CHAPTER 3

HTML, JSX, and CSS Primer

In this chapter, I provide a brief overview of HTML and explain how HTML content can be mixed with JavaScript code when using JSX, which is the superset of JavaScript supported by the React development tools that allows HTML to be mixed with code. I also introduce the Bootstrap CSS framework, which I use to style the content in the examples throughout this book.

■ **Note** Don't worry if not all the features described in this chapter make immediate sense. Some rely on recent additions to the JavaScript language that you may not have encountered before, which are described in Chapter 4 or explained in detail in other chapters.

Preparing for This Chapter

To create the project for this chapter, open a new command prompt, navigate to a convenient location, and run the command shown in Listing 3-1.

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

Listing 3-1. Creating the Example Project

```
npx create-react-app primer
```

Once the project has been created, run the commands shown in Listing 3-2 to navigate to the project folder and install the Bootstrap CSS framework.

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_3

■ **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.

Listing 3-2. Adding the Bootstrap Package to the Project

```
cd primer
npm install bootstrap@4.1.2
```

To include Bootstrap in the application, add the statement shown in Listing 3-3 to the `index.js` file.

Listing 3-3. 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();
```

Preparing the HTML File and the Component

To prepare for the examples in the chapter, replace the contents of the `index.html` file in the `public` folder with the content shown in Listing 3-4.

Listing 3-4. Replacing the Contents of the index.html File in the public Folder

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Primer</title>
  </head>
  <body>
    <h4 class="bg-primary text-white text-center p-2 m-1">
        Static HTML Element
    </h4>
    <div id="domParent"></div>
    <div id="root"></div>
  </body>
</html>
```

Replace the contents of the `App.js` file in the `src` folder with the code shown in Listing 3-5.

Listing 3-5. Replacing the Contents of the App.js File in the src folder

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

export default class App extends Component {
    render = () =>
        <h4 className="bg-primary text-white text-center p-2 m-1">
            Component Element
        </h4>
}
```

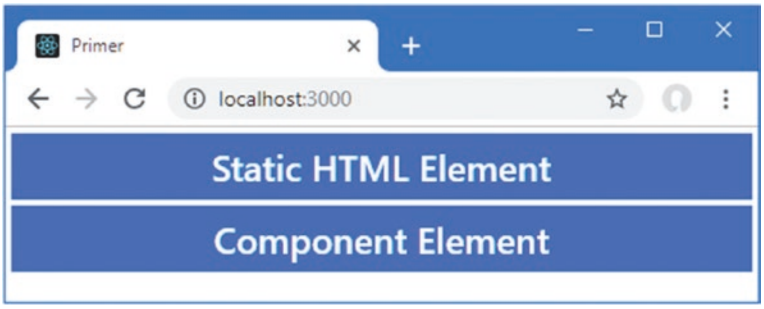
Running the Example Application

Ensure that all the changes are saved and use the command prompt to run the command shown in Listing 3-6 in the `primer` folder.

Listing 3-6. Starting the Development Tools

```
npm start
```

The React development tools will start, and once the initial preparations are complete, a new browser window will open and display the content shown in Figure 3-1.



Static HTML Element
Component Element

Figure 3-1. *Running the example application*

Understanding HTML and DOM Elements

At the heart of all React web applications are HTML elements, which are used to describe the content that will be presented to the user. In a React application, the contents of the static `index.html` file in the `public` folder are combined with the HTML elements created dynamically by React to produce an HTML document that the browser displays to the user.

An HTML element tells the browser what kind of content each part of an HTML document represents. Here is an HTML element from the `index.html` file in the `public` folder:

```
...
<h4 class="bg-primary text-white text-center p-2 m-1">
        Static HTML Element
</h4>
...
```

As illustrated in Figure 3-2, this element has several parts: the start tag, the end tag, the attributes, and the content.

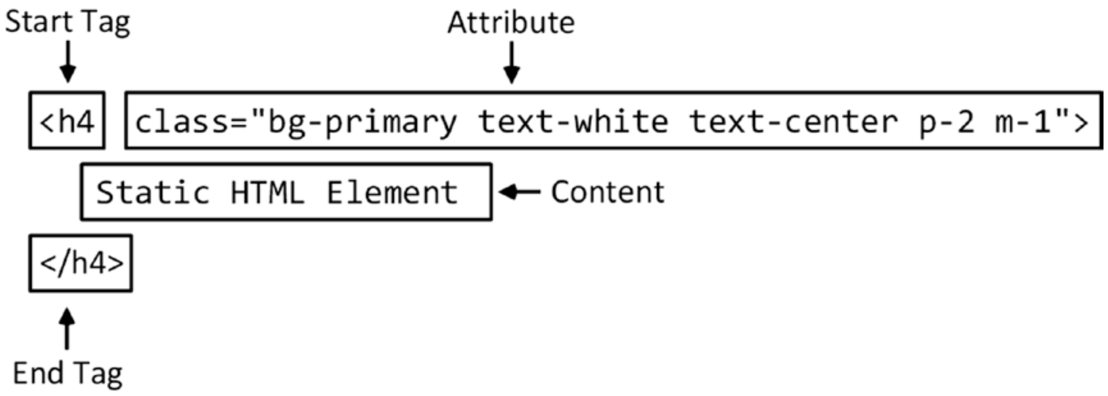


Figure 3-2. *The anatomy of an HTML element*

The *name* of this element (also referred to as the *tag name* or just the *tag*) is `h4`, and it tells the browser that the content between the tags should be treated as a header. There are a range of header elements, ranging from `h1` to `h6`, where `h1` is conventionally used for the most important content, `h2` for slightly less important content, and so on.

When you define an HTML element, you start by placing the tag name in angle brackets (the `<` and `>` characters) and end an element by using the tag in a similar way, except that you also add a `/` character after the left-angle bracket (`<`), to create the *start tag* and *end tag*.

The tag indicates the purpose of the element, and there is a wide range of element types defined by the HTML specification. In Table 3-1, I have described the elements that I used most commonly in this book. For a complete list of tag types, you should consult the HTML specification.

Table 3-1. *Common HTML Elements Used in the Examples*

Element	Description
`a`	A link (more formally known as an anchor), which the user clicks to navigate to a new URL or a new location within the current document
`button`	A button, which can be clicked by the user to initiate an action
`div`	A generic element; often used to add structure to a document for presentation purposes
`h1 to h6`	A header
`input`	A field used to gather a single data item from the user
`table`	A table, used to organize content into rows and columns
`tbody`	The body of the table (as opposed to the header or footer)
`td`	A content cell in a table row
`th`	A header cell in a table row
`thead`	The header of a table
`tr`	A row in a table

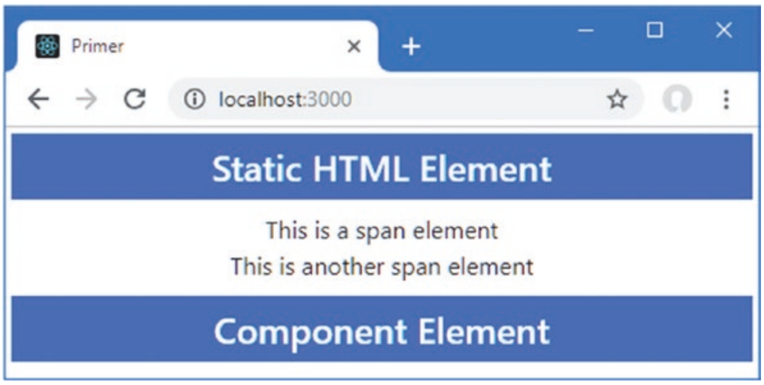
Understanding Element Content

Whatever appears between the start and end tags is the element's content. An element can contain text (such as `Static HTML Element` in this case) or other HTML elements. In Listing 3-7, I have added a new HTML element that contains another element.

Listing 3-7. Adding a New Element in the index.html File in the public Folder

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Primer</title>
  </head>
  <body>
    <h4 class="bg-primary text-white text-center p-2 m-1">
        Static HTML Element
    </h4>
    <div class="text-center m-2">
      <div>This is a span element</div>
      <div>This is another span element</div>
    </div>
    <div id="domParent"></div>
    <div id="root"></div>
  </body>
</html>
```

The outer element is known as the *parent*, while the elements it contains are known as *children*. The additions in Listing 3-7 define a parent `div` element that has two children, also `div` elements. The content of each child `div` element is a text message, producing the result shown in Figure 3-3. Being able to create a hierarchy of elements is an essential HTML feature. It is one of the key building blocks for React applications, and it allows complex content to be created.



Static HTML Element

This is a span element
This is another span element

Component Element

Figure 3-3. *Adding parent and child elements*

Understanding Element Content Restrictions

Some elements have restrictions on the types of elements that can be their children. The `div` elements in the example can contain any other element and are used to add structure to an HTML document, often so that content can be easily styled. Other elements have more specific roles that require specific types of elements to be used as children. For example, a `tbody` element, which you will see in later chapters and which represents the body of a table, can contain only one or more `tr` elements, each of which represents a table row.

■ **Tip** Don't worry about learning all of the HTML elements and their relationships. You will pick up everything you need to know as you follow the examples in later chapters, and most code editors will display a warning if you try to create invalid HTML.

Understanding Void Elements

Some elements are not allowed to contain anything at all. These are called *void* or *self-closing* elements, and they are written without a separate end tag, like this:

```
...
<input />
...
```

A void element is defined in a single tag, and you add a / character before the last angle bracket (the > character). The element shown here is the most common example of a void element, and it is used to gather data from the user in HTML forms. You will see many examples of void elements in later chapters.

Understanding Attributes

You can provide additional information to the browser by adding *attributes* to your elements. Here is the attribute that was applied to the h4 element illustrated in Figure 3-2:

```
...
<h4 class="bg-primary text-white text-center p-2 m-1">
        Static HTML Element
</h4>
...
```

Attributes are always defined as part of the start tag, and most attributes have a name and a value, separated by an equal sign, as illustrated in Figure 3-4.

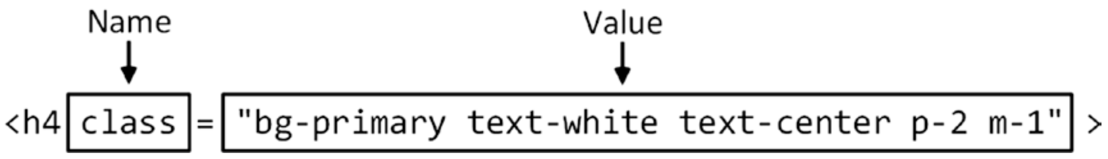


Figure 3-4. *The name and value of an attribute*

The name of this attribute is class, which is used to group related elements, typically so that their appearance can be managed consistently. This is why the class attribute has been used in this example, and the attribute value associates the h4 element with a number of classes that relate to styles provided by the Bootstrap CSS package, which I describe later in the chapter.

Creating HTML Elements Dynamically

The HTML elements defined in the index.html file are static. These elements are received and displayed by the browser just as they are defined, which you can see by right-clicking in the browser window and selecting Inspect or Inspect Element from the pop-up menu. The F12 developer tools will open and display the contents of the HTML document, which will include this element:

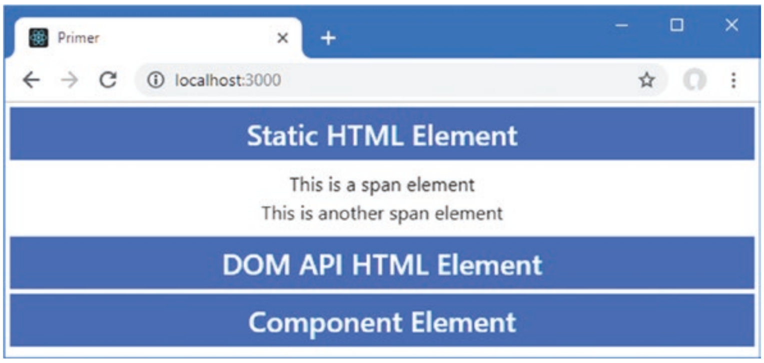
```
...
<h4 class="bg-primary text-white text-center p-2 m-1">
    Static HTML Element
</h4>
...
```

HTML elements can also be dynamically created using JavaScript and the Domain Object Model (DOM) API that all modern browsers support. In Listing 3-8, I have added some JavaScript to the index.html file that uses the DOM API to add a new element to the HTML document.

Listing 3-8. Creating an Element Dynamically in the index.html File in the public Folder

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Primer</title>
  </head>
  <body>
    <h4 class="bg-primary text-white text-center p-2 m-1">
        Static HTML Element
    </h4>
    <div class="text-center m-2">
      <div>This is a span element</div>
      <div>This is another span element</div>
    </div>
    <div id="domParent"></div>
    <div id="root"></div>
    <script>
      let element =  document.createElement("h4")
      element.className = "bg-primary text-white text-center p-2 m-1";
      element.textContent = "DOM API HTML Element";
      document.getElementById("domParent").appendChild(element);
    </script>
  </body>
</html>
```

The `script` element denotes a section of JavaScript code, which the browser will execute when it processes the contents of the `index.html` file and which creates a new HTML element, as shown in Figure 3-5.



Static HTML Element

This is a span element
This is another span element

DOM API HTML Element

Component Element

Figure 3-5. *Creating an element using the DOM API*

The first JavaScript statement in Listing 3-8 creates a new h4 element.

```
...
let element =  document.createElement("h4")
...
```

The `document` object represents the HTML document that the browser is displaying, and the `createElement` method returns an object that represents a new HTML element. The object that the DOM API provides to represent the new HTML element has properties that correspond to the attributes that are used when defining static HTML. The second JavaScript statement in Listing 3-8 uses the property that corresponds to the `class` attribute.

```
...
element.className = "bg-primary text-white text-center p-2 m-1";
...
```

Most of the properties defined by element objects have the same name as the attributes they correspond to. There are some exceptions, including `className`, which is used because the `class` keyword is reserved in many programming languages, including JavaScript.

The remaining JavaScript statements set the text content of the HTML element and add it to the HTML document so it is displayed by the browser. If you examine the new element by right-clicking in the browser window and selecting Inspect from the pop-up menu, you will see that the object created by the JavaScript statements in Listing 3-8 has been represented just like the static element from the `index.html` file.

```
...
<h4 class="bg-primary text-white text-center p-2 m-1">DOM API HTML Element</h4>
...
```

It is worth emphasizing that the `index.html` file does not contain this HTML element. Instead, it contains a series of JavaScript statements that instructed the browser to create the element and add it to the content presented to the user.

Creating Elements Dynamically Using a React Component

If you examine the contents of the `App.js` file, you will see that the `render` method of the `App` component combines aspects of the static and dynamic HTML elements from earlier sections:

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

export default class App extends Component {
    render = () =>
        <h4 className="bg-primary text-white text-center p-2 m-1">
            Component Element
        </h4>
}
...
```

React uses the DOM API to create the HTML elements specified by the `render` method, which it does by creating an object that is configured through its properties. The JSX format used for React development allows HTML elements to be defined declaratively, but the result is still JavaScript when the file is processed by the development tools, which is why the `h4` element is configured using `className` and not `class` in the `App` render method. JSX lets elements appear to be configured using attributes, but they are just the means by which values are specified for properties, and this is why the term *prop* is used so much in React development.

■ **Note** No special steps are required to use JSX, which is supported by the tools added to the project by the `create-react-app` package. I explain how elements defined using JSX are transformed into JavaScript in Chapter 9.

Using Expressions in React Elements

The ability to use expressions to configure elements is one of the key features of React and JSX. Expressions are denoted by curly braces (the { and } characters), and the result is inserted into the content generated by a component. In Listing 3-9, I have used an expression to set the content of the `h4` element rendered by the `App` component.

Listing 3-9. Using an Expression in the App.js File in the src Folder

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

const message = "This is a constant"

export default class App extends Component {

    render = () =>
        <h4 className="bg-primary text-white text-center p-2 m-1">
            { message }
        </h4>
}
```

I have defined a constant named `message` and used an expression to use the `message` value as the content for the `h4` element. To simplify the example, I commented out the static HTML element and the DOM API code from the `index.html` file, as shown in Listing 3-10.

Listing 3-10. Removing Elements in the index.html File in the public Folder

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Primer</title>
  </head>
  <body>
    <!-- <h4 class="bg-primary text-white text-center p-2 m-1">
        Static HTML Element
    </h4>
    <div class="text-center m-2">
      <div>This is a span element</div>
      <div>This is another span element</div>
    </div>
    <div id="domParent"></div> -->
    <div id="root"></div>
    <!-- <script>
      let element =  document.createElement("h4")
      element.className = "bg-primary text-white text-center p-2 m-1";
      element.textContent = "DOM API HTML Element";
      document.getElementById("domParent").appendChild(element);
    </script> -->
  </body>
</html>
```

Save the changes, and you will see the value of the constant defined in Listing 3-9 displayed in the h4 element produced by the App component, as shown in Figure 3-6.

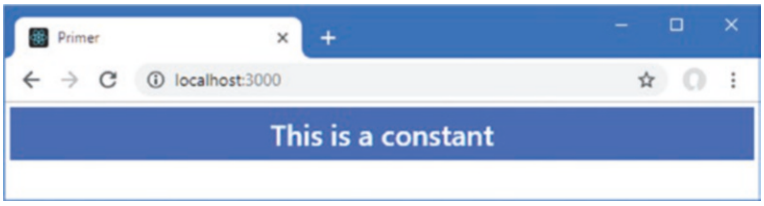


Figure 3-6. *Using an expression to set the content of an element*

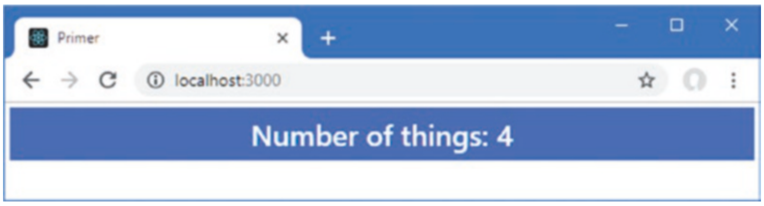
Mixing Expressions and Static Content

Expressions can be combined with static values to create more complex results, as shown in Listing 3-11, which uses an expression to set part of the content for the h4 element.

Listing 3-11. Mixing an Expression with Static Content in the App.js File in the src Folder

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

const count = 4

export default class App extends Component {

    render = () =>
        <h4 className="bg-primary text-white text-center p-2 m-1">
            Number of things: { count }
        </h4>
}
```

The expression includes the count value in the content of the h4 element, which is combined with the static content, producing the result shown in Figure 3-7.



Number of things: 4

Figure 3-7. *Mixing an expression and static content*

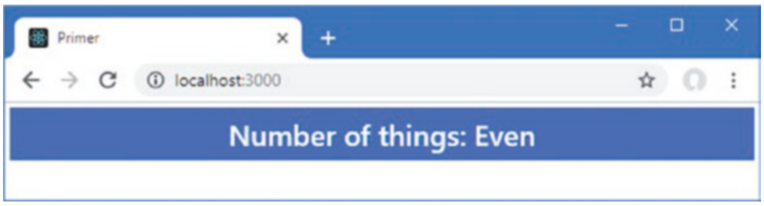
Performing Computation in Expressions

Expressions can do more than inject values into the content rendered by a component and can be used for any computation, as shown in Listing 3-12.

Listing 3-12. Performing a Computation in the App.js File in the src Folder

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

const count = 4

export default class App extends Component {

    render = () =>
        <h4 className="bg-primary text-white text-center p-2 m-1">
            Number of things: { count % 2 === 0 ? "Even" : "Odd" }
        </h4>
}
```

This example uses the ternary operator to determine whether the count value is odd or even and produces the result shown in Figure 3-8.



Number of things: Even

Figure 3-8. *Performing computation in an expression*

Expressions are well-suited to simple operations, but trying to include too much code in an expression results in a confusing component. For more complex operations, a function should be defined and invoked by the expression so that the function result is incorporated into the content produced by the component, as shown in Listing 3-13.

Listing 3-13. Defining a Function in the App.js File in the src Folder

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

const count = 4

function isEven() {
    return count % 2 === 0 ? "Even" : "Odd";
}

export default class App extends Component {

    render = () =>
        <h4 className="bg-primary text-white text-center p-2 m-1">
            Number of things: { isEven() }
        </h4>
}
```

When you use a function in an expression, you must invoke it with parentheses (the (and) characters), as shown in the listing, so that the result of the function is included in the content generated by the component.

Accessing Component Properties and Methods

The `this` keyword is required to specify properties and method defined by the component, as shown in Listing 3-14. As I explain in Part 2, there are different ways to create components, but the technique I use throughout this book is the one shown in the listing, which provides the widest range of features and is suitable for most projects.

Listing 3-14. Using the this Keyword in an Expression 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 = {
            count: 4
        }
    }

    isEven() {
        return this.state.count % 2 === 0 ? "Even" : "Odd";
    }

    render = () =>
        <h4 className="bg-primary text-white text-center p-2 m-1">
            Number of things: { this.isEven() }
        </h4>
}
```

The component in this listing defines a constructor, which is how the initial state of the component is configured, as I explain in Chapter 4. The constructor assigns an object to the `state` property, with a `count` value of `4`. The component also defines a method called `isEven`, which accesses the `count` value as `this.state.count`. The `this` keyword refers to the component instance, as explained in Chapter 4; `state` refers to the state property created in the constructor; and `count` selects the value to use in the computation. This `this` keyword is also used invoke the `isEven` method in the expression. The result is the same as the previous listing. Some methods require arguments, which can be specified as part of the expression, as shown in Listing 3-15.

Listing 3-15. Passing an Argument to a Method 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 = {
            count: 4
        }
    }

    isEven(val) {
        return val % 2 === 0 ? "Even" : "Odd";
    }

    render = () =>
        <h4 className="bg-primary text-white text-center p-2 m-1">
            Number of things: { this.isEven(this.state.count) }
        </h4>
}
```

The expression in this example invokes the `isEven` method, using the `count` value as the argument. The result is the same as the previous listing.

Using Expressions to Set Prop Values

Expressions can also be used to set the value of props, which allows HTML elements and child components to be configured. In Listing 3-16, I have added a method to the `App` component whose result is used to set the `className` prop of the `h4` element.

Listing 3-16. Setting a Prop Value 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 = {
            count: 4
        }
    }
    isEven(val) {
        return val % 2 === 0 ? "Even" : "Odd";
    }

    getClassName(val) {
        return val % 2 === 0
            ? "bg-primary text-white text-center p-2 m-1"
            : "bg-secondary text-white text-center p-2 m-1"
    }

    render = () =>
        <h4 className={this.getClassName(this.state.count)}>
            Number of things: { this.isEven(this.state.count) }
        </h4>
}
```

The result is the same as the previous listing.

Using Expressions to Handle Events

Expressions are used to tell React how to respond to events when they are triggered by an element. In Listing 3-17, I have added a button to the content returned by the App component and used the onClick prop to tell React how to respond when the click event is triggered.

Listing 3-17. Handling an Event 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 = {
            count: 4
        }
    }

    isEven(val) {
        return val % 2 === 0 ? "Even" : "Odd";
    }

    getClassName(val) {
        return val % 2 === 0
            ? "bg-primary text-white text-center p-2 m-1"
            : "bg-secondary text-white text-center p-2 m-1"
    }

    handleClick = () => this.setState({ count: this.state.count + 1});

    render = () =>
        <h4 className={this.getClassName(this.state.count)}>
            <button className="btn btn-info m-2" onClick={ this.handleClick }>
```

```
                    Click Me
                </button>
                Number of things: { this.isEven(this.state.count) }
            </h4>
}
```

The button element is configured using the onClick prop, which tells React to invoke the handleClick method in response to the click event. Note that the method isn't specified using parentheses. Also, note that the handleClick method is defined using the fat arrow syntax; handling events is one of the few times where the way that a method is defined is important, as I explain in Chapter 12. Clicking the button updates the value of the count property, which changes the outcome of the other expressions in the render method, producing the effect shown in Figure 3-9.

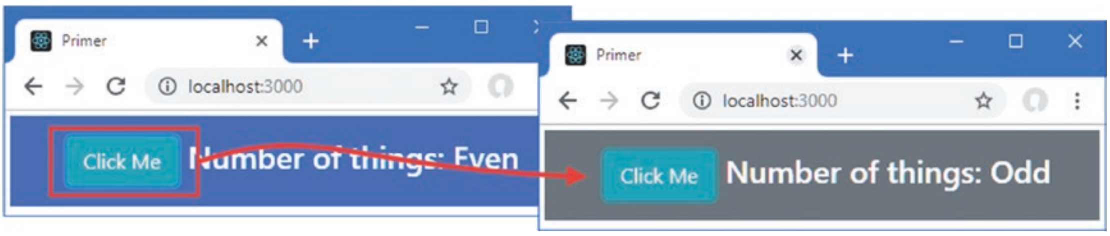


Figure 3-9. *Handling an event*

Understanding Bootstrap

HTML elements tell the browser what kind of content they represent, but they don't provide any information about how that content should be displayed. The information about how to display elements is provided using *Cascading Style Sheets* (CSS). CSS consists of a comprehensive set of *properties* that can be used to configure every aspect of an element's appearance and a set of *selectors* that allow those properties to be applied.

One of the main problems with CSS is that some browsers interpret properties slightly differently, which can lead to variations in the way that HTML content is displayed on different devices. It can be difficult to track down and correct these problems, and CSS frameworks have emerged to help web app developers style their HTML content in a simple and consistent way.

The most popular CSS framework is Bootstrap, which was originally developed at Twitter but has become a widely used open source project. Bootstrap consists of a set of CSS classes that can be applied to elements to style them consistently and some optional JavaScript code that performs additional enhancement (but that I do not use in this book). I use Bootstrap in my own projects; it works well across browsers, and it is simple to use. I use the Bootstrap CSS styles in this book because they let me style my examples without having to define and then list my own custom CSS in each chapter. Bootstrap provides a lot more features than the ones I use in this book; see http://getbootstrap.com for full details.

I don't want to get into too much detail about Bootstrap because it isn't the topic of this book, but I do want to give you enough information so you can tell which parts of an example are React features and which are related to Bootstrap.

Applying Basic Bootstrap Classes

Bootstrap styles are applied via the className prop, which is the counterpart to the class attribute, and is used to group related elements. The className prop isn't just used to apply CSS styles, but it is the most common use, and it underpins the way that Bootstrap and similar frameworks operate. Here is an HTML element with a `classNae` prop, taken from Listing 3-9:

```
...
<h4 className="bg-primary text-white text-center p-2 m-1">
    { message }
</h4>
...
```

The `className` prop assigns the `h4` element to five classes, whose names are separated by spaces: `bg-primary`, `text-white`, `text-center`, `p-2`, and `m-1`. These classes correspond to collections of styles defined by Bootstrap, as described in Table 3-2.

Table 3-2. *The h4 Element Classes*

Name	Description
`bg-primary`	This class applies a style context to provide a visual cue about the purpose of the element. See the “Using Contextual Classes” section.
`text-white`	This class applies a style that sets the text color for the element’s content to white.
`text-center`	This class applies a style that horizontally centers the element’s content.
`p-2`	This class applies a style that adds spacing around the element’s content, as described in the “Using Margin and Padding” section.
`m-1`	This class applies a style that adds spacing around the element, as described in the “Using Margin and Padding” section.

Using Contextual Classes

One of the main advantages of using a CSS framework like Bootstrap is to simplify the process of creating a consistent theme throughout an application. Bootstrap defines a set of *style contexts* that are used to style related elements consistently. These contexts, which are described in Table 3-3, are used in the names of the classes that apply Bootstrap styles to elements.

Table 3-3. *The Bootstrap Style Contexts*

Name	Description
`primary`	Indicates the main action or area of content
`secondary`	Indicates the supporting areas of content
`success`	Indicates a successful outcome
`info`	Presents additional information
`warning`	Presents warnings
`danger`	Presents serious warnings
`muted`	De-emphasizes content
`dark`	Increases contrast by using a dark color
`white`	Increases contrast by using white

Bootstrap provides classes that allow the style contexts to be applied to different types of elements. The h4 element with which I started this section has been added to the bg-primary class, which sets the background color of an element to indicate that it is related to the main purpose of the application. Other classes are specific to a certain set of elements, such as btn-primary, which is used to configure button and a elements so they appear as buttons whose colors are consistent with other elements in the primary context. Some of these context classes must be applied in conjunction with other classes that configure the basic style of an element, such as the btn class, which is combined with the btn-primary class.

Using Margin and Padding

Bootstrap includes a set of utility classes that are used to add *padding*, which is space between an element's edge and its content, and *margin*, which is space between an element's edge and the surrounding elements. The benefit of using these classes is that they apply a consistent amount of spacing throughout the application.

The names of these classes follow a well-defined pattern. Here is the h4 element from Listing 3-9 again:

```
...
<h4 className="bg-primary text-white text-center p-2 m-1">
    { message }
...
```

The classes that apply margin and padding to elements follow a well-defined naming schema: first, the letter m (for margin) or p (for padding), followed by an optional letter selecting specific edges (t for top, b for bottom, l for left, or r for right), then a hyphen, and, finally, a number indicating how much space should be applied (0 for no spacing, or 1, 2, 3, 4 or 5 for increasing amounts). If there is no letter to specify edges, then the margin or padding will be applied to all edges. To help put this schema in context, the p-2 class to which the h4 element has been added applies padding level 2 to all of the element's edges.

Using Bootstrap to Create Grids

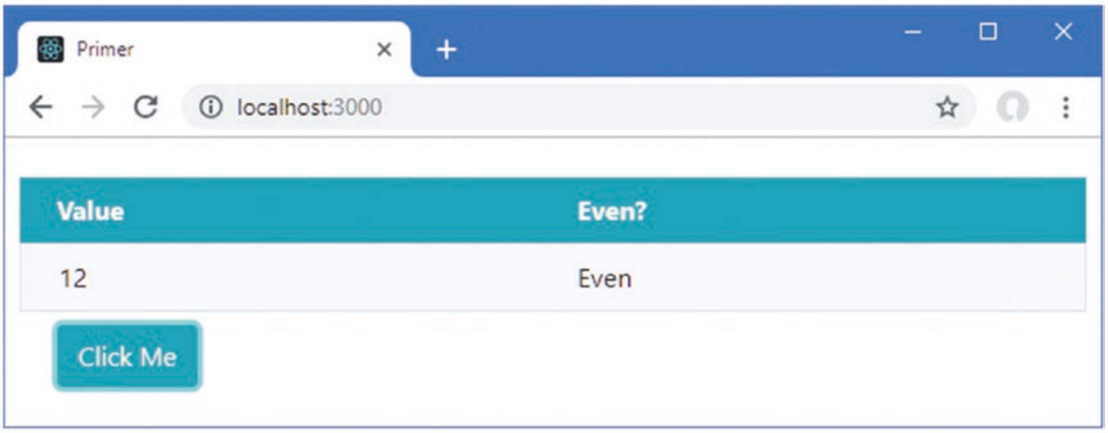
Bootstrap provides style classes that can be used to create different kinds of grid layout, ranging from one to twelve columns. I use the grid layout for many of the examples in this book, and I have created a simple grid layout in Listing 3-18.

Listing 3-18. Creating a Grid 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 = {
            count: 4
        }
    }

    isEven(val) {
        return val % 2 === 0 ? "Even" : "Odd";
    }
    getClassName(val) {
        return val % 2 === 0
            ? "bg-primary text-white text-center p-2 m-1"
            : "bg-secondary text-white text-center p-2 m-1"
    }

    handleClick = () => this.setState({ count: this.state.count + 1});

    render = () =>
        <div className="container-fluid p-4">
            <div className="row bg-info text-white p-2">
                <div className="col font-weight-bold">Value</div>
                <div className="col-6 font-weight-bold">Even?</div>
            </div>
            <div className="row bg-light p-2 border">
                <div className="col">{ this.state.count }</div>
                <div className="col-6">{ this.isEven( this.state.count) }</div>
            </div>
            <div className="row">
                <div className="col">
                    <button className="btn btn-info m-2"
                            onClick={ this.handleClick }>
                        Click Me
                    </button>
                </div>
            </div>
        </div>
}
```

The Bootstrap grid layout system is simple to use. A top-level `div` element is assigned to the `container` class (or the `container-fluid` class if you want it to span the available space). You specify a column by applying the `row` class to a `div` element, which has the effect of setting up the grid layout for the content that the `div` element contains.

Each row defines 12 columns, and you specify how many columns each child element will occupy by assigning a class whose name is `col-` followed by the number of columns. For example, the class `col-1` specifies that an element occupies one column, `col-2` specifies two columns, and so on, right through to `col-12`, which specifies that an element fills the entire row. If you omit the number of columns and just assign an element to the `col` class, then Bootstrap will allocate an equal amount of the remaining columns. The grid in Listing 3-18 produces the layout shown in Figure 3-10.



Value	Even?
12	Even

Figure 3-10. *Using a grid layout*

Using Bootstrap to Style Tables

Bootstrap includes support for styling `table` elements and their contents, which is a feature I use in some of the examples in later chapters. Table 3-4 lists the key Bootstrap classes for working with tables.

Table 3-4. *The Bootstrap CSS Classes for Tables*

Name	Description
`table`	Applies general styling to a `table` element and its rows
`table-striped`	Applies alternate-row striping to the rows in the `table` body
`table-bordered`	Applies borders to all rows and columns
`table-sm`	Reduces the spacing in the table to create a more compact layout

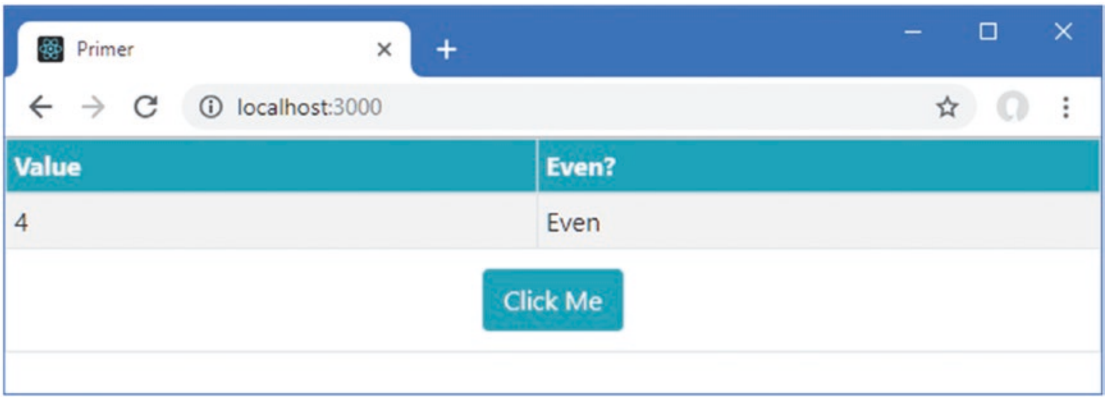
All these classes are applied directly to the `table` element, as shown in Listing 3-19, where I have replaced the grid layout with a table.

Listing 3-19. Using a Table Layout 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 = {
            count: 4
        }
    }
    isEven(val) {
        return val % 2 === 0 ? "Even" : "Odd";
    }

    getClassName(val) {
        return val % 2 === 0
            ? "bg-primary text-white text-center p-2 m-1"
            : "bg-secondary text-white text-center p-2 m-1"
    }

    handleClick = () => this.setState({ count: this.state.count + 1});

    render = () =>
        <table className="table table-striped table-bordered table-sm">
            <thead  className="bg-info text-white">
                <tr><th>Value</th><th>Even?</th></tr>
            </thead>
            <tbody>
                <tr>
                    <td>{ this.state.count }</td>
                    <td>{ this.isEven(this.state.count) } </td>
                </tr>
            </tbody>
            <tfoot className="text-center">
                <tr>
                    <td colSpan="2">
                        <button className="btn btn-info m-2"
                                onClick={ this.handleClick }>
                            Click Me
                        </button>
                    </td>
                </tr>
            </tfoot>
        </table>
}
```

■ **Tip** Notice that I have used the `thead` element when defining the tables in Listing 3-19. Browsers will automatically add any `tr` elements that are direct descendants of `table` elements to a `tbody` element if one has not been used. You will get odd results if you rely on this behavior when working with Bootstrap, and it is always a good idea to use the full set of elements when defining a table.

Figure 3-11 shows the result of using a table instead of a grid.



Value	Even?
4	Even

Figure 3-11. *Styling a table*

Using Bootstrap to Style Forms

Bootstrap includes styling for form elements, allowing them to be styled consistently with other elements in the application. In Listing 3-20, I have added form elements to the content produced by the App component.

Listing 3-20. Adding Form Elements in the App.js File in the src Folder

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

export default class App extends Component {

    render = () =>
        <div className="m-2">
            <div className="form-group">
                <label>Name:</label>
                <input className="form-control" />
            </div>
            <div className="form-group">
                <label>City:</label>
                <input className="form-control" />
            </div>
        </div>
}
```

The basic styling for forms is achieved by applying the form-group class to a div element that contains a label and an input element, where the input element is assigned to the form-control class. Bootstrap styles the elements so that the label is shown above the input element and the input element occupies 100 percent of the available horizontal space, as shown in Figure 3-12.

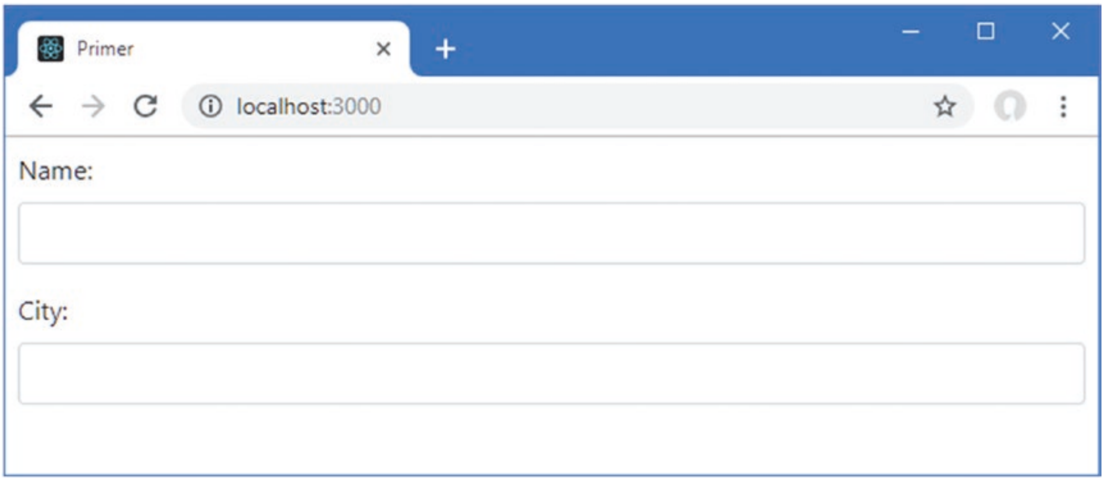


Figure 3-12. *Styling form elements*

Summary

In this chapter, I provided a brief overview of HTML and explained how it can be mixed with JavaScript code in React development, albeit with some changes and restrictions. I also introduced the Bootstrap CSS framework, which I use throughout this book but which is not directly related to React. You need to have a good grasp of HTML and CSS to be truly effective in web application development, but the best way to learn is by firsthand experience, and the descriptions and examples in this chapter will be enough to get you started and provide just enough background information for the examples ahead. In the next chapter, I continue the primer theme and introduce the most important JavaScript features used in this book.