4

REACT.JS

Developers can use the React.js library to create a full-stack application's user interface. React.js is built upon the Node.js ecosystem, and as one of the most commonly used web frameworks, it currently forms the basis of more than 40 percent of the most visited websites.

To work effectively with React.js, you must understand the syntax used to define the appearance of user interface elements and then combine these into React.js components that can dynamically update. This chapter covers everything you need to know to begin developing full-stack applications using this library.

The Role of React.js

Modern frontend architectures split an application's user interface into small, self-contained, and reusable items. Some of these, such as headers, navigations, and logos, might appear only once per page, while others are

repeated elements that form the page's contents, such as headlines, buttons, and teasers. Figure 4-1 shows some of these items. React.js's syntax embraces this pattern; the library focuses on building these independent components and, in doing so, helps us develop our applications more efficiently.

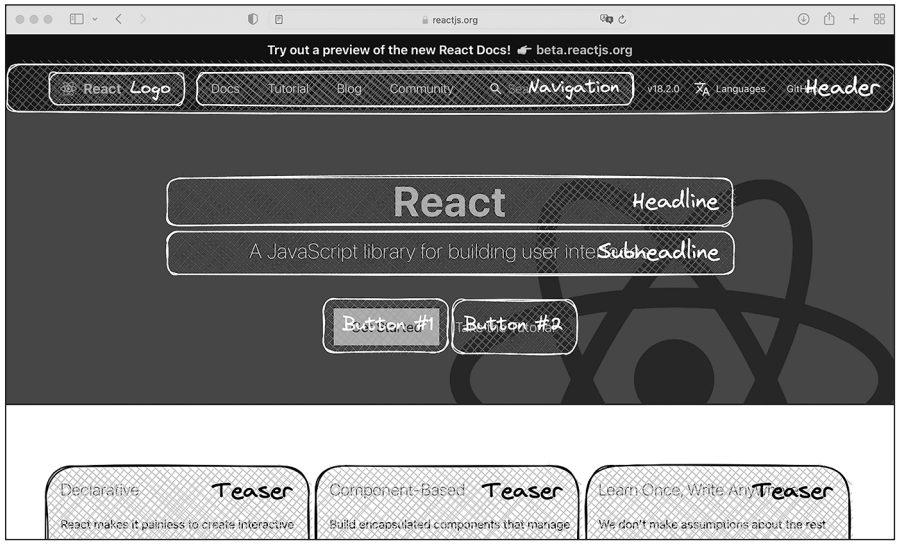

Figure 4-1: User interface components

React.js uses a *declarative* programming paradigm, through which you create a user interface by describing the desired results instead of explicitly listing all the steps necessary to create it, as is done in imperative programming. A classic example of the declarative paradigm is HTML. Using HTML, you describe a web page's elements, and the browser then renders the page. By contrast, you could use JavaScript to write an imperative program that creates each HTML element. In doing so, you would explicitly list the steps to build the website.

In addition, these user interface components are *reactive*. This means two things: first, that they handle their own isolated states, and second, that each component updates the page's HTML as soon as its state changes. Changes to the React.js code instantly affect a browser's *document object model (DOM)*, which represents a website as a tree in which each HTML element is a node. The DOM also provides an API for each node and for the website in general, enabling scripts to modify a website or a specific node.

DOM operations, such as re-rendering a component, are expensive. To update the DOM, React.js uses a *virtual DOM*, which is an in-memory clone of the actual browser DOM that it later syncs with the real thing. This virtual DOM allows for incremental updates that reduce the number of costly operations on the browser. The virtual DOM is a crucial principle of React.js. React.js calculates the difference between the virtual DOM and the real DOM with every call to one of its render functions and then decides what to

update. Usually, React.js performs batch updates to lower the performance impact further. This process of reconciliation lets React.js deliver fast and responsive user interfaces.

Although React.js is primarily a user interface library, developers can also use it to build single-page applications that don't require middleware or a backend. These apps are nothing more than a view layer rendered in the browser. To some extent, they can be dynamic: for example, we can change the page's language, open an image gallery, or toggle an element's visibility. However, all of this occurs in the browser, with additional React.js modules, rather than on the server.

We can also perform more advanced functionality, like updating the browser's location to simulate the existence of distinct pages, purely in the browser, with React.js's Router module. This module lets us define routes, similar to the ones we defined in our Express.js server, on the frontend. As soon as a user clicks an internal link, the routing component updates the view and changes the browser's location. This makes it seem as though they've loaded another HTML page. In reality, we've just changed the current page's contents. In doing so, we avoided another set of server requests, so the simulated page loads much more quickly. Also, because our JavaScript code controls the transition between pages, we can add effects and animations to these transitions.

Setting Up React.js

Unlike, say, the basic Express.js server you created in Exercise 1 on page XX, which uses standard JavaScript and can run directly with Node.js, React.js relies on an advanced setup with a complete build toolchain. For example, it uses a custom JavaScript Syntax Extension (JSX) to describe HTML elements and TypeScript for static typing, both of which require a transpiler to convert the code to JavaScript. Therefore, the manual process for setting up React.js is quite complex.

Thus, we generally rely on other tools. In the case of a single-page application, we use a code generator, such as create-react-app, to scaffold it. During this scaffolding process, create-react-app generates the boilerplate code for a new React.js application, as well as the build chain and folder structure for the project. It also provides a consistent project layout that helps us easily understand other React.js projects.

To run the examples in this chapter, one option is to scaffold a simple TypeScript React.js app with create-react-app by following the steps at *https://create-react-app.dev/docs/getting-started/*. If you don't want to create a dedicated project, you can instead run code using React.js with a TypeScript template in an online playground, such as *https://codesandbox.io* or *https://stackblitz.com*. The playgrounds and create-react-app follow the same file structure. In both cases, you should save your code to the default *App.tsx* file.

For more complex apps, we'd use a complete web application framework such as Next.js, which provides the necessary setup out of the box. Covered

in Chapter 5, Next.js is the most popular framework for full-stack web applications that use React.js. Internally, it employs a variation of create-react-app for scaffolding. We'll rely on it in future chapters to work with React.js.

The JavaScript Syntax Extension

React.js uses JSX to define the appearance of user interface components. JSX is an extension of JavaScript that a transpiler must convert before the browser renders it to the DOM. While it has HTML-like syntax, it is more than a simple templating language. Instead, it allows us to use any JavaScript feature to describe React.js elements. For example, we can use JSX syntax inside conditional statements, assign it to variables, and return it from functions. The compiler will then embed any variable or valid JavaScript expression wrapped in curly brackets ({}) into the HTML.

This logic allows us to, for instance, use `array.map` to loop over an array, check each item for a certain condition, pass the item to another function, and create a set of JSX elements based on the function's return value, directly inside a page's template. While this may sound abstract, we'll use this pattern extensively when we create React.js components in the Food Finder application you'll build in Part II.

An Example JSX Expression

JSX expressions, like those in Listing 4-1, are the most essential part of the React.js user interfaces. This JavaScript code defines a JSX function expression, `getElement`, that takes one string as a parameter and returns a `JSX.Element`.

```
import React from "react";

export default function App() {
    const getElement = (weather: string): JSX.Element => {
        const element = The weather is {weather};
        return element;
    }
    return getElement('sunny');
}
```

Listing 4-1: A minimal example of a JSX expression

The entry point for each React.js application is the `App` function. Like the *index.js* file of our Express.js server, this function is executed when the application starts. Here, we usually set up the global elements, such as stylesheets and the overall page layout.

React.js renders the function's return value to the browser. In Listing 4-1, we immediately return an element. As the smallest building blocks of React.js user interfaces, *elements* describe what you'll see on the screen, just as HTML elements do. Examples of elements include custom buttons, headlines, and images.

After importing the React.js package, we create the JSX element and store it in an `element` constant. At first glance, you might wonder why it isn't wrapped in quotes, as it contains what appears to be a regular HTML `h1` element and looks like a string. The answer is that it isn't a string but a JSX element from which the library creates HTML elements programmatically. As a result, the code will display a message about the weather to the page.

As soon as we call the JSX expression, the React.js library transpiles it into a regular JavaScript function call and creates an HTML string from the JSX element displayed in the browser. In Chapter 3, you learned that all valid JavaScript is also valid TypeScript. Hence, we can use JSX with TypeScript as well. JSX files use a *.jsx* (JavaScript) or *.tsx* (TypeScript) extension. Paste this code into the *App.tsx* file of the project you created, and the browser should render an `h1` HTML element with the text `The weather is sunny` either in the preview pane of the online playground or in your browser.

The ReactDOM Package

One easy way to work with elements is to use the ReactDOM package, which contains APIs for working with the DOM. Note that the elements you create aren't browser DOM elements. Instead, they're plain JavaScript objects that will be rendered, using React.js's `render` function, to the virtual DOM's root element and then attached to the browser DOM.

React.js elements are *immutable*: once created, they cannot be changed. If you do alter any part of the element, React.js will create a new element and re-render the virtual DOM, then compare the virtual DOM with the browser DOM to decide whether the browser DOM needs an update. We'll use JSX abstractions for these tasks; nonetheless, it's good to understand how React.js works under the hood. If you want to dig deeper, consult the official documentation at *https://reactjs.org*.

Organizing Code into Components

We mentioned that components are independent, reusable pieces of code built from React.js elements. Elements are objects that can contain other elements. Once rendered to the virtual or browser DOM, they create DOM nodes or whole DOM subtrees. Meanwhile, React.js *components* are classes or functions that output elements and render them to the virtual DOM. We will build a user interface using React.js components. For more information about this distinction, read the deep dive at the official React.js blog: *https://reactjs.org/blog/2015/12/18/react-components-elements-and-instances.html*.

While other frameworks might separate a user interface's code by technology, splitting it into HTML, CSS, and JavaScript files, React.js instead separates code into these logical building blocks. As a result, a single physical file contains all the information necessary for a component, regardless of underlying technologies.

More concretely, a React.js component is a JavaScript function that, by convention, starts with an uppercase letter. Furthermore, it takes a single object argument, called `props`, and returns a React.js element. This `props`

argument should never be modified inside the component and is considered immutable inside the React.js code.

Listing 4-2 shows a basic React.js component that displays the same weather string as in the previous listings. In addition, we've added a custom interface and a click handler. The custom interface enables us to set an attribute on the JSX component and read its value in the TypeScript code. It's a common way to pass values to a function component without a global state management library.

Here, we simply pass the component the same string used in the previous listings and render it to the DOM, but for a real-world application, the weather string might be part of an API response. To get the weather data, a parent component might query the API and then send this data through the component's attribute to the component's code, or each component in the application would need to query the API to access that data, impacting the overall performance of the application.

The click handler enables us to react to user interactions. In JSX, click handlers have the same names as in HTML, and we add them the way we might add inline DOM events. For example, to react to a user clicking an element, we add an `onClick` attribute with a callback function.

```
import React from "react"

export default function App() {

    interface WeatherProps {
        weather: string;
    }

    const clickHandler = (text: string): void => {
        alert(text);
    }

    const WeatherComponent = (props: WeatherProps): JSX.Element => {
         const text = `The weather is ${props.weather}`;
        return <h1 onClick={()=> clickHandler(text)}>{text}</h1>;
    }

    return (<WeatherComponent weather="sunny" />);
}
```

Listing 4-2: A basic React.js component

First we create a custom interface for our new component's properties. We'll use this interface for the component's `prop` parameter later. Because we set a `weather` attribute on the component and define a matching `weather` property on the interface, we can access the value of the `weather` attribute with `props.weather` in our TypeScript code.

Then we create the event handler as an arrow function with one string parameter. We use an `onClick` event property similar to inline DOM events and assign a callback function, `clickHandler`. As soon as the user clicks the page's headline, we display a simple alert box.

Next, we define the component. As you can see, it's a JSX expression that implements the `WeatherProps` interface and returns a JSX element. Inside the component, we use an untagged template literal to create text and add the dynamic weather information with the value from the `weather` attribute, via `props.weather`. Then we return the JSX element and, finally, return and render the weather component, setting `sunny` as the attribute's value.

Paste this code into the *App.tsx* file. The browser should render an `h1` HTML element with the text `The weather is sunny` in the preview pane. When you click the text, an alert box will display it once more. Change the value of the `weather` attribute to display different weather strings.

Writing Class Components

There are two kinds of components in React.js: class components and functional components. The component in Listing 4-2 is a *functional component*, which borrows heavily from functional programming. In particular, these components follow the pattern of pure functions: they create some output (JSX elements) based on some input (the `props` argument and the JSX component's attributes). While we emphasize this type of component in this chapter, you should know the basics of class components too.

A *class component* follows the typical patterns of object-oriented programming: it is defined as a class and inherits methods from its parent `React.Component` class. Like all components, it has an argument called `props` and returns a JSX element. Class components also have `constructor` and `super` functions, and you can use the `this` keyword to refer to the current component's instance.

Of particular value, the internal property `this.state` provides you an interface to store and access information about the component's internal state, such as opened elements, the current image in an image gallery, or, as in the next example, a simple click counter. Of similar importance are the class's *lifecycle* methods, which run during specific lifecycle steps: for example, whenever the component mounts, renders, updates, or unmounts. In Listing 4-3, we use the `componentDidMount` lifecycle method. React.js runs this method immediately after the component becomes part of the DOM. It is similar to the browser's `DOMReady` event with which you might already be familiar.

Listing 4-3 shows the previously created weather component defined as a class component. To practice accessing the component's state, we've added a counter that will count the clicks on the headline element. Because it records the internal component's state, the counter resets on page reload. Paste this code into the *App.tsx* file and click the headline to count up.

```
import React from "react";

export default function App() {
    interface WeatherProps {
        weather: string;
    }
```

```
    type WeatherState = {
        count: number;
    };

    class WeatherComponent extends React.Component<WeatherProps, WeatherState> {
        constructor(props: WeatherProps) {
            super(props);
            this.state = {
                count: 0,
            };
        }

        componentDidMount() {
            this.setState({ count: 1 });
        }

        clickHandler(): void {
            this.setState({ count: this.state.count + 1 });
        }

        render() {
            return (
                <h1 onClick={() => this.clickHandler()}>
                    The weather is {this.props.weather}, and the counter shows{" "}
                    {this.state.count}
                </h1>
            );
        }
    }

    return <WeatherComponent weather="sunny" />;
}
```

Listing 4-3: A basic React.js class component

First we define the custom interface to use for the component's properties. We also define a type to use in the counter we'll create later.

Next, we define the class component, extending the base class `React.Component`. Following object-oriented programming patterns, the constructor calls a `super` function and initializes the component's state. We set our counter to `0`. As soon as the browser mounts the component, it calls the lifecycle method `componentDidMount`, changing the component's `count` variable to `1`. We modify the click handler to count the number of clicks instead of displaying an alert box, and we call the `render` function. Here we return the JSX elements that display the weather props and the current state as HTML.

Finally, we return the `WeatherComponent`, and React.js initializes it. The preview pane displays the string `The weather is sunny, and the counter shows 1`. We see from the number `1` that the lifecycle method was indeed called. Each click on the headline increases the number instantly, due to the reactive nature of the component's state. As soon as the state changes, React.js re-renders the component and updates the view with the current value of the state.

Using Hooks to Add Reusable Logic to Functional Components

Functional components can use *hooks* to provide reusable behaviors, such as for accessing a component's state. Hooks are functions that offer simple and reusable interfaces to state and lifecycle features. Listing 4-4 shows the same weather component we created in Listing 4-3, this time written as a function component. It uses hooks instead of lifecycle methods to update the component's counter.

```
import React, {useState,useEffect} from "react"

export default function App() {

    interface WeatherProps {
        weather: string;
    }

    const WeatherComponent = (props: WeatherProps): JSX.Element => {

        const [count, setCount] = useState(0);
        useEffect( () => {setCount(1)},[]);

        return (<h1
          onClick={() => setCount(count + 1) }>
              The weather is {props.weather},
              and the counter shows {count}
          </h1>
        );
    }

    return (<WeatherComponent weather="sunny" />);
};
```

Listing 4-4: A React.js function component that uses hooks

We've added two new features to this component: an indicator of the component's state and a way to run code as soon as we mount the component. Therefore, we use the two hooks, `useState` and `useEffect`, by importing them as named imports from the React.js module, then adding them to the function component. The `useState` hook replaces the `this.state` property from the class component, and the `useEffect` hooks the `componentDidMount` lifecycle method. In addition, we replace the `clickHandler` from the previous example with a simple inline function to update the counter.

Each call to a hook produces an entirely isolated state, so we can use the same hook multiple times in the same component and trust that the state will update. This pattern keeps the hook callbacks small and focused. Also note that the runtime does not hoist hooks. They are called in the order in which we define them in the code.

When you compare Listings 4-3 and 4-4, you should instantly see that the functional component is more readable and easier to understand. For this reason, we'll exclusively use functional components in the rest of this book.

Working with Built-in Hooks

React.js provides a collection of built-in hooks. You've just seen the most common ones, `useState` and `useEffect`. Another useful hook is `useContext`, for sharing data between components. Other built-in hooks cover more specific use cases to enhance the performance of your application or handle specific edge cases. You can look them up as needed in the React.js documentation.

You can also create custom hooks whenever you need to break a monolithic component into smaller, reusable packages. Custom hooks follow a specific naming convention. They start with `use`, followed by an action beginning with an uppercase letter. You should define only one functionality per hook to make it easily testable.

This section will guide you through the three most common hooks and how you can benefit from using them.

Managing the Internal State with useState

A pure function uses only the data that is available inside the function. Still, it can react to local state changes, such as the counter in the weather component we created. The `useState` hook is probably the most-used one for handling regional states. This internal component's state is only available inside the component and is never exposed to the outside.

Because the component state is reactive, React.js re-renders the component as soon as we update its state, changing the value across the entire component. However, React.js guarantees that the state is stable and won't change on re-renders.

The `useState` hook returns the reactive state variable and a setter function used to set the state, as shown in Listing 4-5.

```
const [count, setCount] = useState(0);
```

Listing 4-5: The useState hook viewed in isolation

We initialize the `useState` hook with the default value. The hook itself returns the state variable `count` and the setter function we need to modify the state variable's value, because we cannot modify this variable directly. For example, to set the state variable `count` we created in Listing 4-5 to `1`, we need to call the `setCount` function with the new value as a parameter, like this: `setCount(1)`. By convention, the setter function begins with a `set` followed by the state variable's name.

Handling Side Effects with useEffect

Pure functions should rely only on the data passed to them. When a function uses or modifies data outside its local scope, we call this a *side effect*. The simplest example of a side effect is modifying a global variable. This is considered a bad practice both in JavaScript and in functional programming.

Sometimes, however, our components need to interact with the "outside world" or have an external dependency. In these cases, we can use the `useEffect` hook, which handles side effects, providing an escape hatch from the functional aspect of the component. For example, it can manage dependencies, call APIs, and fetch data required for the component.

This hook runs after React.js mounts the component into the layout and the rendering process of the component is completed. It has an optional return object, which runs before the component is unmounted. You can use it for cleanup, for example, to remove event listeners.

One way to use this hook is to observe and react to dependencies. To do this, we can pass it an optional array of dependencies. Any change to one of these dependencies would trigger a rerun of the hook. If the dependency array is empty, the hook won't depend on any external value and never reruns. This is the case in our weather component, where `useEffect` is only executed after mounting and unmounting the component. It has no external dependencies, so the dependency array remains empty and the hook runs only once.

Sharing Global Data with useContext and Context Providers

Ideally, React.js's functional components would be pure functions that operate only on data passed through the `props` parameter. Alas, a component might sometimes need to consume a shared, global state. In this case, React.js implements the *context provider* to share global data with a tree of child components.

The context provider wraps the child components, and we can access the shared data with the `useContext` hook. As the context value changes, React.js automatically re-renders all child components. Thus, it is quite an expensive hook. You shouldn't use it for datasets that change frequently.

In the full-stack application you'll build in Part II, you'll use `useContext` to share session data with child components. Shared contexts are also often employed to keep track of color schemes and themes. Listing 4-6 shows how to consume a theme through a context provider.

```
import React, { useState, createContext, useContext } from 'react';

export default function App() {
    const ThemeContext = createContext('');

    const ContextComponent = (): JSX.Element => {

        const [theme, setTheme] = useState('dark');

        return (
            <div>
                <ThemeContext.Provider value={theme}>
                    <button onClick={() => setTheme(theme == 'dark' ? 'light' : 'dark')}>
                        Toggle theme
                    </button>
                    <Headline />
```

```
            </ThemeContext.Provider>
        </div>
    );
};

const Headline = (): JSX.Element => {
    const theme = useContext(ThemeContext);
    return (<h1 className={theme}>Current theme: {theme}</h1>);
};

return (<ContextComponent />);
}
```

Listing 4-6: A complete context provider example

First we import the necessary functions from the React.js package and use the `createContext` function to initialize the `ThemeContext`. Next, we create the parent component and name it `ContextComponent`. This is the wrapper that holds the context provider and all child components.

In the `ContextComponent`, we create the local `theme` variable with `useState` and set the stateful variable as the content the context provides. This enables us to change the variable in the context from inside a child component. Because we used a reactive stateful variable for the value, all instances of the `theme` variable will instantly update across all child components.

We add a button element and toggle the value of the stateful variable between light and dark whenever a user clicks the button. Finally, we create the `Headline` component, which calls the `useContext` hook to get the `theme` value provided by the `ThemeContext` to all child components. The `Headline` component uses the `theme` value for the HTML class and displays the current `theme`.

Exercise 4: Create a Reactive User Interface for the Express.js Server

Let's use your new knowledge and our weather component to create a reactive user interface for the Express.js server. The new React.js component will allow us to update text on the web page by clicking it.

Adding React.js to the Server

First we'll include React.js in our project. For experimentation purposes, you can add the React.js library and the stand-alone version of the Babel.js transpiler directly inside your HTML `head` tag. Be aware, however, that this technique is not suitable for production. Transpiling code in the browser is a slow process, and the JavaScript libraries we add here aren't optimized. Using React.js with a skeleton Express.js server requires a decent number of tedious setup steps and a decent amount of maintenance. We'll use Next.js in Chapter 5 to simplify developing React.js applications.

Create a folder, named *public*, next to the *package.json* file and then create an empty file called *weather.html* inside it. Add the code in Listing 4-7, which contains our React.js example with the weather component. Later,

we'll create a new endpoint, */components/weather*, that directly returns the HTML file.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Weather Component</title>
 <script src="https://unpkg.com/react@18/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
   <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
  <body>
    <div id="root"></div>

    <script type="text/babel">
        function App() {

                const WeatherComponent = (props) => {

            const [count, setCount] = React.useState(0);
            React.useEffect(() => {
                setCount(1);
            }, []);

            return (
                <h1 onClick={() => setCount(count + 1)}>
                    The weather is {props.weather},
                    and the counter shows {count}
                </h1>
            );
        };
        return <WeatherComponent weather="sunny" />;
      }

      const container = document.getElementById("root");
      const root = ReactDOM.createRoot(container);
      root.render(<App />);
    </script>
  </body>
</html>
```

Listing 4-7: The static file /public/weather.html *renders React.js in the browser.*

First we add three React.js scripts to the *weather.html* file—*react.development*, *react.dom.development*, and the stand-alone *babel.js*, which are all similar to the import of React.js we previously used in the *App.tsx* file. Then we add ReactDOM to let React.js interact with the DOM. The three files add a global property, `React`, to the `window.object`. We use this property as a global variable to reference React.js functions. The stand-alone Babel script adds the Babel.js transpiler, which we need to convert the code from JSX to JavaScript.

Next, we add the weather component's code we developed previously. Instead of referencing the *App.tsx* file, we place app functions directly inside

the HTML file and mark the script block as `text/babel`. This type tells Babel to transpile the code inside the script tag into standard JavaScript.

We make a few simple modifications to the weather component's code. First we remove the type annotations, as they are only allowed in TypeScript files. Then, because we are using the browser environment, we prefix the hooks with their global property name, `React`. Finally, we use ReactDOM to create the React.js root container and render the `<App />` component there.

Creating the Endpoint for the Static HTML File

The second file we'll edit is the *index.ts* file in the root directory. We add the highlighted code in Listing 4-8 to add a new entry point, */components/weather*.

```
import { routeHello, routeAPINames, routeWeather } from "./routes.js";
import express, { Request, Response } from "express";

import path from "path";

const server = express();
const port = 3000;

--snip--
server.get("/components/weather", function (req: Request, res: Response): void {
    const filePath = path.join(process.cwd(), "public", "weather.html");
    res.setHeader("Content-Type", "text/html");
    res.sendFile(filePath);
});

server.listen(port, function (): void {
    console.log("Listening on " + port);
});
```

Listing 4-8: The refactored index.ts

To load the static HTML file, import `path` from Node.js's default path module. The path module provides all kinds of utilities for working with files and directories. In particular, we'll use the `join` function to create a valid path that meets the operation system's format.

We use the default global `process.cwd` function to get the current working directory, and from there, we create the path to our HTML file. Then we add the weather component's entry point and set the response's `Content-Type` header to *text/html*. Finally, we use the `sendFile` function to send to the browser the *weather.html* file we created previously.

Running the Server

We need to transpile the server code to JavaScript, so we run TSC with `npx` on the command line:

```
$ npx tsc
```

The generated files, *index.js* and *routes.js*, are similar to the previously created ones. TSC doesn't touch the static HTML. The stand-alone Babel.js script converts the JSX code on runtime in the browser. Start the server from your command line:

```
$ node index.js
Listening on 3000
```

Now visit *http://localhost:3000/components/weather-component* in your browser. You see the same text you saw when you rendered the weather component in the React.js playground, as in Figure 4-2. As soon as you click the text, the click handler increases the reactive state variable, and the counter shows the new value.

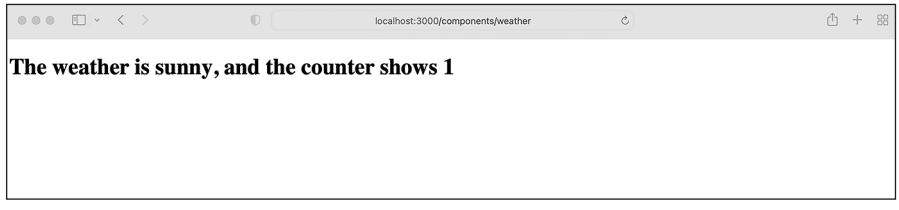

Figure 4-2: The response our browser receives from our Node.js web server

You successfully created your first React.js application. To gain more experience with React.js, try adding a custom button component for the click counter, with a style attribute that uses a JSX expression to change the background color for odd and even counter values.

Summary

You should now have a solid foundation with which to create your React.js apps. JSX elements are the building blocks of React.js components that return JSX to be rendered as HTML in the DOM, via React.js's virtual DOM. You also explored the difference between class components and modern function components, took a deep dive into React.js hooks, and used these hooks to build a functional component.

If you want to explore React.js's full potential, take a look at the React.js tutorials from W3Schools at *https://www.w3schools.com/REACT/DEFAULT.ASP* and those created by the React.js team at *https://reactjs.org/tutorial/tutorial.html*.

In the next chapter, we'll work with Next.js. Built on top of React.js, Next.js is a production-ready full-stack web development framework for single-page applications.