CHAPTER 9

Understanding React Projects

In Part 1 of this book, I created the SportsStore application to demonstrate how different React features can be combined with other packages to create a realistic application. In this part of the book, I dig into the detail of the built-in React features. In this chapter, I describe the structure of a React project and explain the tools that are provided for developers and the process by which code and content is compiled, packaged, and sent to the browser. Table 9-1 puts this chapter in context.

Table 9-1. *Putting React Projects in Context*

Question	Answer
What are they?	The `create-react-app` package is used to create projects and set up the tools that are required for effective React development.
Why are they useful?	Projects created with the `create-react-app` package are designed for the development of complex applications and provide a complete set of tools for development, testing, and deployment.
How are they used?	A project is created using the `npx create-react-app` package, and the development tools are started using the `npm start` command.
Are there any pitfalls or limitations?	The `create-react-app` package is "opiniated," which means that it provides a specific way of working with few configuration options. This can be frustrating if you are used to a different workflow.
Are there any alternatives?	You don't have to use `create-react-app` to create projects. There are alternative packages available as noted later in this chapter.

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_9

Table 9-2 summarizes the chapter.

Table 9-2. *Chapter Summary*

Problem	Solution	Listing
Create a new React project	Use the `create-react-app` package and add optional packages	1–3
Transform HTML to JavaScript	Use the JSX format to mix HTML and code statements	6
Include static content	Add files to the `src` folder and incorporate them into the application using the `import` keyword	9–10
Include static content outside of the development tools	Add files to the public folder and define references using the `PUBLIC_URL` property	11–13
Disabling linting messages	Add comments to JavaScript files	15–19
Configure the React development tools	Create an `.env` file and set configuration properties	20
Debug React applications	Use the React Devtools browser extension or use the browser debugger	22–26

Preparing for This Chapter

To create the example project for this chapter, open a new command prompt, navigate to a convenient location, and run the command shown in Listing 9-1.

Listing 9-1. Creating the Project

```
npx create-react-app projecttools
```

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/Apress/pro-react-16`.

■ **Note** When you create a new project, you may see warnings about security vulnerabilities. React development relies on a large number of packages, each of which has its own dependencies, and security issues will inevitably be discovered. For the examples in this book, it is important to use the package versions specified to ensure you get the expected results. For your own projects, you should review the warnings and update to versions that resolve the problems.

Run the commands shown in Listing 9-2 to navigate to the project folder and add the Bootstrap package to the project.

Listing 9-2. Adding the Bootstrap CSS Framework

```
cd projecttools
npm install bootstrap@4.1.2
```

To include the Bootstrap CSS stylesheet in the application, add the statement shown in Listing 9-3 to the `index.js` file, which can be found in the `src` folder.

Listing 9-3. Including Bootstrap in the index.js File in the src Folder

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

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: http://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Using the command prompt, run the command shown in Listing 9-4 in the `projecttools` folder to start the development tools.

■ **Caution** Notice that the development tools are started using the `npm` command and not the `npx` command that was used in Listing 9-1.

Listing 9-4. Starting the Development Tools

```
npm start
```

Once the initial preparation for the project is complete, a new browser window will open and display the URL `http://localhost:3000` and display the placeholder content shown in Figure 9-1.

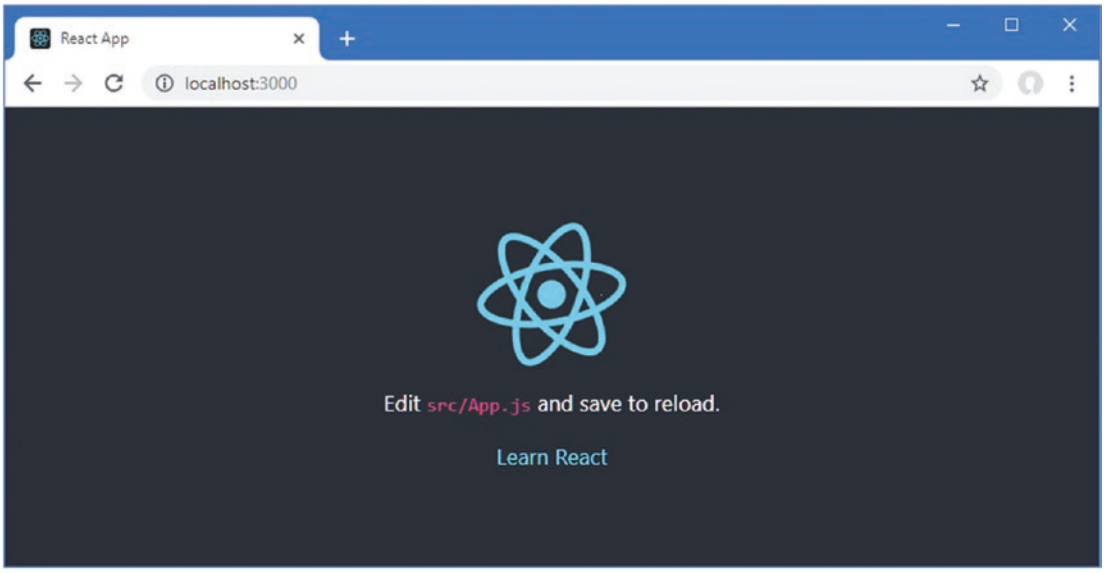

Figure 9-1. *Running the example application*

Understanding the React Project Structure

When you create a new project, you will start with a basic set of React application files, some placeholder content, and a complete set of development tools. Figure 9-2 shows the contents of the `projecttools` folder.

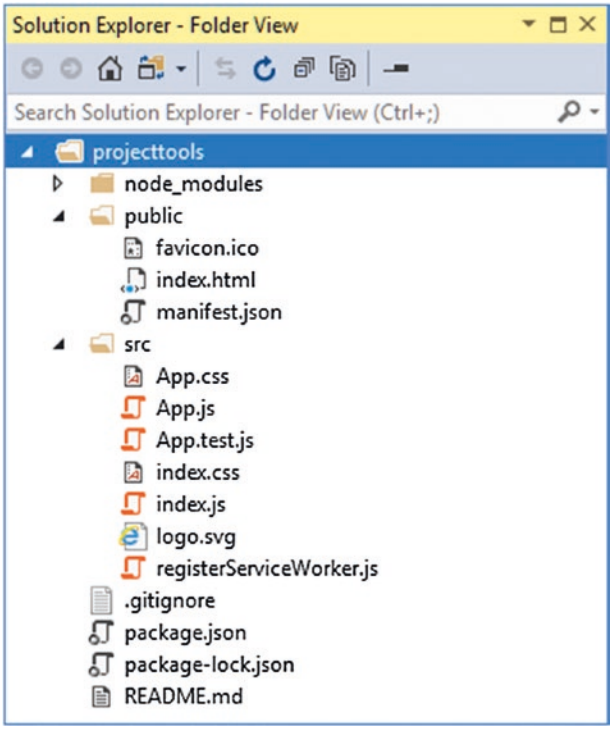

Figure 9-2. *The contents of a new project*

■ **Note** You don't have to use the `create-react-app` package to create React projects, but it is the most common approach, and it takes care of configuring the build tools that support the features described in this chapter. You can create all of the files and configure the tools directly, if you prefer, or use one of the other techniques available for creating a project, which are described at https://reactjs.org/docs/create-a-new-react-app.html.

Table 9-3 describes each of the files in the project, and I provide more details about the most important files in the sections that follow.

Table 9-3. *The Project Files and Folders*

Name	Description
`node_modules`	This folder contains the packages that the application and development tools require, as described in the "Understanding the Packages Folder" section.
`public`	This folder is used for static content and includes the `index.html` file that is used to respond to HTTP requests, as described in the "Understanding Static Content" section.
`src`	This folder contains the application code and content, as described in the "Understanding the Source Code Folder" section.
`.gitignore`	This file is used to exclude files and folders from the Git revision control package.
`package.json`	This folder contains the set of top-level package dependencies for the project, as described in the "Understanding the Packages Folder" section.
`package-lock.json`	This file contains a complete list of the package dependencies for the project, as described in the "Understanding the Packages Folder" section.
`README.md`	This file contains information about the project tools, and the same content can be found at `https://github.com/facebook/create-react-app`.

Understanding the Source Code Folder

The `src` folder is the most important in the project because it is where the application's code and content files are placed and where you will define the custom features required by your project. The `create-react-app` package adds files to jump-start development, as described in Table 9-4.

Table 9-4. *The Files in the src Folder*

Name	Description
`index.js`	This file is responsible for configuring and starting the application.
`index.css`	This file contains the global CSS styles for the application. See the "Understanding Static Content" section for details of using CSS files.
`App.js`	This file contains the top-level React component. Components are described in Chapters 10 and 11.
`App.css`	This file contains the placeholder CSS styles for new projects. See the "Understanding Static Content" section for details.
`App.test.js`	This file contains unit tests for the top-level component. See Chapter 17 for details of unit testing.
`registerServiceWorker.js`	This file is used by progressive web applications, which can work offline. I do not describe progressive applications in this book, but you can find details at `https://facebook.github.io/create-react-app/docs/making-a-progressive-web-app`.
`logo.svg`	This image file contains the React logo and is displayed by the placeholder component added to the project when it is created. See the "Understanding Static Content section.

Understanding the Packages Folder

JavaScript application development depends on a rich ecosystem of packages, ranging from those that contain the code that will be sent to the browser to small packages that are used behind the scenes during development for a specific task. A lot of packages are required in a React project: the example project created at the start of this chapter, for example, requires more than 900 packages.

There is a complex hierarchy of dependencies between these packages that is too difficult to manage manually and that is handled using a *package manager*. React projects can be created using two different package managers: NPM, which is the Node Package Manager and that was installed alongside Node.js in Chapter 1, and Yarn, which is a recent competitor designed to improve package management. I use NPM throughout this book for simplicity.

■ **Tip** You should use NPM to follow the examples in this book, but you can find details of Yarn at `https://yarnpkg.com` if you want to use it in your own projects.

When a project is created, the package manager is given an initial list of packages required for React development, each of which is inspected to get the set of packages it depends on. The process is performed again to get the dependencies of those packages and repeated until a complete list of packages is built up. The package manager downloads and installs all of the packages and installs them into the `node_modules` folder.

The initial set of packages is defined in the `package.json` file using the `dependencies` and `devDependencies` properties. The `dependencies` section is used to list the packages that the application will require to run. The `devDependencies` section is used to list the packages that are required for development but that are not deployed as part of the application.

You may see different details in your project, but here is the `dependencies` section from the `package.json` file from my example project:

```
...
"dependencies": {
    "bootstrap": "^4.1.2",
    "react": "^16.7.0",
    "react-dom": "^16.7.0",
    "react-scripts": "2.1.2"
},
...
```

Only three packages are required in the `dependencies` section for a React project: the `react` package contains the main features, the `react-dom` package contains the features required for web applications, and the `react-scripts` package contains the development tool commands that I describe in this chapter. The fourth package is the Bootstrap CSS framework, added to the project in Listing 9-2. For each package, the `package.json` file includes details of the version numbers that are acceptable, using the format described in Table 9-5.

Table 9-5. *The Package Version Numbering System*

Format	Description
`16.7.0`	Expressing a version number directly will accept only the package with the exact matching version number, e.g., 16.7.0.
*	Using an asterisk accepts any version of the package to be installed.
`>16.7.0 >=16.7.0`	Prefixing a version number with > or >= accepts any version of the package that is greater than or greater than or equal to a given version.
`<16.7.0 <=16.7.0`	Prefixing a version number with < or <= accepts any version of the package that is less than or less than or equal to a given version.
`~16.7.0`	Prefixing a version number with a tilde (the ~ character) accepts versions to be installed even if the patch level number (the last of the three version numbers) doesn't match. For example, specifying ~16.7.0 will accept version 16.7.1 or 16.7.2 (which would contain patches to version 16.7.0) but not version 16.8.0 (which would be a new minor release).
`^16.7.0`	Prefixing a version number with a caret (the ^ character) will accept versions even if the minor release number (the second of the three version numbers) or the patch number doesn't match. For example, specifying ^16.7.0 will allow versions 16.8.0 and 16.9.0, for example, but not version 17.0.0.

The version numbers specified in the `dependencies` section of the `package.json` file will accept minor updates and patches.

UNDERSTANDING GLOBAL AND LOCAL PACKAGES

Package managers can install packages so they are specific to a single project (known as a *local install*) or so they can be accessed from anywhere (known as a *global install*). Few packages require global installs, but one exception is the `create-react-app` package that I installed in Chapter 1 as part of the preparations for this book. The `create-react-app` package requires a global install because it is used to create new projects. The individual packages required for the project are installed locally, into the `node_modules` folder.

All the packages required for development are automatically downloaded and installed into the `node_modules` folder when you create a React project, but Table 9-6 lists some NPM commands that you may find useful during development. All of these commands should be run inside the project folder, which is the one that contains the `package.json` file.

Table 9-6. *Useful NPM Commands*

Command	Description
`npx create-react-app <name>`	This command creates a new React project.
`npm install`	This command performs a local install of the packages specified in the `package.json` file.
`npm install package@version`	This command performs a local install of a specific version of a package and updates the `package.json` file to add the package to the `dependencies` section.
`npm install --save-dev package@version`	This command performs a local install of a specific version of a package and updates the `package.json` file to add the package to the `devDependencies` section, which is used to add packages to the project that are required for development but are not part of the application.
`npm install --global package@version`	This command will perform a global install of a specific version of a package.
`npm list`	This command will list all of the local packages and their dependencies.
`npm run`	This command will execute one of the scripts defined in the `package.json` file, as described next.

The last command described in Table 9-6 is an oddity, but package managers have traditionally included support for running commands that are defined in the `scripts` section of the `package.json` file. In a React project, this feature is used to provide access to the tools that are used during development and that prepare the application for deployment. Here is the `scripts` section of the `package.json` file in the example project:

```
...
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
...
```

These commands are summarized in Table 9-7, and I demonstrate their use in later sections.

Table 9-7. *The Commands in the Scripts Section of the package.json File*

Name	Description
`start`	This command starts the development tools, as described in the "Using the React Development Tools" section.
`build`	This command performs the build process.
`test`	This command runs the unit tests, as described in Chapter 17.
`eject`	This command copies the configuration files for all the development tools into the project folder. This is a one-way operation and should be used only when the default configuration of the development tools is unsuitable for a project.

The commands in Table 9-7 are executed by using `npm run` followed by the name of the command that you require, and this must be done in the folder that contains the `package.json` file. So, if you want to run the `build` command in the example project, you would navigate to the `projecttools` folder and type `npm run build`. The exception is the `start` command, which is executed using `npm start`.

Using the React Development Tools

The development tools added to a project automatically detect changes in the `src` folder, compile the application, and package the files ready to be used by the browser. These are tasks you could perform manually, but having automatic updates makes for a more pleasant development experience. If they are not already running, start the development tools by opening a command prompt, navigating to the `projecttools` folder, and running the command shown in Listing 9-5.

Listing 9-5. Starting the Development Tools

```
npm start
```

The key package used by the development tools is *webpack*, which is the backbone for many JavaScript development tools and frameworks. Webpack is a *module bundler*, which means that it packages JavaScript modules for use in a browser—although that's a bland description for an important function, and it is one of the key tools that you will rely on while developing a React application.

When you run the command in Listing 9-5, you will see a series of messages as webpack prepares the bundles required to run the example application. Webpack starts with the `index.js` file and loads all of the modules for which there are `import` statements to create a set of dependencies. This process is repeated for each of the modules that `index.js` depends on, and webpack keeps working its way through the application until it has a complete set of dependencies for the entire application, which is then combined into a single file, known as the *bundle*.

The bundling process can take a moment, but it needs to be performed only when you start the development tools. Once the initial preparation has been completed, you will see a message like this one, which tells you that the application has been compiled and bundled:

```
...
Compiled successfully!

You can now view projecttools in the browser.

  Local:            http://localhost:3000/
  On Your Network:  http://192.168.0.77:3000/

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

As the initial process completes, a new browser window will be opened for `http://localhost:3000`, showing the placeholder content in Figure 9-3.

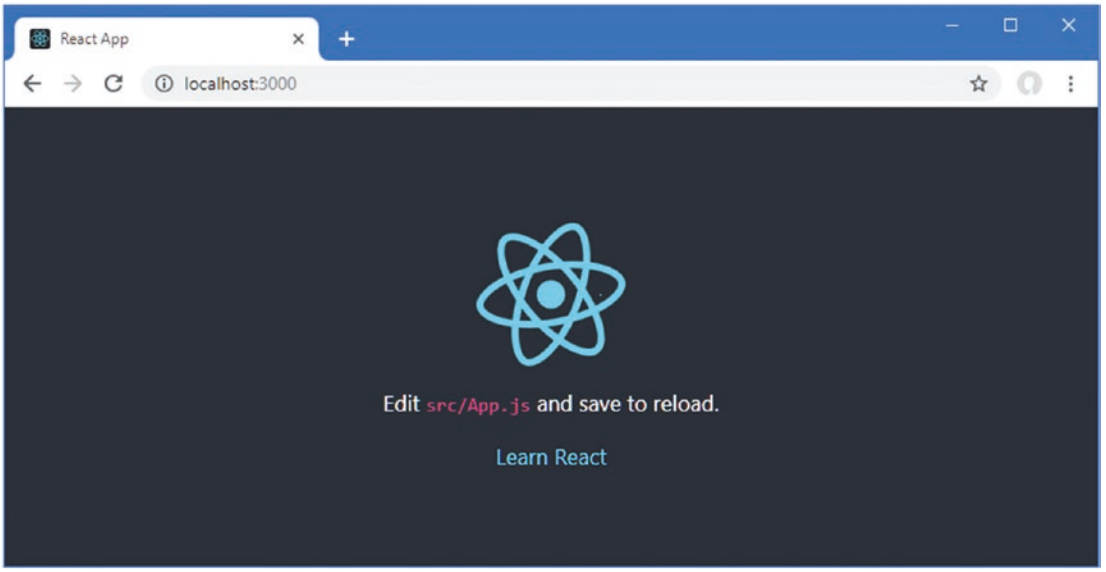

Figure 9-3. *Using the development tools*

Understanding the Compilation and Transformation Process

Webpack is responsible for the build process, and one of the key steps is code transformation performed by the *Babel* package. Babel has two important tasks in a React project: transforming JSX content and transforming JavaScript code that uses the latest JavaScript features into code that can be executed by older browsers.

Understanding the JSX Transformation

As I explained in Chapter 3, the JSX format is a superset of JavaScript that allows HTML to be mixed with regular code statements. JSX doesn't support entirely standard HTML, and the most obvious difference is that attributes such as `class` in pure HTML are expressed as `className` in a JSX file. The reason for these oddities is that the content of a JSX file is converted into calls to the React API by Babel during the build process so that every HTML element is translated into a call to the `React.createElement` method.
In Listing 9-6, I have replaced the placeholder content in the `App.js` file with a component whose `render` method returns some simple HTML elements.

Listing 9-6. Replacing the Placeholder Content in the App.js File in the src Folder

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

export default class extends Component {

    render = () =>
        <h4 className="bg-primary text-white text-center p-3">
            This is an HTML element
        </h4>
}
```

During the transformation process, the h4 element is replaced with a call to the React.createElement method, producing a result that is entirely JavaScript and that requires no special understanding of JSX by the browser. As a simple demonstration, Listing 9-7 uses the React.createElement method directly to achieve the same result.

Listing 9-7. Using the React API Directly in the App.js File in the src Folder

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

export default class extends Component {

    render = () => React.createElement("h4",
                    { className: "bg-primary text-white text-center p-3" },
                    "This is an HTML element")
}
```

Listing 9-6 and Listing 9-7 produce the same result, and when Babel processes the contents of the App.js file from Listing 9-6, it produces the code from Listing 9-7. When React executes the JavaScript code in the browser, it then uses the DOM API to create the HTML element, as demonstrated in Chapter 3. This may seem like a circular approach, but the JSX transformation is performed only during the build process and is intended to make writing React features easier.

Understanding the JavaScript Language Transformation

After years of stagnation, the JavaScript language has been revitalized and modernized with features that simplify development and provide features that are common in other programming languages, such as those features described in Chapter 4. Not all recent language features are supported by all browsers, especially older browsers or those used in corporate environments where updates are often rolled out slowly (if at all). Babel solves this problem by translating modern features into code that uses features that are supported by a much wider range of browsers, including those that pre-date the JavaScript renaissance.

In Listing 9-8, I have returned the App.js file to use HTML elements and used recent JavaScript features to set the content of the h4 element.

Listing 9-8. Using Modern JavaScript Features in the App.js File in the src Folder

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

let name = "Adam";
const city = "London";

export default class extends Component {

    message = () => `Hello ${name} from ${city}`;

    render = () =>
        <h4 className="bg-primary text-white text-center p-3">
            { this.message() }
        </h4>
}
```

This component relies on several recent JavaScript features: the `class` and `extends` keywords for defining classes, the `let` and `const` keywords for defining variables and constants, and a lambda function and template string in the `message` method. When you save the changes, the React development tools will automatically compile and bundle the JavaScript code and send it to the browser, producing the content shown in Figure 9-4.

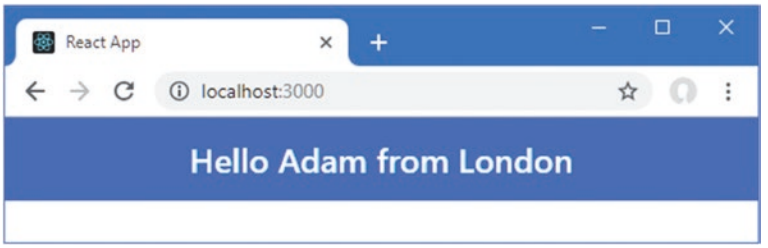

Figure 9-4. *Using modern language features*

To see how Babel has handled the modern JavaScript features, open the F12 developer tools, select the Sources tab, and locate the `main.chunk.js` item in the tree on the left side of the window, as shown in Figure 9-5. For the version of Chrome that was current at the time of writing, the file is under the `localhost:3000` > `static/js` part of the tree.

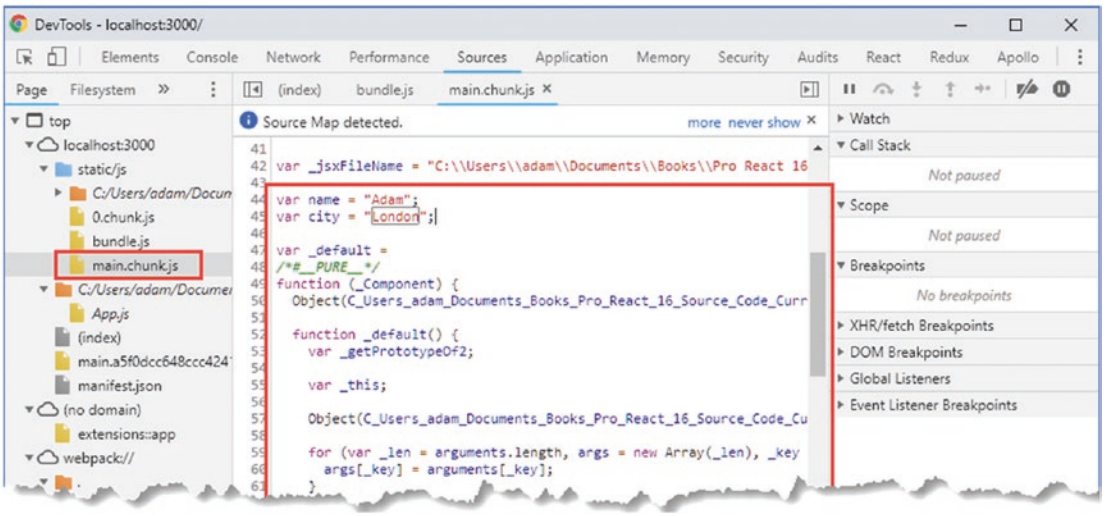

Figure 9-5. *Locating the compiled source code*

■ **Tip** The Google Chrome developer tools change often, and you may have to hunt around to locate the code produced by Babel. Using Ctrl+F and searching for *London* is a good way to locate the code you are looking for. An alternative approach is to paste the code from Listing 9-8 into the interpreter at `https://babeljs.io/repl`, which will produce a similar result.

If you scroll down—or search for *London*, as noted earlier—then you will see the code that Babel has produced. All the features that are not supported by older browsers are replaced with backward-compatible code, like this:

```
...
var name = "Adam";
var city = "London";

var App = function (_Component) {
    _inherits(App, _Component);

    function App() {
        var _ref;

        var _temp, _this, _ret;

        _classCallCheck(this, App);

        for (var _len = arguments.length, args = Array(_len), _key = 0;
                _key < _len; _key++) {
            args[_key] = arguments[_key];
        }

        return _ret = (_temp = (_this = _possibleConstructorReturn(this,
            (_ref = App.__proto__ || Object.getPrototypeOf(App)).call.apply(_ref,
                [this].concat(args))), _this), _this.message = function () {
                   return "Hello " + name + " from " + city;
                }, _temp), _possibleConstructorReturn(_this, _ret);
    }

    _createClass(App, [{
        key: "render",
        value: function render() {
            return __WEBPACK_IMPORTED_MODULE_0_react___default.a.createElement(
                "div",
                { className: "h1 bg-primary text-white text-center p-3", __source: {
                        fileName: _jsxFileName,
                        lineNumber: 12
                    },
                    __self: this
                },
                this.message()
            );
        }
    }]);
    return App;
}
...
```

You don't have to understand how this code works in detail, not least because some of it is convoluted and difficult to read. What's important is how the features used in the `App.js` file are handled, such as the `let` and `const` keywords, which are replaced with the traditional `var` keyword.

```
...
var name = "Adam";
var city = "London";
...
```

You can also see that the template string has been replaced with string concatenation, as shown here:

```
...
return "Hello " + name + " from " + city;
...
```

Some of the features, such as classes, are handled using functions that Babel adds to the bundle sent to the browser. The JSX HTML fragment is translated into a call to the `React.createElement` method.

The translation of modern features is complex, but recent additions to the JavaScript language are largely syntactic sugar intended to make coding more pleasant for the developer. Translating these features robs the code of these leasing features and requires some contortions to create an equivalent effect that older browsers can execute.

UNDERSTANDING THE LIMITS OF BABEL

Babel is an excellent tool, but it deals only with JavaScript language features. Babel is not able to add support for recent JavaScript APIs to browsers that do not implement them. You can still use these APIs—as I demonstrated in Part 1 when I used the Local Storage API—but doing so restricts the range of browsers that can run the application.

Understanding the Development HTTP Server

To simplify the development process, the project incorporates the `webpack-dev-server` package, which is an HTTP server that is integrated with webpack. The server is configured to start listening for HTTP requests on port 3000 as soon as the initial bundling process is complete. When an HTTP request is received, the development HTTP server returns the contents of the `public/index.html` file. When it processes the `index.html` file, the development server makes some important additions, which you can see by right-clicking in the browser window and selecting View Page Source from the pop-up menu.

```
<!DOCTYPE html>
<html lang="en">
  <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="/manifest.json">
    <link rel="shortcut icon" href="/favicon.ico">
    <title>React App</title>
  </head>
```

```
  <body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>
    <div id="root"></div>
    <script src="/static/js/bundle.js"></script>
    <script src="/static/js/0.chunk.js"></script>
    <script src="/static/js/main.chunk.js"></script>
    <script src="/main.a5f0dcc648ccc4241725.hot-update.js"></script>
  </body>
</html>
```

The development server adds `script` elements that tell the browser to load the files that contain the React framework, the application code, static content (such as CSS), and some additional features that support the development tools and automatically reload the browser when a change has been detected.

Understanding Static Content

There are two ways to include static content, such as images or CSS stylesheets in a React application. In most circumstances, the best approach is to add the files you need to the `src` folder and then declare dependencies on them in your code files using `import` statements.

To demonstrate how static content in the `src` folder is handled, I replaced the contents of the `App.css` file, which was added to the project when it was created, with the CSS style shown in Listing 9-9.

Listing 9-9. Replacing the Styles in the App.css File in the src Folder

```
img {
  background-color: lightcyan;
  width: 50%;
}
```

The style I defined selects `img` elements and sets the background color and width. In Listing 9-10, I added dependencies to two static files in the `src` folder to the `App` component, including the CSS file I updated in the previous listing and the placeholder image added to the project when it is created.

■ **Tip** The `index.css` file is imported by the `index.js` file, which is the JavaScript file responsible for starting the React application. You can define global styles in the CSS file, and they will be included in the content sent to the browser.

Listing 9-10. Declaring a Static Dependency in the App.js File in the src Folder

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

```
let name = "Adam";
const city = "London";

export default class extends Component {

    message = () => `Hello ${name} from ${city}`;

    render = () =>
        <div className="text-center">
            <h4 className="bg-primary text-white text-center p-3">
                { this.message() }
            </h4>
            <img src={ reactLogo } alt="reactLogo" />
        </div>
}
```

To import content that doesn't need to be referred to in order to be used, such as a CSS stylesheet, the `import` keyword is followed by the file name, which must include the file extension, like this:

```
...
import "./App.css";
...
```

To import content that will be referred to in an HTML element, such an image, then the form of the `import` statement that assigns a name to the imported feature must be used, like this statement:

```
...
import reactLogo from "./logo.svg";
...
```

This `statement` imports the `logo.svg` file and assigns it the name `reactLogo`, which I can then use in an expression in an `img` element, like this:

```
...
<img src={ reactLogo } alt="reactLogo" />
...
```

When you use the `import` keyword to declare a dependency on static content, the decision about how to handle the content is left to the development tools. For files that are smaller than 10Kb, the content will be included in the `bundle.js` file, along with the JavaScript code required to add the content to the HTML document. This is what happens with the `App.css` file that was imported in Listing 9-10: the contents of the CSS file will be included in the `bundle.js` file, along with the code required to create a `style` element.

For larger files—and any SVG files of any size—the imported file is requested in a separate HTTP request. The relative path specified by the `import` statement is automatically replaced by a URL that will locate the file, and the file name is changed so that it includes a checksum, which ensures that stale data won't be cached by the browser.

You can see the effect of the static content used in Listing 9-10 by saving the changes to the App.js file, waiting for the browser to reload, and then using the F12 developer's tools to examine the Elements tab, which will show the following HTML (although I have omitted the large number of Bootstrap CSS styles for brevity):

```
<html lang="en">
  <head>
    <meta charset="utf-8">
    <link rel="shortcut icon" href="/favicon.ico">
    <meta name="viewport" content="width=device-width,
         initial-scale=1, shrink-to-fit=no">
    <meta name="theme-color" content="#000000">
    <link rel="manifest" href="/manifest.json">
    <title>React App</title>
    <style type="text/css">
      img { background-color: lightcyan; width: 50% }
     </style>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root">
        <div class="text-center">
          <h4 class="bg-primary text-white text-center p-3">
            Hello Adam from London
          </h4>
          <img src="/static/media/logo.5d5d9eef.svg" alt="reactLogo">
        </div>
    </div>
    <script src="/static/js/bundle.js"></script>
    <script src="/static/js/1.chunk.js"></script>
    <script src="/static/js/main.chunk.js"></script>
    <script src="/main.00ec8a0c115561c18137.hot-update.js"></script>
  </body>
</html>
```

You can see that the CSS styles have been unpacked from the JavaScript bundle and added to the HTML document through a style element, whereas the image file is accessed through the URL /static/media/logo.5d5d9eef.svg. During the build process, large files are automatically copied into the location specified by the URLs that are included in the application's code, which means you don't have to worry about them being available. The changes in Listing 9-10 produce the result shown in Figure 9-6.

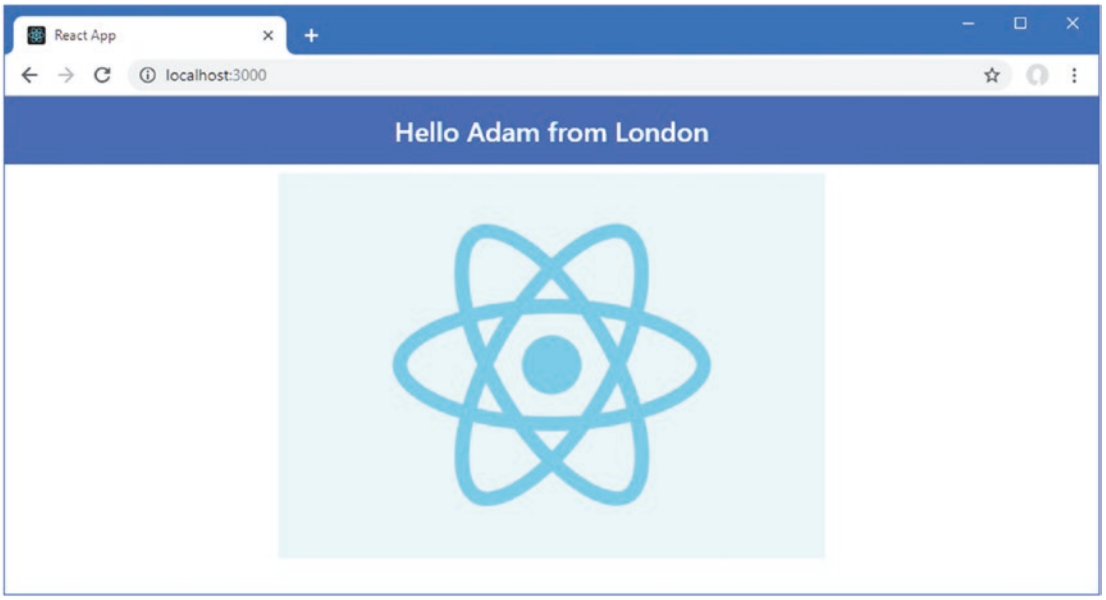

Figure 9-6. *Static content in the src folder*

Using the Public Folder for Static Content

There are several advantages to using the `src` folder for static content, but you may find that it isn't always suitable for every project, especially where static content isn't available at build time and cannot be processed by the React development tools. In these situations, you can put static content in the `public` folder, although this means you are responsible for ensuring the application has the files it requires. To demonstrate the use of the `public` folder, I added to it a file called `static.css`, with the content shown in Listing 9-11.

Listing 9-11. The Contents of the static.css File in the public Folder

```
img {
    border: 8px solid black;
}
```

Open a new command prompt, navigate to the `projecttools` folder, and run the command shown in Listing 9-12, which will copy the `logo.svg` file from the `src` folder into the `public` folder.

Listing 9-12. Copying an Image File into the Public Folder

```
cp src/logo.svg public/
```

In Listing 9-13, I have added HTML elements to the content rendered by the `App` component for the image and stylesheet in the `public` folder.

Listing 9-13. Accessing Static Files in the App.js File in the src Folder

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

let name = "Adam";
const city = "London";

export default class extends Component {

    message = () => `Hello ${name} from ${city}`;

    render = () =>
        <div className="text-center">
            <h4 className="bg-primary text-white text-center p-3">
                { this.message() }
            </h4>
            <img src={ reactLogo } alt="reactLogo" />
            <link rel="stylesheet"
                  href={ process.env.PUBLIC_URL + "/static.css"} />
            <img src={ process.env.PUBLIC_URL + "/logo.svg" } alt="reactLogo" />
        </div>
}
```

To specify the URL for the static files, the `process.env.PUBLIC_URL` property is combined with the file name in an expression. Notice that I have added a `link` element for the stylesheet because I cannot rely on the code in the `bundle.js` file to create the styles automatically. The result of the elements added to the component is shown in Figure 9-7.

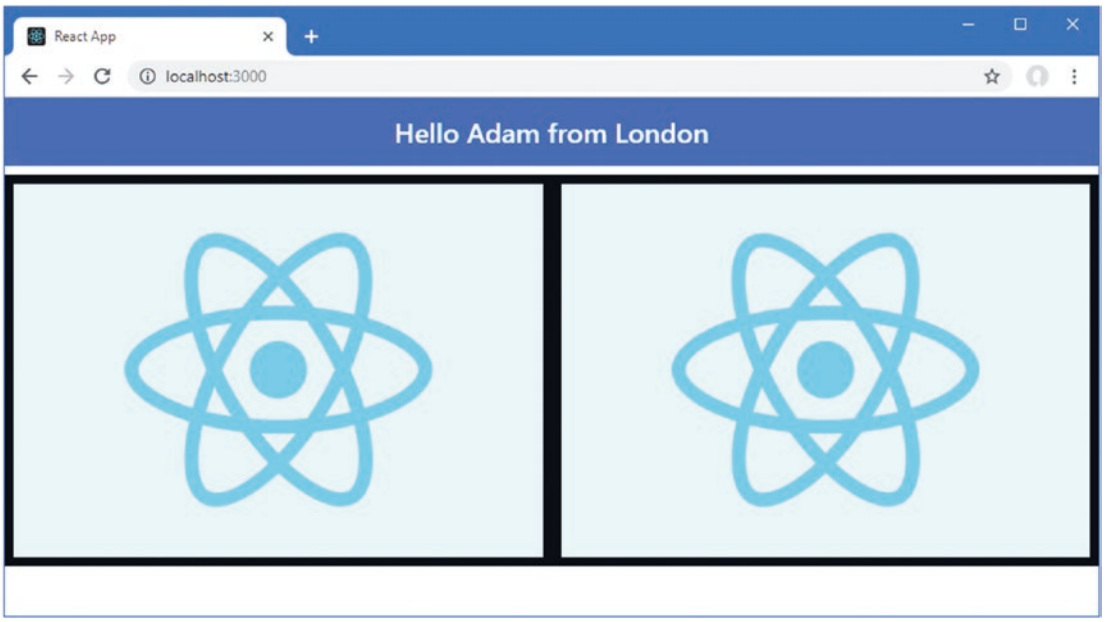

Figure 9-7. *Static content in the public folder*

Understanding the Error Display

One effect of the immediacy provided by the automatic-reload feature is that you will tend to stop watching the console output during development because your focus will naturally gravitate to the browser window. The risk is that the content displayed by the browser remains static when the code contains errors because the compilation process can't produce a new module to send to the browser through the HMR feature. To address this, the bundle produced by webpack includes an integrated error display that shows details of problems in the browser window. To demonstrate the way that an error is handled, I added the statement shown in Listing 9-14 to the App.js file.

Listing 9-14. Creating an Error in the App.js File in the src Folder

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

let name = "Adam";
const city = "London";

not a valid statement

export default class extends Component {

    message = () => `Hello ${name} from ${city}`;

    render = () =>
        <div className="text-center">
            <h4 className="bg-primary text-white text-center p-3">
                { this.message() }
            </h4>
            <img src={ reactLogo } alt="reactLogo" />
            <link rel="stylesheet"
                    href={ process.env.PUBLIC_URL + "/static.css"} />
            <img src={ process.env.PUBLIC_URL + "/logo.svg" } alt="reactLogo" />
        </div>
}
```

The addition isn't a valid JavaScript statement. When the change to the file is saved, the build process tries to compile the code and generates the following error message at the command prompt:

```
...
Failed to compile.

./src/App.js
  Line 8:  Parsing error: Unexpected token, expected ";"

   6 | const city = "London";
   7 |
>  8 | not a valid statement
     |     ^
   9 |
  10 | export default class extends Component {
```

```
  11 |
...
```

The same error message is displayed in the browser window so you will realize there is a problem even if you are not paying attention to the command-line messages. If you click the stack trace, then the browser will send an HTTP request to the development server, which will try to figure out which code editor you are using and highlight the problem, as shown in Figure 9-8.

■ **Tip** You may need to configure the React development tools to specify your editor, as described in the "Configuring the Development Tools" section, and not all editors are supported. Figure 9-8 shows Visual Studio Code, which is one of the editors for which support is provided.

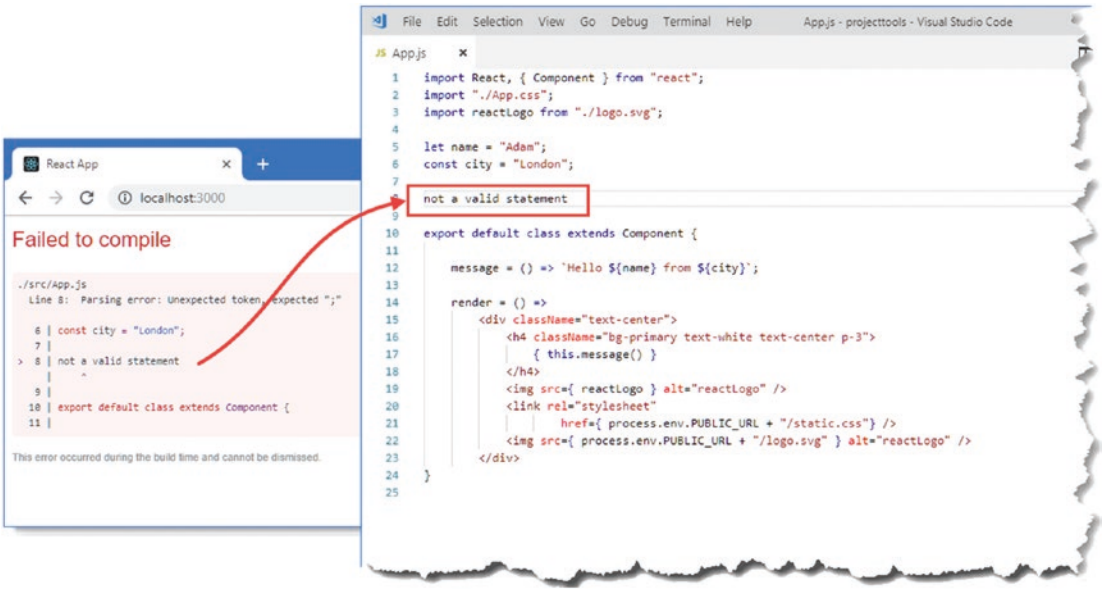

Figure 9-8. *Following an error to the source code file*

Understanding the Linter

The React development tools include a linter, which is responsible for checking the code and content in a project conform to a set of rules. When you create a project using the `create-react-app` package, the ESLint package is used as the linter with a set of rules that are intended to help programmers avoid common errors. As a demonstration, I added a variable to the `App.js` file, as shown in Listing 9-15. (This change also has the effect of removing the statement that causes the compiler error in the previous section).

Listing 9-15. Adding a Variable in the App.js File in the src Folder

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

```
let name = "Adam";
const city = "London";

let error = "not a valid statement";

export default class extends Component {

    message = () => `Hello ${name} from ${city}`;

    render = () =>
        <div className="text-center">
            <h4 className="bg-primary text-white text-center p-3">
                { this.message() }
            </h4>
            <img src={ reactLogo } alt="reactLogo" />
            <link rel="stylesheet"
                    href={ process.env.PUBLIC_URL + "/static.css"} />
            <img src={ process.env.PUBLIC_URL + "/logo.svg" } alt="reactLogo" />
        </div>
}
```

When you save the file, you will see the following warning displayed at the command line and also in the browser's JavaScript console:

```
...
Compiled with warnings.

./src/App.js
  Line 8:  'error' is assigned a value but never used  no-unused-vars
...
```

The linter cannot be disabled or reconfigured, which means that you will receive linting warnings for a fixed set of rules, including the `no-unused-vars` rule, which is the one broken by Listing 9-15. You can see the set of rules that are applied in React projects at https://github.com/facebook/create-react-app/tree/master/packages/eslint-config-react-app.

When you receive a warning, a search for the rule name will provide you with a description of the problem. In this case, a search for no-unused-vars will lead you to https://eslint.org/docs/rules/no-unused-vars, which explains that variables cannot be defined and not used.

Disabling Linting for Individual Statements and Files

Although the linter cannot be disabled, you can add comments to files to prevent warnings. In Listing 9-16, I have disabled the `no-unused-var` rule for a single statement by adding a comment.

Listing 9-16. Disabling a Single Linting Rule in the App.js File in the src Folder

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

```
let name = "Adam";
const city = "London";

// eslint-disable-next-line no-unused-vars
let error = "not a valid statement";

export default class extends Component {

    message = () => `Hello ${name} from ${city}`;

    render = () =>
        <div className="text-center">
            <h4 className="bg-primary text-white text-center p-3">
                { this.message() }
            </h4>
            <img src={ reactLogo } alt="reactLogo" />
            <link rel="stylesheet"
                    href={ process.env.PUBLIC_URL + "/static.css"} />
            <img src={ process.env.PUBLIC_URL + "/logo.svg" } alt="reactLogo" />
        </div>
}
```

If you want to disable every rule for a next statement, then you can omit the rule name, as shown in Listing 9-17.

Listing 9-17. Disabling All Linting Rules in the App.js File in the src Folder

```
...
// eslint-disable-next-line
let error = "not a valid statement";
...
```

If you want to disable a rule for an entire file, then you can add a comment to the top of the file, as shown in Listing 9-18.

Listing 9-18. Disabling a Single Rule for a File in the App.js File in the src Folder

```
/* eslint-disable no-unused-vars */

import React, { Component } from "react";
import "./App.css";
import reactLogo from "./logo.svg";

let name = "Adam";
const city = "London";

let error = "not a valid statement";

export default class extends Component {

    message = () => `Hello ${name} from ${city}`;
```

```
    render = () =>
        <div className="text-center">
            <h4 className="bg-primary text-white text-center p-3">
                { this.message() }
            </h4>
            <img src={ reactLogo } alt="reactLogo" />
            <link rel="stylesheet"
                    href={ process.env.PUBLIC_URL + "/static.css"} />
            <img src={ process.env.PUBLIC_URL + "/logo.svg" } alt="reactLogo" />
        </div>
}
```

If you want to disable linting for all rules for a single file, then you can omit the rule name from the comment, as shown in Listing 9-19.

Listing 9-19. Disabling All Rules for a File in the App.js File in the src Folder

```
...
/* eslint-disable */

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

let name = "Adam";
const city = "London";
...
```

The linter will ignore the contents of the App.js file but will still check the contents of the other files in the project.

USING TYPESCRIPT OR FLOW

Linting isn't the only way to detect common errors and a good complementary technique is *static type checking*, in which you add details of the data types for variables and function results to your code to create a policy that is enforced by the compiler. For example, you might specify that a function always returns a string or that its first parameter can only be a number. When the application is compiled, the code that uses that function is checked to ensure that it only passes number values as arguments and treats the result only as a string.

There are two common ways to add static type checking to a React project. The first is to use TypeScript, which is a superset of JavaScript produced by Microsoft. TypeScript makes working with JavaScript more like C# or Java and includes support for static type checking. If you want to use TypeScript, then use the --scripts-version argument when you create the project, like this:

```
...
npx create-react-app projecttools --scripts-version=react-scripts-ts
...
```

The `react-scripts-ts` value produces a project that is set up with the TypeScript tools and features. You can learn more about TypeScript at `https://www.typescriptlang.org`.

An alternative is a package called Flow, which is focused solely on type checking and doesn't have the broader features of TypeScript. You can learn more about Flow at `https://flow.org`

Configuring the Development Tools

The React development tools provide a small number of configuration options, although these won't be required in most projects. The available options are described in Table 9-8.

Table 9-8. *The React Development Tools Configuration Options*

Name	Description
BROWSER	This option is used to specify the browser that is opened when the development tools complete the initial build process. You can specify a browser by specifying its path or disable this feature by using `none`.
HOST	This option is used to specify the hostname that the development HTTP server binds to, which defaults to `localhost`.
PORT	This option is used to specify the port that the development HTTP server uses, which defaults to `3000`.
HTTPS	When set to `true`, this option enables SSL for the development HTTP server, which generates a self-signed certificate. The default is `false`.
PUBLIC_URL	This option is used to change the URL used to request content from the `public` folder, as described in the *Understanding Static Content* section.
CI	When set to `true`, this option treats all warnings as errors in the build process. The default value is `false`.
REACT_EDITOR	This option is used to specify the editor for the feature that opens the code file when you click on a stack trace in the browser, as described in the *Understanding the Error Display* section.
CHOKIDAR_USEPOLLING	This option should be set to `true` when the development tools cannot detect changes to the `src` folder, which may happen if you are working in a virtual machine or a container.
GENERATE_SOURCEMAP	Settings this option to `false` disables the generation of source maps, which the browser uses to correlate the bundled JavaScript code with the source files in the project during debugging. The default is `true`.
NODE_PATH	This setting is used to specify the locations that will be searched for Node.js modules.

These options are specified either by setting environment variables or by creating an `.env` file, which is the approach that I find most reliable. To demonstrate the configuration process, I added a file called `.env` to the `projecttools` folder and added the configuration statements shown in Listing 9-20.

Listing 9-20. The Contents of the .env File in the projecttools Folder

```
PORT=3500
HTTPS=true
```

I used the `PORT` option to specify port 3500 for receiving requests and the `HTTPS` option to enable SSL in the development server. To see the effect of the changes, stop the development tools and run the command shown in Listing 9-21 to start them again.

Listing 9-21. Starting the React Development Tools

```
npm start
```

When the initial build process is complete, the browser window that is opened will navigate to `https://localhost:3500`. Most browsers will display a warning about the self-signed certificate and then display the web application once you have clicked on the Advanced link (or its equivalent) and told the browser to proceed, as shown in Figure 9-9.

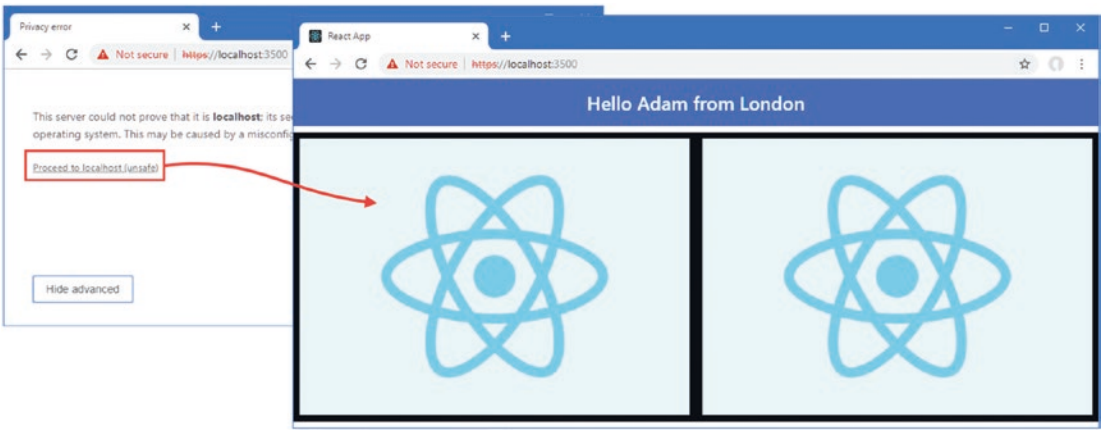

Figure 9-9. *Configuring the Development Tools*

Debugging React Applications

Not all problems can be detected by the compiler or the linter, and code that compiles perfectly can behave in unexpected ways. There are two ways to understand the behavior of your application, as described in the sections that follow. To help demonstrate the debugging features, I added a file called `Display.js` to the `src` folder and used it to define the component shown in Listing 9-22.

Listing 9-22. The Contents of the Display.js File in the src Folder

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

export class Display extends Component {
```

```
    constructor(props) {
        super(props);
        this.state = {
            counter: 1
        }
    }

    incrementCounter = () => {
        this.setState({ counter: this.state.counter + 1 });
    }

    render() {
        return (
            <div>
                <h2 className="bg-primary text-white text-center p-2">
                    <div>Props Value: { this.props.value }</div>
                    <div>Local Value: { this.state.counter } </div>
                </h2>
                <div className="text-center">
                    <button className="btn btn-primary m-2"
                            onClick={ this.props.callback }>
                        Parent
                    </button>
                    <button  className="btn btn-primary m-2"
                            onClick={ this.incrementCounter }>
                        Local
                    </button>
                </div>
            </div>
        )
    }
}
```

The component displays its own state property and a prop value it receives from its parents. It displays two `button` elements, one of which changes the state property and one of which invokes a callback provided as a prop. In Listing 9-23, I replaced the existing contents of the `App` component to prepare for the debugging section.

Listing 9-23. Replacing the Contents of the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            city: "London"
        }
    }
```

```
    changeCity = () => {
        this.setState({ city: this.state.city === "London" ? "New York" : "London"})
    }

    render() {
        return (
            <Display value={ this.state.city } callback={ this.changeCity } />
        );
    }
}
```

When you save the changes to the JavaScript files, the application will be compiled, and you will see the content shown in Figure 9-10.

■ **Note** You may find that the browser doesn't update automatically when the `HTTPS` option is set to true in the `.env` file. You can reload the browser manually to see the changes or disable this option and restart the development tools.

Figure 9-10. *Adding functionality to the example application*

Exploring the Application State

The React Devtools browser extension is an excellent tool for exploring the state of a React application. There are versions available for Google Chrome and Mozilla Firefox, and details of the project—including support for other platforms and details of a stand-alone version—can be found at `https://github.com/facebook/react-devtools`. Once you have installed the extension, you will see an additional tab in the browser's developer tools window, which is accessed by pressing the F12 button (which is why these are also known as the F12 tools).

The React tab in the F12 tools window allows you to explore and alter the application's structure and state. You can see the set of components that provide the application functionality, along with their state data and their props.

For the example application, if you open the React tab and expand the application structure in the left pane, you will see the App and Display components in the left pane, displayed with the view of the HTML elements presented by the application. When you select a component in the left page, its props and state data are displayed in the right pane, as shown in Figure 9-11.

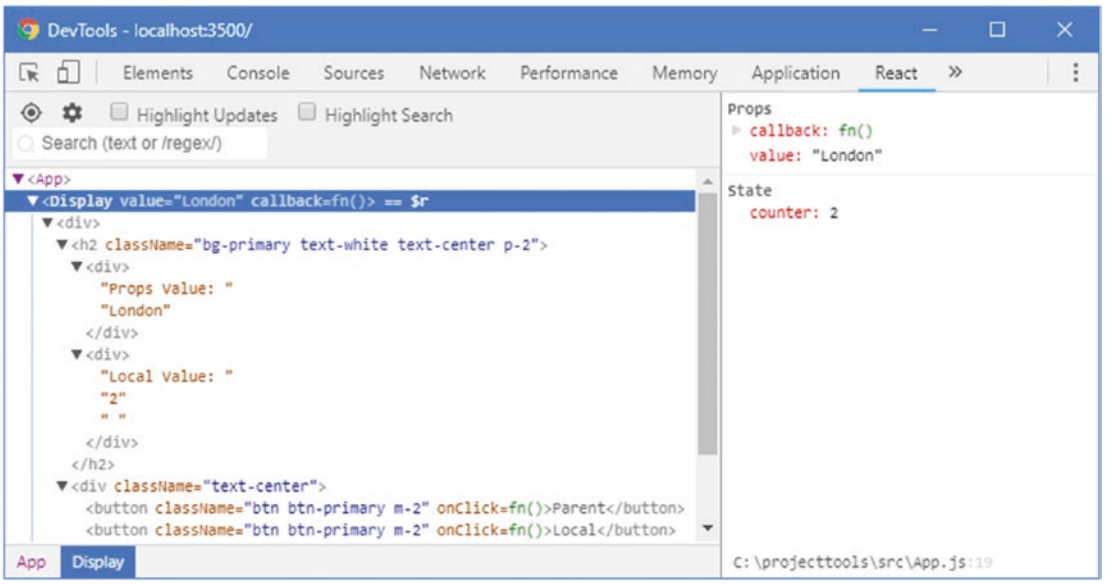

Figure 9-11. *Exploring components using the React Devtools*

If you click the buttons in the browser window, you will see the values displayed by the React Devtools change, reflecting the live state of the application. You can also click a state data value and change its values through React Devtools, which allows the state of the application to be manipulated directly.

■ **Tip** There are also debugging tools for the Redux data store package, which I describe in Chapter 19 and which is often used to manage the data for complex projects.

Using the Browser Debugger

Modern browsers include sophisticated debuggers that can be used to control the execution of an application and examine its state. The React development tools include support for creating source maps, which allow the browser to correlate the minified and bundled code that it is executing with the developer-friendly source code required for productive debugging.

Some browsers let you navigate through the application's source code using these source maps and create breakpoints, which will halt the execution of the application when they are reached and pass control to the debugger. As I write this, the ability to create breakpoints is a fragile feature that doesn't work on Chrome and has mixed reliability in other browsers. As a consequence, the most reliable way to pass control of the application to the debugger is to use the JavaScript debugger keyword, as shown in Listing 9-24.

Listing 9-24. Triggering the Debugger in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            city: "London"
        }
    }

    changeCity = () => {
        debugger
        this.setState({ city: this.state.city === "London" ? "New York" : "London"})
    }

    render() {
        return (
            <Display value={ this.state.city } callback={ this.changeCity } />
        );
    }
}
```

To use the debugger effectively, disable the HTTPS option in the `.env` file, as shown in Listing 9-25. If you do not disable this option, you will only see the code generated by Babel and not your original source code.

Listing 9-25. Disabling Secure Connections in the .env File in the projecttools Folder

```
PORT=3500
HTTPS=false
```

Stop the development tools and start them again by running the command shown in Listing 9-26 in the `projecttools` folder.

Listing 9-26. Starting the Development Tools

```
npx start
```

The application will be executed as normal, but when the Parent button is clicked and the `changeCity` method is invoked, the browser will encounter the `debugger` keyword and halt the execution of the application. You can then use the controls in the F12 tools window to inspect the variables and their values at the point at which execution was stopped and manually control execution, as shown in Figure 9-12. The browser is executing the minified and bundled code created by the development tools but displaying the corresponding code from the source map.

■ **Tip** Most browsers ignore the `debugger` keyword unless the F12 tools window is open, but it is good practice to remove it at the end of a debugging session.

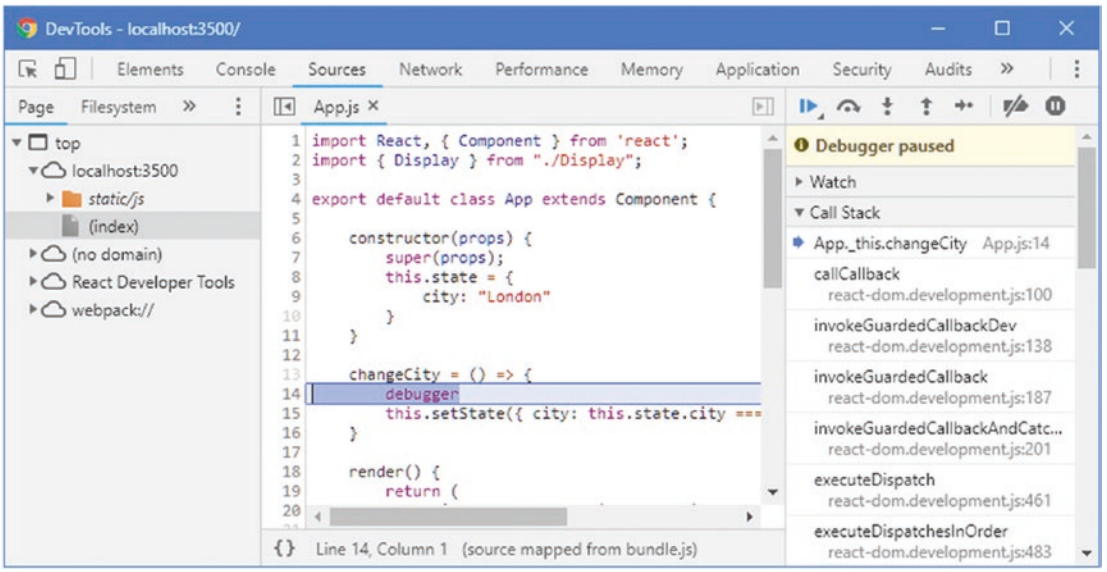

Figure 9-12. *Using the browser debugger*

Summary

In this chapter, I described the structure of React projects created with the `create-react-app` package and explained the purpose of the files and folders used in React development. I also explained how the React development tools are used, how applications are bundled for use in the browser, how the error display and linter help avoid common problems, and how you can debug applications when you don't receive the results you are expecting. In the next chapter, I introduce components, which are the key building block for React applications.