

13

Setting Up Your React Dev Environment Easily

The last major new topic we're going to look at is less about React and more about setting up your development environment to build a React app. Until now, we've been building our React apps by including a few script files:

[Click here to view code image](#)

```
<script src="https://unpkg.com/react@16/umd/react.development.js">
</script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js">
</script>
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js">
</script>
```

These script files not only loaded the React libraries, but they also loaded Babel to help our browser do what needs to be done when encountering bizarre things like JSX (Figure 13.1):

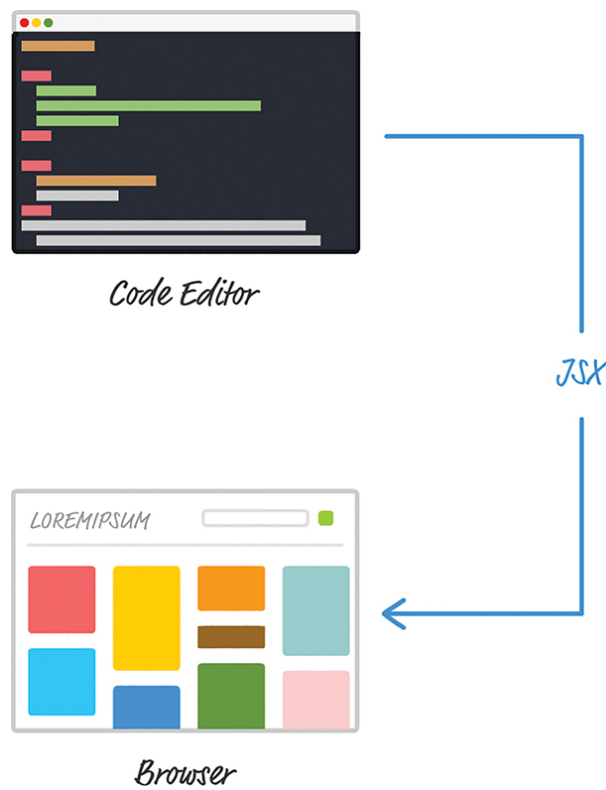


Figure 13.1 What our in-browser JSX transformer does.

To review what we mentioned earlier when talking about this approach, the downside is performance. As your browser handles all of the page loading things it normally does, it is also responsible for turning your JSX into actual JavaScript. That conversion is a time-consuming process that is fine during development but not fine if every user of your app has to pay that performance penalty.

The solution is to set up your development environment so that your JSX-to-JS conversion is handled as part of getting your app built (Figure 13.2):

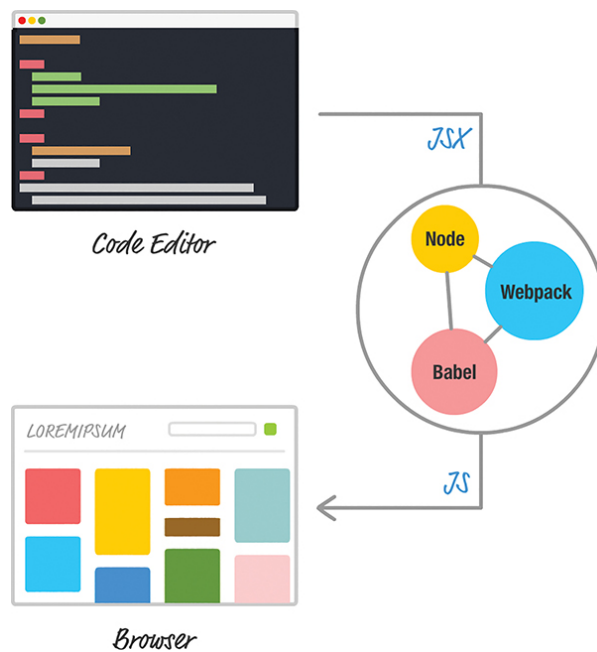


Figure 13.2 What the proper dev setup does with our JSX!

With this solution, your browser is loading your app and dealing with an already converted (and potentially optimized) JavaScript file. Good stuff, right? The only reason we delayed talking about all of this until now is for **simplicity**. Learning React is difficult enough. Adding the complexity of build tools and setting up your environment as part of learning React is just not cool. Now that you have a solid grasp of everything React does, it's time to change that with this chapter.

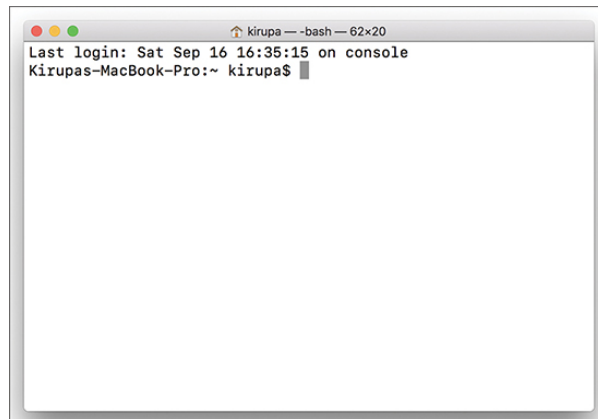
In the following sections, we're going to look at one way to set up your development environment using a combination of Node, Babel, and webpack. If all of this sounds bizarre to you, don't worry. We'll use a really nifty solution created by Facebook that makes all of this a breeze.

Onward!

MEET CREATE REACT

A few years ago, getting your build environment set up would have been a huge pain because it involved manually configuring all the tools we've talked about. You would have had to ask your really smart friend for some advice. You might even have questioned your decision to learn programming and React in the first place. Fortunately, the Create React project (<https://github.com/facebookincubator/create-react-app>) came about and greatly simplified the process of setting up your React environment. You just run a few commands on your command line, and your React project is automatically created with all the proper behind-the-scenes configurations.

To get started, first make sure you have the latest version of Node installed (<https://nodejs.org/>). Then bring up your favorite command line. If you aren't too familiar with command lines, don't worry. On Windows, launch either the command prompt or the BASH shell. On Mac, launch the Terminal. You'll see something that looks like this:



It's basically some bizarre window with a blinking cursor that allows you to type things into it. The first thing you need to do is install the Create React project. Type the following in your command line and press Enter/Return:

[Click here to view code image](#)

```
npm install -g create-react-app
```

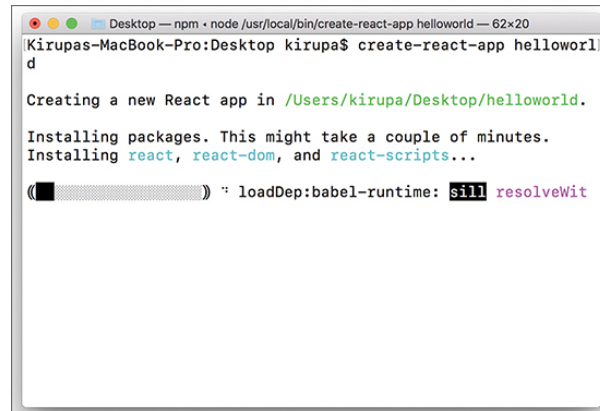
It can take anywhere from a few seconds to a few minutes, but once your installation has completed, it's time to create our new React project. Navigate to the folder where you want to create your new project—this can be your desktop, a location under Documents, and so on. When you've navigated to a

folder in your command line, enter the following to create a new project at this location:

[Click here to view code image](#)

```
create-react-app helloworld
```

You'll see something that looks as follows:



After the command has fully executed, you'll have a project called **helloworld** created for you. Don't worry too much about everything that's going on; we'll look at the project contents later. For now, the first thing to do is test this project. Navigate into the newly created project's `helloworld` folder by typing the following:

[Click here to view code image](#)

```
cd helloworld
```

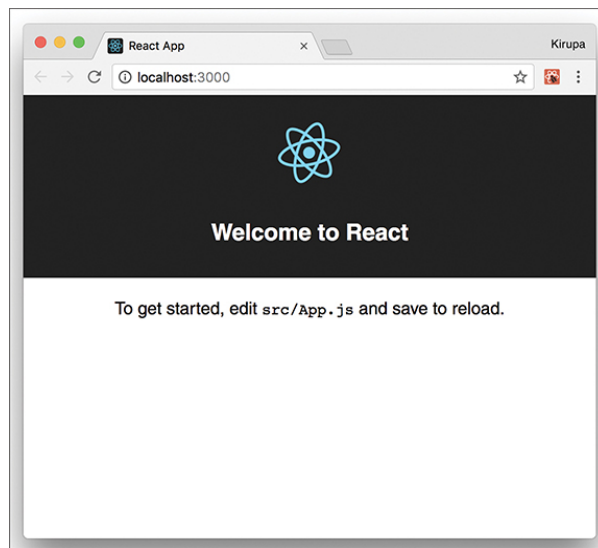
From inside this folder, enter the following to test the app:

[Click here to view code image](#)

```
npm start
```

If you have yarn installed, Create will prefer it over npm for the install and you'll see onscreen instructions saying to use `yarn start` instead of `npm start`.

Your project will get built, a local web server will get started, and you'll see your project running, similar to the following image:



If everything worked out properly, you should see the same thing. If this is your first time creating a new React project using the command line, congratulations! This is a really big step. You aren't done, though. Now, we need to take a few steps back and revisit what exactly just happened.

Making Sense of What Happened

Right now, we just see whatever default content the `create-react-app` command generated for us. That isn't very helpful. First, let's take a look at what exactly gets generated. Your file and folder structure after running `create-react-app helloworld` will look as in [Figure 13.3](#):

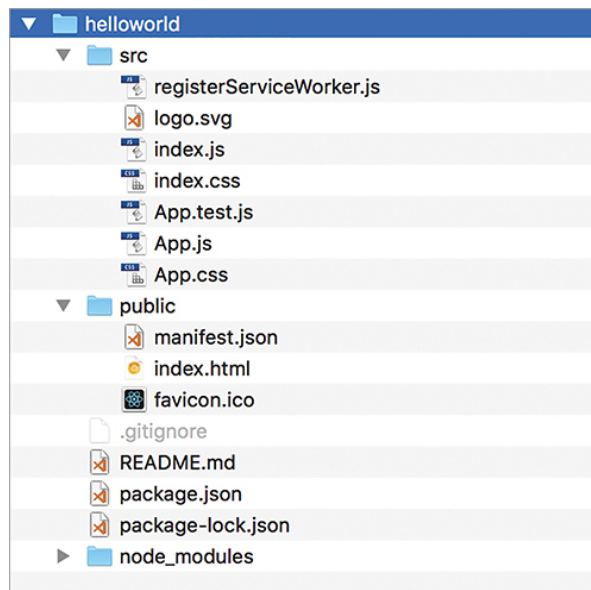


Figure 13.3 What our file and folder structure looks like.

The `index.html` in your `public` folder gets loaded in your browser. If you take a look at this file, you'll

realize that it's very basic. Here are the contents of this file with all the comments removed:

[Click here to view code image](#)

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-
width, initial-scale=1, shrink-to-
fit=no">
    <meta name="theme-
color" content="#000000">

    <link rel="manifest" href="%PUBLIC_URL%/manifest.json">
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">

    <title>React App</title>
  </head>
  <body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>

    <div id="root"></div>

  </body>
</html>
```

The important part to look at is the `div` element with an `id` value of `root`. This is where the contents of our React app ultimately get printed to. Speaking of that, the contents of our React app with all the JSX are contained inside the `src` folder. The starting point for our React app is contained in `index.js`:

[Click here to view code image](#)

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

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

Notice the `ReactDOM.render` call that looks for the `root` element we called out inside `index.html`. You'll also see a bunch of `import` statements at the

top of the page. These `import` statements are part of something in JavaScript known as **modules**. The goal of modules is to divide the functionality of your app into increasingly smaller pieces. When it comes time to use a piece, you import only what you need instead of everything and the entire kitchen sink. Some of the modules you import are a part of code in your project. Other modules, like `React` and `ReactDOM`, are external to your project but still capable of being imported. I can say a lot about module loading, but for your sanity (and mine!), let's just leave that topic alone for now.

In our code right now, we're importing both the `React` and `React-DOM` libraries. That should be familiar from when we included the script tags for them earlier. We're also importing a CSS file, a service worker script that we'll reference as `registerServiceWorker`, and a `React` component that we'll reference as `App`.

Our `App` component seems like our next stop, so to see what's inside it, open `App.js`:

[Click here to view code image](#)

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

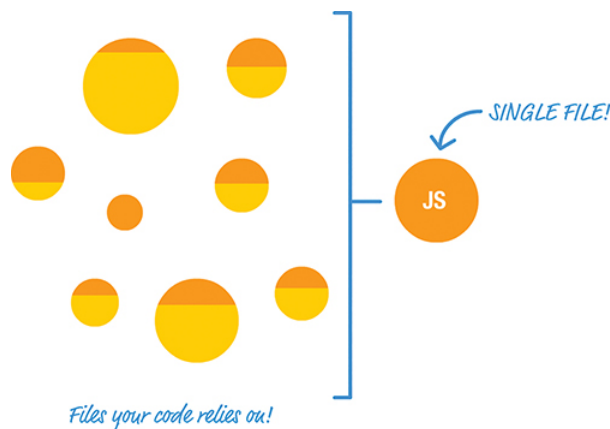
class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src=
{logo} className="App-
logo" alt="logo" />
          <h1 className="App-
title">Welcome to React</h1>
        </header>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to rel
        </p>
      </div>
    );
  }
}
```

`export default App;`

Notice that our `App.js` file has `import` statements of its own. Some, such as the one for `React` and `Component`, seem necessary, given what our code is doing. The last line here is interesting: `export default app`. It contains the `export` command and the name that our project will use to identify the exported module. You'll use this exported name when importing the `App` module in other parts of the project, such as `index.js`. Closing out what this file is doing, it also imports an image and CSS file that are needed to make this page work.

You've now seen a different way of structuring code using some potentially new keywords. What's the purpose of all of this? These modules, `import` statements, and `export` statements are just niceties to make our app's code more manageable. Instead of having everything defined in one giant file, you can break your code and related assets across multiple files. Depending on which files you reference and what files get loaded ahead of other files, our mysterious build process (currently kicked off with an `npm start`) can optimize the final output in a variety of ways that we don't need to worry about.

The important point to note is that none of these things you are doing to your code affect the functionality of your final app in any major way. Behind the scenes, when we're ready to test our app, a build step takes place. This build step makes sense of all of the various files and components you are importing, to present them as an easily digestible set of combined files for the browser to take care of. We'll get one JS file with all the relevant pieces represented:



We'll also get one combined CSS file. Depending on what else you might have configured, you could get other combined files for your HTML and more. All of

these will be in a form that your browser will immediately know what to do with. Your browser will have no additional work to do, as in our in-browser solution we were using initially. Everything gets generated as vanilla HTML, CSS, and JavaScript.

CREATING OUR HELLOWORLD APP

Now that you've gotten a better idea of what this project is doing, let's modify the example. We want to display the words `Hello, world!` to our screen.

We'll go about this by creating a component, appropriately called `HelloWorld`, to handle it for us.

The new part in this isn't that you get some text to display onscreen; you're a pro at that by this point. The part to focus on is how to structure the files in your project to ensure that you're creating your app the *right* way.

To get started, go to your `src` directory and delete all the files you see there. Then create a new file called `index.js`. Inside that file, add the following contents:

[Click here to view code image](#)

```
import React from "react";
import ReactDOM from "react-dom";
import HelloWorld from "./HelloWorld";

ReactDOM.render(
  <HelloWorld/>,
  document.getElementById("root")
);
```

We're importing our React and ReactDOM modules here. We're also importing a component called `HelloWorld` that we are specifying in our `ReactDOM.render` call. That component doesn't exist, so we are going to fix that next.

In the same `src` directory that we're in right now, create a file called `HelloWorld.js`. Then go ahead and modify it by adding in the following:

[Click here to view code image](#)

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

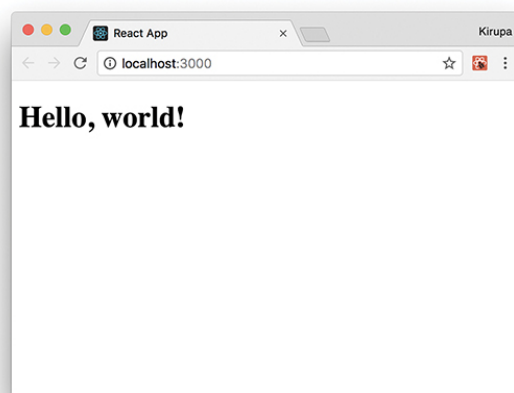
class HelloWorld extends Component {
  render() {
```

```
    return (  
      <div className="helloContainer">  
        <h1>Hello, world!</h1>  
      </div>  
    );  
  }  
}  
  
export default HelloWorld;
```

Take a moment to look through what you've added. You shouldn't see anything really exciting going on here—just a boring `import` statement, our `HelloWorld` component that prints some text to the screen, and (in the last line) code that tags our `HelloWorld` component for exporting so that it can be imported by another module, such as our `index.js`.

With these changes made, we can test the application. Make sure you've saved all your changes. Go back to the command line and type in `npm start`. If your app was already running behind the scenes, you would automatically see it update with the latest changes. If that didn't happen or your app stopped, press `Ctrl+C` to stop the session and enter `npm start` again.

You should see something similar to this on your screen:



If this is what you see, great! Our example is working now, but it looks a little too plain. Let's fix that by adding some CSS. Create a stylesheet called `index.css`, and add the following style rule into it:

[Click here to view code image](#)

```
body {
  display: flex;
  align-items: center;
  justify-content: center;
  min-height: 100vh;
  margin: 0;
}
```

In this approach for building apps, creating the stylesheet is only one part of what you have to do. The other part requires you to reference the newly created `index.css` in the `index.js` file. Open `index.js` and add the highlighted `import` statement for it:

[Click here to view code image](#)

```
import React from "react";
import ReactDOM from "react-dom";
import HelloWorld from "./HelloWorld";
import "./index.css";

ReactDOM.render(
  <HelloWorld/>,
  document.getElementById("root")
);
```

If you go back to your browser, you'll notice that the current setup automatically refreshes your page with all the latest changes. You'll see the words `Hello, world!` centered vertically and horizontally for you. Not bad, but we can do better.

The last thing we want to do is make our text appear in a more stylish fashion. We could add the appropriate style rules to `index.css` itself, but the more appropriate solution is to create a new CSS file that we reference only in our `HelloWorld` component. The end result of both approaches is identical, but you want to get into the practice of grouping related files (and their dependencies) together, as part of being a better developer.

Create a new file called `HelloWorld.css` inside the `src` folder. Add the following style rule into it:

[Click here to view code image](#)

```
h1 {
  font-family: sans-serif;
  font-size: 56px;
  padding: 5px;
```

```
padding-left: 15px;
padding-right: 15px;
margin: 0;
background: linear-
gradient(to bottom,
                                white 0%,
                                white 62%,
                                gold 62%,
                                gold 100%);
}
```

All that's left is to reference this stylesheet in the `HelloWorld.js` file, so open that file and add the highlighted `import` statement:

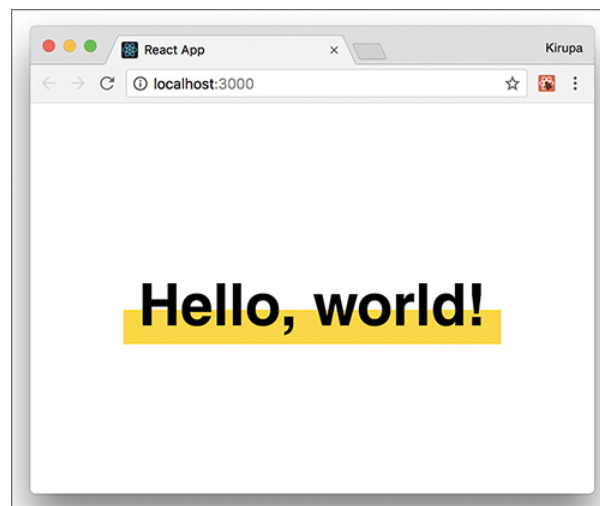
[Click here to view code image](#)

```
import React, { Component } from "react";
import "./HelloWorld.css";

class HelloWorld extends Component {
  render() {
    return (
      <div className="helloContainer">
        <h1>Hello, world!</h1>
      </div>
    );
  }
}

export default HelloWorld;
```

If you go back to your browser, you know that everything worked out fine if you see something like the following:



You'll see the words `Hello, world!` displayed, but with a little more style and pizzazz (as the cool kids say

these days) than they did a few moments ago.

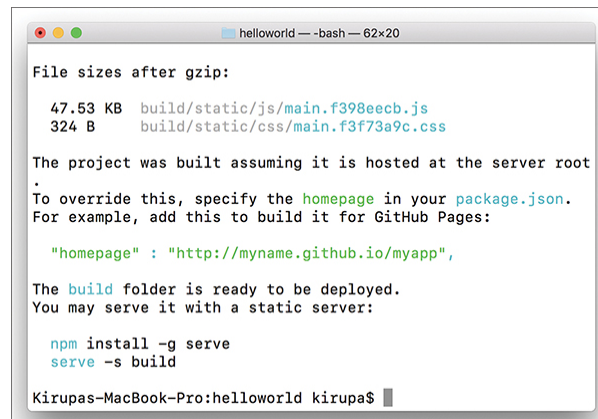
CREATING A PRODUCTION BUILD

We're almost done. We've got just one more thing left to do. So far, we've been building this app in **development mode**. In this mode, our code isn't minified and some of the things run in a slow/verbose setting so that we can debug issues more easily. When it's time to send the app live to our real users, we want the fastest and most compact solution possible. For that, we can go back to the command line and enter the following (after stopping the build by pressing Ctrl+C):

[Click here to view code image](#)

```
npm run build
```

The script takes a few minutes to create an optimized set of files for you. Once it has run to completion, you'll see some confirmation text that looks as follows:

A terminal window titled 'helloworld — bash — 62x20' showing the output of the 'npm run build' command. The output includes file sizes after gzip, instructions on how to override the homepage in package.json, and a list of commands to serve the build. The prompt at the bottom is 'Kirupas-MacBook-Pro:helloworld kirupa\$'.

```
File sizes after gzip:

47.53 KB   build/static/js/main.f398eecb.js
324 B      build/static/css/main.f3f73a9c.css

The project was built assuming it is hosted at the server root
.
To override this, specify the homepage 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:

  npm install -g serve
  serve -s build

Kirupas-MacBook-Pro:helloworld kirupa$
```

When this has completed, you can follow the onscreen prompts to deploy it to your server or just test it locally using the popular `serve` node package.

Also take a moment to browse through all the files that were generated. The end result is just plain HTML, CSS, and JS files. No JSX. No multiple JS files. We have just a single JS file that contains all the logic our app needs to work.

CONCLUSION

So that just happened! In the preceding sections, we used the awesome Create React solution to create our React app in a modern way. If this is your first time building apps like this, you'll want to get more familiar with this approach. We use the `create-react-app` command for future React examples; our earlier in-browser approach was just to help you learn the basics without fiddling with all of what you saw here. Under the covers, Create React hides a lot of the complexity that goes with tweaking Node, Babel, webpack, and other components. That is its greatest strength, as well as its greatest weakness.

If you want to go beyond the happy path that Create React provides, you'll need to learn a lot of the complexity hidden underneath. Covering all of that goes beyond this book. As a starting point, take a look at what's specified in the various JS files under the `node_modules/react_scripts/scripts` path.

Note: If you run into any issues, ask!

If you have any questions or your code isn't running like you expect, don't hesitate to ask! Post on the forums at <https://forum.kirupa.com> and get help from some of the friendliest and most knowledgeable people the Internet has ever brought together!

[Recommended](#) / [Playlists](#) / [History](#) / [Topics](#) / [Tutorials](#) / [Settings](#) / [Get the App](#) / [Sign Out](#)



PREV

[12 Accessing DOM Elements in React](#)

NEXT



[14 Working with External Data in React](#)