CHAPTER 1

Your First React Application

The best way to get started with React is to dive in. In this chapter, I take you through a simple development process to create an application to keep track of to-do items. In Chapters 5–8, I show you how to create a more complex and realistic application, but, for now, a simple example will be enough to demonstrate how React applications are created and how the basic features work. Don't worry if you don't understand everything in this chapter—the idea is to get an overall sense of how React works. I explain everything in detail in later chapters.

Note If you want a conventional description of React features, you can jump to Part 2 of this book, where I start the process of describing individual features in depth. Before you go, make sure you install the development tools and packages described in this chapter.

Preparing the Development Environment

There is some preparation required for React development. In the sections that follow, I explain how to get set up and ready to create your first project.

Installing Node.js

The tools used for React development rely on Node.js—also known as Node—which was created in 2009 as a simple and efficient runtime for server-side applications written in JavaScript. Node.js is based on the JavaScript engine used in the Chrome browser and provides an API for executing JavaScript code outside of the browser environment.

Node.js has enjoyed success as an application server, but for this book it is interesting because it has provided the foundation for a new generation of cross-platform development and build tools.

It is important that you download the same version of Node.js that I use throughout this book. Although Node.js is relatively stable, there are still breaking API changes from time to time that may stop the examples I include in the chapters from working. The version I have used is 10.14.1, which is the current Long-Term Support release at the time of writing. There may be a later version available by the time you read this, but you should stick to the 10.14.1 release for the examples in this book. A complete set of 10.14.1 releases, with installers for Windows and macOS and binary packages for other platforms, is available at `https://nodejs.org/dist/v10.14.1`.

When you install Node.js, make sure you select the option to add the Node.js executables to the path. When the installation is complete, run the command shown in Listing 1-1.

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_1

Listing 1-1. Checking the Node Version

```
node -v
```

If the installation has gone as it should, then you will see the following version number displayed:

```
v10.14.1
```

The Node.js installer includes the Node Package Manager (NPM), which is used to manage the packages in a project. Run the command shown in Listing 1-2 to ensure that NPM is working.

Listing 1-2. Checking NPM Works

```
npm -v
```

If everything is working as it should, then you will see the following version number:

```
6.4.1
```

Installing the create-react-app Package

The `create-react-app` package is the standard way to create and manage complex React packages and provides developers with a complete toolchain. There are other ways to get started with React, but this is the approach that best suits most projects and is the one that I use throughout this book.

To install the package, open a new command prompt and run the command shown in Listing 1-3. If you are using Linux or macOS, you may need to use `sudo`.

Listing 1-3. Installing the create-react-app Package

```
npm install --global create-react-app@2.1.2
```

Installing Git

The Git revision control tool is required to manage some of the packages required for React development. If you are using Windows or macOS, then download and run the installer from `https://git-scm.com/downloads`. (On macOS, you may have to change your security settings to open the installer, which has not been signed by the developers.)

Git is already included in most Linux distributions. If you want to install the latest version, then consult the installation instructions for your distribution at `https://git-scm.com/download/linux`. As an example, for Ubuntu, which is the Linux distribution I use, I used the command shown in Listing 1-4.

Listing 1-4. Installing Git

```
sudo apt-get install git
```

Once you have completed the installation, open a new command prompt and run the command shown in Listing 1-5 to check that Git is installed and available.

Listing 1-5. Checking Git

```
git --version
```

This command prints out the version of the Git package that has been installed. At the time of writing, the latest version of Git for Windows and Linux is 2.20.1, and the latest version of Git for macOS is 2.19.2.

Installing an Editor

React development can be done with any programmer's editor, from which there is an endless number to choose. Some editors have enhanced support for working with React, including highlighting keywords and expressions. If you don't already have a preferred editor for web application development, then you can consider some of the popular options in Table 1-1. I don't rely on any specific editor for this book, and you should use whichever editor you are comfortable working with.

Table 1-1. *Popular Programming Editors*

Name	Description
Sublime Text	Sublime Text is a commercial cross-platform editor that has packages to support most programming languages, frameworks, and platforms. See `www.sublimetext.com` for details.
Atom	Atom is an open-source, cross-platform editor that has a particular emphasis on customization and extensibility. See `atom.io` for details.
Brackets	Brackets is a free open-source editor developed by Adobe. See `brackets.io` for details.
Visual Studio Code	Visual Studio Code is an open-source, cross-platform editor from Microsoft, with an emphasis on extensibility. See `code.visualstudio.com` for details.
Visual Studio	Visual Studio is Microsoft's flagship developer tool. There are free and commercial editions available, and it comes with a wide range of additional tools that integrate into the Microsoft ecosystem.

Installing a Browser

The final choice to make is the browser that you will use to check your work during development. All the current-generation browsers have good developer support and work well with React, but there is a useful extension for Chrome and Firefox called `react-devtools` that provides insights into the state of a React application and that is especially useful in complex projects. See `https://github.com/facebook/react-devtools` for details of installing the extension. I used Google Chrome throughout this book, and this is the browser I recommend you use to follow the examples.

Creating the Project

Projects are created and managed from the command line. Open a new command prompt, navigate to a convenient location, and run the command shown in Listing 1-6 to create the project for this chapter.

■ **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`.

Listing 1-6. Creating the Project

```
npx create-react-app todo
```

The `npx` command was installed as part of the Node.js/NPM package in the previous section and is used to run Node.js packages. The `create-react-app` argument tells `npx` to run the `create-react-app` package that is used to create new React projects and was installed in Listing 1-3. The final argument is `todo`, which is the name of the project to create. When you run this command, the project will be created, and all of the packages required for developing and running a React project will be downloaded and installed. The setup process can take a while because there are a large number of packages to download.

■ **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.

Understanding the Project Structure

Open the `todo` folder using your preferred editor, and you will see the project structure shown in Figure 1-1. The figure shows the layout in my preferred editor—Visual Studio—and you may see the project content presented slightly differently if you have chosen a different editor.

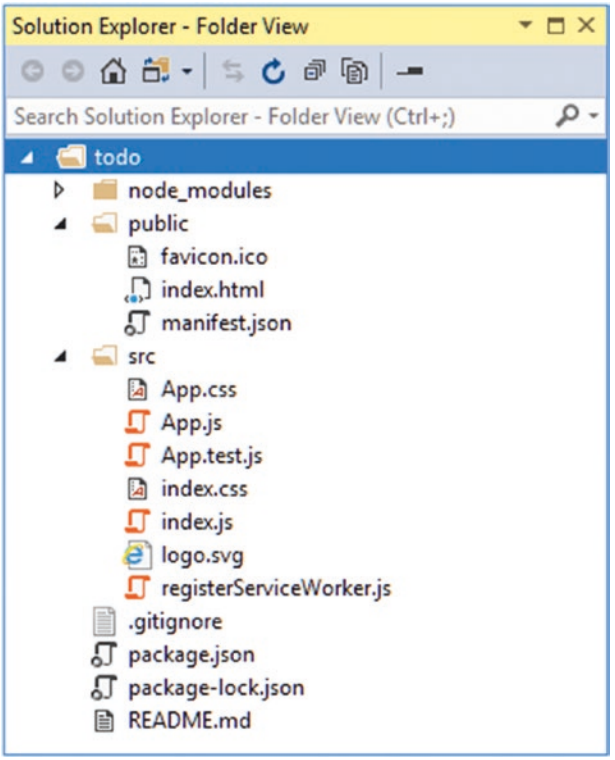

Figure 1-1. *The project structure*

This is the starting point for all projects, and while the purpose of each file may not be obvious at the moment, you will know what each file and folder is for by the end of the book. For the moment, Table 1-2 briefly describes the files that are important for this chapter, and I provide a detailed explanation of React projects in Chapter 9.

Table 1-2. *The Important Files in the Project for This Chapter*

Name	Description
`public/index.html`	This is the HTML file that is loaded by the browser. It contains an element in which the application is displayed and a `script` element that loads the application's JavaScript files.
`src/index.js`	This is the JavaScript file that is responsible for configuring and starting the React application. I use this file to add the Bootstrap CSS framework to the application in the next section.
`src/App.js`	This is the React component, which contains the HTML content that will be displayed to the user and the JavaScript code required by the HTML. Components are the main building blocks in a React application, and you will see them used throughout this book.

Adding the Bootstrap CSS Framework

I use the excellent Bootstrap CSS framework to style the HTML presented by the examples in this book. I describe the basic use of Bootstrap in Chapter 3, but to get started in this chapter, run the commands shown in Listing 1-7 to navigate to the `todo` folder and add the Bootstrap package to the project.

■ **Tip** The command used to manage the packages in a project is `npm`, which is confusingly similar to `npx`, which is used only when creating a new project. It is important not to confuse the two commands.

Listing 1-7. Adding the Bootstrap CSS Framework

```
cd todo
npm install bootstrap@4.1.2
```

To include Bootstrap in the application, add the statement shown in Listing 1-8 to the `index.js` file.

Listing 1-8. 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();
```

As I explain in Chapter 4, the `import` statement is used to declare a dependency so that it becomes part of the application. The `import` keyword is most often used to declare dependencies on JavaScript code, but it can also be used for CSS stylesheets.

Starting the Development Tools

When you create a project using the `create-react-app` package, a complete set of development tools is installed so that the project can be compiled, packaged up, and delivered to the browser. Using the command prompt, run the commands shown in Listing 1-9 in the `todo` folder to start the development tools.

Listing 1-9. Starting the Development Tools

```
npm start
```

There is an initial preparation process when the development tools start, which can take a moment to complete. Don't be put off by the amount of time the preparation takes because this process is required only when you start a development session. When the startup process is complete, you will see a message like this one, which confirms that the application is running and tells you which HTTP port to connect to:

```
Compiled successfully!
You can now view todo 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.
```

The default port used to listen for HTTP requests is 3000, although a different port will be selected if 3000 is in use. Once the initial preparation for the project is complete, a new browser window will open and display the URL `http://localhost:3000` and the placeholder content shown in Figure 1-2.

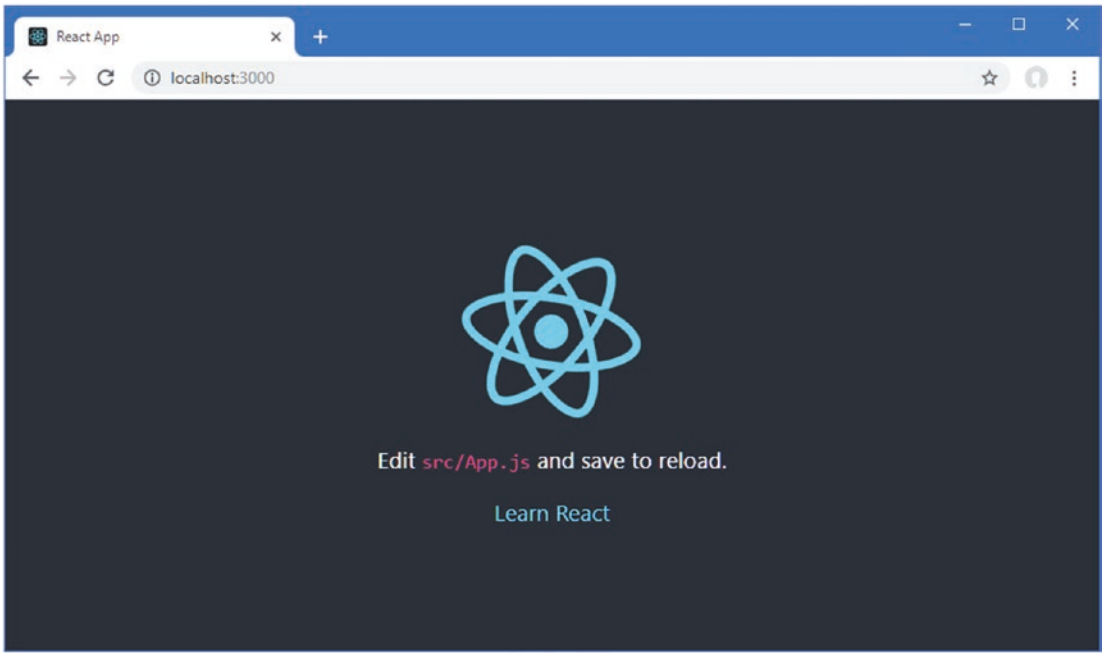

Figure 1-2. *Running the example application*

Replacing the Placeholder Content

The content that is displayed in Figure 1-2 is a placeholder that is used to ensure that the development tools are working. To replace the default content, I changed the `App.js` file, as shown in Listing 1-10.

Listing 1-10. Removing the Placeholder in the App.js File in the src Folder

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

export default class App extends Component {

    render() {
        return (
            <div>
                <h4 className="bg-primary text-white text-center p-2">
                    To Do List
                </h4>
            </div>
        )
    };
}
```

The `App.js` file contains a React *component*, which is named `App`. Components are the main building block for React applications, and they are written using JSX, which is a superset of JavaScript that allows HTML to be included in code files without requiring any special quoting. I describe JSX in more detail in Chapter 3, but in this listing, the `App` component defines a `render` method that React calls to get the content to display to the user.

■ **Tip** React supports recent additions to the JavaScript language, such as the `class` keyword, which is used in Listing 1-10. I provide a primer for the most useful JavaScript features in Chapter 4.

When you save the `App.js` file, the React development tools automatically detect the changes, rebuild the application, and instruct the browser to reload, showing the content in Figure 1-3.

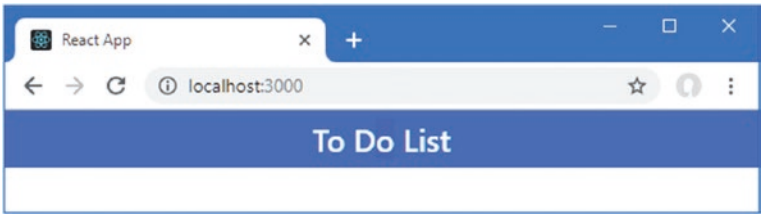

Figure 1-3. *Replacing the placeholder content*

The JSX files used in React development make it easy to mix HTML and JavaScript, but there are some important differences from regular HTML files. You can see a common example in the h4 element in Listing 1-10, shown here:

```
...
<h4 className="bg-primary text-white text-center p-2">
    To Do List
</h4>
...
```

In regular HTML, the class attribute is used to assign elements to classes, which is how elements are styled when using the Bootstrap CSS framework. Even though it might not appear so, JSX files are JavaScript files, and JavaScript configures classes through the className property. The differences between pure HTML and JSX can be jarring when you first begin React development, but they soon will become second nature.

■ **Tip** I provide a brief overview of working with the Bootstrap CSS framework in Chapter 3, where I explain the meaning of the classes to which the h4 element has been assigned in Listing 1-10, such as bg-primary, text-white, and p-2. You can ignore these classes for the moment, however, and just focus on the basic structure of the application.

React will write a warning message to the browser's JavaScript console if you forget you are working with JSX and use standard HTML instead. If you use the class attribute instead of className, for example, you will see the Invalid DOM property 'class'. Did you mean 'className'? warning. To see the browser's JavaScript console, press the F12 key and select the Console or JavaScript Console tab.

Displaying Dynamic Content

All web applications need to display dynamic content to the user, and React makes this easy by supporting the *expressions* feature. An expression is a fragment of JavaScript that is evaluated when a component's render method is called and provides the means to display data to the user. Many expressions are used to display data values defined by the component to keep track of the state of the application, known as *state data*. State data and expressions are easier to understand when you see an example, and Listing 1-11 adds both to the App component.

Listing 1-11. Adding State Data and Data Bindings in the App.js File in the src Folder

```
import React, { Component } from 'react';
export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            userName: "Adam"
        }
    }
```

```
    render() {
        return (
            <div>
                <h4 className="bg-primary text-white text-center p-2">
                    { this.state.userName }'s To Do List
                </h4>
            </div>
        )
    };
}
```

The `constructor` is a special method that is invoked when the component is initialized, and calling the `super` method within the constructor is required to ensure that the component is set up properly, as I explain in Chapter 11. The `props` parameter defined by the constructor is important in React development because it allows one component to configure another, which you will see shortly.

■ **Tip** The term *props* is short for *properties*, and it reflects the way React creates the HTML content that is displayed in the browser, as I explain in Chapter 3.

React components have a special property named `state`, which is used to define state data, like this:

```
...
this.state = {
    userName: "Adam"
}
...
```

The `this` keyword refers to the current object and is used to access its properties and methods. The highlighted statement assigns an object with a `userName` property to `this.state`, which is all that is required to set up state data. Once state data has been defined, it can be included in the content generated by the component in an expression, like this:

```
...
<h4 className="bg-primary text-white text-center p-2">
    { this.state.userName }'s To Do List
</h4>
...
```

Expressions are denoted with curly braces (the { and } characters). When the `render` method is invoked, the expression is evaluated, and its result is included in the content presented to the user. The expression in Listing 1-11 reads the value of the `userName` state data property, producing the result shown in Figure 1-4.

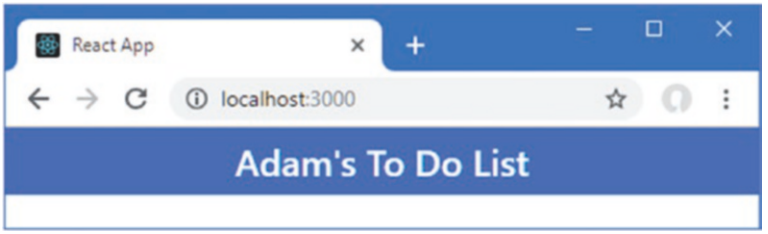

Figure 1-4. *Using state data and expressions in the App.js file in the src Folder*

Understanding State Data Changes

The dynamic nature of a React application is based on changes to state data, which React responds to by invoking the component's `render` method again, which causes the expressions to be re-evaluated using the new state data values. In Listing 1-12, I have updated the `App` component so that the value of the `userName` state data property is changed.

Listing 1-12. Changing State Data in the App.js File in the src Folder

```
import React, { Component } from 'react';
export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            userName: "Adam"
        }
    }

    changeStateData = () => {
        this.setState({
            userName: this.state.userName === "Adam" ? "Bob" : "Adam"
        })
    }

    render() {
        return (
            <div>
                <h4 className="bg-primary text-white text-center p-2">
                    { this.state.userName }'s To Do List
                </h4>
                <button className="btn btn-primary m-2"
                        onClick={ this.changeStateData }>
                    Change
                </button>
            </div>
        )
    };
}
```

Save the changes to the App.js file, and you will see a button in the browser window. Clicking the button changes the username, as shown in Figure 1-5.

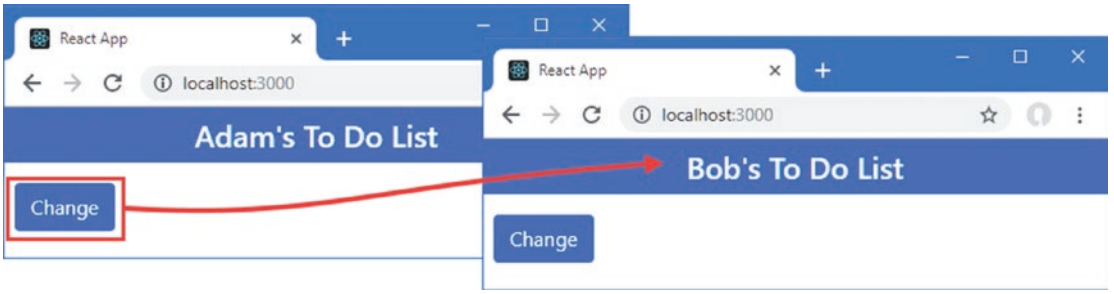

Figure 1-5. *Changing the username*

This example contains several important React features working together. The first is the onClick attribute on the button element.

```
...
<button className="btn btn-primary m-2" onClick={ this.changeStateData }>
    Change
</button>
...
```

The onClick attribute is assigned an expression that React evaluates when the button is clicked. Clicking a button triggers an *event*, and onClick is an example of an event-handler prop. The function or method that is specified by onClick will be invoked each time the button is clicked. The expression in Listing 1-12 specifies the changeStateData method, which is defined using the *fat arrow* syntax, which allows functions to be expressed concisely, as shown here:

```
...
changeStateData = () => {
    this.setState({ userName: this.state.userName === "Adam" ? "Bob" : "Adam" })
}
...
```

As I explain in Chapter 4, fat arrow functions are used to simplify responding to events, but they be used more widely and help keep the mix of HTML and JavaScript readable in a React application. The changeStateData method uses the setState method to set a new value for the userName property. When the setState method is called, React updates the component's state data with the new values and then invokes the render method so that the expressions will generate updated content. This is why clicking the button changes the name shown in the browser window from Adam to Bob. I didn't have to explicitly tell React that the value used by the expression changed—I just called the setState method to set the new value and left React to update the content in the browser.

■ **Tip** The this keyword is required whenever you use the properties and methods defined by a component, including the setState method. Forgetting to use this is a common error in React development, and it is the first thing to check if you don't get the behavior you expect.

Functions defined using the fat arrow syntax don't use the `return` keyword or require curly braces around the function body, which can result in simpler and clearer `render` methods, for example, as shown in Listing 1-13.

Listing 1-13. Redefining a Method Using a Fat Arrow Function in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            userName: "Adam"
        }
    }

    changeStateData = () => {
        this.setState({
            userName: this.state.userName === "Adam" ? "Bob" : "Adam"
        })
    }

    render = () =>
        <div>
            <h4 className="bg-primary text-white text-center p-2">
                { this.state.userName }'s To Do List
            </h4>
            <button className="btn btn-primary m-2"
                    onClick={ this.changeStateData }>
                Change
            </button>
        </div>
}
```

I use both styles to define functions and methods in this book. For the most part, you can choose between conventional JavaScript functions and fat arrow functions, although there are some important considerations explained in Chapter 12.

Adding the To-Do Application Features

Now that you have seen how React can display dynamic content, it is time to start adding the features required by the application, starting with additional state data and expressions, as shown in Listing 1-14.

Listing 1-14. Adding Application Features in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
```

```
        this.state = {
            userName: "Adam",
            todoItems: [{ action: "Buy Flowers", done: false },
                        { action: "Get Shoes", done: false },
                        { action: "Collect Tickets", done: true },
                        { action: "Call Joe", done: false }],
            newItemText: ""
        }
    }

    updateNewTextValue = (event) => {
        this.setState({ newItemText: event.target.value });
    }

    createNewTodo = () => {
        if (!this.state.todoItems
                 .find(item => item.action === this.state.newItemText)) {
            this.setState({
                todoItems: [...this.state.todoItems,
                    { action: this.state.newItemText, done: false }],
                newItemText: ""
            });
        }
    }

    render = () =>
        <div>
            <h4 className="bg-primary text-white text-center p-2">
                {this.state.userName}'s To Do List
                ({ this.state.todoItems.filter(t => !t.done).length} items to do)
            </h4>
            <div className="container-fluid">
                <div className="my-1">
                    <input className="form-control"
                        value={ this.state.newItemText }
                        onChange={ this.updateNewTextValue } />
                    <button className="btn btn-primary mt-1"
                        onClick={ this.createNewTodo }>Add</button>
                </div>
            </div>
        </div>
}
```

Because React expressions are JavaScript, they can be used to inspect data values and generate results dynamically, like this expression:

```
...
<h4 className="bg-primary text-white text-center p-2">
    {this.state.userName}'s To Do List
    ({ this.state.todoItems.filter(t => !t.done).length} items to do)
</h4>
...
```

This expression filters the objects in the `todoItems` state data array so that only incomplete items are selected and then reads the value of the `length` property, which is the value that the binding will display to the user. The JSX format makes it easy to mix HTML elements and code like this, although complex expressions can be difficult to read and are often defined in a property or method to keep the HTML as simple as possible.

The changes in Listing 1-14 introduce an `input` element, which allows the user to enter the text for a new to-do item. The input element has two props, which are used to manage the content of the element and respond to changes, shown here:

```
...
<input className="form-control"
    value={ this.state.newItemText } onChange={ this.updateNewTextValue } />
...
```

The `value` prop is used to set the contents of the `input` element. In this case, the expression that the `value` prop contains will return the value of the `newItemText` state data property, which means that any change to the state data property will update the contents of the `input` element. The `onChange` prop tells React what to do when the `change` event is triggered, which will happen when the user types into the `input` element. This expression tells React to invoke the component's `updateNewTextValue` method, which uses the `setState` method to update the `newItemText` state data property. This may seem like a circular approach, but it ensures that React knows how to deal with changes performed by code and by the user.

The `button` element uses the `onClick` prop to tell React to invoke the `createNewTodo` method in response to the `click` event. The `createNewTodo` method checks that there an existing item with the same text and, if there is not, uses the `setState` method to add a new item to the `todoItems` array and resets the `newItemText` property, which has the effect of clearing the `input` element. The statement that adds the new item to the array does so with the JavaScript *spread* operator, which is a recent addition to the JavaScript language.

```
...
todoItems: [...this.state.todoItems,
    { action: this.state.newItemText, done: false }],
...
```

The spread operator is three periods, and it expands an array. The tools used for React development allow recent JavaScript features to be used and translates them into compatible code that can be understood by older web browsers. I describe the spread operator and other useful JavaScript features in Chapter 4.

To see the effect of the changes in Listing 1-14, enter a description of a task into the text field and click the Add button. React responds to the event by invoking the method specified by the button's `onClick` prop, which uses the value of the `input` element to create a new to-do item. You can't see the description of the task yet, but you will see that the number of incomplete tasks increases, as shown in Figure 1-6.

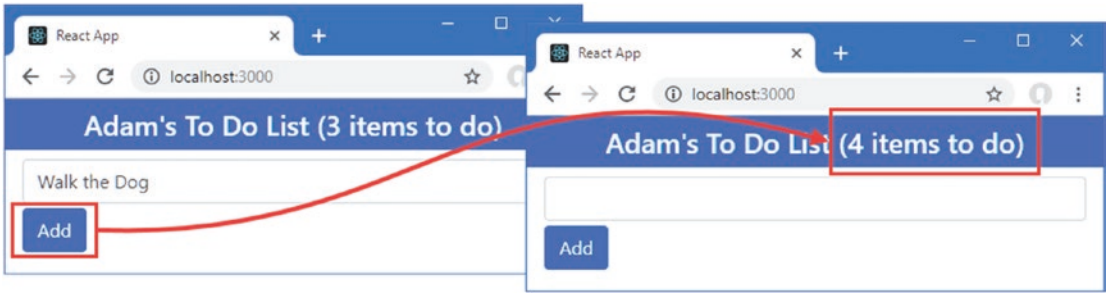

Figure 1-6. *Adding a new task*

Displaying the To-Do Items

The next step is to display each to-do item to the user so they can see details of the task and mark them complete when they are done, as shown in Listing 1-15.

Listing 1-15. Displaying To-Do Items in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            userName: "Adam",
            todoItems: [{ action: "Buy Flowers", done: false },
                        { action: "Get Shoes", done: false },
                        { action: "Collect Tickets", done: true },
                        { action: "Call Joe", done: false }],
            newItemText: ""
        }
    }

    updateNewTextValue = (event) => {
        this.setState({ newItemText: event.target.value });
    }

    createNewTodo = () => {
        if (!this.state.todoItems
                .find(item => item.action === this.state.newItemText)) {
            this.setState({
                todoItems: [...this.state.todoItems,
                    { action: this.state.newItemText, done: false }],
                newItemText: ""
            });
        }
    }
```

```
    toggleTodo = (todo) => this.setState({ todoItems:
        this.state.todoItems.map(item => item.action === todo.action
            ? { ...item, done: !item.done } : item) });

    todoTableRows = () => this.state.todoItems.map(item =>
        <tr key={ item.action }>
            <td>{ item.action}</td>
            <td>
                <input type="checkbox" checked={ item.done }
                    onChange={ () => this.toggleTodo(item) } />
            </td>
        </tr> );

    render = () =>
        <div>
            <h4 className="bg-primary text-white text-center p-2">
                {this.state.userName}'s To Do List
                ({ this.state.todoItems.filter(t => !t.done).length} items to do)
            </h4>
            <div className="container-fluid">
                <div className="my-1">
                    <input className="form-control"
                        value={ this.state.newItemText }
                        onChange={ this.updateNewTextValue } />
                    <button className="btn btn-primary mt-1"
                        onClick={ this.createNewTodo }>Add</button>
                </div>
                <table className="table table-striped table-bordered">
                    <thead>
                        <tr><th>Description</th><th>Done</th></tr>
                    </thead>
                    <tbody>{ this.todoTableRows() }</tbody>
                </table>
            </div>
        </div>
}
```

So far, the emphasis in the `App.js` file has been embedding a JavaScript expression in fragments of HTML. But the JSX format allows HTML and JavaScript to be mixed freely, which means that JavaScript methods can return HTML content. You can see an example in Listing 1-15, where the `todoTableRows` method uses the JavaScript `map` method to produce a sequence of HTML elements for each object in the `todoItems` array, like this:

```
...
todoTableRows = () => this.state.todoItems.map(item =>
    <tr key={ item.action }>
        <td>{ item.action}</td>
        <td>
            <input type="checkbox" checked={ item.done }
                onChange={ () => this.toggleTodo(item) } />
        </td>
    </tr> );
...
```

Each item in the array is mapped to a `tr` element, which is the HTML element for a table row. Within the `tr` element is a set of `td` elements that define HTML table cells. The HTML content produced by the `map` method contains further JavaScript expressions that populate the `td` elements with state data values or functions that will be invoked to handle an event.

React does enforce some restrictions on the content it handles, such as the `key` prop added to each `tr` element by the `todoTableRows` method, shown here:

```
...
<tr key={ item.action }>
...
```

As you will learn in detail in Chapter 13, React invokes a component's `render` method when there is a change and compares the result with the HTML that is displayed in the browser so that only the differences are applied. React requires the `key` prop so that it can correlate the content is displayed with the data that produced it and manage changes efficiently.

The result of the changes in Listing 1-15 is that each to-do item is displayed with a checkbox that the user toggles to indicate that the task is complete. Each table row generated by the `todoTableRows` method contains an `input` element configured as a checkbox.

The result of the changes in Listing 1-15 is that the list of to-do items is displayed in a table and that checking an item as complete reduces the number displayed in the header, as shown in Figure 1-7.

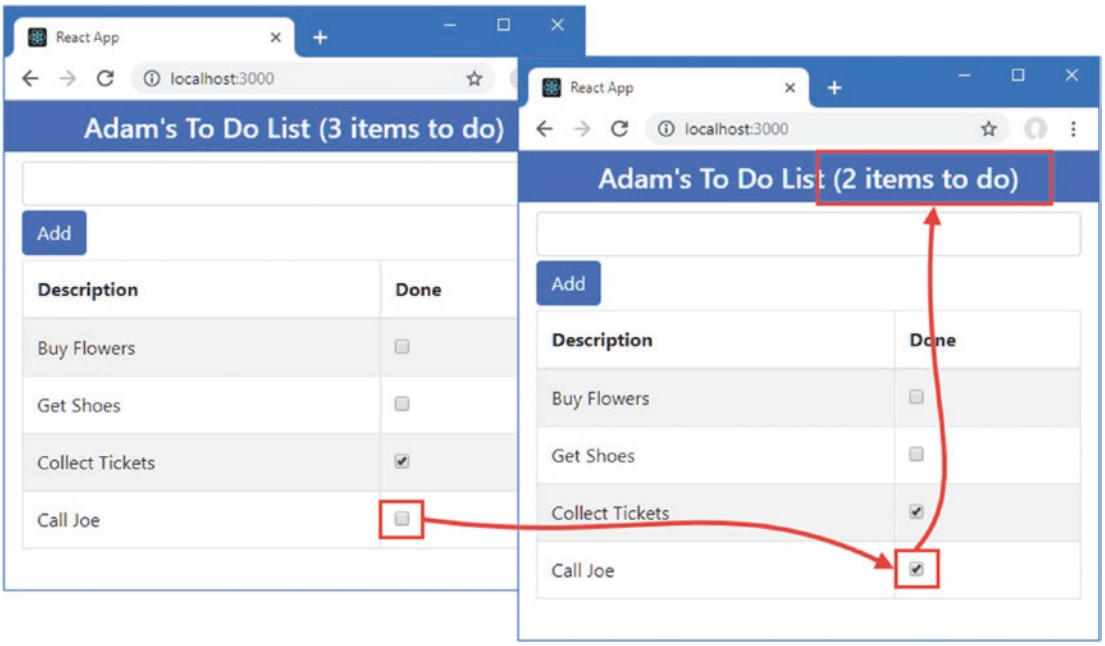

Figure 1-7. *Displaying the to-do items*

Introducing Additional Components

At the moment, all of the example application's functionality is contained in a single component, which can become difficult to manage as new features are added. To help keep components manageable, functionality is delegated up into separate components that are responsible for specific features. These are known as *child components*, while the component that delegated the functionality is known as the *parent*.

In this section, I am going to introduce several child components, each of which will be responsible for a single feature. I started by adding a file called `TodoBanner.js` to the `src` folder and using it to define the component shown in Listing 1-16.

Listing 1-16. The Contents of the TodoBanner.js File in the src Folder

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

export class TodoBanner extends Component {

    render = () =>
        <h4 className="bg-primary text-white text-center p-2">
            { this.props.name }'s To Do List
            ({ this.props.tasks.filter(t => !t.done).length } items to do)
        </h4>
}
```

This component is responsible for displaying the banner. Parent components provide their children with data using *props*, and the data values are accessed through the `props` property, accessed via the `this` keyword. This component, which is called `TodoBanner`, expects to receive two props: a `name` prop, which contains the user's name, and a `tasks` prop, which contains the set of tasks and which is filtered to display the number that are incomplete. To display the value of the `name` prop, for example, the component uses an expression that contains `this.props.name`, like this:

```
...
{ this.props.name }'s To Do List
...
```

When React invokes the `TodoBanner` component's `render` method, the value of the `name` prop provided by the parent component will be included in the result. The other expression in the `TodoBanner` component's `render` method uses the JavaScript `filter` method to select the incomplete items and determine how many there are, showing that props can be used in expressions that do more than just display their value.

Next, I created a file called `TodoRow.js` in the `src` folder and used it to define the component shown in Listing 1-17.

Listing 1-17. The Contents of the TodoRow.js File in the src Folder

```
import React, { Component } from 'react';
export class TodoRow extends Component {

    render = () =>
        <tr>
            <td>{ this.props.item.action}</td>
            <td>
                <input type="checkbox" checked={ this.props.item.done }
```

```
                    onChange={ () => this.props.callback(this.props.item) }
                />
            </td>
        </tr>
}
```

This component will be responsible for displaying a single row in the table, showing details of a to-do item. The data that is received by a child component through its props is read-only and must not be altered. To make changes, parent components can use *function props* to provide children with callback functions that are invoked when something important happens. This combination allows collaboration between components: data props allow a parent to provide data to a child, and function props allow a child to communicate with this parent.

The component in Listing 1-17 defines a data prop named `item` that is used to receive the to-do item to be displayed, and it defines a function prop named `callback` that provides a function that is invoked when the user toggles the checkbox. For the final child component, I added a file called `TodoCreator.js` to the `src` folder and added the code shown in Listing 1-18.

Listing 1-18. The Contents of the TodoCreator.js File in the src Folder

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

export class TodoCreator extends Component {

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

    updateNewTextValue = (event) => {
        this.setState({ newItemText: event.target.value});
    }

    createNewTodo = () => {
        this.props.callback(this.state.newItemText);
        this.setState({ newItemText: ""});
    }

    render = () =>
        <div className="my-1">
            <input className="form-control" value={ this.state.newItemText }
                onChange={ this.updateNewTextValue } />
            <button className="btn btn-primary mt-1"
                onClick={ this.createNewTodo }>Add</button>
        </div>
}
```

Child components can have their own state data, which is what this component uses to handle the content of its `input` element. The component invokes a function prop to notify its parent when the user clicks the Add button.

Using the Child Components

The components I defined in the previous section take responsibility for specific features of the to-do application. In Listing 1-19, I have updated the App component to use the three new components, each of which is configured using props to provide them with the data and callback functions they require.

Listing 1-19. Applying Child Components in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { TodoBanner } from "./TodoBanner";
import { TodoCreator } from "./TodoCreator";
import { TodoRow } from "./TodoRow";

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            userName: "Adam",
            todoItems: [{ action: "Buy Flowers", done: false },
                        { action: "Get Shoes", done: false },
                        { action: "Collect Tickets", done: true },
                        { action: "Call Joe", done: false }],
            //newItemText: ""
        }
    }

    updateNewTextValue = (event) => {
        this.setState({ newItemText: event.target.value });
    }

    createNewTodo = (task) => {
        if (!this.state.todoItems.find(item => item.action === task)) {
            this.setState({
                todoItems: [...this.state.todoItems, { action: task, done: false }]
            });
        }
    }

    toggleTodo = (todo) => this.setState({ todoItems:
        this.state.todoItems.map(item => item.action === todo.action
            ? { ...item, done: !item.done } : item) });

    todoTableRows = () => this.state.todoItems.map(item =>
        <TodoRow key={ item.action } item={ item } callback={ this.toggleTodo } />)

    render = () =>
        <div>
            <TodoBanner name={ this.state.userName } tasks={this.state.todoItems } />
            <div className="container-fluid">
                <TodoCreator callback={ this.createNewTodo } />
                <table className="table table-striped table-bordered">
```

```
                    <thead>
                        <tr><th>Description</th><th>Done</th></tr>
                    </thead>
                    <tbody>{ this.todoTableRows() }</tbody>
                </table>
            </div>
        </div>
}
```

The new `import` statements declare dependencies on the child components, which ensures they are included in the application during the build process. Child components are used as custom HTML elements, with attributes and expressions defining the props that the component will receive, like this:

```
...
<TodoBanner name={ this.state.userName } tasks={this.state.todoItems } />
...
```

The expressions used to set the prop values provide a child component with access to specific data and methods defined by its parent. In this case, the `name` and `tasks` props are used to provide the `TodoBanner` component with the values of the `userName` and `todoItems` state data properties.

Adding the Finishing Touches

The basic features of the application are in place, and the set of components that provide those features are all working together. In this section, I add some finishing touches to complete the to-do application.

Managing the Visibility of Completed Tasks

At the moment, tasks always remain visible to the user even when they have been completed. To address this, I will present the user with separate lists of complete and incomplete tasks and allow the incomplete tasks to be hidden. I added a file called `VisibilityControl.js` to the `src` folder and used it to define the component shown in Listing 1-20.

Listing 1-20. The Contents of the VisibilityControl.js File in the src Folder

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

export class VisibilityControl extends Component {

    render = () =>
        <div className="form-check">
            <input className="form-check-input" type="checkbox"
                checked={ this.props.isChecked }
                onChange={ (e) => this.props.callback(e.target.checked) } />
            <label className="form-check-label">
                Show { this.props.description }
            </label>
        </div>
}
```

Using props to receive data and callback functions from a parent makes it easy to add new features to an application. The component defined in Listing 1-20 is a general-purpose feature that has no knowledge of the content that it is being used to manage, and it works entirely through its props: the `description` prop provides the label text it displays, the `isChecked` prop provides the initial state for the checkbox, and the `callback` prop provides the function that is invoked when the user toggles the checkbox and triggers the change event.

In Listing 1-21, I have updated the `App` component to apply the `VisibilityControl` component as a child, along with the changes required to display the completed and incomplete tasks separately.

Listing 1-21. Managing Completed Tasks in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { TodoBanner } from "./TodoBanner";
import { TodoCreator } from "./TodoCreator";
import { TodoRow } from "./TodoRow";
import { VisibilityControl } from "./VisibilityControl";

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            userName: "Adam",
            todoItems: [{ action: "Buy Flowers", done: false },
                        { action: "Get Shoes", done: false },
                        { action: "Collect Tickets", done: true },
                        { action: "Call Joe", done: false }],
            showCompleted: true
        }
    }

    updateNewTextValue = (event) => {
        this.setState({ newItemText: event.target.value });
    }

    createNewTodo = (task) => {
        if (!this.state.todoItems.find(item => item.action === task)) {
            this.setState({
                todoItems: [...this.state.todoItems, { action: task, done: false }]
            });
        }
    }

    toggleTodo = (todo) => this.setState({ todoItems:
        this.state.todoItems.map(item => item.action === todo.action
            ? { ...item, done: !item.done } : item) });

    todoTableRows = (doneValue) => this.state.todoItems
        .filter(item => item.done === doneValue).map(item =>
            <TodoRow key={ item.action } item={ item }
                  callback={ this.toggleTodo } />)
```

```
    render = () =>
        <div>
            <TodoBanner name={ this.state.userName }
                tasks={this.state.todoItems } />
            <div className="container-fluid">
                <TodoCreator callback={ this.createNewTodo } />
                <table className="table table-striped table-bordered">
                    <thead>
                        <tr><th>Description</th><th>Done</th></tr>
                    </thead>
                    <tbody>{ this.todoTableRows(false) }</tbody>
                </table>
                <div className="bg-secondary text-white text-center p-2">
                    <VisibilityControl description="Completed Tasks"
                        isChecked={this.state.showCompleted}
                        callback={ (checked) =>
                            this.setState({ showCompleted: checked })} />
                </div>

                { this.state.showCompleted &&
                    <table className="table table-striped table-bordered">
                        <thead>
                            <tr><th>Description</th><th>Done</th></tr>
                        </thead>
                        <tbody>{ this.todoTableRows(true) }</tbody>
                    </table>
                }
            </div>
        </div>
}
```

The `VisibilityControl` component is configured so it changes the value of the `App` component's state data property named `showCompleted` when the user toggles the checkbox. To separate the complete and incomplete tasks, I added a parameter to the `todoTableRows` method and used the `filter` method to select objects from the state data array based on the value of the `done` property.

To display the completed tasks, I added a second `table` element. The table will be displayed only when the `showCompleted` property is `true`, so I placed the `table` and its content inside a data binding expression and used the && operator, like this:

```
...
{ this.state.showCompleted && <table className="table table-striped table-bordered">
...
```

When the expression is evaluated, the `table` element will be included in the component's content only if the `showCompleted` property is `true`. This is another example of how JSX mixes content and code. For the most part, JSX does a good job at blending elements and code statements, but it doesn't excel at everything, and the syntax required for conditional statements is awkward, as this example shows.

When you save the changes to the `App.js` file, you will see the separate sets of tasks. When you toggle the checkbox for a task, it will be moved to the other table, as shown in Figure 1-8. When you toggle the Show Completed Tasks checkbox, the second table will be hidden.

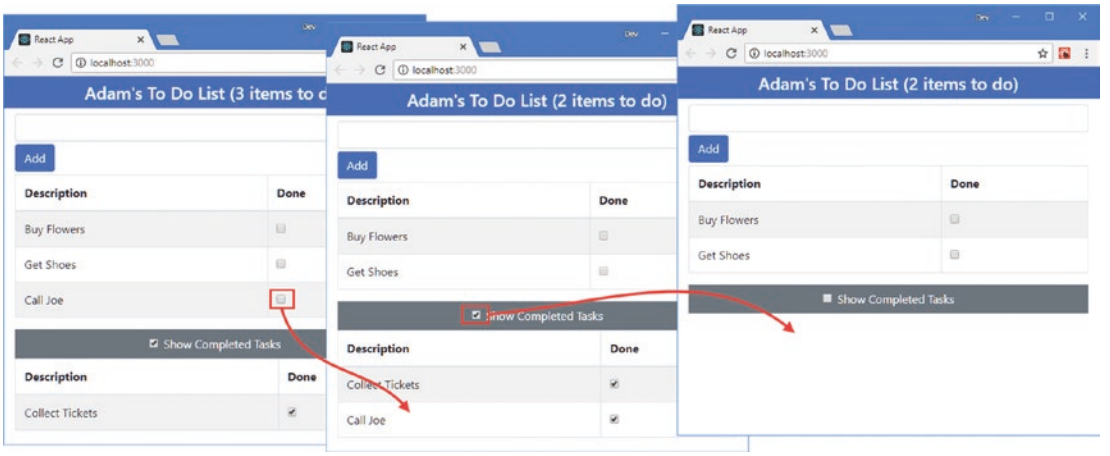

Figure 1-8. *Changing the task display*

Persistently Storing Data

The final change is to store the data so that the user's list is preserved when navigating away from the application. Later in the book, I demonstrate different ways of working with data stored on a server, but for this chapter I am going to keep the application simple and ask the browser to store the data using the Local Storage API, as shown in Listing 1-22.

■ **Tip** The Local Storage API is a standard browser feature and isn't specific to React development. See `https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage` for a good description of how local storage works.

Listing 1-22. Persistently Storing Data in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { TodoBanner } from "./TodoBanner";
import { TodoCreator } from "./TodoCreator";
import { TodoRow } from "./TodoRow";
import { VisibilityControl } from "./VisibilityControl";

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            userName: "Adam",
            todoItems: [{ action: "Buy Flowers", done: false },
                        { action: "Get Shoes", done: false },
                        { action: "Collect Tickets", done: true },
```

```
                        { action: "Call Joe", done: false }],
            showCompleted: true
        }
    }

    updateNewTextValue = (event) => {
        this.setState({ newItemText: event.target.value });
    }

    createNewTodo = (task) => {
        if (!this.state.todoItems.find(item => item.action === task)) {
            this.setState({
                todoItems: [...this.state.todoItems, { action: task, done: false }]
            }, () => localStorage.setItem("todos", JSON.stringify(this.state)));
        }
    }

    toggleTodo = (todo) => this.setState({ todoItems:
        this.state.todoItems.map(item => item.action === todo.action
            ? { ...item, done: !item.done } : item) });

    todoTableRows = (doneValue) => this.state.todoItems
        .filter(item => item.done === doneValue).map(item =>
            <TodoRow key={ item.action } item={ item }
                callback={ this.toggleTodo } />)

    componentDidMount = () => {
        let data = localStorage.getItem("todos");
        this.setState(data != null
            ? JSON.parse(data)
            :  {
                userName: "Adam",
                todoItems: [{ action: "Buy Flowers", done: false },
                            { action: "Get Shoes", done: false },
                            { action: "Collect Tickets", done: true },
                            { action: "Call Joe", done: false }],
                showCompleted: true
            });
    }

    render = () =>
        <div>
            <TodoBanner name={ this.state.userName }
                tasks={this.state.todoItems } />
            <div className="container-fluid">
                <TodoCreator callback={ this.createNewTodo } />
                <table className="table table-striped table-bordered">
                    <thead>
                        <tr><th>Description</th><th>Done</th></tr>
                    </thead>
                    <tbody>{ this.todoTableRows(false) }</tbody>
                </table>
```

```
                <div className="bg-secondary text-white text-center p-2">
                    <VisibilityControl description="Completed Tasks"
                        isChecked={this.state.showCompleted}
                        callback={ (checked) =>
                            this.setState({ showCompleted: checked })} />
                </div>

                { this.state.showCompleted &&
                    <table className="table table-striped table-bordered">
                        <thead>
                            <tr><th>Description</th><th>Done</th></tr>
                        </thead>
                        <tbody>{ this.todoTableRows(true) }</tbody>
                    </table>
                }
            </div>
        </div>
}
```

The Local Storage API is accessed through the `localStorage` object, and the component uses the `setItem` method to store the to-do items when a new to-do item is created. The local storage feature is only able to store string values, so I serialize the data objects as JSON before they can be stored. The `setState` method can accept a function that will be updated once the state data has been updated, as described in Chapter 11, and that ensures that the most recent data is stored.

Components have a well-defined lifecycle, which is described in Chapter 13, and can implement methods to receive notifications about important events. The component in the listing implements the `componentDidMount` method, which is invoked early in the component's life and provides a good opportunity to perform tasks such as loading data.

To retrieve the stored data, I have used the Local Storage API's `getItem` method. I use the `setState` method to update the component with the stored data or with some default data if there is no stored data available.

There is no visual change, but the application will persistently store any to-do items you create, which means they will still be available when you reload the browser window or navigate away to a different URL, such as the Apress home page, and then back to `http://localhost:3000`, as shown in Figure 1-9.

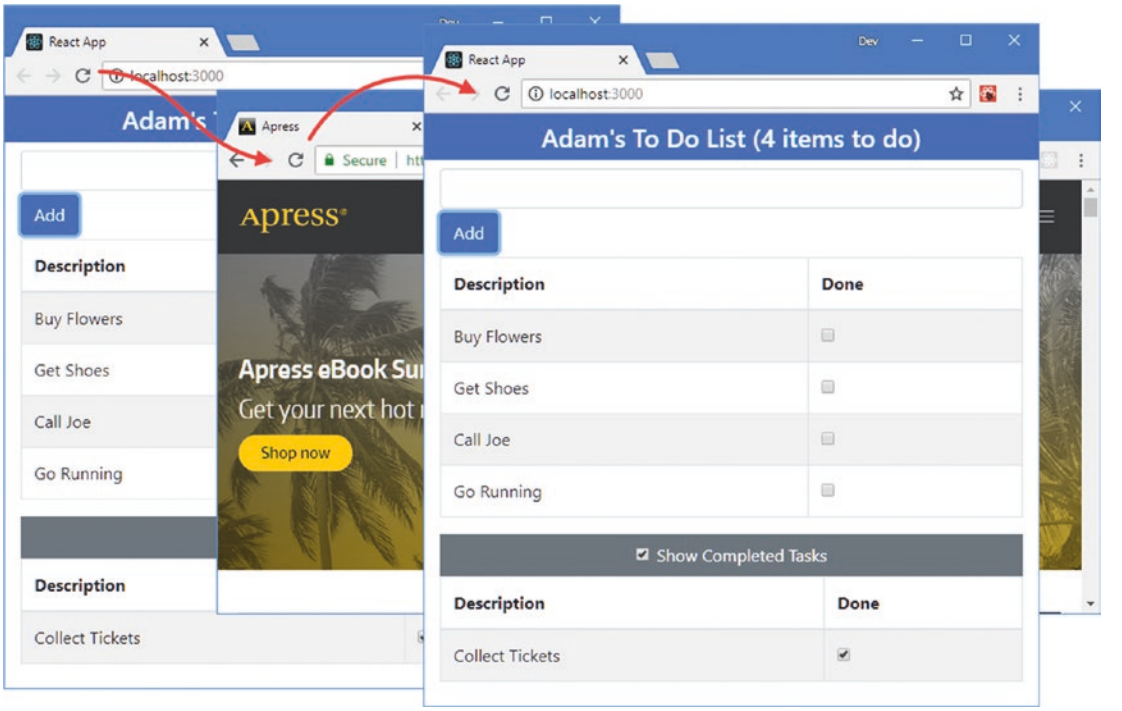

Figure 1-9. *Storing data*

Summary

In this chapter, I created a simple example application to introduce you to the React development process and to demonstrate some important React concepts. You saw that React development is focused on components, which are defined in JSX files that combine JavaScript code and HTML content. When you create a project, everything that is required to work with JSX files, build the application, and deliver it to the browser for testing is included so that you can get started quickly and easily.

You also learned that React applications can contain multiple components, each of which is responsible for a specific feature and which receive the data and callback functions they require using props.

Many more React features are available, as you can tell from the size of this book, but the basic application I created in this chapter has shown you the most essential characteristics of React development and will provide a foundation for later chapters. In the next chapter, I put React in context and describe the structure and content of this book.