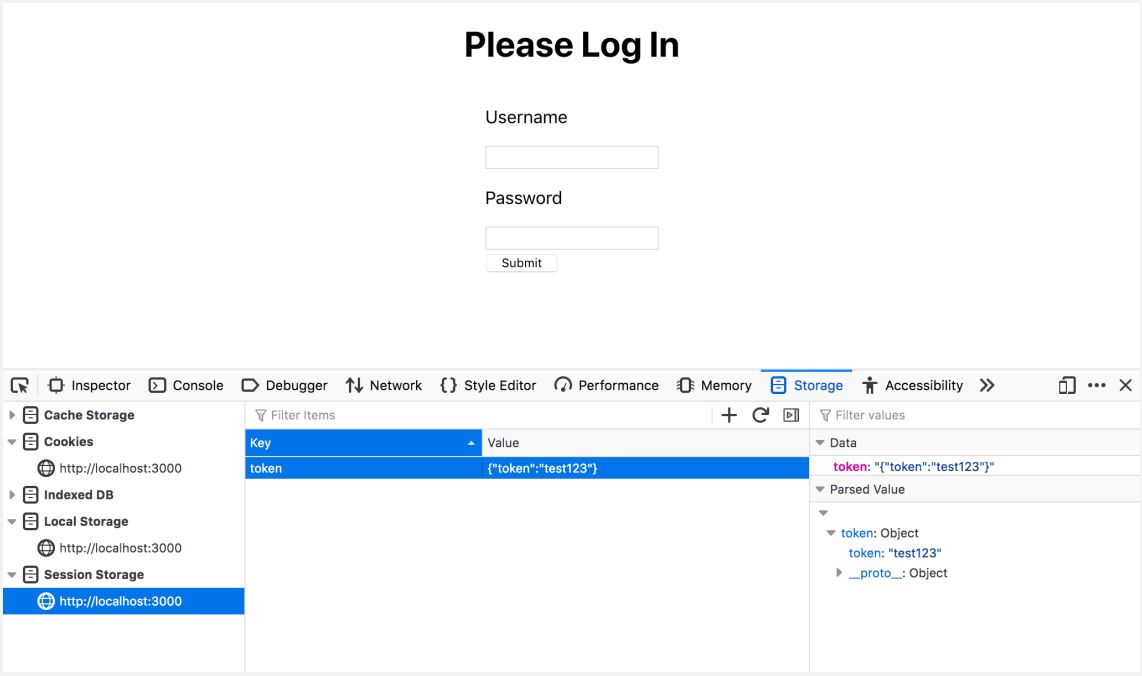
Inside of `setToken`, save the `userToken` argument to `sessionStorage` using the `setItem` method. This method takes a key as a first argument and a string as the second argument. That means you'll need to convert the `userToken` from an object to a string using the `JSON.stringify` function. Call `setItem` with a key of `token` and the converted object.

auth-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom'

import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Login from '../Login/Login';
import Preferences from '../Preferences/Preferences';

function setToken(userToken) {
  sessionStorage.setItem('token', JSON.stringify(userToken));
}

function getToken() {
}

function App() {
  const token = getToken();

  if(!token) {
    return <Login setToken={setToken} />
  }

  return (
    <div className="wrapper">
      ...
```

```
    </div>
  );
}

export default App;
```

Save the file. When you do the browser will reload. If you type in a username and password and submit, the browser will still render the login page, but if you look inside your browser console tools, you'll find the token is stored in `sessionStorage`. This image is from Firefox, but you'll find the same results in Chrome or other modern browsers.



Token in sessionStorage

Now you need to retrieve the token to render the correct page. Inside the `getToken` function, call `sessionStorage.getItem`. This method takes a `key` as an argument and returns the string value. Convert the string to an object using `JSON.parse`, then return the value of `token`:

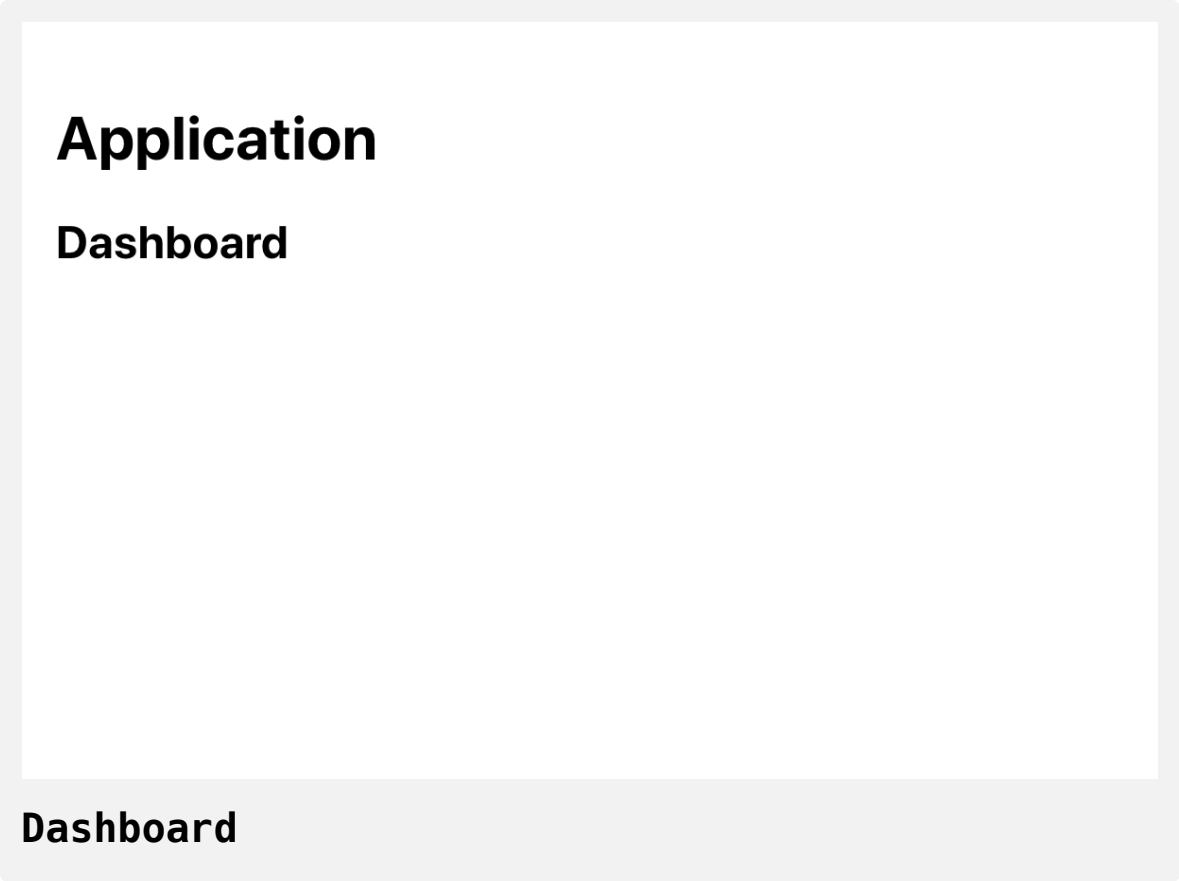
auth-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Login from '../Login/Login';
import Preferences from '../Preferences/Preferences';

function setToken(userToken) {
  sessionStorage.setItem('token', JSON.stringify(userToken));
}

function getToken() {
  const tokenString = sessionStorage.getItem('token');
  const userToken = JSON.parse(tokenString);
  return userToken?.token
}

function App() {
  const token = getToken();

  if(!token) {
    return <Login setToken={setToken} />
  }

  return (
```

```
    <div className="wrapper">
      ...
    </div>
  );
}

export default App;
```

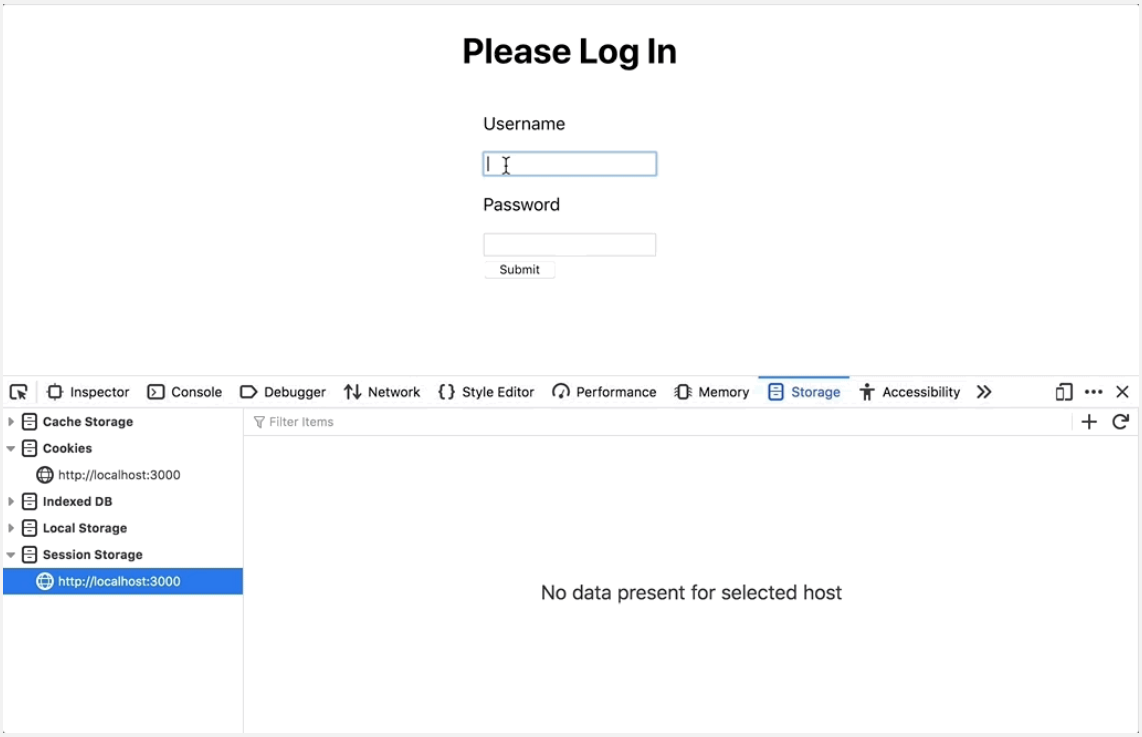
You need to use the optional chaining operator—`?.`—when accessing the `token` property because when you first access the application, the value of `sessionStorage.getItem('token')` will be `undefined`. If you try to access a property, you will generate an error.

Save and close the file. In this case, you already have a token stored, so when the browser refreshes, you will navigate to the private pages:



Clear out the token by either deleting the token in the **Storage** tab in your developer tools or by typing `sessionStorage.clear()` in your developer console.

There’s a little problem now. When you log in, the browser saves the token, but you still see the login page.



Token stored still not logged in

The problem is your code never alerts React that the token retrieval was successful. You'll still need to set some state that will trigger a re-render when the data changes. Like most problems in React, there are multiple ways to solve it. One of the most elegant and reusable is to create a custom Hook.

Creating a Custom Token Hook

A custom Hook is a function that wraps custom logic. A custom Hook usually wraps one or more built-in React Hooks along with custom implementations. The primary advantage of a custom Hook is that you can remove the implementation logic from the component and you can reuse it across multiple components.

By convention, custom Hooks start with the keyword `use*`.

Open a new file in the `App` directory called `useToken.js`:

```
nano src/components/App/useToken.js
```

This will be a small Hook and would be fine if you defined it directly in `App.js`. But moving the custom Hook to a different file will show how Hooks work outside of a component. If you start to reuse this Hook across multiple components, you might also want to move it to a separate directory.

Inside `useToken.js`, import `useState` from `react`. Notice that you do not need to import `React` since you will have no JSX in the file. Create and export a function called `useToken`. Inside this function, use the `useState` Hook to create a `token` state and a `setToken` function:

auth-tutorial/src/components/App/useToken.js

```
import { useState } from 'react';

export default function useToken() {
  const [token, setToken] = useState();

}
```

Next, copy the `getToken` function to `useHook` and convert it to an arrow function, since you placed it inside `useToken`. You could leave the function as a standard, named function, but it can be easier to read when top-level functions are standard and internal functions are arrow functions. However, each team will be different. Choose one style and stick with it.

Place `getToken` before the state declaration, then initialize `useState` with `getToken`. This will fetch the token and set it as the initial state:

auth-tutorial/src/components/App/useToken.js

```
import { useState } from 'react';

export default function useToken() {
  const getToken = () => {
    const tokenString = sessionStorage.getItem('token');
    const userToken = JSON.parse(tokenString);
    return userToken?.token
  };
  const [token, setToken] = useState(getToken());

}
```

Next, copy the `setToken` function from `App.js`. Convert to an arrow function and name the new function `saveToken`. In addition to saving the token to `sessionStorage`, save the token to state by calling `setToken`:

auth-tutorial/src/components/App/useToken.js

```
import { useState } from 'react';

export default function useToken() {
  const getToken = () => {
    const tokenString = sessionStorage.getItem('token');
    const userToken = JSON.parse(tokenString);
    return userToken?.token
  };

  const [token, setToken] = useState(getToken());

  const saveToken = userToken => {
    sessionStorage.setItem('token', JSON.stringify(userToken));
    setToken(userToken.token);
  };
}
```

Finally, return an object that contains the `token` and `saveToken` set to the `setToken` property name. This will give the component the same interface. You can also return the values as an array, but an object will give users a chance to destructure only the values they want if you reuse this in another component.

auth-tutorial/src/components/App/useToken.js

```
import { useState } from 'react';

export default function useToken() {
  const getToken = () => {
    const tokenString = sessionStorage.getItem('token');
    const userToken = JSON.parse(tokenString);
    return userToken?.token
  };

  const [token, setToken] = useState(getToken());

  const saveToken = userToken => {
    sessionStorage.setItem('token', JSON.stringify(userToken));
    setToken(userToken.token);
  };

  return {
    setToken: saveToken,
    token
  }
}
```

Save and close the file.

Next, open `App.js`:

```
nano src/components/App/App.js
```

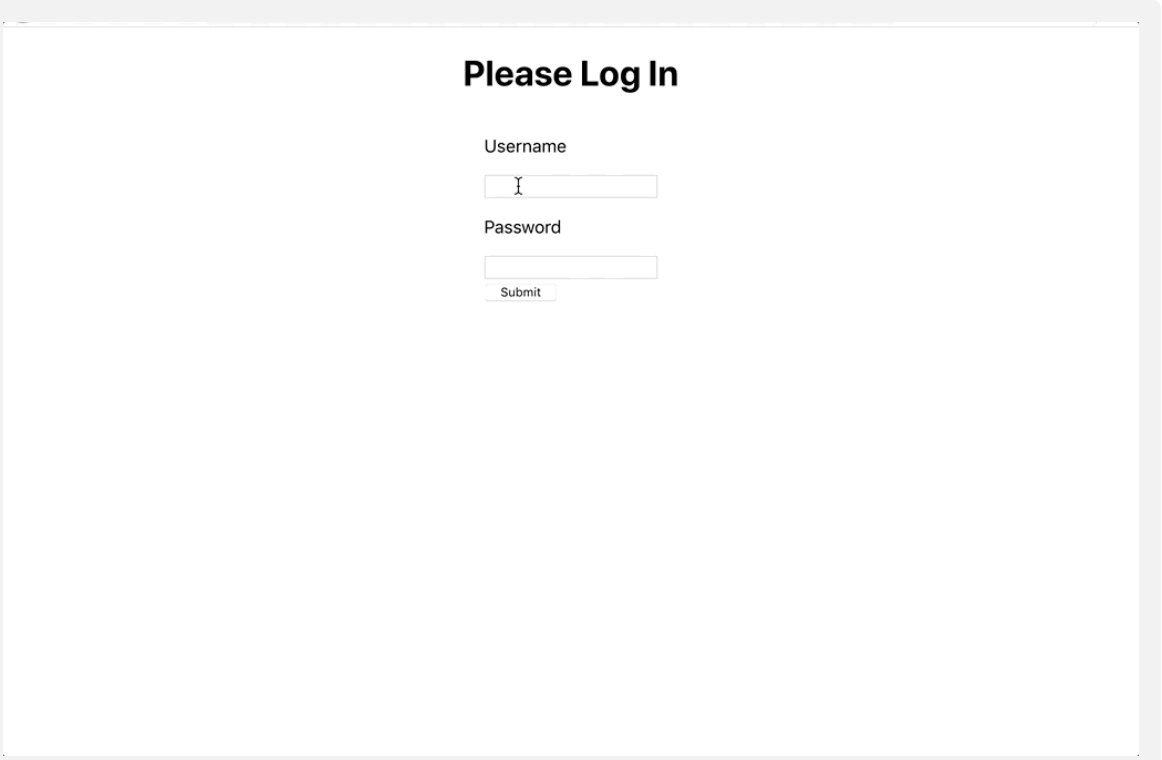
Remove the `getToken` and `setToken` functions. Then import `useToken` and call the function destructuring the `setToken` and `token` values. You can also remove the import of `useState` since you are no longer using the Hook:

auth-tutorial/src/components/App/App.js

```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom'
import './App.css';
import Dashboard from '../Dashboard/Dashboard';
import Login from '../Login/Login';
import Preferences from '../Preferences/Preferences';
import useToken from './useToken';

function App() {

  const { token, setToken } = useToken();

  if(!token) {
    return <Login setToken={setToken} />
  }

  return (
    <div className="wrapper">
      <h1>Application</h1>
      <BrowserRouter>
        <Switch>
          <Route path="/dashboard">
            <Dashboard />
          </Route>
          <Route path="/preferences">
```

```
            <Preferences />
          </Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}

export default App;
```

Save and close the file. When you do, the browser will refresh, and when you log in, you will immediately go to the page. This is happening because you are calling `useState` in your custom Hook, which will trigger a component re-render:



Login immediately

You now have a custom Hook to store your token in `sessionStorage`. Now you can refresh your page and the user will remain logged in. But if you try to open the application in another tab, the user will be logged out. `sessionStorage` belongs only to the specific window session. Any data will not be available in a new tab and will be lost when the active tab is closed. If you want to save the token across tabs, you'll need to convert to `localStorage`.

Using `localStorage` to Save Data Across Windows

Unlike `sessionStorage`, `localStorage` will save data even after the session ends. This can be more convenient, since it lets users open multiple windows and tabs without a new login, but it does have some security

problems. If the user shares their computer, they will remain logged in to the application even though they close the browser. It will be the user's responsibility to explicitly log out. The next user would have immediate access to the application without a login. It's a risk, but the convenience may be worth it for some applications.

To convert to `localStorage`, open `useToken.js`:

```
nano src/components/App/useToken.js
```

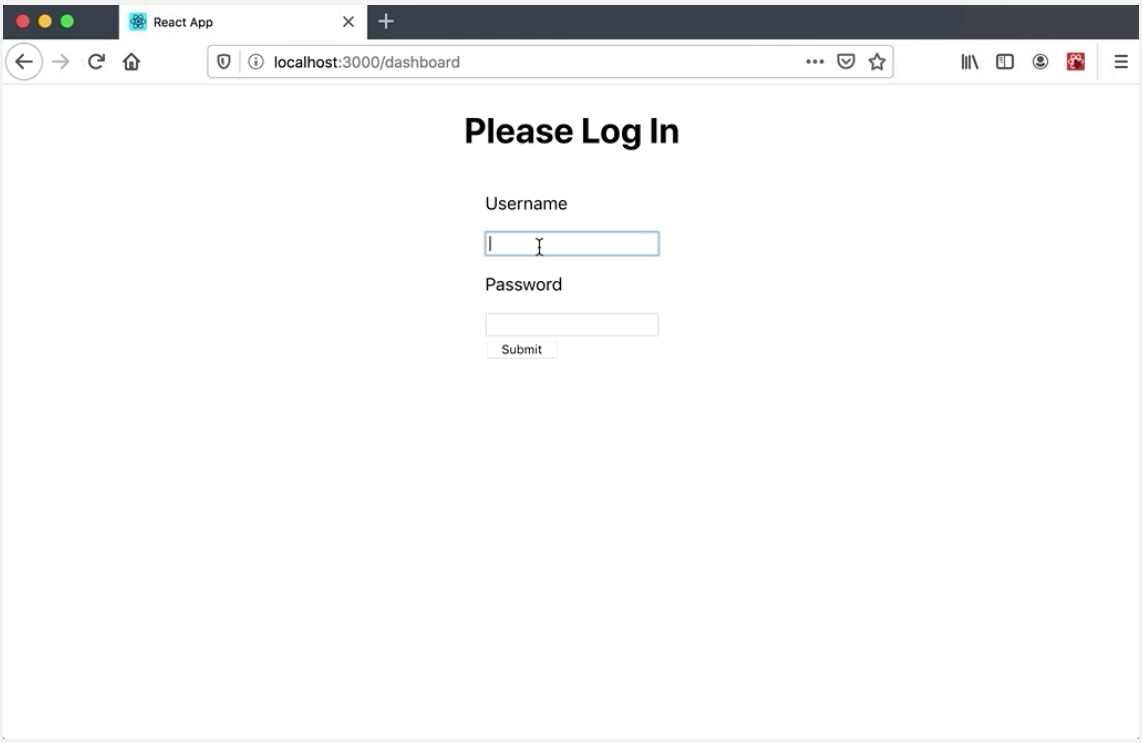
Then change every reference of `sessionStorage` to `localStorage`. The methods you call will be the same:

auth-tutorial/src/components/App/useToken.js

```
import { useState } from 'react';

export default function useToken() {
  const getToken = () => {
    const tokenString = localStorage.getItem('token');
    const userToken = JSON.parse(tokenString);
    return userToken?.token
  };

  const [token, setToken] = useState(getToken());

  const saveToken = userToken => {
    localStorage.setItem('token', JSON.stringify(userToken));
    setToken(userToken.token);
  };

  return {
    setToken: saveToken,
    token
  }
}
```

Save the file. When you do, the browser will refresh. You will need to log in again since there is no token yet in `localStorage`, but after you do, you will

remain logged in when you open a new tab.



Still logged into tab

In this step, you saved tokens with `sessionStorage` and `localStorage`. You also created a custom Hook to trigger a component re-render and to move component logic to a separate function. You also learned about how `sessionStorage` and `localStorage` affect the user's ability to start new sessions without login.

Conclusion

Authentication is a crucial requirement of many applications. The mixture of security concerns and user experience can be intimidating, but if you

focus on validating data and rendering components at the correct time, it can become a lightweight process.

Each storage solution offers distinct advantages and disadvantages. Your choice may change as your application evolves. By moving your component logic into an abstract custom Hook, you give yourself the ability to refactor without disrupting existing components.

If you would like to read more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Avoid Performance Pitfalls in React with memo, useMemo, and useCallback

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

In React applications, performance problems can come from network latency, overworked APIs, inefficient third-party libraries, and even well-structured code that works fine until it encounters an unusually large load. Identifying the root cause of performance problems can be difficult, but many of these problems originate from component re-rendering. Either the component re-renders more than expected or the component has a data-heavy operation that can cause each render to be slow. Because of this, learning how to prevent unneeded re-renders can help to optimize the performance of your React application and create a better experience for your user.

In this tutorial, you'll focus on optimizing performance in React components. To explore the problem, you'll build a component to analyze a block of text. You'll look at how different actions can trigger re-renders and how you can use Hooks and memoization to minimize expensive data calculations. By the end of this tutorial, you'll be familiar with many performance enhancing Hooks, such as the `useMemo` and `useCallback` Hook, and the circumstances that will require them.

Prerequisites

- You will need a development environment running Node.js; this tutorial was tested on Node.js version 10.22.0 and npm version 6.14.6. To install this on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04.

- A React development environment set up with Create React App, with the non-essential boilerplate removed. To set this up, follow **Step 1 — Creating an Empty Project** of the How To Manage State on React Class Components tutorial. This tutorial will use `performance-tutorial` as the project name.

- If you are new to debugging in React, check out the tutorial How To Debug React Components Using React Developer Tools, and familiarize yourself with the developer tools in the browser you are using, such as Chrome DevTools and Firefox Developer Tools.

- You will also need a basic knowledge of JavaScript, HTML, and CSS, which you can find in our How To Build a Website With HTML series, How To Build a Website With CSS series, and in How To Code in JavaScript.

Step 1 — Preventing Re-renders with `memo`

In this step, you'll build a text analyzing component. You'll create an input to take a block of text and a component that will calculate the frequency of letters and symbols. You'll then create a scenario where the text analyzer performs poorly and you'll identify the root cause of the performance problem. Finally, you'll use the React `memo` function to prevent re-renders on the component when a parent changes, but the props to the child component do not change.

By the end of this step, you'll have a working component that you'll use throughout the rest of the tutorial and an understanding of how parent re-rendering can create performance problems in child components.

Building a Text Analyzer

To begin, add a `<textarea>` element to `App.js`.

Open `App.js` in a text editor of your choice:

```
nano src/components/App/App.js
```

Then add a `<textarea>`input with a `<label>`. Place the label inside a `<div>` with a `className` of `wrapper` by adding the following highlighted code:

performance-tutorial/src/components/App/App.js

```
import React from 'react';
import './App.css';

function App() {
  return(
    <div className="wrapper">
      <label htmlFor="text">
        <p>Add Your Text Here:</p>
        <textarea
          id="text"
          name="text"
          rows="10"
          cols="100"
        >
        </textarea>
      </label>
    </div>
  )
}

export default App;
```

This will create an input box for the sample application. Save and close the file.

Open `App.css` to add some styles:

```
nano src/components/App/App.css
```

Inside `App.css`, add padding to the `.wrapper` class, then add a `margin` to the `div` elements. Replace the CSS with the following:

performance-tutorial/src/components/App/App.css

```
.wrapper {
    padding: 20px;
}

.wrapper div {
    margin: 20px 0;
}
```

This will add separation between the input and the data display. Save and close the file.

Next, create a directory for the `CharacterMap` component. This component will analyze the text, calculate the frequency of each letter and symbol, and display the results.

First, make the directory:

```
mkdir src/components/CharacterMap
```

Then open `CharacterMap.js` in a text editor:

```
nano src/components/CharacterMap/CharacterMap.js
```

Inside, create a component called `CharacterMap` that takes `text` as a prop and displays the `text` inside a `<div>`:

performance-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React from 'react';
import PropTypes from 'prop-types';

export default function CharacterMap({ text }) {
  return(
    <div>
      Character Map:
      {text}
    </div>
  )
}

CharacterMap.propTypes = {
  text: PropTypes.string.isRequired
}
```

Notice that you are adding a `PropType` for the `text` prop to introduce some weak typing.

Add a function to loop over the text and extract the character information. Name the function `itemize` and pass the `text` as an argument. The `itemize` function is going to loop over every character several times and will be

very slow as the text size increases. This will make it a good way to test performance:

performance-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React from 'react';
import PropTypes from 'prop-types';

function itemize(text){
  const letters = text.split('')
    .filter(l => l !== ' ')
    .reduce((collection, item) => {
      const letter = item.toLowerCase();
      return {
        ...collection,
        [letter]: (collection[letter] || 0) + 1
      }
    }, {})
  return letters;
}

export default function CharacterMap({ text }) {
  return(
    <div>
      Character Map:
      {text}
    </div>
  )
}
```

```
CharacterMap.propTypes = {
  text: PropTypes.string.isRequired
}
```

Inside `itemize`, you convert the text into an array by using `.split` on every character. Then you remove the spaces using the `.filter` method and use the `.reduce` method to iterate over each letter. Inside the `.reduce` method, use an object as the initial value, then normalize the character by converting it to lower case and adding `1` to the previous total or `0` if there was no previous total. Update the object with the new value while preserving previous values using the Object spread operator.

Now that you have created an object with a count for each character, you need to sort it by the highest character. Convert the object to an array of pairs with `Object.entries`. The first item in the array is the character and the second item is the count. Use the `.sort` method to place the most common characters on top:

performance-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React from 'react';
import PropTypes from 'prop-types';

function itemize(text){
  const letters = text.split('')
    .filter(l => l !== ' ')
    .reduce((collection, item) => {
      const letter = item.toLowerCase();
      return {
        ...collection,
        [letter]: (collection[letter] || 0) + 1
      }
    }, {})
  return Object.entries(letters)
    .sort((a, b) => b[1] - a[1]);
}

export default function CharacterMap({ text }) {
  return(
    <div>
      Character Map:
      {text}
    </div>
  )
```

```
}


CharacterMap.propTypes = {
  text: PropTypes.string.isRequired
}
```

Finally, call the `itemize` function with the text and display the results:

performance-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React from 'react';
import PropTypes from 'prop-types';

function itemize(text){
  const letters = text.split('')
    .filter(l => l !== ' ')
    .reduce((collection, item) => {
      const letter = item.toLowerCase();
      return {
        ...collection,
        [letter]: (collection[letter] || 0) + 1
      }
    }, {})
  return Object.entries(letters)
    .sort((a, b) => b[1] - a[1]);
}

export default function CharacterMap({ text }) {
  return(
    <div>
      Character Map:
      {itemize(text).map(character => (
        <div key={character[0]}>
```

```
          {character[0]}: {character[1]}
        </div>
      ))}
    </div>
  )
}

CharacterMap.propTypes = {
  text: PropTypes.string.isRequired
}
```

Save and close the file.

Now import the component and render inside of `App.js`. Open `App.js`:

```
nano src/components/App/App.js
```

Before you can use the component, you need a way to store the text. Import `useState` then call the function and store the values on a variable called `text` and an update function called `setText`.

To update the `text`, add a function to `onChange` that will pass the `event.target.value` to the `setText` function:

performance-tutorial/src/components/App/App.js

```
import React, { useState } from 'react';
import './App.css';

function App() {
  const [text, setText] = useState('');

  return(
    <div className="wrapper">
      <label htmlFor="text">
        <p>Your Text</p>
        <textarea
          id="text"
          name="text"
          rows="10"
          cols="100"
          onChange={event => setText(event.target.value)}
        >
        </textarea>
      </label>
    </div>
  )
}

export default App;
```

Notice that you are initializing `useState` with an empty string. This will ensure that the value you pass to the `CharacterMap` component is always a string even if the user has not yet entered text.

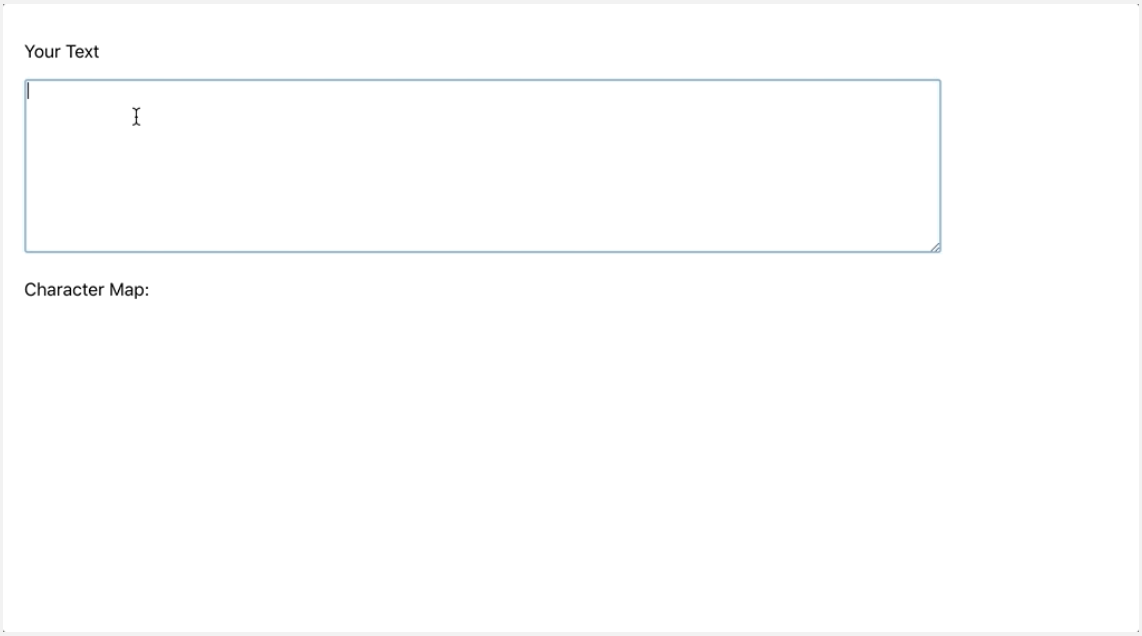
Import `CharacterMap` and render it after the `<label>` element. Pass the `text` state to the `text` prop:

performance-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React, { useState } from 'react';
import './App.css';

import CharacterMap from '../CharacterMap/CharacterMap';

function App() {
  const [text, setText] = useState('');

  return(
    <div className="wrapper">
      <label htmlFor="text">
        <p>Your Text</p>
        <textarea
          id="text"
          name="text"
          rows="10"
          cols="100"
          onChange={event => setText(event.target.value)}
        >
        </textarea>
      </label>
      <CharacterMap text={text} />
    </div>
  )
```

```
}


export default App;
```

Save the file. When you do, the browser will refresh and when you add text, you'll find the character analysis after the input:



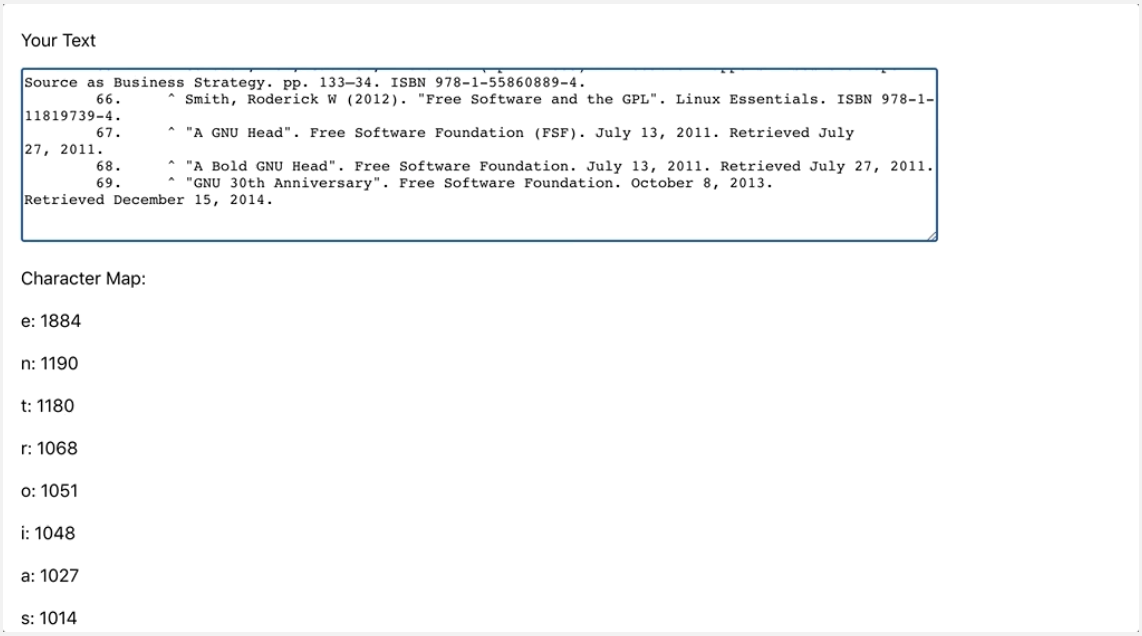
Input with results below

As shown in the example, the component performs fairly well with a small amount of text. With every keystroke, React will update the `CharacterMap` with new data. But performance locally can be misleading. Not all devices will have the same memory as your development environment.

Testing Performance

There are multiple ways to test performance of your application. You can add large volumes of text or you can set your browser to use less memory. To push the component to a performance bottleneck, copy the Wikipedia entry for GNU and paste it in the text box. Your sample may be slightly different depending on how the Wikipedia page is edited.

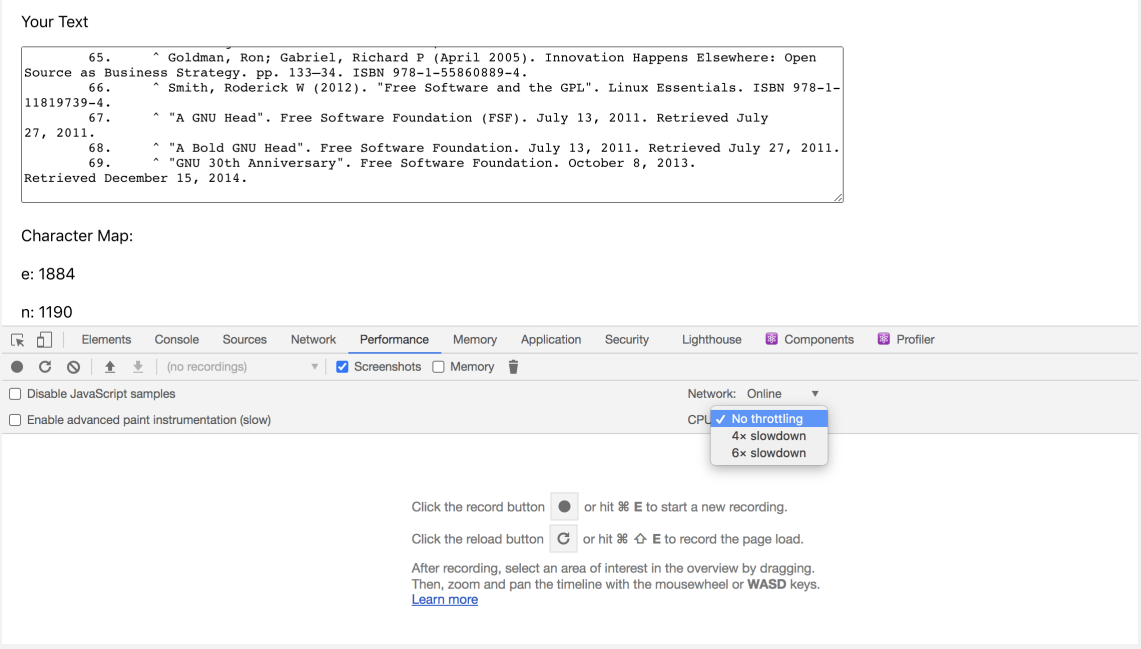
After pasting the entry into your text box, try typing the additional letter `e` and notice how long it takes to display. There will be a significant pause before the character map updates:



Animation showing the delay when typing “e” into the application

If the component is not slow enough and you are using Firefox, Edge, or some other browser, add more text until you notice a slowdown.

If you are using Chrome, you can throttle the CPU inside the performance tab. This is a great way to emulate a smartphone or an older piece of hardware. For more information, check out the Chrome DevTools documentation.



Performance Throttling in Chrome DevTools

If the component is too slow with the Wikipedia entry, remove some text. You want to receive a noticable delay, but you do not want to make it unusably slow or to crash your browser.

Preventing Re-Rendering of Child Components

The `itemize` function is the root of the delay identified in the last section. The function does a lot of work on each entry by looping over the contents several times. There are optimizations you can perform directly in the

function itself, but the focus of this tutorial is how to handle component re-rendering when the text does not change. In other words, you will treat the `itemize` function as a function that you do not have access to change. The goal will be to run it only when necessary. This will show how to handle performance for APIs or third-party libraries that you can't control.

To start, you will explore a situation where the parent changes, but the child component does not change.

Inside of `App.js`, add a paragraph explaining how the component works and a button to toggle the information:

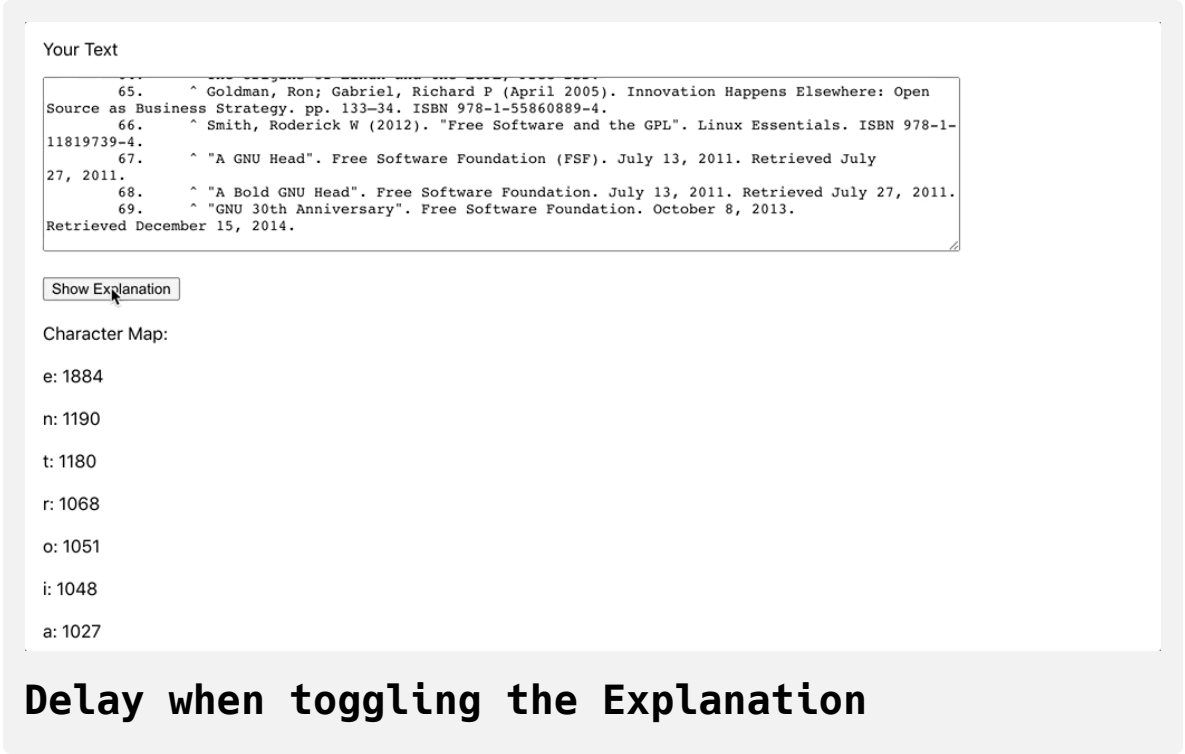
performance-tutorial/src/components/App/App.js

```
import React, { useReducer, useState } from 'react';
import './App.css';

import CharacterMap from '../CharacterMap/CharacterMap';

function App() {
  const [text, setText] = useState('');
  const [showExplanation, toggleExplanation] = useReducer(state

  return(
    <div className="wrapper">
      <label htmlFor="text">
        <p>Your Text</p>
        <textarea
          id="text"
          name="text"
          rows="10"
          cols="100"
          onChange={event => setText(event.target.value)}
        >
        </textarea>
      </label>
      <div>
        <button onClick={toggleExplanation}>Show Explanation</b
      </div>
```

```
      {showExplanation &&
        <p>
          This displays a list of the most common characters.
        </p>
      }
      <CharacterMap text={text} />
    </div>
  )
}

export default App;
```

Call the `useReducer` Hook with a reducer function to reverse the current state. Save the output to `showExplanation` and `toggleExplanation`. After the `<label>`, add a button to toggle the explanation and a paragraph that will render when `showExplanation` is truthy.

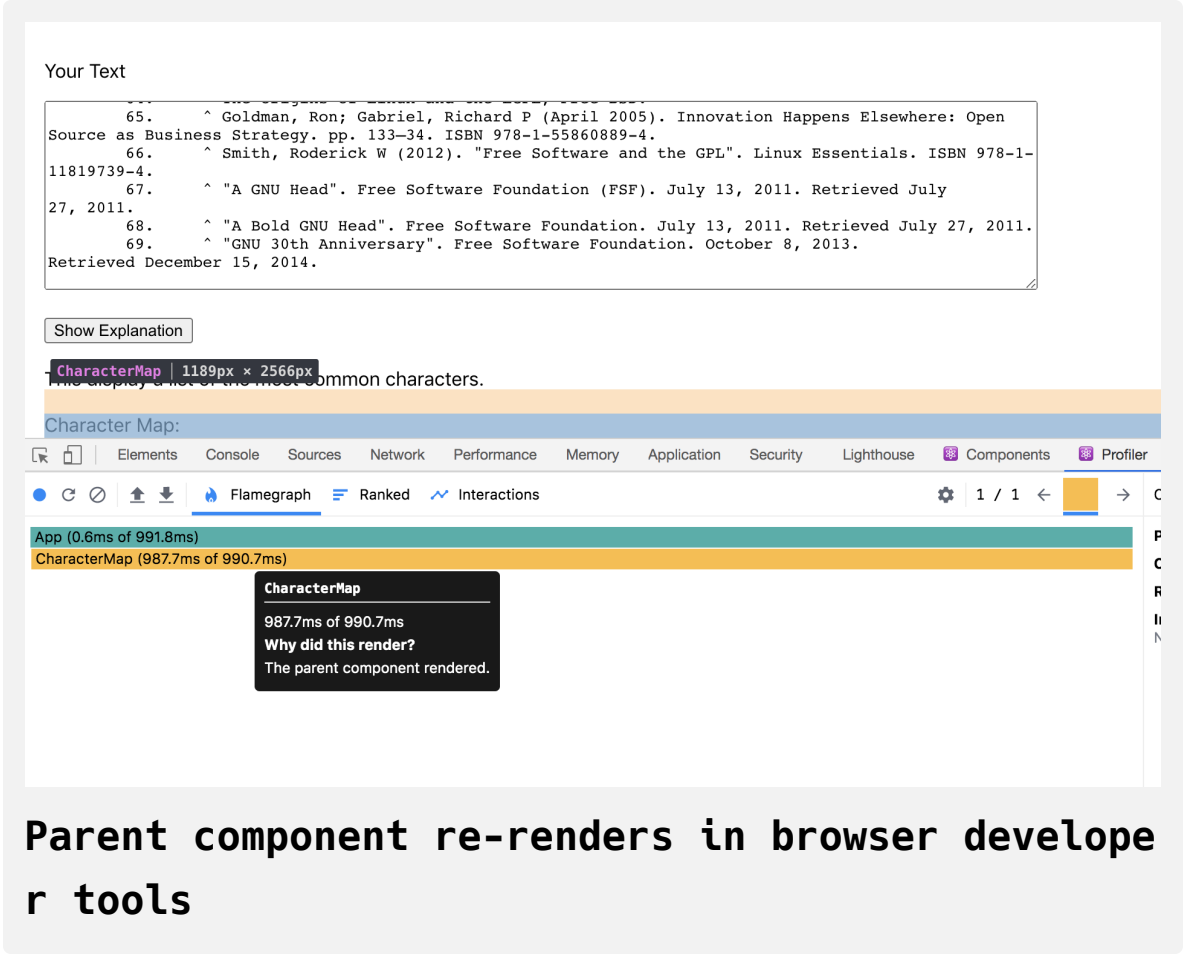
Save and exit the file. When the browser refreshes, click on the button to toggle the explanation. Notice how there is a delay.



Delay when toggling the Explanation

This presents a problem. Your users shouldn't encounter a delay when they are toggling a small amount of JSX. The delay occurs because when the parent component changes—`App.js` in this situation—the `CharacterMap` component is re-rendering and re-calculating the character data. The `text` prop is identical, but the component still re-renders because React will re-render the entire component tree when a parent changes.

If you profile the application with the browser's developer tools, you'll discover that the component re-renders because the parent changes. For a review of profiling using the developer tools, check out How To Debug React Components Using React Developer Tools.



Parent component re-renders in browser developer tools

Since `CharacterMap` contains an expensive function, it should only re-render it when the props change.

Open `CharacterMap.js`:

```
nano src/components/CharacterMap/CharacterMap.js
```

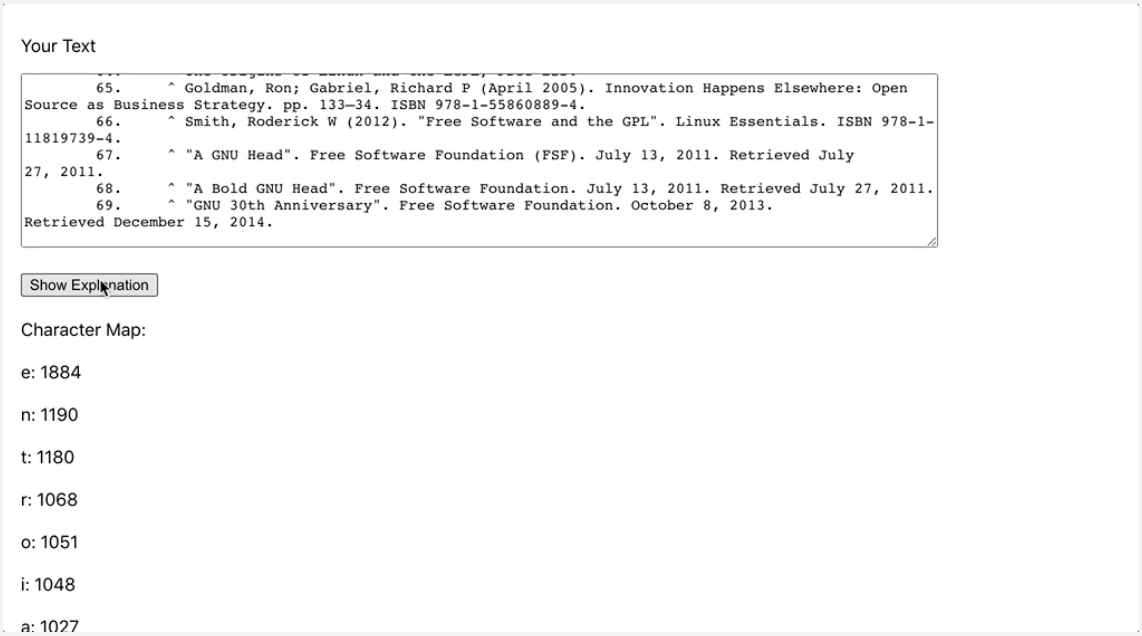
Next, import `memo`, then pass the component to `memo` and export the result as the default:

performance-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React { memo } from 'react';
import PropTypes from 'prop-types';

function itemize(text){
  ...
}

function CharacterMap({ text }) {
  return(
    <div>
      Character Map:
      {itemize(text).map(character => (
        <div key={character[0]}>
          {character[0]}: {character[1]}
        </div>
      ))}
    </div>
  )
}

CharacterMap.propTypes = {
  text: PropTypes.string.isRequired
}
```

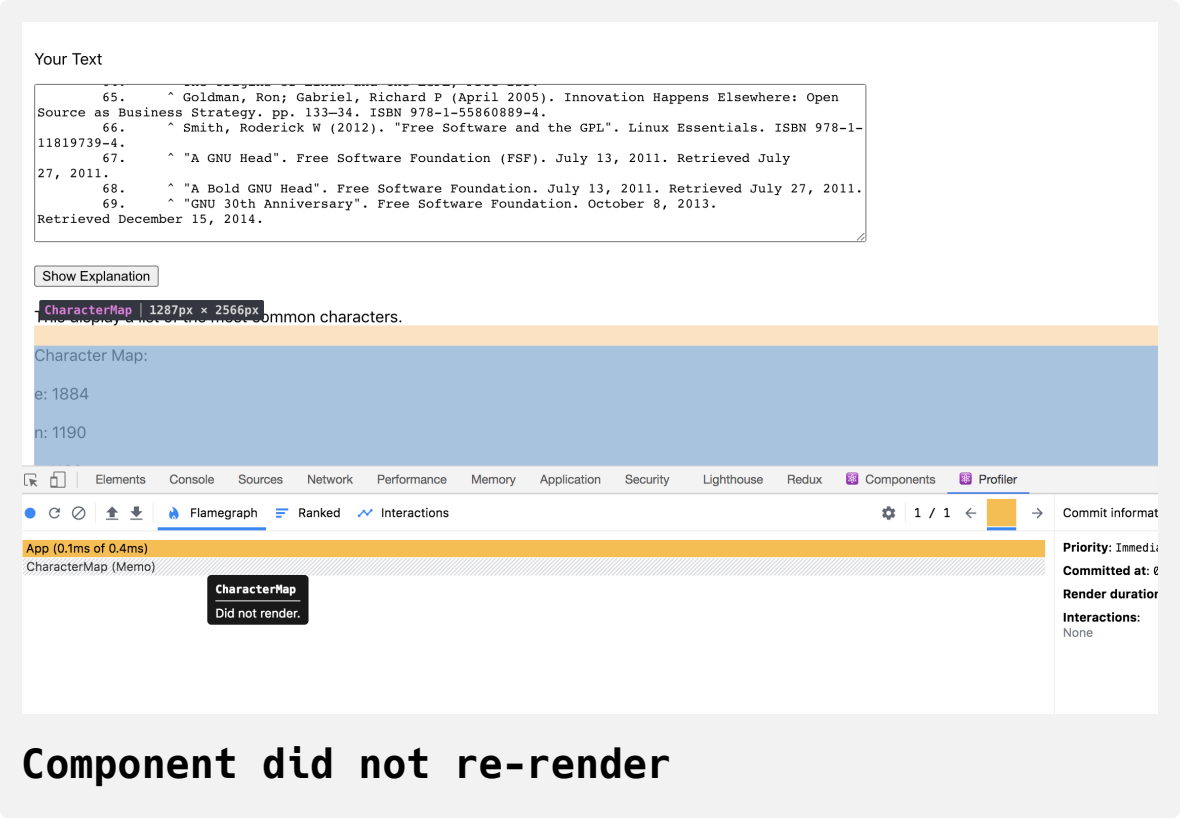
```
export default memo(CharacterMap);
```

Save the file. When you do, the browser will reload and there will no longer be a delay after you click the button before you get the result:



No delay when toggling the explanation in the test app

If you look at the developer tools, you'll find that the component no longer re-renders:



Component did not re-render

The `memo` function will perform a shallow comparison of props and will re-render only when the props change. A shallow comparison will use the `===` operator to compare the previous prop to the current prop.

It's important to remember that the comparison is not free. There is a performance cost to check the props, but when you have a clear performance impact such as an expensive calculation, it is worth it to prevent re-renders. Further, since React performs a shallow comparison, the component will still re-render when a prop is an object or a function. You will read more about handling functions as props in Step 3.

In this step, you created an application with a long, slow calculation. You learned how parent re-rendering will cause a child component to re-render

and how to prevent the re-render using `memo`. In the next step, you'll memoize actions in a component so that you only perform actions when specific properties change.

Step 2 — Caching Expensive Data Calculations with `useMemo`

In this step, you'll store the results of slow data calculations with the `useMemo` Hook. You'll then incorporate the `useMemo` Hook in an existing component and set conditions for data re-calculations. By the end of this step, you'll be able to cache expensive functions so that they will only run when specific pieces of data change.

In the previous step, the toggled explanation of the component was part of the parent. However, you could instead add it to the `CharacterMap` component itself. When you do, `CharacterMap` will have two properties, the `text` and `showExplanation`, and it will display the explanation when `showExplanation` is truthy.

To start, open `CharacterMap.js`:

```
nano src/components/CharacterMap/CharacterMap.js
```

Inside of `CharacterMap`, add a new property of `showExplanation`. Display the explanation text when the value of `showExplanation` is truthy:

performance-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React, { memo } from 'react';
import PropTypes from 'prop-types';

function itemize(text){
  ...
}

function CharacterMap({ showExplanation, text }) {
  return(
    <div>
      {showExplanation &&
        <p>
          This display a list of the most common characters.
        </p>
      }
      Character Map:
      {itemize(text).map(character => (
        <div key={character[0]}>
          {character[0]}: {character[1]}
        </div>
      ))}
    </div>
  )
}
```

```
CharacterMap.propTypes = {
  showExplanation: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired
}

export default memo(CharacterMap);
```

Save and close the file.

Next, open `App.js`:

```
nano src/components/App/App.js
```

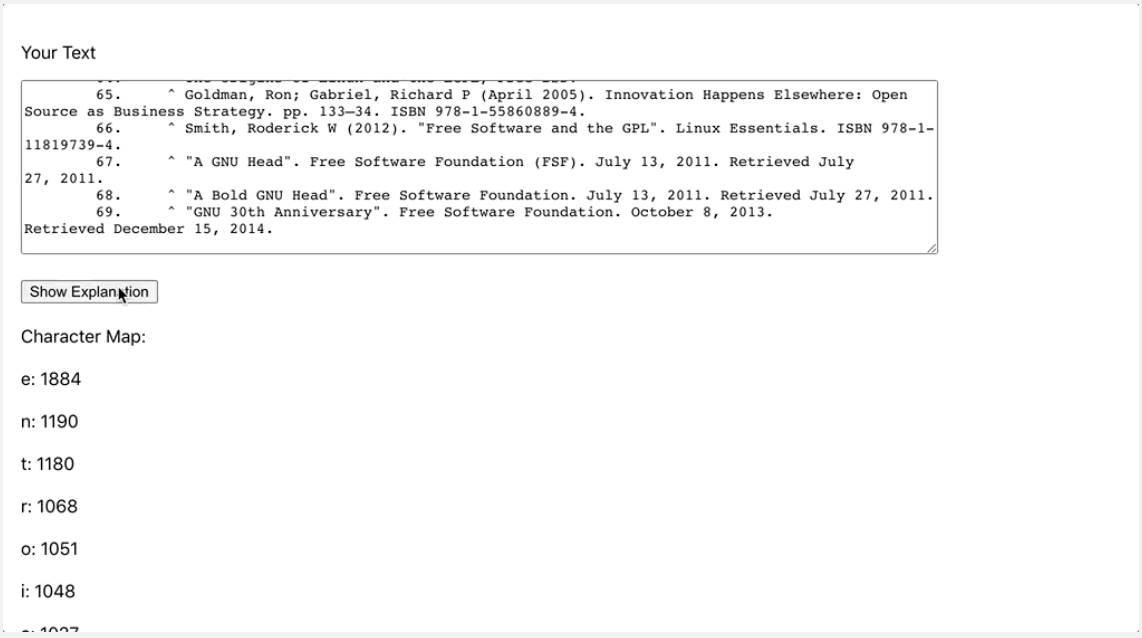
Remove the paragraph of explanation and pass `showExplanation` as a prop to `CharacterMap`:

performance-tutorial/src/components/App/App.js

```
import React, { useReducer, useState } from 'react';
import './App.css';

import CharacterMap from '../CharacterMap/CharacterMap';

function App() {
  const [text, setText] = useState('');
  const [showExplanation, toggleExplanation] = useReducer(state

  return(
    <div className="wrapper">
      <label htmlFor="text">
        <p>Your Text</p>
        <textarea
          id="text"
          name="text"
          rows="10"
          cols="100"
          onChange={event => setText(event.target.value)}
        >
        </textarea>
      </label>
      <div>
        <button onClick={toggleExplanation}>Show Explanation</b
      </div>
```

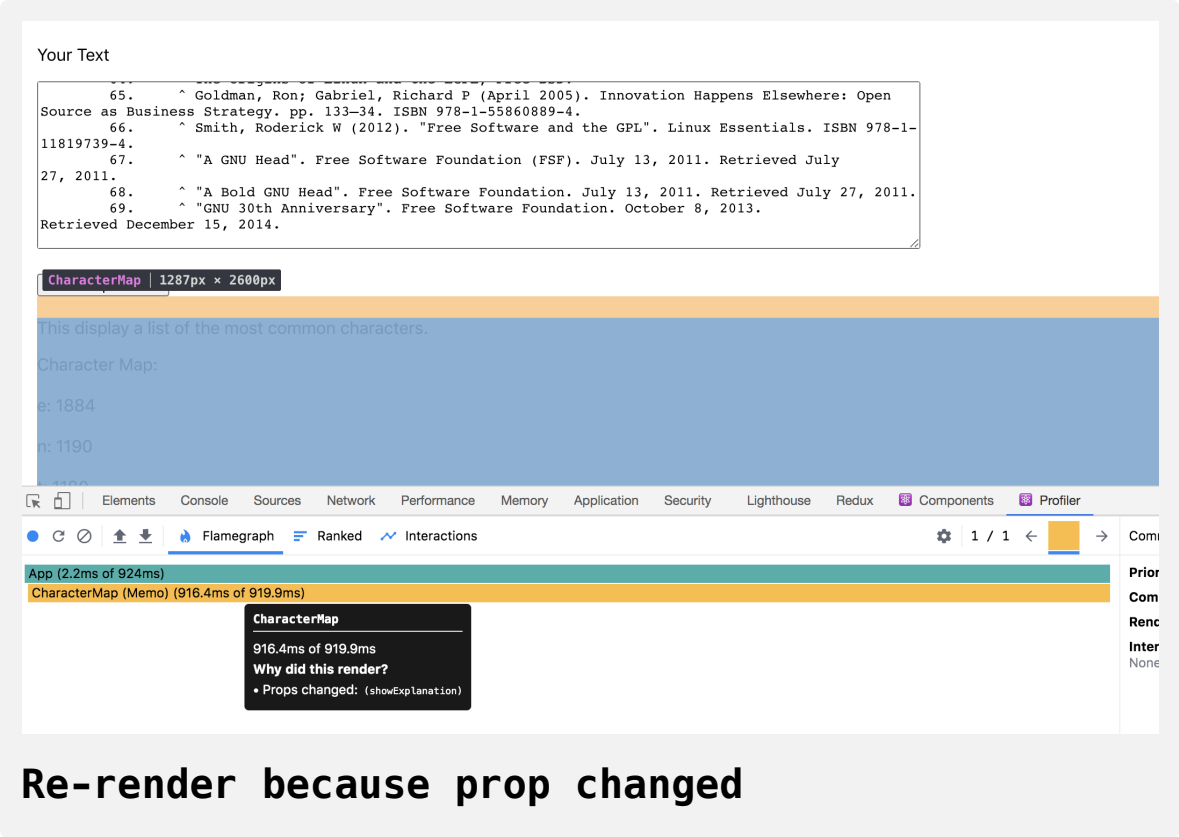
```
      <CharacterMap showExplanation={showExplanation} text={tex
    </div>
  )
}

export default App;
```

Save and close the file. When you do, the browser will refresh. If you toggle the explanation, you will again receive the delay.



Delay when toggling explanation

If you look at the profiler, you'll discover that the component re-rendered because the `showExplanation` prop changed:



Re-render because prop changed

The `memo` function will compare props and prevent re-renders if no props change, but in this case the `showExplanation` prop does change, so the whole component will re-render and the component will re-run the `itemize` function.

In this case, you need to memoize specific parts of the component, not the whole component. React provides a special Hook called `useMemo` that you can use to preserve parts of your component across re-renders. The Hook takes two arguments. The first argument is a function that will return the value you want to memoize. The second argument is an array of dependencies. If a dependency changes, `useMemo` will re-run the function and return a value.

To implement `useMemo`, first open `CharacterMap.js`:

```
nano src/components/CharacterMap/CharacterMap.js
```

Declare a new variable called `characters`. Then call `useMemo` and pass an anonymous function that returns the value of `itemize(text)` as the first argument and an array containing `text` as the second argument. When `useMemo` runs, it will return the result of `itemize(text)` to the `characters` variable.

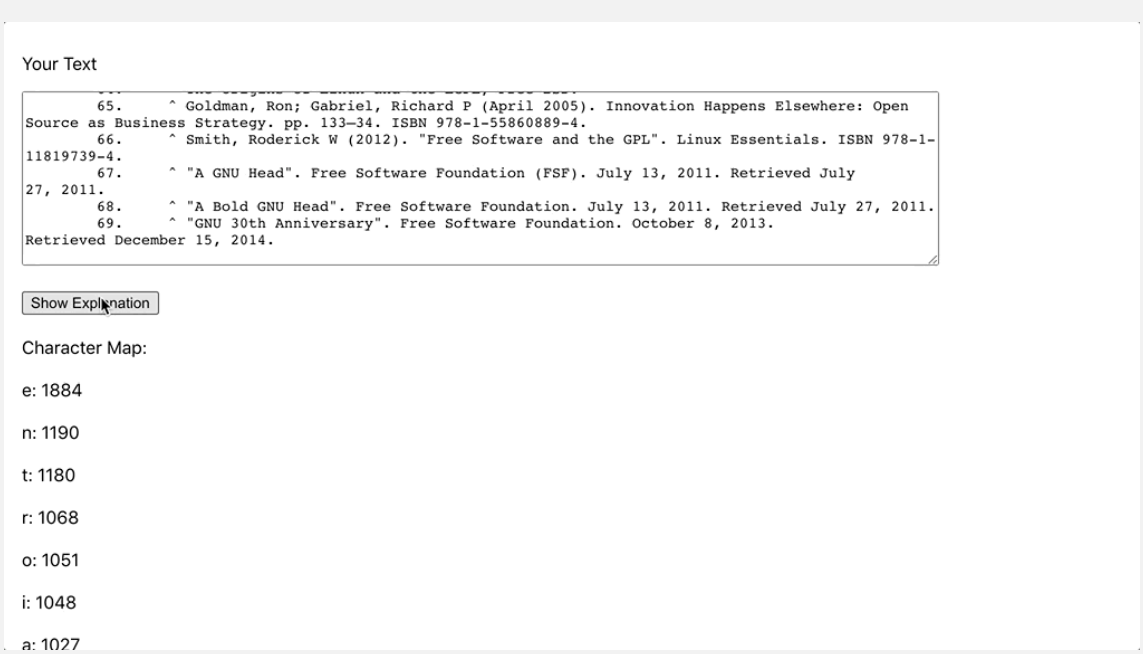
Replace the call to `itemize` in the JSX with `characters`:

performance-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React, { memo, useMemo } from 'react';
import PropTypes from 'prop-types';

function itemize(text){
  ...
}

function CharacterMap({ showExplanation, text }) {
  const characters = useMemo(() => itemize(text), [text]);
  return(
    <div>
      {showExplanation &&
        <p>
          This display a list of the most common characters.
        </p>
      }
      Character Map:
      {characters.map(character => (
        <div key={character[0]}>
          {character[0]}: {character[1]}
        </div>
      ))}
    </div>
  )
```

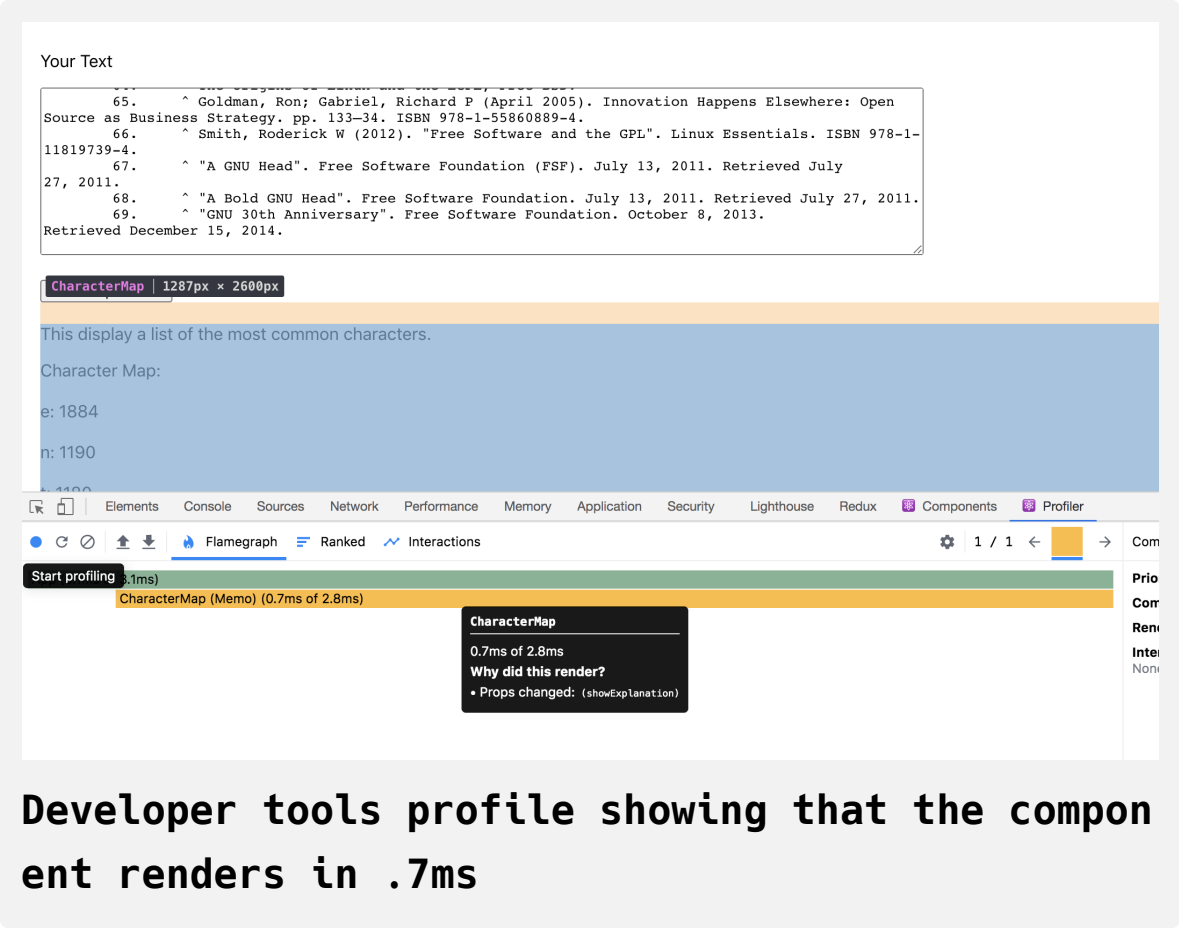
```
}


CharacterMap.propTypes = {
  showExplanation: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired
}


export default memo(CharacterMap);
```

Save the file. When you do, the browser will reload and there will be no delay when you toggle the explanation.



Animation showing no delay when toggling explanation

If you profile the component, you will still find that it re-renders, but the time it takes to render will be much shorter. In this example it took .7 milliseconds compared to 916.4 milliseconds without the `useMemo` Hook. That's because React is re-rendering the component, but it is not re-running the function contained in the `useMemo` Hook. You're able to preserve the result while still allowing other parts of the component to update:



Developer tools profile showing that the component renders in .7ms

If you change the text in the textbox, there will still be a delay because the dependency—`text`—changed, so `useMemo` will re-run the function. If it did not re-run, you would have old data. The key point is that it only runs when the data it needs changes.

In this step, you memoized parts of your component. You isolated an expensive function from the rest of the component and used the `useMemo` Hook to run the function only when certain dependencies change. In the next step, you'll memoize functions to prevent shallow comparison re-renders.

Step 3 — Managing Function Equality Checks with `useCallback`

In this step, you'll handle props that are difficult to compare in JavaScript. React uses strict equality checking when props change. This check determines when to re-run Hooks and when to re-render components. Since JavaScript functions and objects are difficult to compare, there are situations where a prop would be effectively the same, but still trigger a re-render.

You can use the `useCallback` Hook to preserve a function across re-renders. This will prevent unnecessary re-renders when a parent component recreates a function. By the end of this step, you'll be able to prevent re-renders using the `useCallback` Hook.

As you build your `CharacterMap` component, you may have a situation where you need it to be more flexible. In the `itemize` function, you always convert the character to lower case, but some consumers of the component may not want that functionality. They may want to compare upper and lowercase characters or want to convert all characters to upper case.

To facilitate this, add a new prop called `transformer` that will change the character. The `transformer` function will be anything that takes a character as an argument and returns a string of some sort.

Inside of `CharacterMap`, add `transformer` as a prop. Give it a `PropType` of function with a default of `null`:

performance-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React, { memo, useMemo } from 'react';
import PropTypes from 'prop-types';

function itemize(text){
  const letters = text.split('')
    .filter(l => l !== ' ')
    .reduce((collection, item) => {
      const letter = item.toLowerCase();
      return {
        ...collection,
        [letter]: (collection[letter] || 0) + 1
      }
    }, {})
  return Object.entries(letters)
    .sort((a, b) => b[1] - a[1]);
}

function CharacterMap({ showExplanation, text, transformer }) {
  const characters = useMemo(() => itemize(text), [text]);
  return(
    <div>
      {showExplanation &&
        <p>
          This display a list of the most common characters.
```

```
        </p>
      }
      Character Map:
      {characters.map(character => (
        <div key={character[0]}>
          {character[0]}: {character[1]}
        </div>
      ))}
    </div>
  )
}

CharacterMap.propTypes = {
  showExplanation: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired,
  transformer: PropTypes.func
}

CharacterMap.defaultProps = {
  transformer: null
}

export default memo(CharacterMap);
```

Next, update `itemize` to take `transformer` as an argument. Replace the `.toLowerCase` method with the transformer. If `transformer` is truthy, call the function with the `item` as an argument. Otherwise, return the `item`:

performance-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React, { memo, useMemo } from 'react';
import PropTypes from 'prop-types';

function itemize(text, transformer){
  const letters = text.split('')
    .filter(l => l !== ' ')
    .reduce((collection, item) => {
      const letter = transformer ? transformer(item) : item;
      return {
        ...collection,
        [letter]: (collection[letter] || 0) + 1
      }
    }, {})
  return Object.entries(letters)
    .sort((a, b) => b[1] - a[1]);
}

function CharacterMap({ showExplanation, text, transformer }) {
    ...
}

CharacterMap.propTypes = {
  showExplanation: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired,
```

```
  transformer: PropTypes.func
}

CharacterMap.defaultProps = {
  transformer: null
}

export default memo(CharacterMap);
```

Finally, update the `useMemo` Hook. Add `transformer` as a dependency and pass it to the `itemize` function. You want to be sure that your dependencies are exhaustive. That means you need to add anything that might change as a dependency. If a user changes the `transformer` by toggling between different options, you'd need to re-run the function to get the correct value.

performance-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React, { memo, useMemo } from 'react';
import PropTypes from 'prop-types';

function itemize(text, transformer){
  ...
}

function CharacterMap({ showExplanation, text, transformer }) {
  const characters = useMemo(() => itemize(text, transformer),
  return(
    <div>
      {showExplanation &&
        <p>
          This display a list of the most common characters.
        </p>
      }
      Character Map:
      {characters.map(character => (
        <div key={character[0]}>
          {character[0]}: {character[1]}
        </div>
      ))}
    </div>
  )
```

```
}

CharacterMap.propTypes = {
  showExplanation: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired,
  transformer: PropTypes.func
}

CharacterMap.defaultProps = {
  transformer: null
}

export default memo(CharacterMap);
```

Save and close the file.

In this application, you don't want to give users the ability to toggle between different functions. But you do want the characters to be lower case. Define a `transformer` in `App.js` that will convert the character to lower case. This function will never change, but you do need to pass it to the `CharacterMap`.

Open `App.js`:

```
nano src/components/App/App.js
```

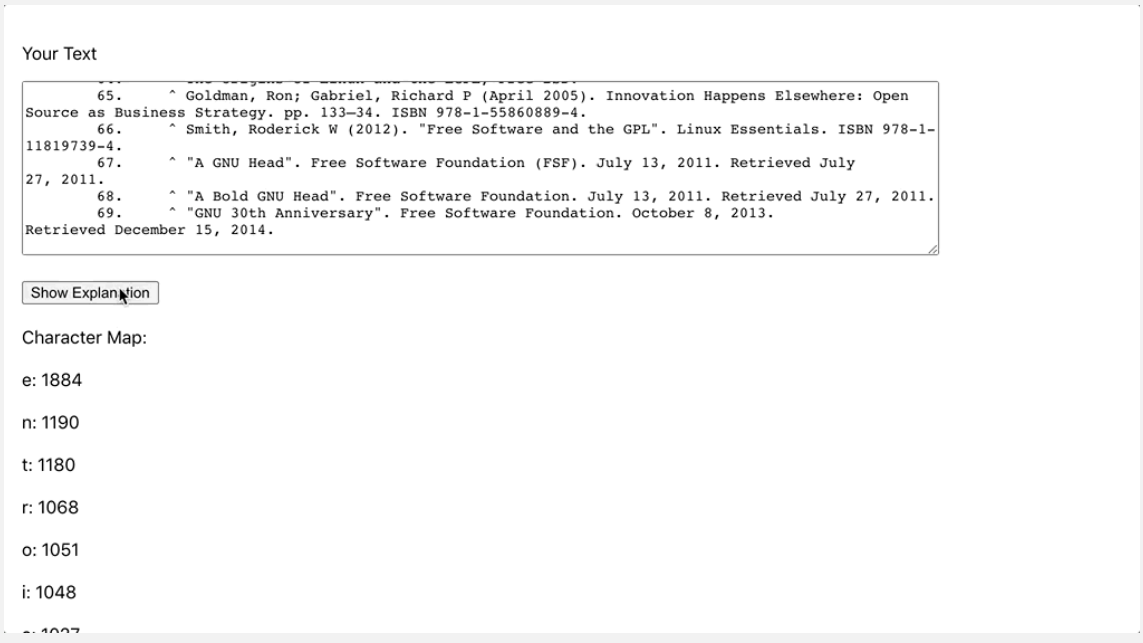
Then define a function called `transformer` that converts a character to lower case. Pass the function as a prop to `CharacterMap`:

performance-tutorial/src/components/CharacterMap/CharacterMap.js

```
import React, { useReducer, useState } from 'react';
import './App.css';

import CharacterMap from '../CharacterMap/CharacterMap';

function App() {
  const [text, setText] = useState('');
  const [showExplanation, toggleExplanation] = useReducer(state
  const transformer = item => item.toLowerCase();

  return(
    <div className="wrapper">
      <label htmlFor="text">
        <p>Your Text</p>
        <textarea
          id="text"
          name="text"
          rows="10"
          cols="100"
          onChange={event => setText(event.target.value)}
        >
        </textarea>
      </label>
      <div>
```

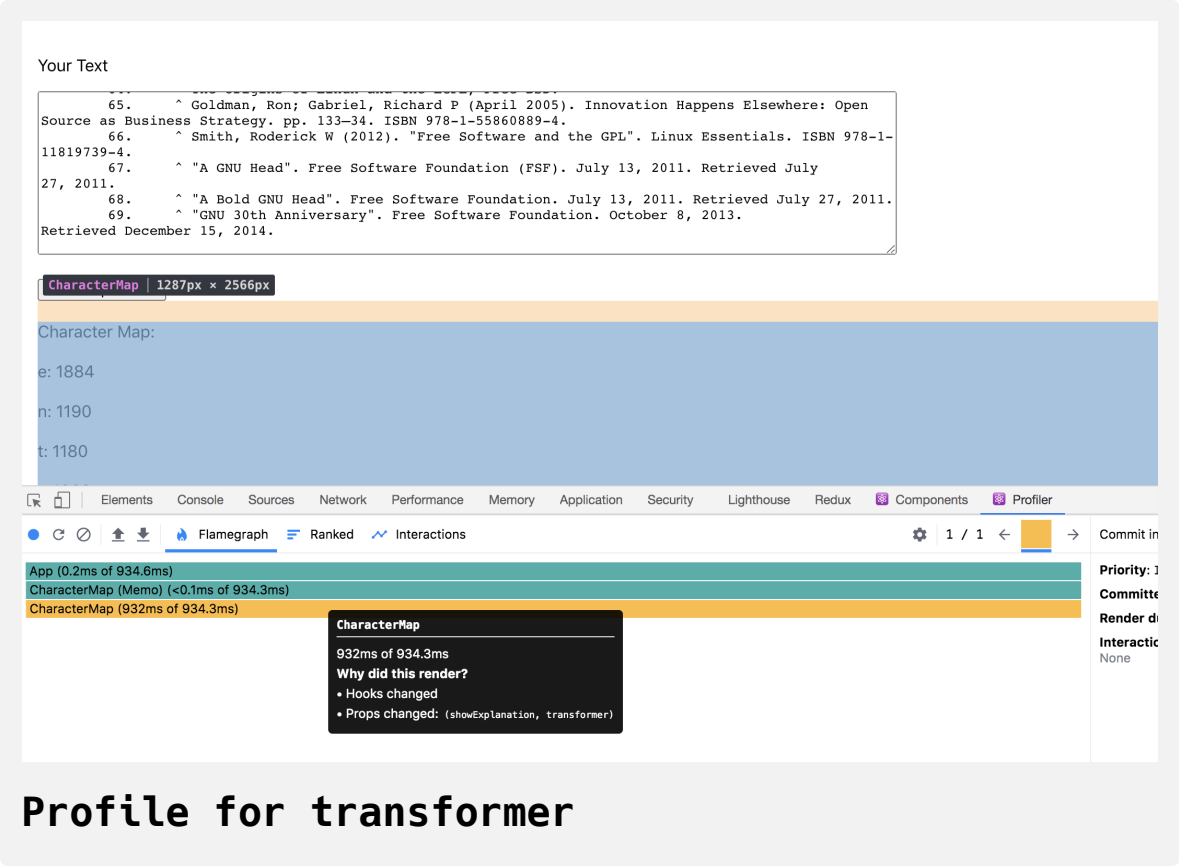
```
        <button onClick={toggleExplanation}>Show Explanation</b
      </div>
      <CharacterMap showExplanation={showExplanation} text={tex
        transformer={transformer} />
    </div>
  )
}


export default App;
```

Save the file. When you do, you will find that the delay has returned when you toggle the explanation.



Animation showing a delay when toggling explanation

If you profile the component, you will find that the component re-renders because the props change and the Hooks changed:



Profile for transformer

If you look closely, you'll find that the `showExplanation` prop changed, which makes sense because you clicked the button, but the `transformer` prop also changed.

When you made a state change in `App` by clicking on the button, the `App` component re-rendered and redeclared the `transformer`. Even though the function is the same, it is not referentially identical to the previous function. That means it's not strictly identical to the previous function.

If you open the browser console and compared identical functions, you'd find that the comparison is false, as shown in the following code block:

```
const a = () => {};
const b = () => {};

a === a
// This will evaluate to true

a === b
// This will evaluate to false
```

Using the `===` comparison operator, this code shows that two functions are not considered equal, even if they have the same values.

To avoid this problem, React provides a Hook called `useCallback`. The Hook is similar to `useMemo`: it takes a function as the first argument and an array of dependencies as the second argument. The difference is that `useCallback` returns the function and not the result of the function. Like the `useMemo` Hook, it will not recreate the function unless a dependency changes. That means that the `useMemo` Hook in `CharacterMap.js` will compare the same value and the Hook will not re-run.

Inside of `App.js`, import `useCallback` and pass the anonymous function as the first argument and an empty array as the second argument. You never

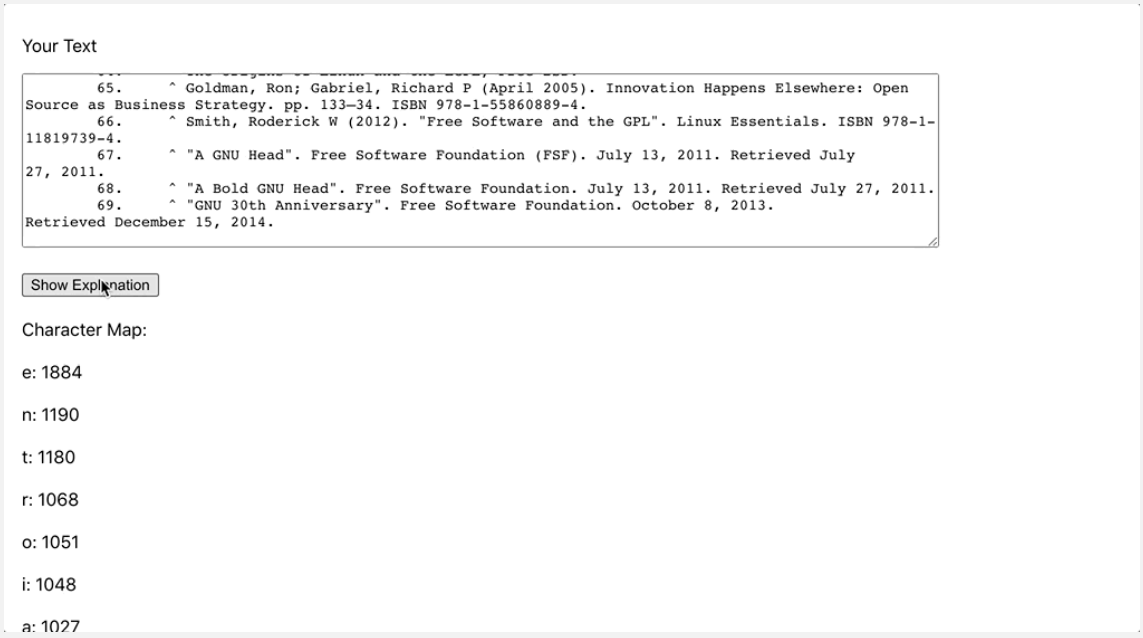
want `App` to recreate this function:

performance-tutorial/src/components/App/App.js

```
import React, { useCallback, useReducer, useState } from 'react
import './App.css';

import CharacterMap from '../CharacterMap/CharacterMap';

function App() {
  const [text, setText] = useState('');
  const [showExplanation, toggleExplanation] = useReducer(state
  const transformer = useCallback(item => item.toLowerCase(), [

  return(
    <div className="wrapper">
      <label htmlFor="text">
        <p>Your Text</p>
        <textarea
          id="text"
          name="text"
          rows="10"
          cols="100"
          onChange={event => setText(event.target.value)}
        >
        </textarea>
      </label>
      <div>
        <button onClick={toggleExplanation}>Show Explanation</b
```

```
      </div>
      <CharacterMap showExplanation={showExplanation} text={tex
        transformer={transformer} />
    </div>
  )
}


export default App;
```

Save and close the file. When you do, you'll be able to toggle the explanation without re-running the function.



Animation showing no delay when toggling the explanation component

If you profile the component, you'll find that the Hook no longer runs:

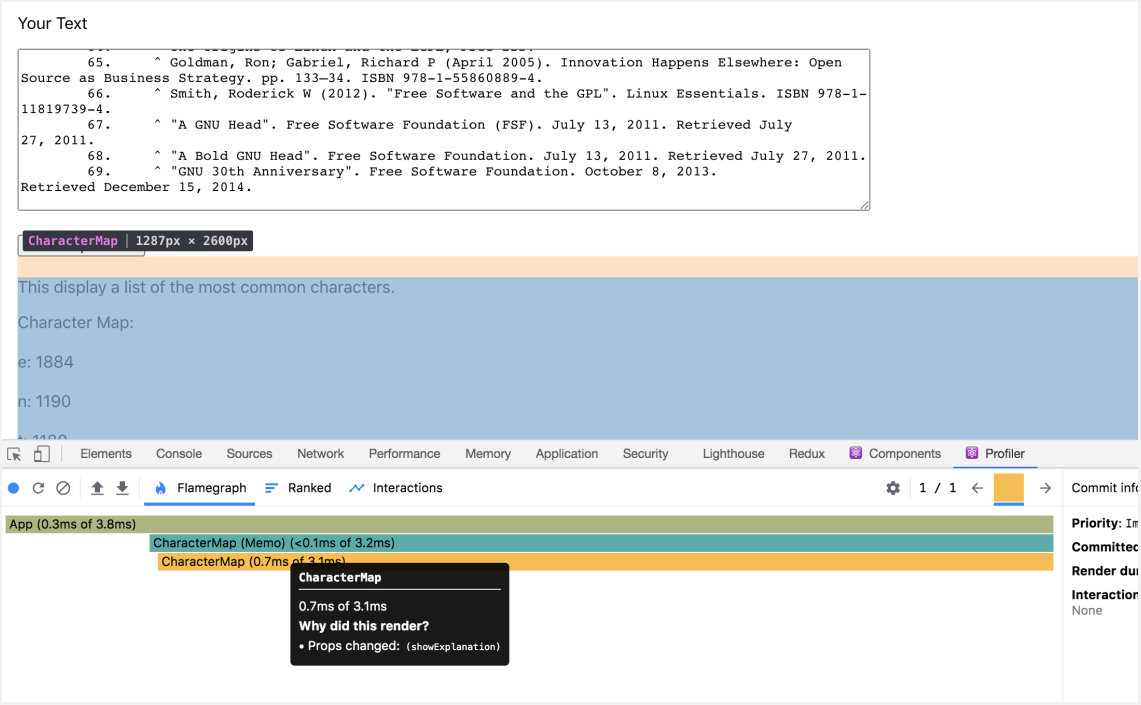


Image of the browser developer tools showing th at the Hook does not run

In this particular component, you do not actually need the `useCallback` Hook. You could declare the function outside of the component and it would never re-render. You should only declare functions inside of your component if they require some sort of prop or stateful data. But there are times when you need to create functions based on internal state or props and in those situations you can use the `useCallback` Hook to minimize re-renders.

In this step, you preserved functions across re-renders using the `useCallbac k` Hook. You also learned how those functions will retain equality when

compared as props or dependencies in a Hook.

Conclusion

You now have the tools to improve performance on expensive components. You can use `memo`, `useMemo`, and `useCallback` to avoid costly component re-renders. But all these strategies include a performance cost of their own. `memo` will take extra work to compare properties, and the Hooks will need to run extra comparisons on each render. Only use these tools when there is a clear need in your project, otherwise you risk adding in your own latency.

Finally, not all performance problems require a technical fix. There are times when the performance cost is unavoidable—such as slow APIs or large data conversions—and in those situations you can solve the problem using design by either rendering loading components, showing placeholders while asynchronous functions are running, or other enhancements to the user experience.

If you would like to read more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Deploy a React Application with Nginx on Ubuntu 20.04

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

You can quickly deploy React applications to a server using the default Create React App build tool. The `build` script compiles the application into a single directory containing all of the JavaScript code, images, styles, and HTML files. With the assets in a single location, you can deploy to a web server with minimal configuration.

In this tutorial, you'll deploy a React application on your local machine to an Ubuntu 20.04 server running Nginx. You'll build an application using Create React App, use an Nginx config file to determine where to deploy files, and securely copy the build directory and its contents to the server. By the end of this tutorial, you'll be able to build and deploy a React application.

Prerequisites

- On your local machine, you will need a development environment running Node.js; this tutorial was tested on Node.js version 10.22.0 and npm version 6.14.6. To install this on macOS or Ubuntu 20.04, follow the steps in How to Install Node.js and Create a Local

Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 20.04.

- One Ubuntu 20.04 server for deployment, set up by following this initial server setup for Ubuntu 20.04 tutorial, including a sudo-enabled non-**root** user, a firewall, and SSH access from your local machine. To gain SSH access on a DigitalOcean Droplet, read through How to Connect to Droplets with SSH.

- A registered domain name. This tutorial will use `your_domain` throughout. You can purchase a domain name from Namecheap, get one for free with Freenom, or use the domain registrar of your choice.

- Both of the following DNS records set up for your server. If you are using DigitalOcean, please see our DNS documentation for details on how to add them.

    - An A record with `your_domain` pointing to your server's public IP address.
    - An A record with `www.your_domain` pointing to your server's public IP address.

- Nginx installed by following How To Install Nginx on Ubuntu 20.04. Be sure that you have a server block for your domain. This tutorial will use `/etc/nginx/sites-available/your_domain` as an example.

- It is recommended that you also secure your server with an HTTPS certificate. You can do this with the How To Secure Nginx with Let's Encrypt on Ubuntu 20.04 tutorial.

- You will also need a basic knowledge of JavaScript, HTML, and CSS, which you can find in our How To Build a Website With HTML series, How To Build a Website With CSS series, and in How To Code in JavaScript.

Step 1 — Creating a React Project

In this step, you'll create an application using Create React App and build a deployable version of the boilerplate app.

To start, create a new application using Create React App in your local environment. In a terminal, run the command to build an application. In this tutorial, the project will be called `react-deploy`:

```
npx create-react-app react-deploy
```

The `npx` command will run a Node package without downloading it to your machine. The `create-react-app` script will install all of the dependencies needed for your React app and will build a base project in the `react-deploy` directory. For more on Create React App, check out out the tutorial How To Set Up a React Project with Create React App.

The code will run for a few minutes as it downloads and installs the dependencies. When it is complete, you will receive a success message. Your version may be slightly different if you use `yarn` instead of `npm`:

Output

```
Success! Created react-deploy at your_file_path/react-deploy
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm eject
    Removes this tool and copies build dependencies, configura
tion files
    and scripts into the app directory. If you do this, you ca
n’t go back!

We suggest that you begin by typing:

  cd react-deploy
  npm start

Happy hacking!
```

Following the suggestion in the output, first move into the project folder:

```
cd react-deploy
```

Now that you have a base project, run it locally to test how it will appear on the server. Run the project using the `npm start` script:

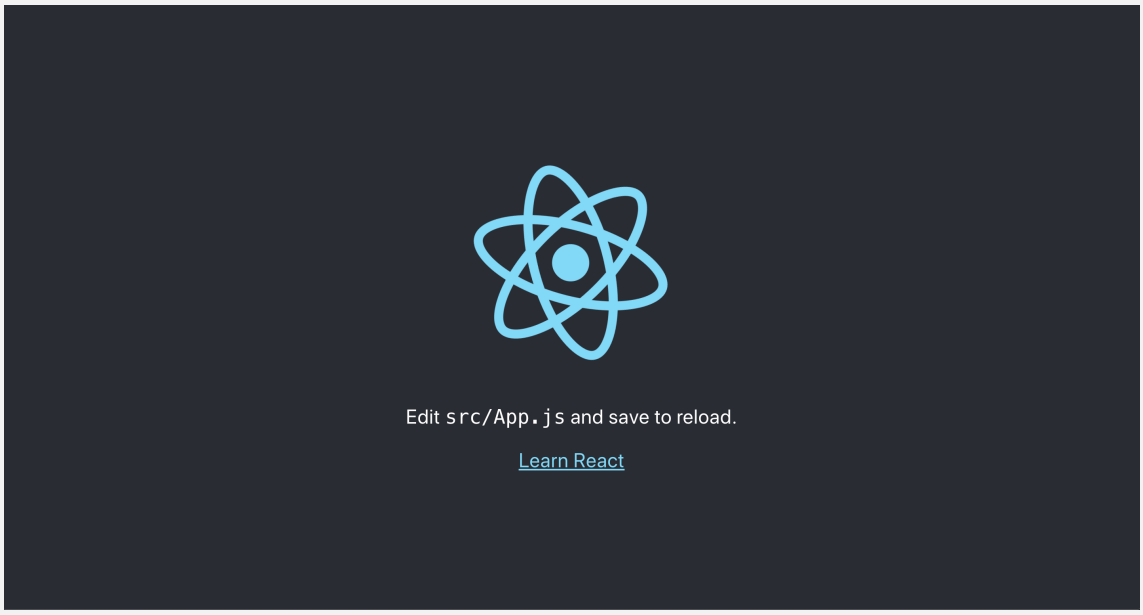
```
npm start
```

When the command runs, you'll receive output with the local server info:

Output

```
Compiled successfully!

You can now view react-deploy in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.1.110:3000

Note that the development build is not optimized.
To create a production build, use npm build.
```

Open a browser and navigate to `http://localhost:3000`. You will be able to access the boilerplate React app:



React project template running locally

Stop the project by entering either `CTRL+C` or `⌘+C` in a terminal.

Now that you have a project that runs successfully in a browser, you need to create a production build. Run the `create-react-app` build script with the following:

```
npm run build
```

This command will compile the JavaScript and assets into the `build` directory. When the command finishes, you will receive some output with data about your build. Notice that the filenames include a hash, so your output will be slightly different:

Output

```
Creating an optimized production build...
Compiled successfully.

File sizes after gzip:

  41.21 KB  build/static/js/2.82f639e7.chunk.js
  1.4 KB    build/static/js/3.9fbaa076.chunk.js
  1.17 KB   build/static/js/runtime-main.1caef30b.js
  593 B     build/static/js/main.e8c17c7d.chunk.js
  546 B     build/static/css/main.ab7136cd.chunk.css

The project was built assuming it is hosted at /.
You can control this with the homepage field in your package.j
son.

The build folder is ready to be deployed.
You may serve it with a static server:

  serve -s build

Find out more about deployment here:

  https://cra.link/deployment
```

The `build` directory will now include compiled and minified versions of all the files you need for your project. At this point, you don't need to worry about anything outside of the `build` directory. All you need to do is deploy the directory to a server.

In this step, you created a new React application. You verified that the application runs locally and you built a production version using the Create React App `build` script. In the next step, you'll log onto your server to learn where to copy the `build` directory.

Step 2 — Determining Deployment File Location on your Ubuntu Server

In this step, you'll start to deploy your React application to a server. But before you can upload the files, you'll need to determine the correct file location on your deployment server. This tutorial uses Nginx as a web server, but the approach is the same with Apache. The main difference is that the configuration files will be in a different directory.

To find the directory the web server will use as the root for your project, log in to your server using `ssh`:

```
ssh username@server_ip
```

Once on the server, look for your web server configuration in `/etc/nginx/sites-enabled`. There is also a directory called `sites-allowed`; this directory includes configurations that are not necessarily activated. Once

you find the configuration file, display the output in your terminal with the following command:

```
cat /etc/nginx/sites-enabled/your_domain
```

If your site has no HTTPS certificate, you will receive a result similar to this:

Output

```
server {
        listen 80;
        listen [::]:80;

        root /var/www/your_domain/html;
        index index.html index.htm index.nginx-debian.html;

        server_name your_domain www.your_domain;

        location / {
                try_files $uri $uri/ =404;
        }
}
```

If you followed the Let’s Encrypt prerequisite to secure your Ubuntu 20.04 server, you will receive this output:

Output

```
server {

        root /var/www/your_domain/html;
        index index.html index.htm index.nginx-debian.html;

        server_name your_domain www.your_domain;

        location / {
                try_files $uri $uri/ =404;
        }

    listen [::]:443 ssl ipv6only=on; # managed by Certbot
    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/your_domain/fullchai
n.pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/your_domain/priv
key.pem; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed
by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed b
y Certbot



}
server {
```

```
    if ($host = www.your_domain) {
        return 301 https://$host$request_uri;
    } # managed by Certbot



    if ($host = your_domain) {
        return 301 https://$host$request_uri;
    } # managed by Certbot



        listen 80;
        listen [::]:80;

        server_name your_domain www.your_domain;
    return 404; # managed by Certbot




}
```

In either case, the most important field for deploying your React app is `root`. This points HTTP requests to the `/var/www/your_domain/html` directory. That means you will copy your files to that location. In the next line, you can see that Nginx will look for an `index.html` file. If you look in your

local `build` directory, you will see an `index.html` file that will serve as the main entry point.

Log off the Ubuntu 20.04 server and go back to your local development environment.

Now that you know the file location that Nginx will serve, you can upload your build.

Step 3 — Uploading Build Files with `scp`

At this point, your `build` files are ready to go. All you need to do is copy them to the server. A quick way to do this is to use `scp` to copy your files to the correct location. The `scp` command is a secure way to copy files to a remote server from a terminal. The command uses your `ssh` key if it is configured. Otherwise, you will be prompted for a username and password.

The command format will be `scp files_to_copy username@server_ip:path_on_server`. The first argument will be the files you want to copy. In this case, you are copying all of the files in the `build` directory. The second argument is a combination of your credentials and the destination path. The destination path will be the same as the `root` in your Nginx config: `/var/www/your_domain/html`.

Copy all the `build` files using the `*` wildcard to `/var/www/your_domain/html`:

```
scp -r ./build/* username@server_ip:/var/www/your_domain/html
```

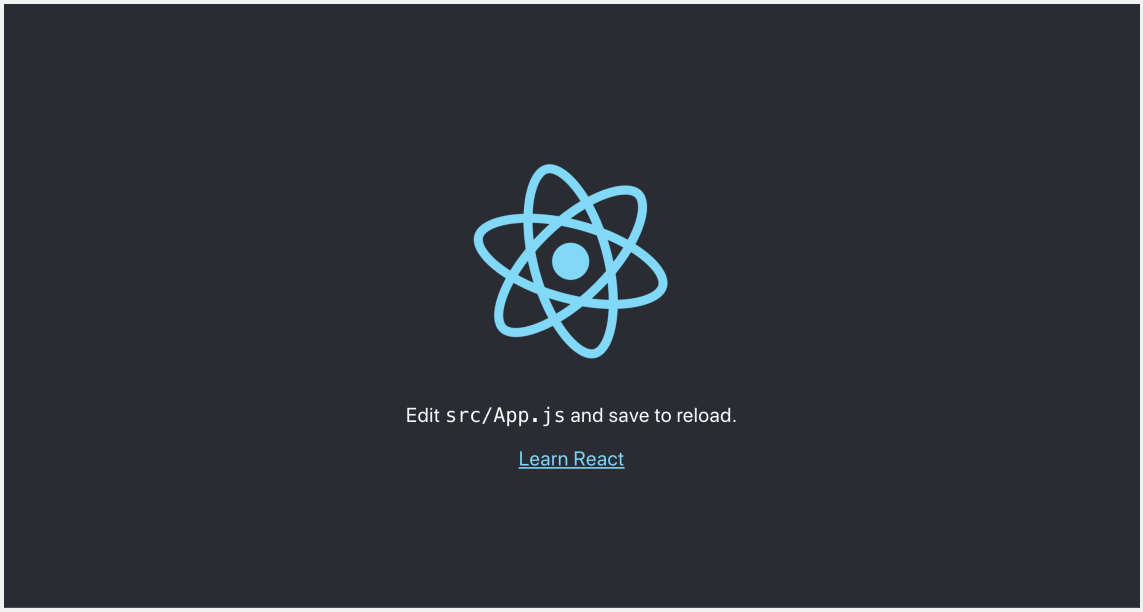
When you run the command, you will receive output showing that your files are uploaded. Your results will be slightly different:

Output

```
asset-manifest.json
100% 1092    22.0KB/s   00:00
favicon.ico
100% 3870    80.5KB/s   00:00
index.html
100% 3032    61.1KB/s   00:00
logo192.png
100% 5347    59.9KB/s   00:00
logo512.png
100% 9664    69.5KB/s   00:00
manifest.json
100%  492    10.4KB/s   00:00
robots.txt
100%   67     1.0KB/s   00:00
main.ab7136cd.chunk.css
100%  943    20.8KB/s   00:00
main.ab7136cd.chunk.css.map
100% 1490    31.2KB/s   00:00
runtime-main.1caef30b.js.map
100%   12KB  90.3KB/s   00:00
3.9fbaa076.chunk.js
100% 3561    67.2KB/s   00:00
2.82f639e7.chunk.js.map
100%  313KB 156.1KB/s   00:02
runtime-main.1caef30b.js
```

```
100% 2372    45.8KB/s   00:00
main.e8c17c7d.chunk.js.map
100% 2436    50.9KB/s   00:00
3.9fbaa076.chunk.js.map
100% 7690   146.7KB/s   00:00
2.82f639e7.chunk.js
100%  128KB 226.5KB/s   00:00
2.82f639e7.chunk.js.LICENSE.txt
100% 1043    21.6KB/s   00:00
main.e8c17c7d.chunk.js
100% 1045    21.7KB/s   00:00
logo.103b5fa1.svg
100% 2671    56.8KB/s   00:00
```

When the command completes, you are finished. Since a React project is built of static files that only need a browser, you don’t have to configure any further server-side language. Open a browser and navigate to your domain name. When you do, you will find your React project:



Browser with React Project on Server

In this step, you deployed a React application to a server. You learned how to identify the root web directory on your server and you copied the files with `scp`. When the files finished uploading, you were able to view your project in a web browser.

Conclusion

Deploying React applications is a quick process when you use Create React App. You run the `build` command to create a directory of all the files you need for a deployment. After running the build, you copy the files to the correct location on the server, pushing your application live to the web.

If you would like to read more React tutorials, check out our React Topic page, or return to the How To Code in React.js series page.

How To Deploy a React Application to DigitalOcean App Platform

Written by Joe Morgan

The author selected Creative Commons to receive a donation as part of the Write for DOnations program.

DigitalOcean's App Platform is a Platform as a Service (PaaS) product that lets you configure and deploy applications from a source repository. After configuring your app, DigitalOcean will build and deploy the application on every change, giving you the benefit of a full web server and deployment pipeline with minimal configuration. This can be a quick and efficient way to deploy your React applications, and if you are using React to build a site with no backend, you can use App Platform's free tier.

In this tutorial, you'll deploy a React application to the DigitalOcean App Platform using the free **Starter** tier. You'll build an application with Create React App, push the code to a GitHub repository, then configure the application as a DigitalOcean app. You'll connect the app to your source code and deploy the project as a set of static files.

By the end of this tutorial, you'll be able to configure a React application to deploy automatically on every push to the main branch of a GitHub repository.

Prerequisites

- On your local machine, you will need a development environment running Node.js; this tutorial was tested on Node.js version 10.22.0 and npm version 6.14.6. To install this on macOS or Ubuntu 20.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 20.04.

- Git installed on your local machine. You can follow the tutorial Contributing to Open Source: Getting Started with Git to install and set up Git on your computer.

- A DigitalOcean account.

- An account on GitHub, which you can create by going to the **Create your Account** page.

- You will also need a basic knowledge of JavaScript, HTML, and CSS, which you can find in our How To Build a Website With HTML series, How To Build a Website With CSS series, and in How To Code in JavaScript.

Step 1 — Creating a React Project

In this step, you'll create a React application using Create React App and build a deployable version of it.

To start, create a new application using Create React App on your local machine. In a terminal, run the command to build an application called `digital-ocean-app`:

```
npx create-react-app digital-ocean-app
```

The `npx` command will run a Node package without downloading it to your machine. The `create-react-app` script will install all of the dependencies and will build a base project in the `digital-ocean-app` directory. For more on Create React App, check out the tutorial How To Set Up a React Project with Create React App.

The code will download the dependencies and will create a base project. It may take a few minutes to finish. When it is complete, you will receive a success message. Your version may be slightly different if you use `yarn` instead of `npm`:

Output

```
Success! Created digital-ocean-app at your_file_path/digital-ocean-app
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can’t go back!

We suggest that you begin by typing:

  cd digital-ocean-app
  npm start

Happy hacking!
```

Now that you have a base project, run it locally so you can test how the project will appear on the server. First, change into the directory:

```
cd digital-ocean-app
```

Run the project using the `npm start` script:

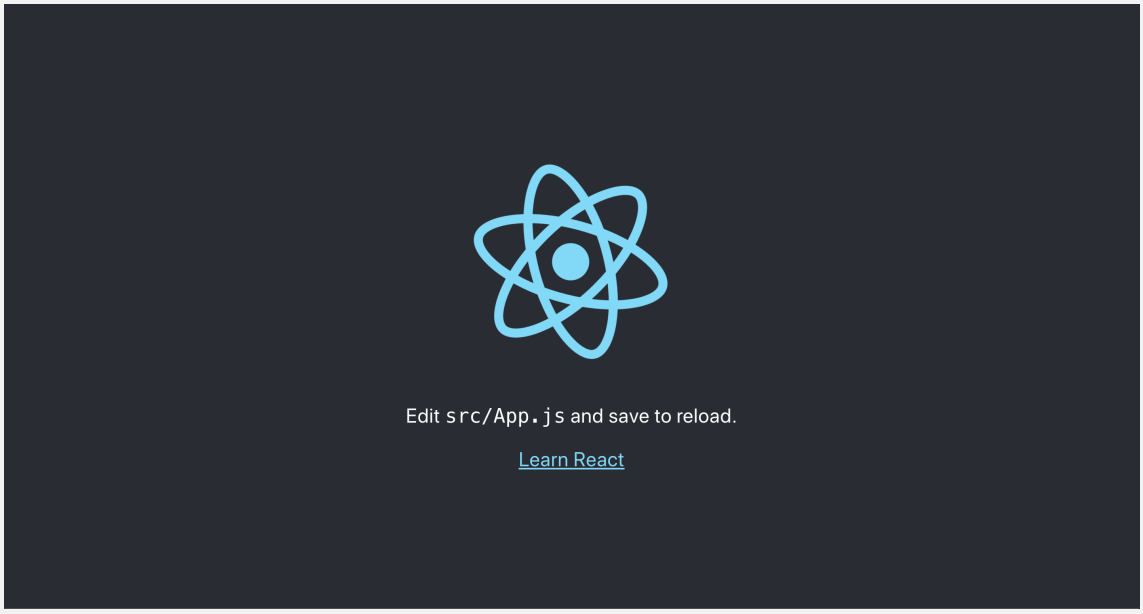
```
npm start
```

When the command runs, you'll receive output with the URL for the local development server:

Output

```
Compiled successfully!

You can now view digital-ocean-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.1.110:3000

Note that the development build is not optimized.
To create a production build, use npm build.
```

Open a browser to `http://localhost:3000` and you'll find your project:



React project template running locally

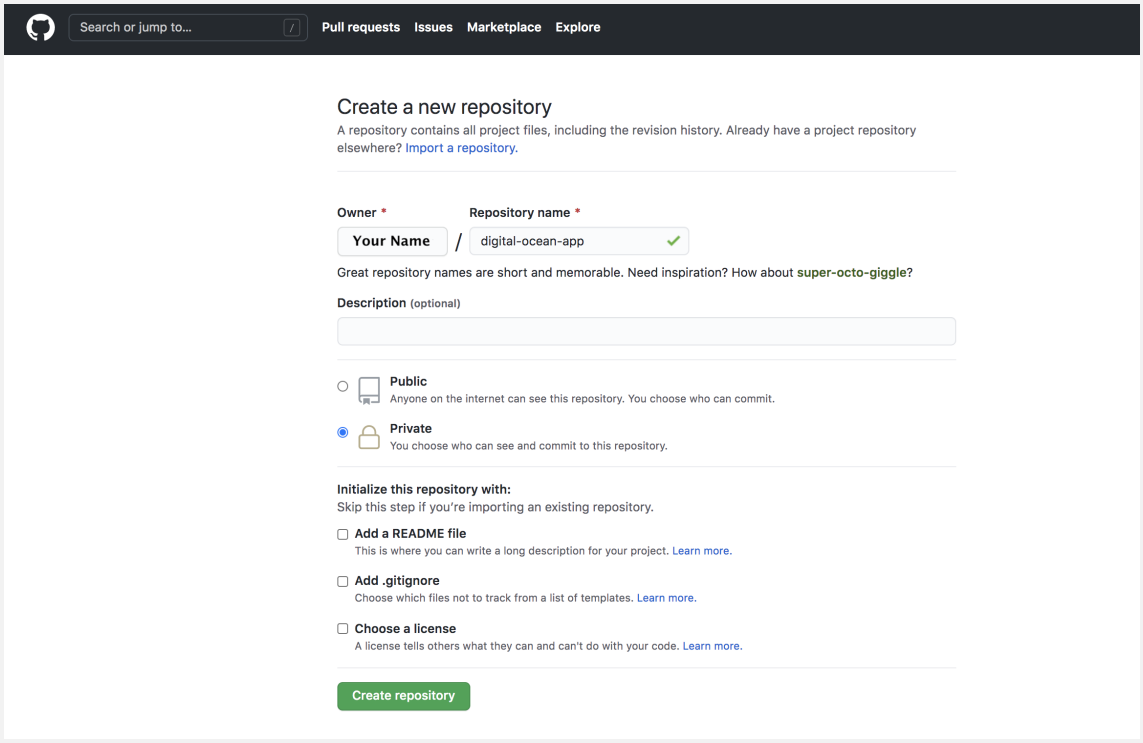
Stop the project by typing either `CTRL+C` or `⌘+C` in the terminal.

Now that you have a working React application, you can push the code to a GitHub repository.

Step 2 — Pushing the Code to GitHub

To deploy your app, App Platform retrieves your source code from a hosted code repository. In this step, you will push your React app code to a GitHub repository so that App Platform can access it later.

Log in to your GitHub account. After you log in, create a new repository called **digital-ocean-app**. You can make the repository either private or public:



New GitHub repository page

Create React App automatically initializes your project with git, so you can set up to push the code directly to GitHub. First, add the repository that you'd like to use with the following command:

```
git remote add origin git@github.com:your_name/digital-ocean-app.git
```

Next, declare that you want to push to the `main` branch with the following:

```
git branch -M main
```

Finally, push the code to your repository:

```
git push -u origin main
```

Enter your GitHub credentials when prompted to push your code.

When you push the code you will receive a success message. Your message will be slightly different:

```
Output
Counting objects: 22, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (22/22), done.
Writing objects: 100% (22/22), 187.50 KiB | 6.94 MiB/s, done.
Total 22 (delta 0), reused 0 (delta 0)
To github.com:your_name/digital-ocean-app.git
   4011c66..647d2e1  main -> main
```

You've now copied your code to the GitHub repository.

In this step, you pushed your project to GitHub so that you can access it using DigitalOcean Apps. Next, you'll create a new DigitalOcean App using your project and set up automatic deployment.

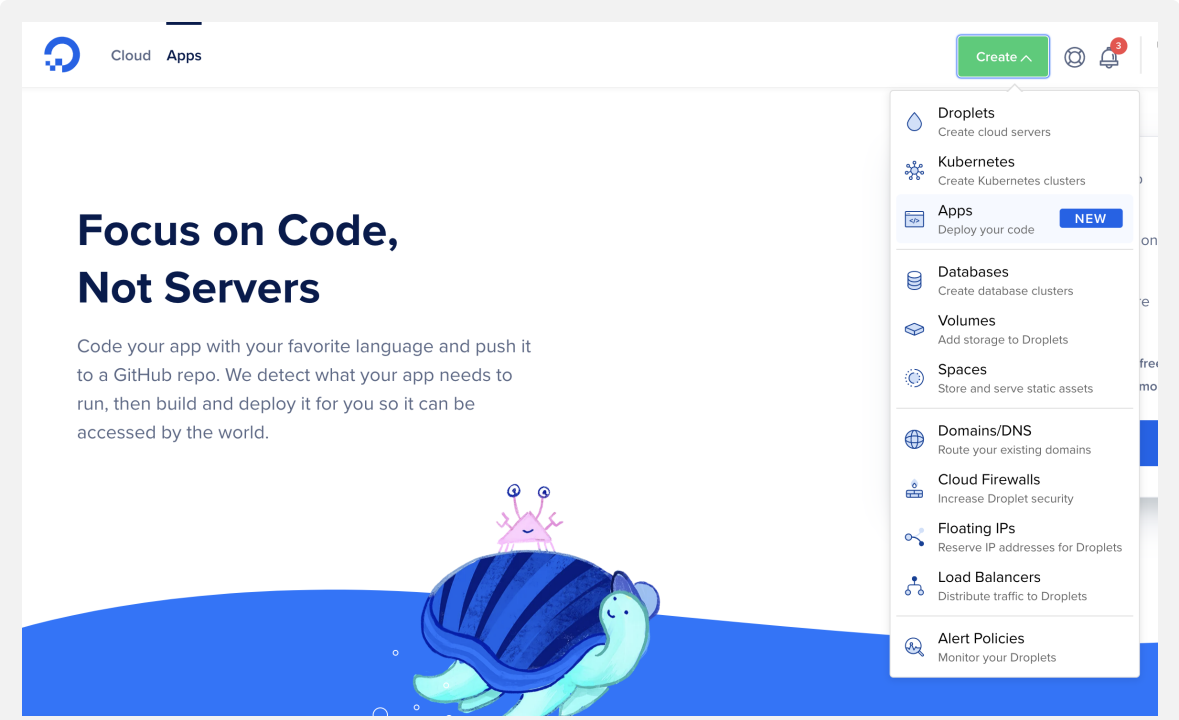
Step 3 — Deploying to DigitalOcean App Platform

In this step, you'll deploy a React application to the DigitalOcean App Platform. You'll connect your GitHub repository to DigitalOcean, configure

the project to build, and build your initial project. After the project is live, each change will trigger a new build and update.

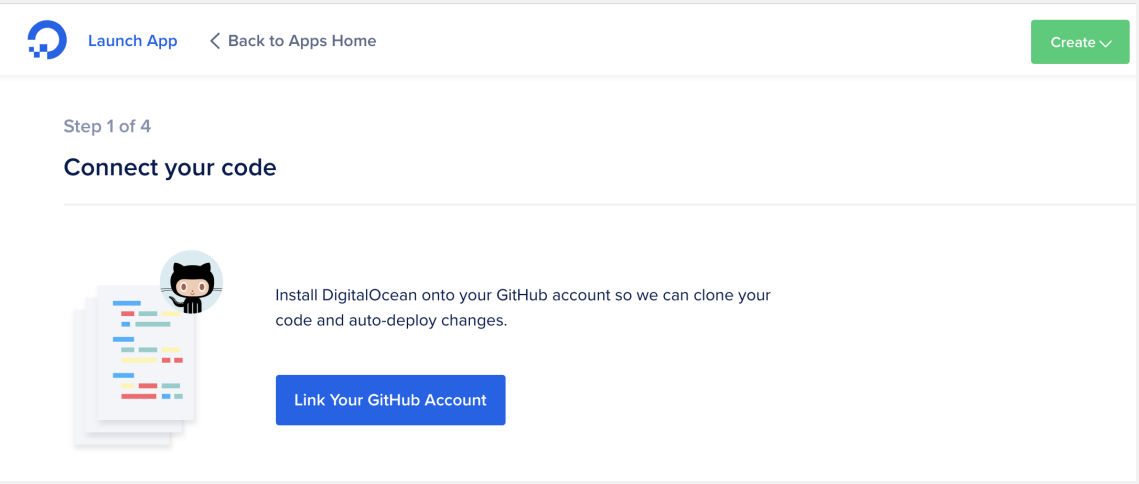
By the end of this step, you'll be able to deploy an application with continous delivery on DigitalOcean.

To begin, log in to your DigitalOcean account and press the **Create** button, then select **Apps**:



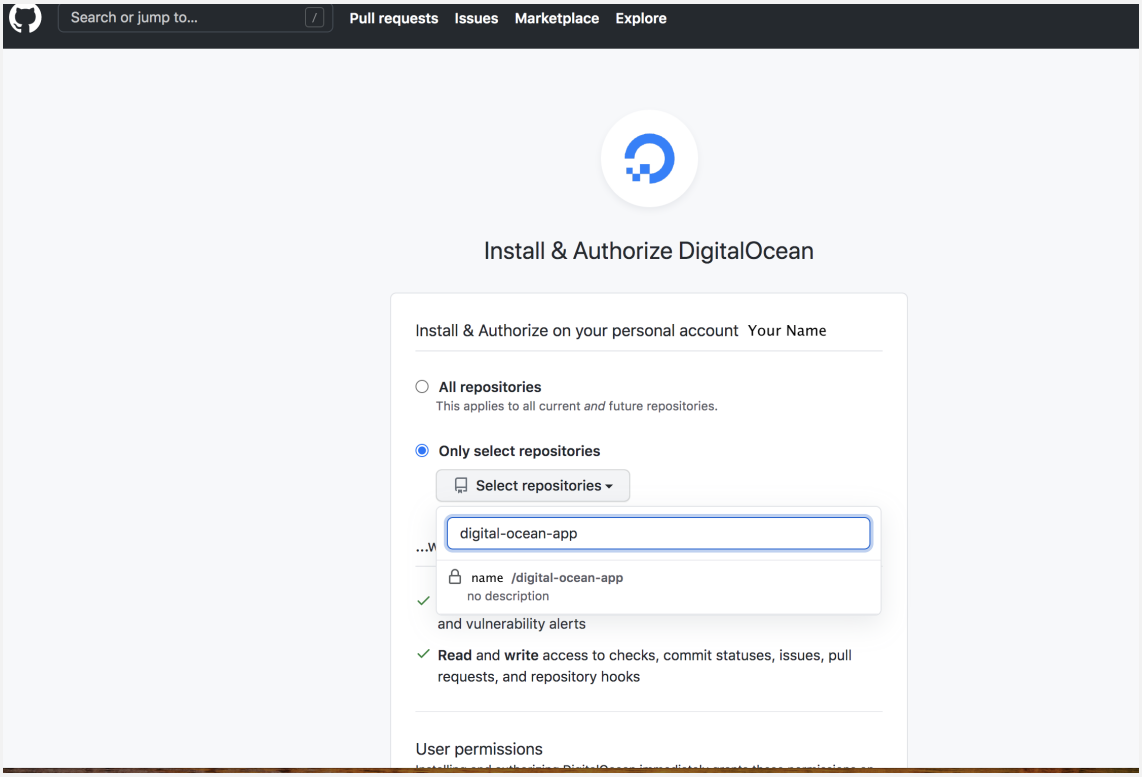
Create a new app page in the DigitalOcean interface

You'll next be prompted to link your GitHub repository. If you have not yet connected it, you will need to log in with your username and password and give DigitalOcean authorization to access your repositories:



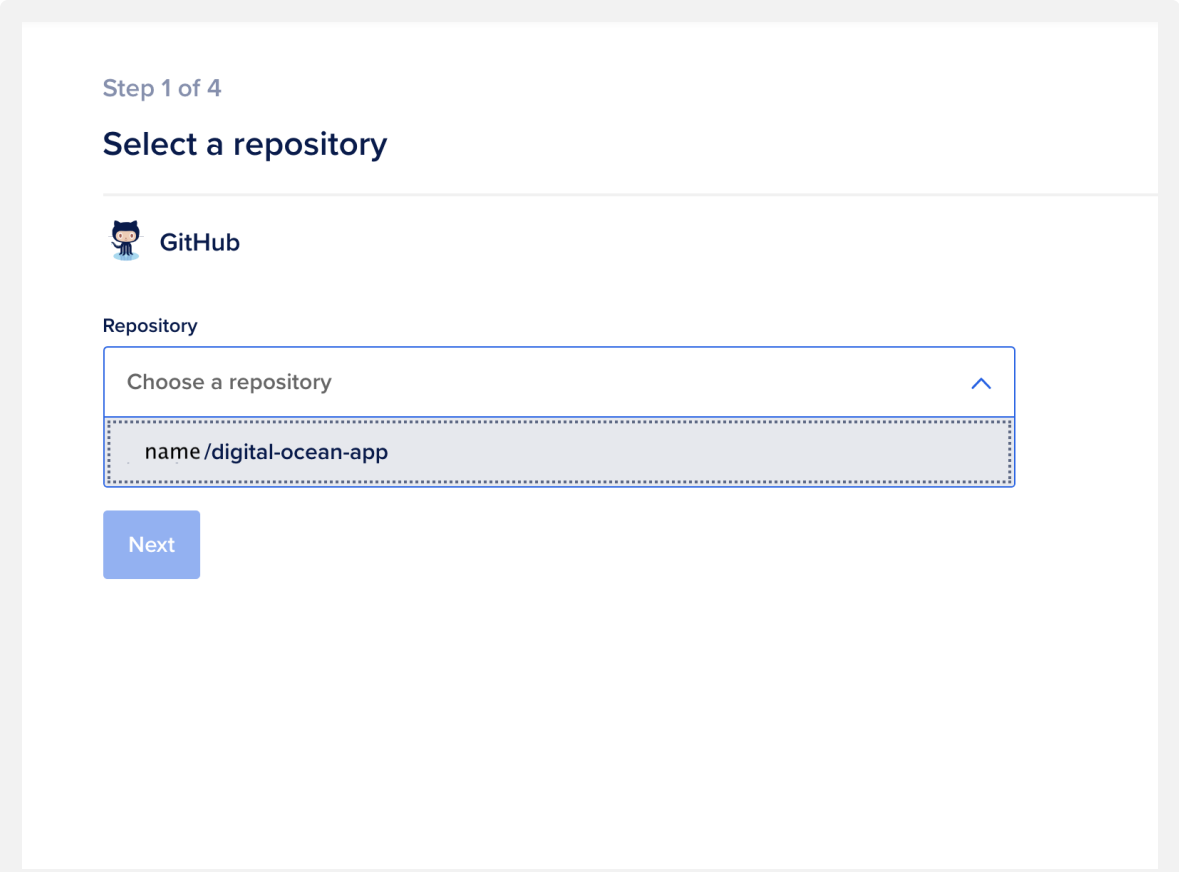
Link GitHub to DigitalOcean page

Once you link your account, select the repository you'd like to connect on the GitHub authorization screen. In this case, you are using the **digital-ocean-app** repository, but you can connect more repositories if you would like:



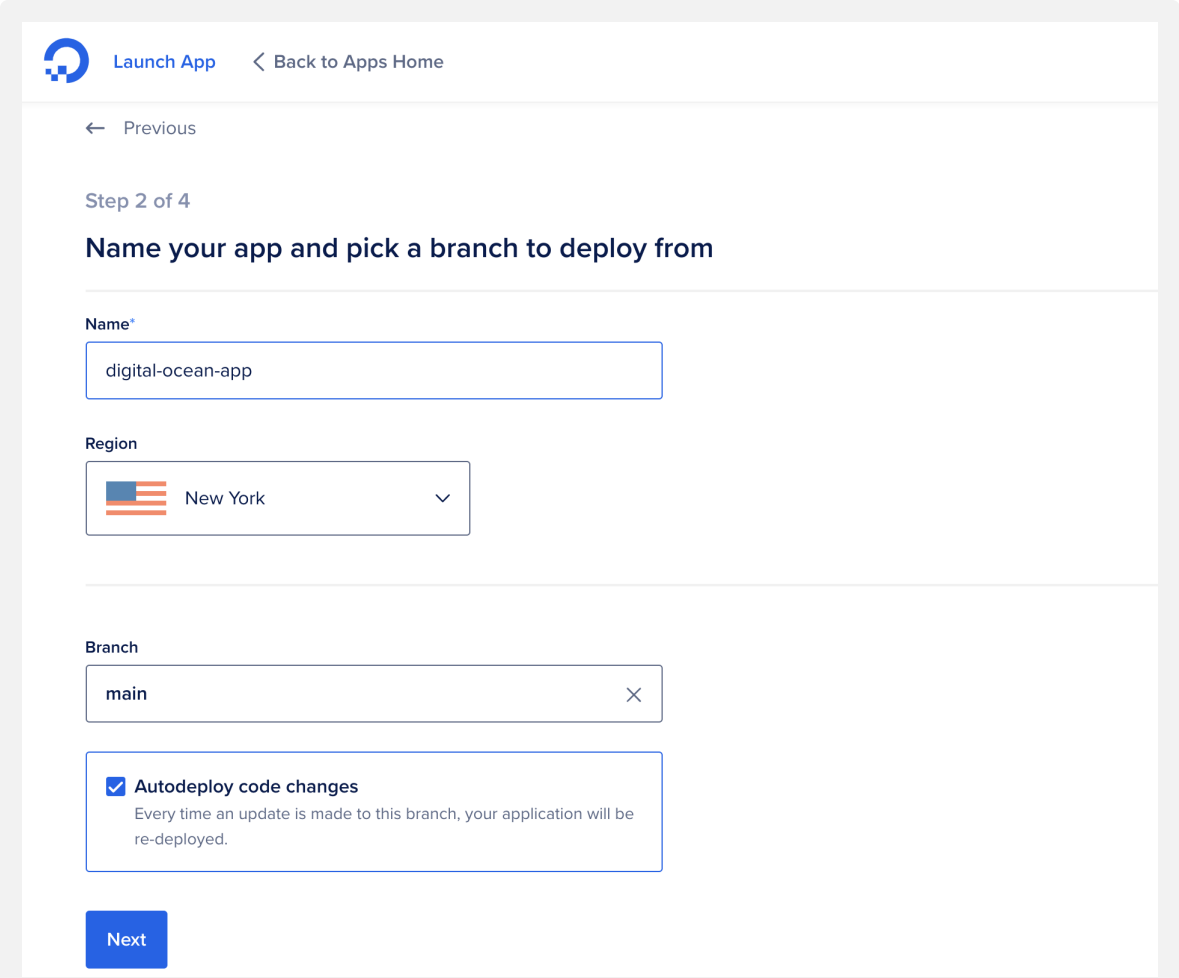
Select a repo on GitHub authorization page

After selecting the repository, you will reconnect to the DigitalOcean interface. Select **digital-ocean-app** from the list of repositories, then press **Next**. This will connect your App directly to the GitHub repo:



Select repo in the DigitalOcean UI

Now that you have selected your repository, you need to configure the DigitalOcean App. In this example, the server will be based in North America by choosing **New York** in the **Region** field, and the deployment branch will be **main**. For your app, choose the region that is closest to your physical location:



Select branch and location in the DigitalOcean interface

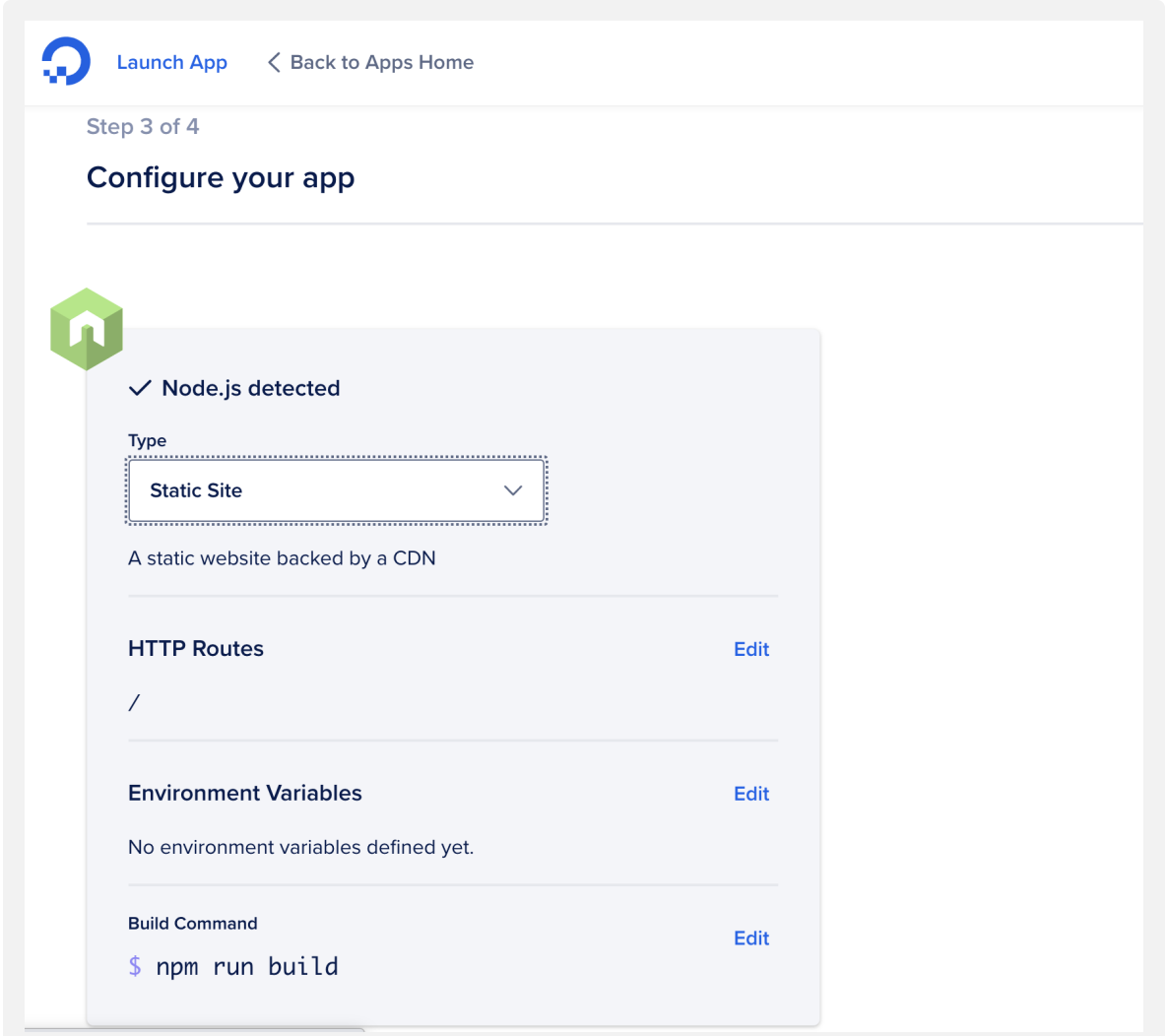
For this tutorial, you are checking **Autodeploy code changes**. This will automatically rebuild your app every time you update the code.

Press **Next** to move on to the **Configure your app** screen.

Next, select the type of application you will run. Since React will build static assets, select **Static Site** from the dropdown menu in the **Type** field.

Note: Create React App is not a static site generator like Gatsby, but you are using static assets, since the server does not need to run any server-side code such as Ruby or PHP. The app will use Node to run the install and build steps, but will not execute application code at the free tier.

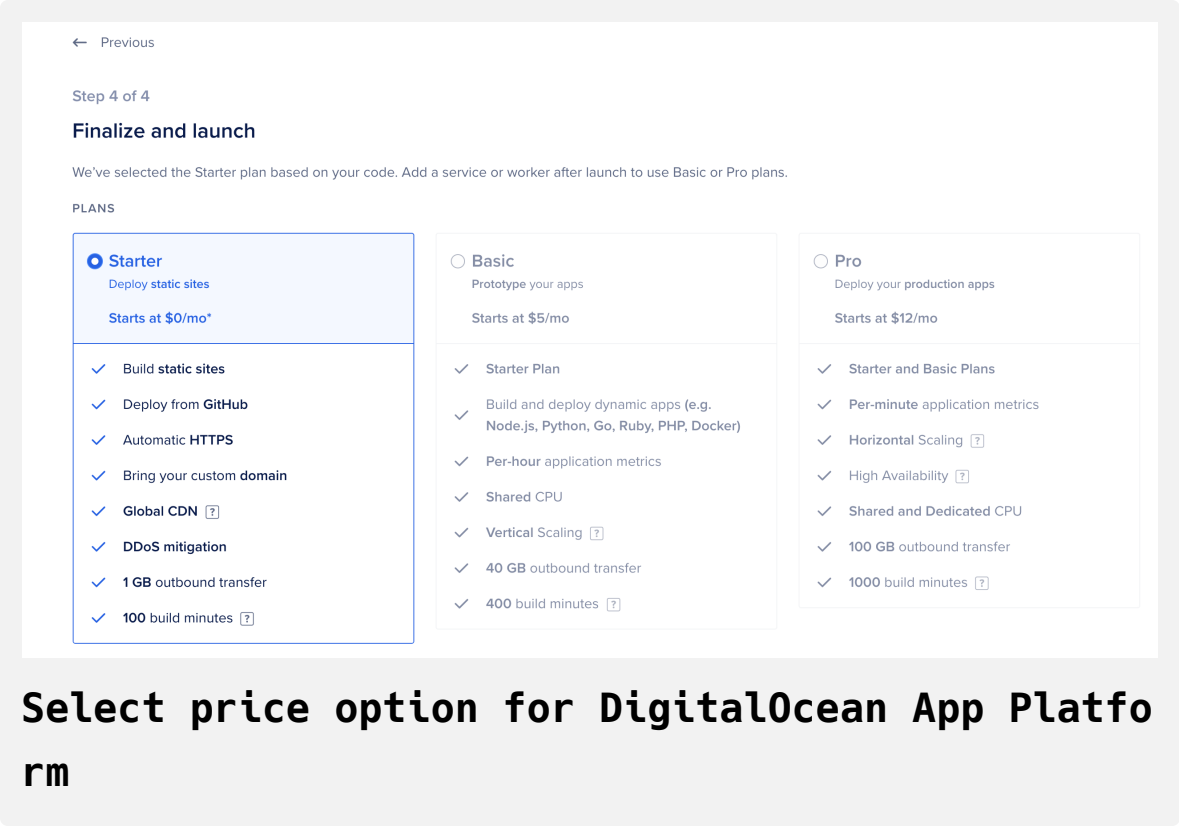
You also have the option to use a custom build script. But in this case, you can stick with the default `npm run build` command. You may want to create a custom build script if you have a different build script for a quality assurance (QA) or a production environment:



Select Static Site on the app configuration page

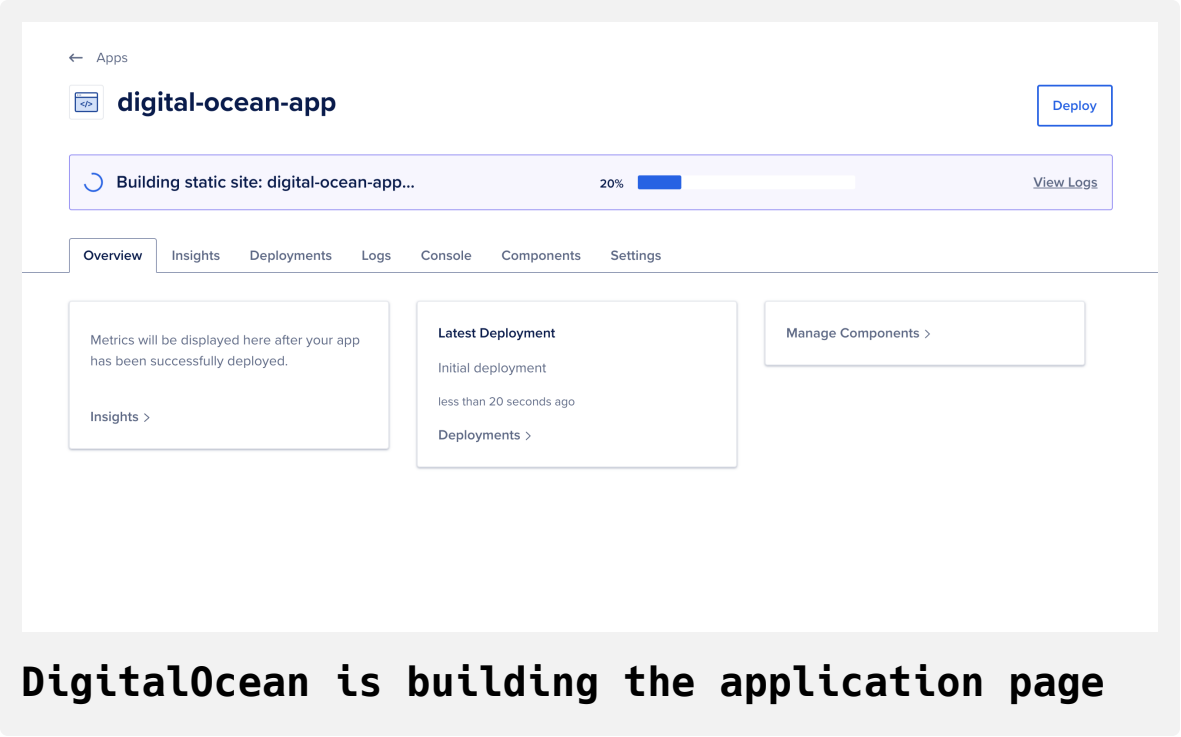
Press **Next** to move on to the **Finalize and launch** page.

Here you can select the price plan. The free **Starter** tier is made for static sites, so choose that one:



Select price option for DigitalOcean App Platform

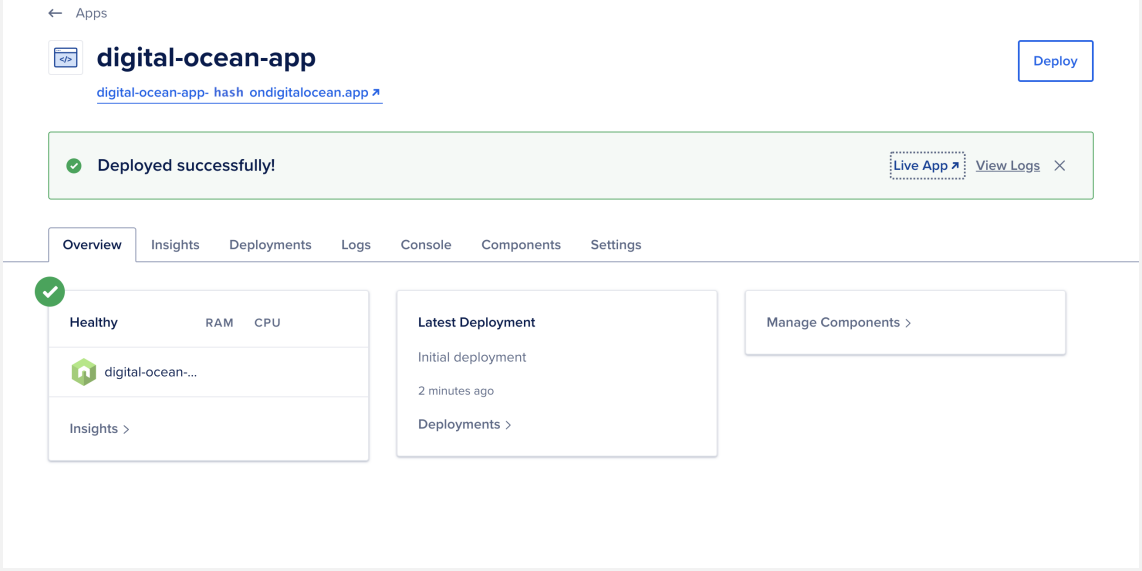
Press the **Launch Starter App** button and DigitalOcean will start building your application.



DigitalOcean is building the application page

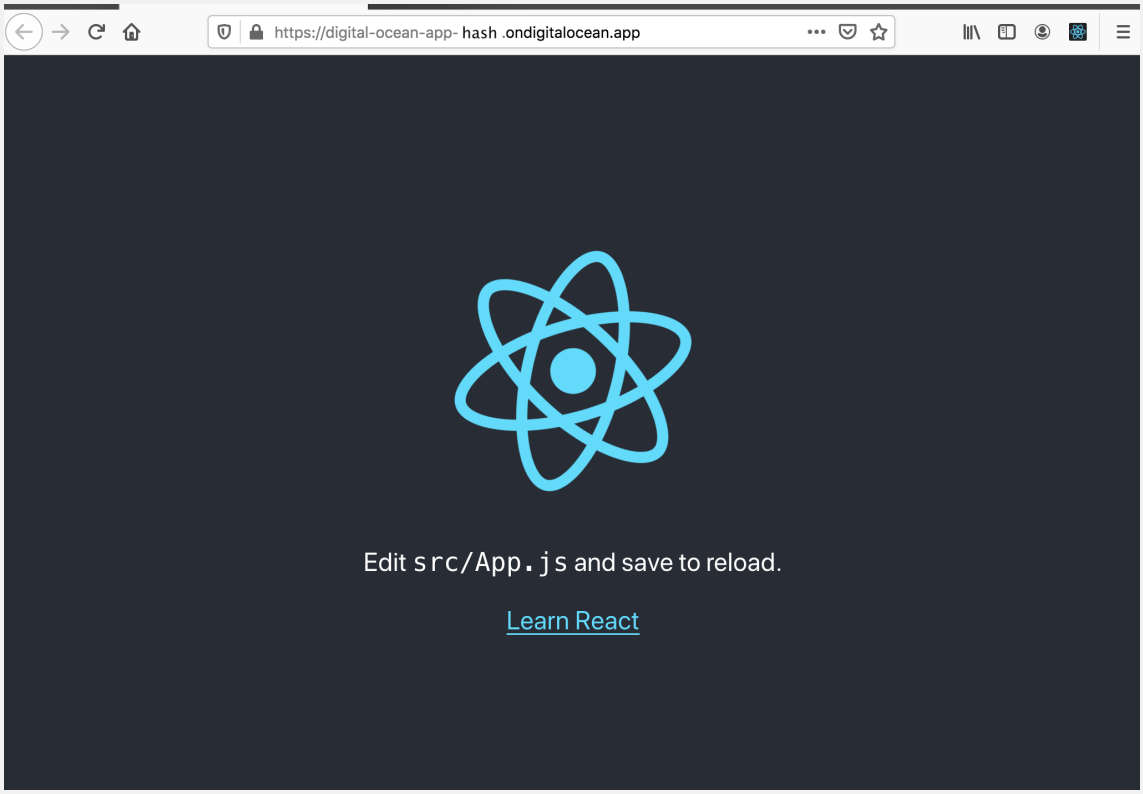
The app will run the `npm ci` and `npm build` scripts in your repo. This will download all of the dependencies and create the `build` directory with a compiled and minified version of all of your JavaScript, HTML files, and other assets. You could also create a custom script in your `package.json` and update the **Commands** in the **Components** tab of your application on App Platform.

It will take a few minutes for the build to run, but when it is finished, you will receive a success message and a link to your new site. Your link will have a unique name and will be slightly different:



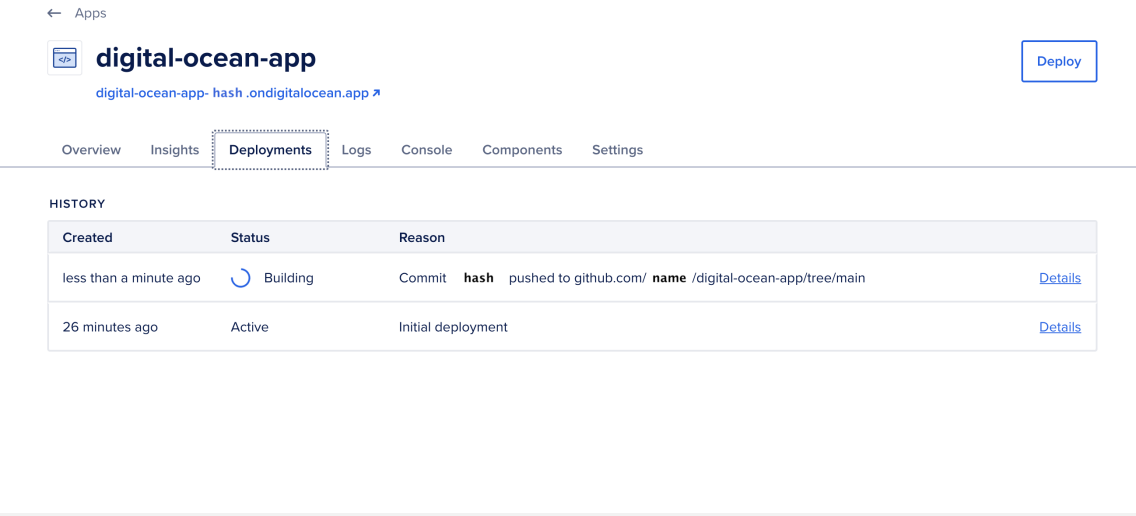
Deploy complete page

Press **Live App** to access your project in the browser. It will be the same as the project you tested locally, but this will be live on the web with a secure URL:



Live application

Now that your project is configured, any time you make a change to the linked repository, you'll run a new build. In this case, if you push a change to the **main** branch, DigitalOcean will automatically run a new deployment. There is no need to log in; it will run as soon as you push the change:



New deploy

In this step, you created a new DigitalOcean app on App Platform. You connected your GitHub account and configured the app to build the **main** branch. After configuring the application, you learned that the app will deploy a new build after every change.

Conclusion

DigitalOcean's App Platform gives you a quick tool for deploying applications. With a small initial set up, your app will deploy automatically after every change. This can be used in conjunction with React to quickly get your web application up and running.

A possible next step for a project like this would be to change the domain name of the app. To do this, take a look at the official documentation for App Platform.