



# Components and Props

In this chapter, I describe the key building block in React applications: the component. I focus on the simplest type of component in this chapter, which is the *stateless component*. I describe the more complex alternative, stateful components, in Chapter 11. I also explain how the props feature works in this chapter, which allows one component to provide another with the data it requires to render its content and the functions it should invoke when something important happens. Table 10-1 puts stateless components and props in context.

**Table 10-1.** *Putting Stateless Components and Props in Context*

Question	Answer
What are they?	Components are the key building blocks in React applications. Stateless components are JavaScript functions that render content that React can present to the user. Props are the means by which one component provides data to another so that it can adapt the content it renders.
Why are they useful?	Components are useful because they provide access to the React support for creating features by combining JavaScript, HTML, and other components. Props are useful because they allow components to adapt the content they produce.
How are they used?	Stateless components are defined as JavaScript functions that return a React element, which is usually defined using HTML in the JSX format. Props are defined as properties on elements.
Are there any pitfalls or limitations?	React requires components to behave in specific ways, such as returning a single React element and always returning a result, and it can take time to become used to these restrictions. The most common pitfall with props is specifying literal values when a JavaScript expression was required.
Are there any alternatives?	Components are the key building block in React applications, and there is no way to avoid their use. There are alternative to props that can be useful in larger and more complex projects, as described in Chapter 14 and in Part 3.

Table 10-2 summarizes the chapter.

**Table 10-2.** Chapter Summary

Problem	Solution	Listing
Add content to a React application	Define a function that returns HTML elements or invokes the <code>React.createElement</code> method	1–9
Add additional features to a React application	Define components and compose them in a parent-child relationship using elements that correspond to the component name	10–14
Configure a child component	Define props when applying the component	15–19
Render HTML elements for each object in a data array	Use the <code>map</code> method to create elements, ensuring that they have a key prop	20–24
Render multiple elements from a component	Use the <code>React.Fragment</code> element or use elements without tags	25–28
Render no content	Return <code>null</code>	29
Receive notifications from a child component	Configure the component with a function prop	31–34
Pass on props to a child	Use the prop values received from the parent or use the destructuring operator	35–39
Define default prop values	Use the <code>defaultProps</code> property	40, 41
Check prop types	Use the <code>propTypes</code> property	42–44

# Preparing for This Chapter

To create the example project for this chapter, open a new command prompt, navigate to a convenient location, and run the command shown in Listing 10-1.

■ **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 10-1.** Creating the Example Project

```
npx create-react-app components
```

Run the commands shown in Listing 10-2 to navigate to the project folder and add the Bootstrap package to the project.

**Listing 10-2.** Adding the Bootstrap CSS Framework

---

```
cd components
npm install bootstrap@4.1.2
```

---

To include the Bootstrap CSS stylesheet in the application, add the statement shown in Listing 10-3 to the `index.js` file, which can be found in the `src` folder.

**Listing 10-3.** Including Bootstrap in the `index.js` File in the `src` Folder

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

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

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: http://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Using the command prompt, run the command shown in Listing 10-4 in the `components` folder to start the development tools.

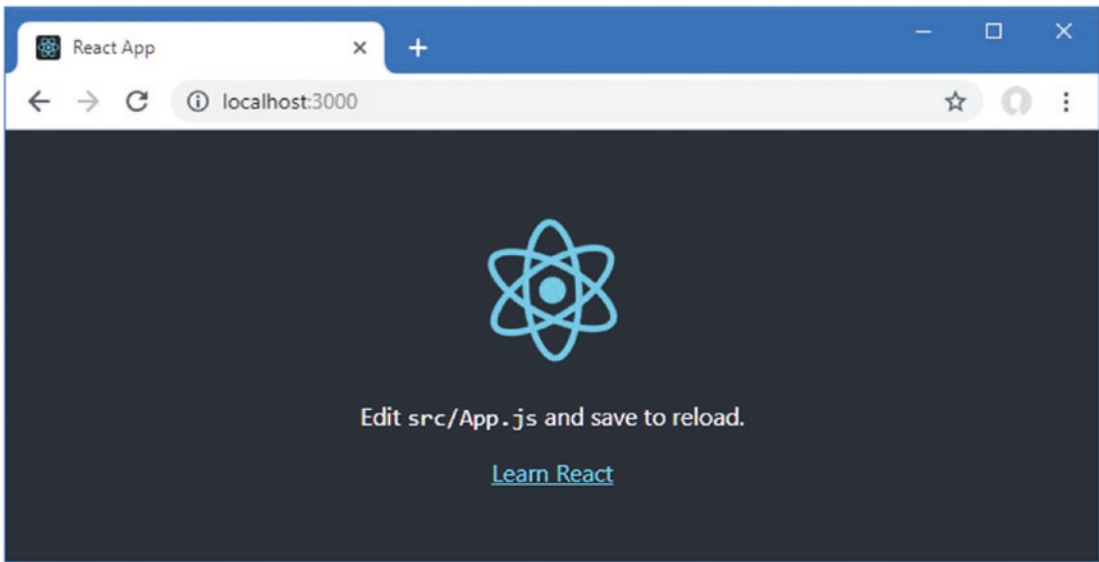
**Listing 10-4.** Starting the Development Tools

---

```
npm start
```

---

Once the initial preparation for the project is complete, a new browser window will open and display the URL `http://localhost:3000` and display the placeholder content shown in Figure 10-1.



**Figure 10-1.** Running the example application

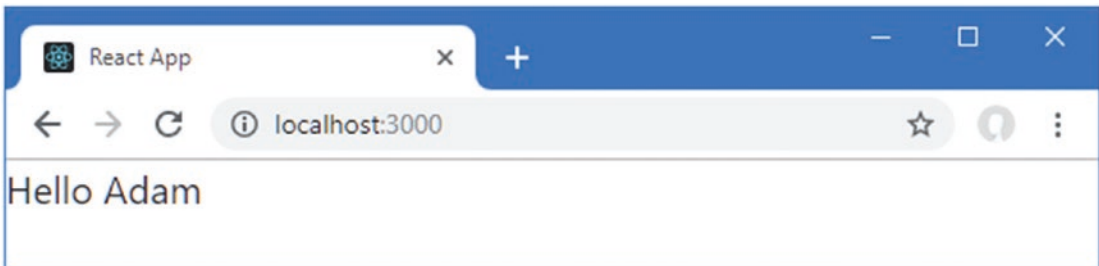
## Understanding Components

The best place to start with components is by defining one and seeing how it works. In Listing 10-5, I replaced the contents of the `App.js` file with a simple component.

**Listing 10-5.** Defining a Component in the `App.js` File in the `src` Folder

```
export default function App() {  
  return "Hello Adam";  
}
```

This is an example of a *stateless component*, and it is just about as simple as a component can be: a function that returns content that React will display to the user, which is known as *rendering*. When the application starts, the code in the `index.js` file is executed, including the statement that renders the `App` component. React invokes the function and displays the result to the user, as shown in Figure 10-2.



**Figure 10-2.** Defining and applying a component

As simple as the result might be, it reveals the key purpose of components, which is to provide React with content to display to the user.

## Rendering HTML Content

When a component renders a string value, it is included as text content in the parent element. Components become more useful when they return HTML content, which is most easily done by taking advantage of JSX and the way it allows HTML to be mixed with JavaScript code. In Listing 10-6, I changed the result of the component so that it renders a fragment of HTML.

---

■ **Tip** You must declare a dependency on React from the `react` module when you use JSX, as shown in the listing. You will receive a warning if you forget.

---

*Listing 10-6.* Rendering HTML in the App.js File in the src Folder

```
import React from "react";

export default function App() {
  return <h1 className="bg-primary text-white text-center p-2">
    Hello Adam
  </h1>
}
```

You remember to use the `return` keyword inside the component's function to render the result. This can feel awkward, but remember that the HTML fragment in a JSX file is converted to a call to the `createElement` method, which produces an object that React can display to the user.

The use of the `return` keyword makes sense when you consider what the code looks like once the HTML fragment has been replaced with the `createElement` method during the build process.

```
...
import React from "react";

export default function App() {
  return React.createElement("h1",
    { className: "bg-primary text-white text-center p-2" },
    "Hello Adam");
}
...
```

The component function returns the result from the `React.createElement` method, which is an element that React can use to add content to the Domain Object Model (DOM).

If you want to start the HTML on a separate line from the `return` keyword, then you can use parentheses to enclose the result, as shown in Listing 10-7.

**Listing 10-7.** Using Parentheses in the App.js File in the src Folder

```
import React from "react";

export default function App() {
  return (
    <h1 className="bg-primary text-white text-center p-2">
      Hello Adam
    </h1>
  )
}
```

This allows the HTML elements to be consistently indented, although the dangling ( and ) characters can strike some developers as awkward.

Functional components can also be defined using the fat arrow syntax, which omits the return keyword, as shown in Listing 10-8.

**Listing 10-8.** Using a Fat Arrow Function in the App.js File in the src Folder

```
import React from "react";

export default () =>
  <h1 className="bg-primary text-white text-center p-2">
    Hello Adam
  </h1>
```

The fat arrow function is exported without a name, which works in the example application because the statement in the index.js file that imports the component from the App.js file uses the default export, like this:

```
...
import App from './App';
...
```

Exporting a fat arrow function by name and as the default requires an additional statement, as shown in Listing 10-9.

**Listing 10-9.** Creating a Named and Default Export in the App.js File in the src Folder

```
import React from "react";

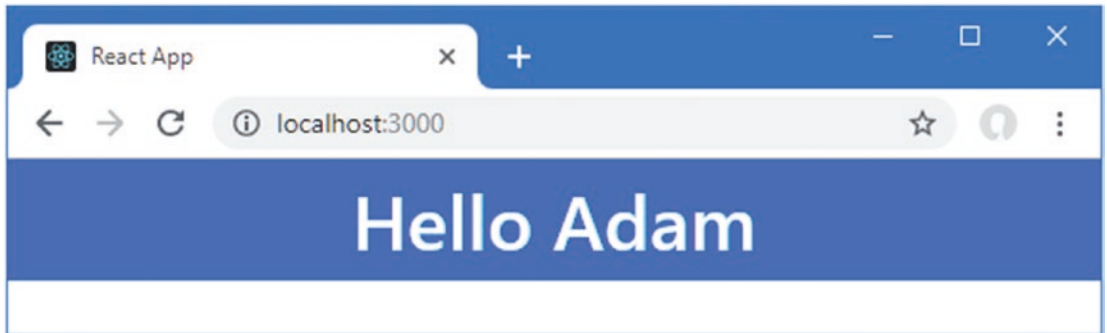
export const App = () =>
  <h1 className="bg-primary text-white text-center p-2">
    Hello Adam
  </h1>

export default App;
```

The fat arrow function is assigned to a const that is exported by name, and a separate statement uses the name to create the default export, which allows the component to be imported by name and as the default.

■ **Note** I have included this example because module exports cause confusion, but in real projects they use either named or default exports throughout and don't have to accommodate both styles of working. I prefer using named exports, and that is the approach I have taken in the examples in this book.

I use regular functions in this chapter and use parentheses where they help make the HTML content more readable, but all of the examples in this section produce the same result, as shown in Figure 10-3.



**Figure 10-3.** Returning HTML content

## Rendering Other Components

One of the most important React features is that the content rendered by a component can contain other components, allowing features to be combined to create complex applications. I added a file called `Message.js` to the `src` folder and used it to define the component shown in Listing 10-10.

**Listing 10-10.** The Contents of the `Message.js` File in the `src` Folder

```
import React from "react";

export function Message() {
  return <h4 className="bg-success text-white text-center p-2">
    This is a message
  </h4>
}
```

The `Message` component renders an `h4` element that contains a message. In Listing 10-11, I have updated the `App` component so that it renders the `Message` content as part of its content.

**Listing 10-11.** Rendering Another Component in the `App.js` File in the `src` Folder

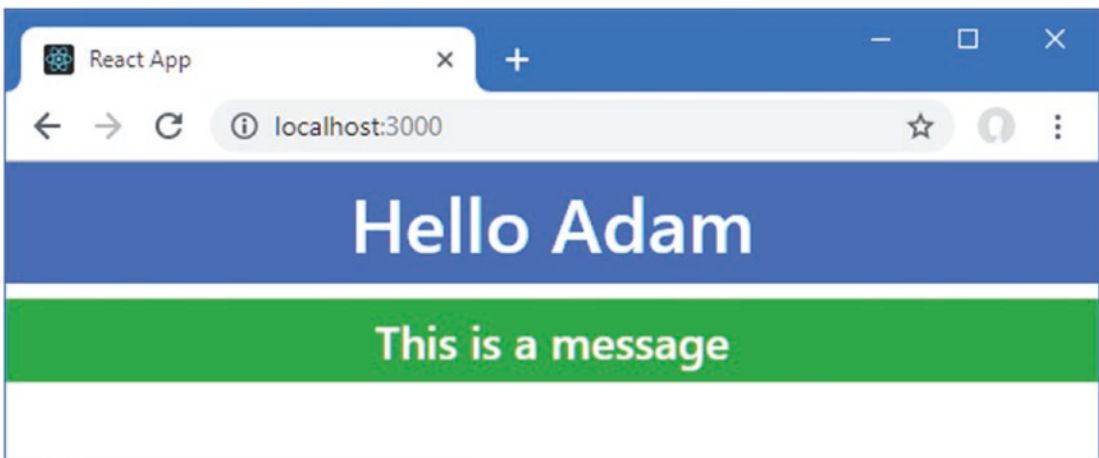
```
import React from "react";
import { Message } from "../Message";
```

```

export default function App() {
  return (
    <div>
      <h1 className="bg-primary text-white text-center p-2">
        Hello Adam
      </h1>
      <Message />
    </div>
  )
}

```

The `import` statement declares a dependency on the `Message` component, which is rendered using a `Message` element. When React receives the content rendered by the `App` component, it will contain the `Message` element, which it will deal with by invoking the `Message` component's function and replacing the `Message` element with the content it renders, producing the result shown in Figure 10-4.



**Figure 10-4.** *Rendering other content*

When one component uses another like this, a parent-child relationship is formed. In this example, the `App` component is the parent to the `Message` component, and the `Message` component is the child of the `App` component. A component can apply the same component more than once by defining multiple elements for the child component, as shown in Listing 10-12.

**Listing 10-12.** Applying a Child Component in the `App.js` File in the `src` Folder

```

import React from "react";
import { Message } from "../Message";

export default function App() {
  return (
    <div>
      <h1 className="bg-primary text-white text-center p-2">
        Hello Adam
      </h1>

```

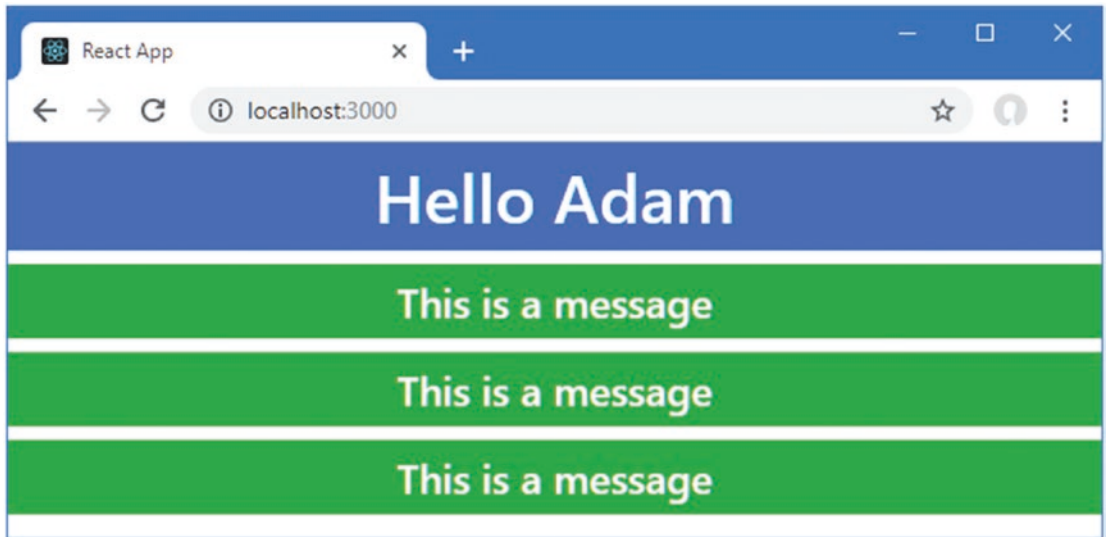


```

    <Message />
    <Message />
    <Message />
  </div>
)
}

```

Each time that React encounters the `Message` element, it invokes the `Message` component and uses the content it renders to replace the `Message` element, as shown in Figure 10-5.



**Figure 10-5.** Applying multiple children

A component can have children of different types, which means that one component can take advantage of the features that multiple components offer. I created another simple component by adding a file called `Summary.js` to the `src` folder with the code shown in Listing 10-13.

**Listing 10-13.** The Contents of the `Summary.js` File in the `src` Folder

```

import React from "react";

export function Summary() {
  return <h4 className="bg-info text-white text-center p-2">
    This is a summary
  </h4>
}

```

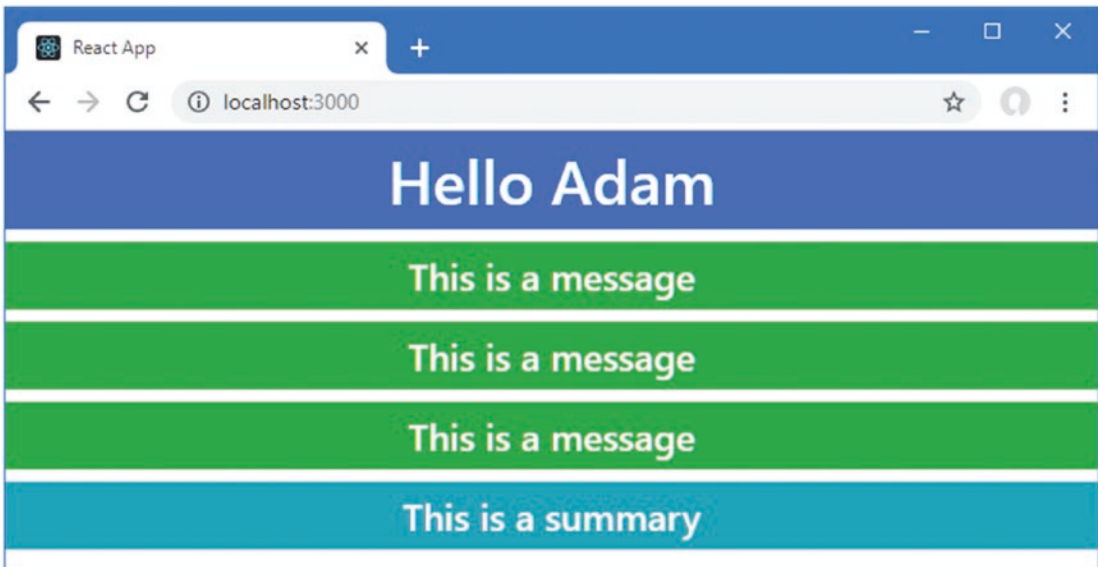
In Listing 10-14, I have updated the `App` component to declare a dependency on the `Summary` component and render its contents using a `Summary` element.

**Listing 10-14.** Adding a Child Component in the App.js File in the src Folder

```
import React from "react";
import { Message } from "../Message";
import { Summary } from "../Summary";

export default function App() {
  return (
    <div>
      <h1 className="bg-primary text-white text-center p-2">
        Hello Adam
      </h1>
      <Message />
      <Message />
      <Message />
      <Summary />
    </div>
  )
}
```

When React processes the content rendered by the App component, it encounters the elements for the child components, invokes their function, and replaces the Message and Summary elements with the content they render. The result is shown in Figure 10-6.

**Figure 10-6.** Using different child components

## Understanding Props

Being able to render content from multiple children isn't that useful when each component renders identical content. Fortunately, React supports *props*—short for *properties*—which allows a parent component to provide data to its children, which they can use to render their content. In the sections that follow, I explain how props work and demonstrate the different ways they can be used.

## Defining Props in the Parent Component

Props are defined by adding properties to the custom HTML elements that apply components. The name of the property is the name of the prop, and the value can be a static value or an expression. In Listing 10-15, I have added props to the `Message` elements used by the `App` component.

**Listing 10-15.** Defining Props in the `App.js` File in the `src` Folder

```
import React from "react";
import { Message } from "../Message";
import { Summary } from "../Summary";

export default function App() {
  return (
    <div>
      <h1 className="bg-primary text-white text-center p-2">
        Hello Adam
      </h1>
      <Message greeting="Hello" name="Bob" />
      <Message greeting="Hola" name={ "Alice" + "Smith" } />
      <Message greeting="Hi there" name="Dora" />
      <Summary />
    </div>
  )
}
```

I have provided two props, `greeting` and `name`, for each `Message` component. Most of the prop values are static values, which are expressed as literal strings. The value for the `greeting` prop on the second `Message` element is an expression, which concatenates two string values. (You will see a linter warning about the expression in Listing 10-15 because concatenating string literal values is on the list of poor practices that the linter is configured to detect. The linter warning can be ignored for this purposes of this chapter.)

### DEFINING PROPS

Props can be used to pass static values or the results of dynamic expressions to child components. Static values are quoted literally, like this:

```
...
<Message greeting="Hello" name="Bob" />
...
```

This prop provides the child component with the value Bob for its name prop. If you want to use the result of a JavaScript expression as the value for the prop, then use a data binding expression, like this:

```
...
<Message greeting="Hola" name={ "Alice" + "Smith" } />
...
```

React will evaluate the expression and use the result, which is the concatenation of two strings in this example, as the value for the prop. A common mistake is to put the JavaScript expression in quotes, like this:

```
...
<Message greeting="Hola" name="{ "Alice" + "Smith" }" />
...
```

React will interpret this as a request to use the static value { "Alice" + "Smith" } as the value for the prop. When using expressions for props, you must remember not to use quotes. If you prefer not to use JSX and want to create React elements using pure JavaScript, then props are provided as the second argument to the `createElement` method, like this:

```
...
React.createElement(Message, { greeting: "Hola", name: "Alice" + "Smith"})
...
```

If you don't get the results you expect, in JSX or pure JavaScript, the React Devtools browser extension (described in Chapter 9) can display the props that are received by each component in the application, which makes it easy to see where things have gone wrong.

---

## Receiving Props in the Child Component

Props are received in components by defining a parameter called `props` (although that is just a convention, and you can give the parameter any legal JavaScript name). The `props` object has a property for each of the props, which is assigned the prop value. As an example, these props from Listing 10-15:

```
...
<Message greeting="Hello" name="Bob" />
...
```

will be translated into an object like this:

```
...
{
  greeting: "Hello",
  name: "Bob"
}
...
```

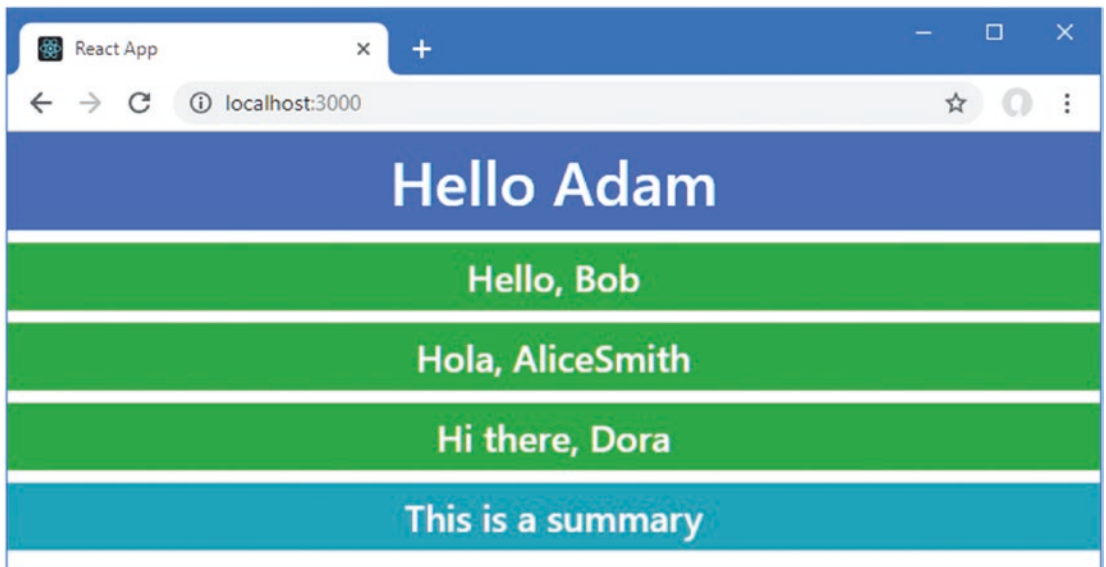
In Listing 10-16, I have changed the `Message` component so that it defines a prop parameter and uses the values provided by the parent component in the result it produces.

**Listing 10-16.** Using Props in the `Message.js` File in the `src` Folder

```
import React from "react";

export function Message(props) {
  return <h4 className="bg-success text-white text-center p-2">
    {props.greeting}, {props.name}
  </h4>
}
```

The child component doesn't need to worry about whether a prop value was specified statically or with an expression and uses the props like any other JavaScript object. In the listing, I used the `greeting` and `name` props in an expression to set the contents of the `h4` element rendered by the component, producing the result shown in Figure 10-7.



**Figure 10-7.** Rendering content using props

## Combining JavaScript and Props to Render Content

The prop values provided to each `Message` element defined by the `App` component in Listing 10-16 results in different content, allowing the same functionality to be employed by the parent component in different ways.

## Selectively Rendering Content

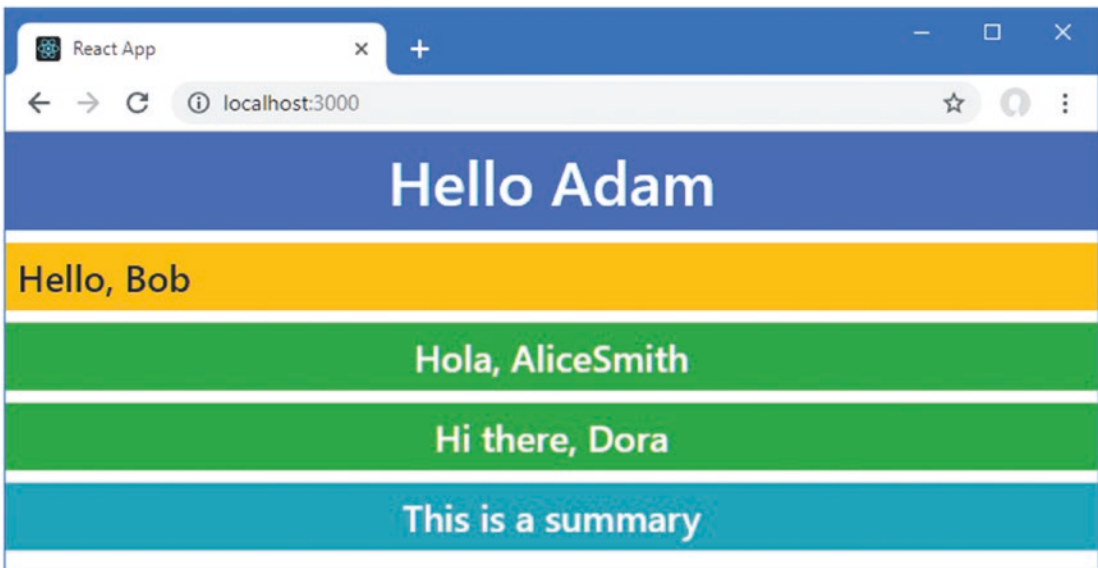
Components can use the JavaScript `if` keyword to inspect a prop and render different content based on its value. In Listing 10-17, I used the `if` statement to alter the content rendered by the `Message` component.

**Listing 10-17.** Selectively Rendering in the `Message.js` File in the `src` Folder

```
import React from "react";

export function Message(props) {
  if (props.name === "Bob") {
    return <h4 className="bg-warning p-2">{props.greeting}, {props.name}</h4>
  } else {
    return <h4 className="bg-success text-white text-center p-2">
      {props.greeting}, {props.name}
    </h4>
  }
}
```

If the value of the `name` prop is `Bob`, the component will render an `h4` element with different class memberships, as shown in Figure 10-8.



**Figure 10-8.** Using an `if` statement to select content

This type of selective rendering, where only the value of a prop changes, can be expressed with less duplication by separating the value of the property from the rest of the HTML, as shown in Listing 10-18.

**Listing 10-18.** Selecting a Property Value in the Message.js File in the src Folder

```
import React from "react";

export function Message(props) {

  let classes = props.name === "Bob" ? "bg-warning p-2"
    : "bg-success text-white text-center p-2";

  return <h4 className={ classes }>
    {props.greeting}, {props.name}
  </h4>
}
```

I have used the JavaScript ternary conditional operator to select the classes that the h4 element will be assigned to and applied those classes with an expression for the `className` property. The result is the same as Listing 10-17 but without duplicating the unchanging parts of the HTML element.

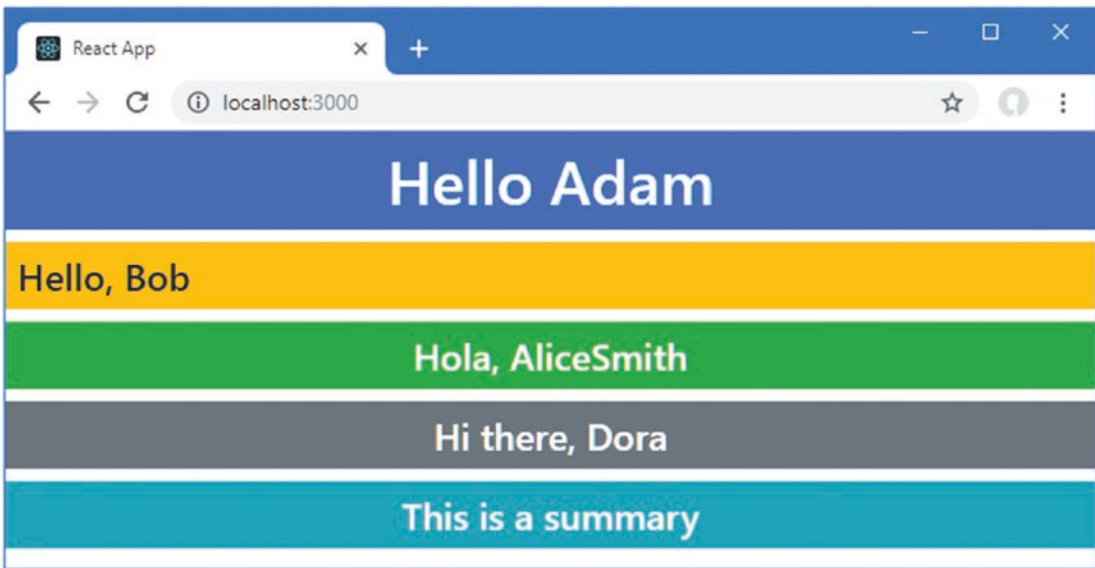
A switch statement can be used when a component needs to select content from a more complex list, as shown in Listing 10-19.

**Listing 10-19.** Using a switch Statement in the Message.js File in the src Folder

```
import React from "react";

export function Message(props) {
  let classes;
  switch (props.name) {
    case "Bob":
      classes = "bg-warning p-2";
      break;
    case "Dora":
      classes = "bg-secondary text-white text-center p-2";
      break;
    default:
      classes = "bg-success text-white text-center p-2"
  }
  return <h4 className={ classes }>
    {props.greeting}, {props.name}
  </h4>
}
```

This example uses the switch statement on the `props.name` value to select the classes for the h4 element, producing the result shown in Figure 10-9.



**Figure 10-9.** Using a switch statement to select content

## Rendering Arrays

Components often have to create HTML elements for each element in an array, often to display items in a list or as rows in a table. The technique required for dealing with arrays causes confusion and is worth approaching carefully. To prepare, I updated the App component so that it configures the Summary component with a prop, as shown in Listing 10-20. (I also removed elements to keep the example simple.)

**Listing 10-20.** Adding a Prop in the App.js File in the src Folder

```
import React from "react";
//import { Message } from "../Message";
import { Summary } from "../Summary";

export default function App() {
  return (
    <div>
      <h1 className="bg-primary text-white text-center p-2">
        Hello Adam
      </h1>
      <Summary names={ ["Bob", "Alice", "Dora"]} />
    </div>
  )
}
```



The `names` prop provides the `Summary` component with an array of string values. In Listing 10-21, I have changed the content rendered by the `Summary` component so that it produces elements for each of the values in the array.

**Listing 10-21.** Rendering an Array in the `Summary.js` File in the `src` Folder

```
import React from "react";

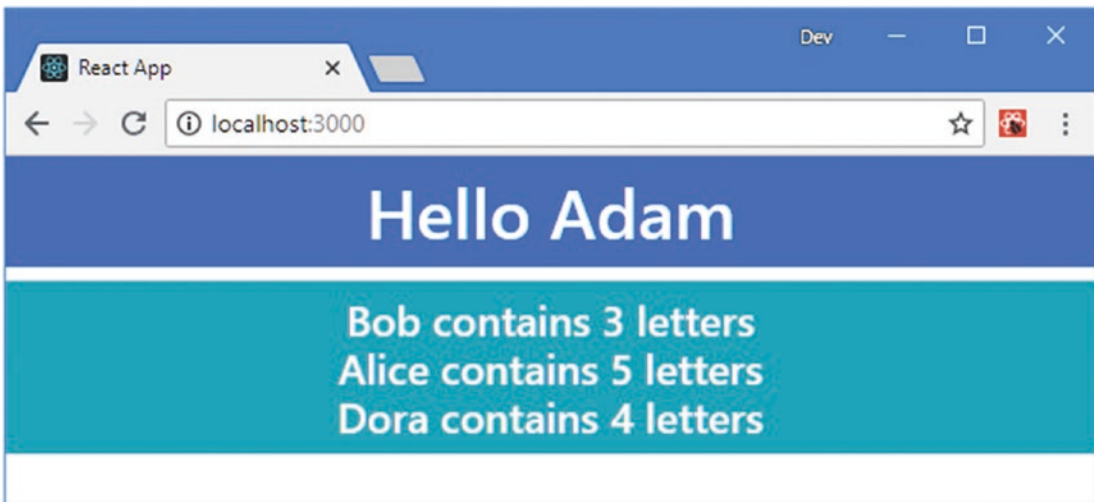
function createInnerElements(names) {
  let arrayElems = [];
  for (let i = 0; i < names.length; i++) {
    arrayElems.push(
      <div>
        `${names[i]} contains ${names[i].length} letters`
      </div>
    )
  }
  return arrayElems;
}

export function Summary(props) {
  return <h4 className="bg-info text-white text-center p-2">
    { createInnerElements(props.names) }
  </h4>
}
```

The component function uses an expression to set the content of the `h4` element, which it does by invoking the `createInnerElements` function. The `createInnerElements` function uses a JavaScript `for` loop to enumerate the contents of the `names` array and adds a `div` element to a result array.

```
...
arrayElems.push(<div>`${names[i]} contains ${names[i].length} letters`</div>)
...
```

The content of each `div` element is set with another expression, which uses a template string to create a message specific to each element in the array. The array of `div` elements is returned as the result of the `createInnerElements` function and used as the content for the `h4` element, producing the result shown in Figure 10-10.



**Figure 10-10.** Creating React elements for the objects in an array

## Using the Map Method to Process Array Objects

Although the `for` loop is the way that most programmers are used to enumerating arrays, it isn't the most elegant way to deal with arrays in React. The `map` method, described in Chapter 4, can be used to transform objects in an array into HTML elements, as shown in Listing 10-22.

**Listing 10-22.** Transforming an Array in the `Summary.js` File in the `src` Folder

```
import React from "react";

function createInnerElements(names) {
  return names.map(name =>
    <div>
      `${name} contains ${name.length} letters`
    </div>
  )
}

export function Summary(props) {
  return <h4 className="bg-info text-white text-center p-2">
    { createInnerElements(props.names)}
  </h4>
}
```

The argument to the `map` method is a function that is invoked for each object in the array. Each time the function passed to the `map` method is invoked, the next item in the array is passed to the function, which I use to create the element that represents that object. The result from each call to the function is added to an array that is used as the `map` result. The code in Listing 10-22 produces the same result as Listing 10-21.

---

■ **Tip** You don't have to use fat arrow functions with the `map` method, but it produces a more concise component.

---

Now that the `createElement` function contains a single line of code, I can further simplify the component by moving the statement that creates the inner elements into the component function, as shown in Listing 10-23.

**Listing 10-23.** Simplifying the Code in the `Summary.js` File in the `src` Folder

```
import React from "react";

export function Summary(props) {
  return (
    <h4 className="bg-info text-white text-center p-2">
      { props.names.map(name =>
        <div>
          `${name} contains ${name.length} letters`
        </div>
      ) }
    </h4>
  )
}
```

This change doesn't alter the output and produces the same result as Listing 10-21 and Listing 10-22.

## RECEIVING OTHER ARGUMENTS WHEN USING THE MAP METHOD

In Listing 10-23, the function I passed to the `map` method receives the current array object as its argument. The `map` method also provides two additional arguments: the zero-based index of the current object in the array and the complete array of objects. You can see an example of the array index in the “Rendering Multiple Elements” section later in this chapter.

---

## Adding the Key Prop

One final change is required to complete the example. React requires a `key` prop to be added to elements that are generated for the objects in an array so that changes can be handled efficiently, as I explain in Chapter 13. The value of the `key` prop should be an expression whose value identifies the object uniquely within the array, as shown in Listing 10-24.

**Listing 10-24.** Adding the Key Prop in the Summary.js File in the src Folder

```
import React from "react";

export function Summary(props) {
  return (
    <h4 className="bg-info text-white text-center p-2">
      { props.names.map(name =>
        <div key={ name }>
          {`${name} contains ${name.length} letters`}
        </div>
      )}
    </h4>
  )
}
```

I used the value of the name variable, to which each object in the array is assigned when the function passed to the map method is invoked and which allows React to differentiate between the elements created from the array objects.

React will display elements that do not have a key prop, as the earlier examples in this section demonstrate, but a warning will be displayed in the browser's JavaScript console.

## Rendering Multiple Elements

React requires components to return a single top-level element, although that element is able to contain as many other elements as the application requires. The Summary component, for example, returns a top-level h4 element that contains a series of div elements that are generated for the elements in the names prop.

There are times when the requirement for a single top-level element causes a problem. The HTML specification applies restrictions on how elements can be combined, which can conflict with the single element React requirement. To demonstrate the problem, I have changed the content rendered by the App component so that it contains a table, where the contents for each tr element are produced by a child component, as shown in Listing 10-25.

**Listing 10-25.** Rendering a Table in the App.js File in the src Folder

```
import React from "react";
import { Summary } from "./Summary";

let names = ["Bob", "Alice", "Dora"]

export default function App() {
  return (
    <table className="table table-sm table-striped">
      <thead>
        <tr><th>#</th><th>Name</th><th>Letters</th></tr>
      </thead>
      <tbody>
```

```

        { names.map((name, index) =>
          <tr key={ name }>
            <Summary index={index} name={name} />
          </tr>
        )}
      </tbody>
    </table>
  )
}

```

The Summary component is passed index and name props. In Listing 10-26, I have updated the Summary component so that it generates a series of table cells using the prop values.

**Listing 10-26.** Rendering Table Cells in the Summary.js File in the src Folder

```

import React from "react";

export function Summary(props) {
  return <td>{ props.index + 1} </td>
    <td>{ props.name } </td>
    <td>{ props.name.length } </td>
}

```

The Summary component renders a set of td elements because that's what the HTML specification requires as the children of td elements. But when you save the changes, you will see the following error:

```

...
Syntax error: src/Summary.js: Adjacent JSX elements must be wrapped
in an enclosing tag (5:12)

3 | export function Summary(props) {
4 |   return <td>{ props.index + 1} </td>
> 5 |     <td>{ props.name } </td>
    |     ^
6 |     <td>{ props.name.length } </td>
7 | }
...

```

This error message indicates that the content rendered by the component doesn't meet the React requirement of a single top-level element. There isn't an HTML element that can be used to wrap the td elements and still be a legal addition to the table. For these situations, React provides a special element, as shown in Listing 10-27.

**Listing 10-27.** Wrapping Elements in the Summary.js File in the src Folder

```

import React from "react";

export function Summary(props) {
  return <React.Fragment>
    <td>{ props.index + 1} </td>


```

```

      <td>{ props.name } </td>
      <td>{ props.name.length } </td>
    </React.Fragment>
  }

```

When React processes the elements rendered by the Summary component, it discards the `React.Fragment` element and uses the remaining content to replace the Summary element that applied the component, as shown in Figure 10-11.



#	Name	Letters
1	Bob	3
2	Alice	5
3	Dora	4

**Figure 10-11.** Rendering multiple elements

React supports an alternative syntax for these situations, which is to use an enclosing element without a tag name, as shown in Listing 10-28.

**Listing 10-28.** Wrapping Elements in the Summary.js File in the src Folder

```

import React from "react";

export function Summary(props) {
  return <>
    <td>{ props.index + 1 } </td>
    <td>{ props.name } </td>
    <td>{ props.name.length } </td>
  </>
}

```

This is equivalent to Listing 10-27 and produces the same result. I use the `React.Fragment` for the examples in this book or wrap multiple elements in a `div` where that produces a legal combination of HTML elements.

## Rendering No Content

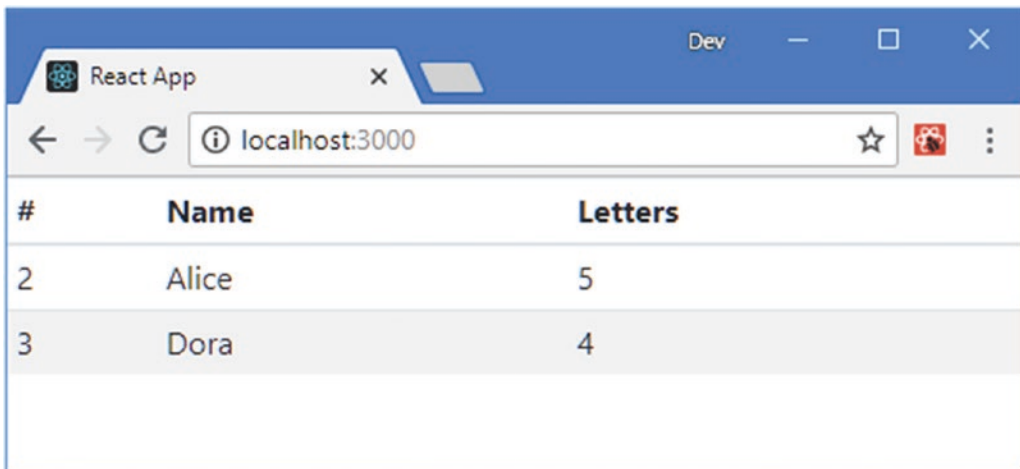
A component must always return a result, even when it doesn't produce any content for React to display. In these situations, the component's function should return `null`, and in Listing 10-29, I have modified the `Summary` component so that it doesn't produce any content when the length of its `name` prop is less than four characters.

**Listing 10-29.** Rendering No Content in the `Summary.js` File in the `src` Folder

```
import React from "react";

export function Summary(props) {
  if (props.name.length >= 4) {
    return <React.Fragment>
      <td>{ props.index + 1 } </td>
      <td>{ props.name } </td>
      <td>{ props.name.length } </td>
    </React.Fragment>
  } else {
    return null;
  }
}
```

The parent component still applies the `Summary` element three times, each of which results in the `Summary` component's function being invoked, but only two of those invocations produce a result, as shown in Figure 10-12.



#	Name	Letters
2	Alice	5
3	Dora	4

**Figure 10-12.** Rendering no content

## Attempting to Change Props

Props are read-only and must not be changed by a component. When React creates the `props` object, it configures its properties so that an error is displayed if any changes are made. In Listing 10-30, I have added a statement to the `Summary` component that changes the value of the `name` prop.

**Listing 10-30.** Changing a Prop Value in the `Summary.js` File in the `src` Folder

```
import React from "react";

export function Summary(props) {
  props.name = `Name: ${props.name}`;
  if (props.name.length >= 4) {
    return <React.Fragment>
      <td>{ props.index + 1} </td>
      <td>{ props.name } </td>
      <td>{ props.name.length } </td>
    </React.Fragment>
  } else {
    return null;
  }
}
```

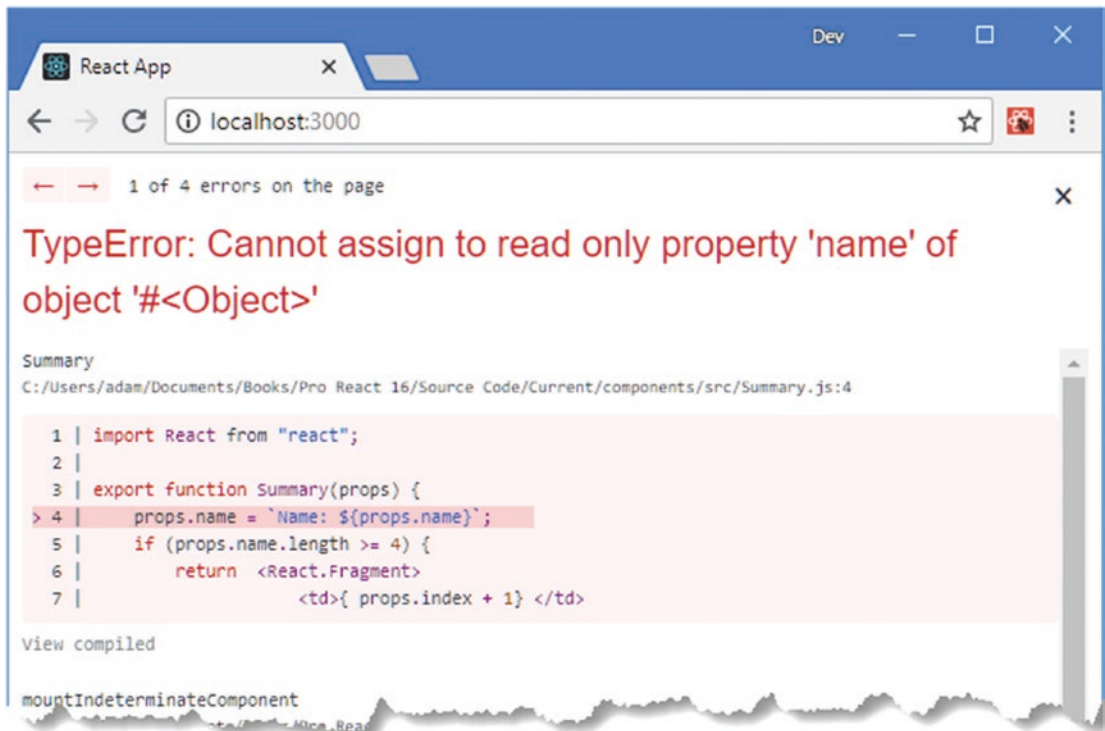
When you save the changes and the browser reloads, you will see the error message shown in Figure 10-13. This is a runtime error, which means that no warning is displayed by the compiler at the command prompt.

---

■ **Tip** This error isn't displayed when the application has been built for deployment using the process described in Chapter 8, which means you should test thoroughly during development to ensure your components don't inadvertently try to change a prop.

---





**Figure 10-13.** Attempting to modify a prop

## Using Function Props

All of the props I have used so far in this chapter have been *data props*, which provide a child component with a read-only data value. React also supports function props, where the parent component provides a child with a function that it can invoke to notify the parent that something important has happened. The parent component can respond by changing the value of the data props, which will trigger an update and allow the child to present updated content to the user.

To show how this works, I have defined a function in the file that contains the App component that changes the order of the values that are used for the name props for the Summary elements, as shown in Listing 10-31.

**Listing 10-31.** Defining a Change Function in the App.js File in the src Folder

```
import React from "react";
import { Summary } from "../Summary";
import ReactDOM from "react-dom";

let names = ["Bob", "Alice", "Dora"]

function reverseNames() {
  names.reverse();
  ReactDOM.render(<App />, document.getElementById('root'));
}
```

```

export default function App() {
  return (
    <table className="table table-sm table-striped">
      <thead>
        <tr><th>#</th><th>Name</th><th>Letters</th></tr>
      </thead>
      <tbody>
        { names.map((name, index) =>
          <tr key={ name }>
            <Summary index={index} name={name}
              reverseCallback={reverseNames} />
          </tr>
        )}
      </tbody>
    </table>
  )
}

```

The function I defined is called `reverseNames`, and it uses the JavaScript `reverse` method to reverse the order of the values in the `names` array. The `reverseNames` function is provided to the `Summary` component as the value for a prop named `reverseCallback`, like this:

```

...
<Summary index={index} name={name} reverseCallback={reverseNames} />
...

```

The `Summary` component will receive a prop object with three properties: the `index` prop provides the index of the current object being processed by the `map` method, the `name` prop provides the current value from the array, and the `reverseCallback` prop provides the function that will reverse the order of the array's contents. In Listing 10-32, I have updated the `Summary` component to make use of the function it receives as a prop. (I have also removed the statement that attempts to change the prop value and removed the `if` statement that prevents the component from rendering content for short names.)

**Listing 10-32.** Using a Function Prop in the `Summary.js` File in the `src` Folder

```

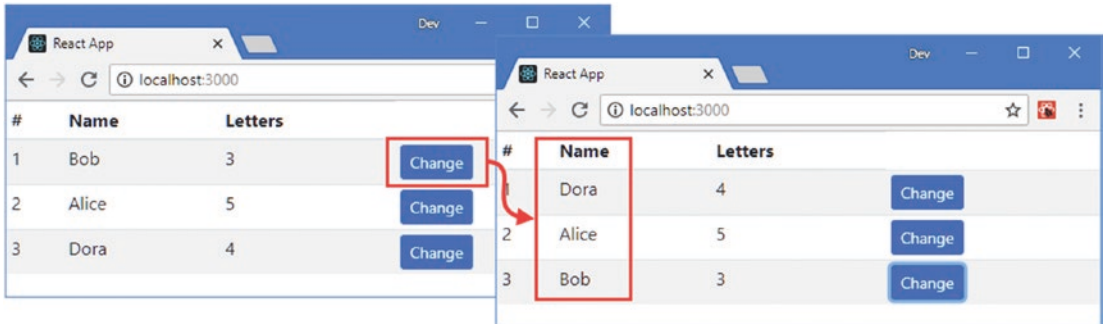
import React from "react";

export function Summary(props) {
  return (
    <React.Fragment>
      <td>{ props.index + 1} </td>
      <td>{ props.name } </td>
      <td>{ props.name.length } </td>
      <td>
        <button className="btn btn-primary btn-sm"
          onClick={ props.reverseCallback }>
          Change
        </button>
      </td>
    </React.Fragment>
  )
}

```

The component renders a button element whose `onClick` prop selects the function prop it receives from its parent. I describe the `onClick` prop in Chapter 12, but, as you have seen in earlier chapters, this property tells React how to respond when the user clicks an element, and, in this case, the expression tells React to invoke the `reverseCallback` prop, which is the function that has been provided by the parent component.

The result is that clicking a button element causes React to invoke the `changeValues` function defined in the `App.js` file, which reverses the order of the values used for the name props, producing the result shown in Figure 10-14.



**Figure 10-14.** Using a function received as a prop

## UNDERSTANDING THE UPDATE STATEMENT

When the `Summary` component invokes the function prop, the `reverseCallback` function is called, and this statement from Listing 10-31 is executed:

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

The `render` method is used to add a component's content to the Document Object Model (DOM) displayed by the browser and is used in the `index.js` file to start the application; it is described in Chapter 13. This is not a feature that is normally used directly, but I needed to be able to perform an update in response to the function prop being invoked. I describe the features that are normally used to perform updates in Chapter 11. For the moment, it is enough to know that calling this method updates the HTML elements displayed to the user, reflecting the change in the data values used for prop values.

## Invoking Prop Functions with Arguments

In Listing 10-32, the expression for the `onClick` property specifies the function prop, like this:

```
...
<button className="btn btn-primary btn-sm" onClick={ props.reverseCallback } >
  Change
</button>
...
```

When the function is selected by an expression, it will be passed an event object, which I describe in Chapter 12 and which provides the function that is invoked with details of the HTML element that triggered the event.

This isn't always useful when invoking a function prop, because it requires the parent to have sufficient knowledge of the child component to make sense of the event and act accordingly. Often, a more helpful approach can be to provide a custom argument to the function that gives the parent component the detail it needs directly. In Listing 10-33, I added a function to the `App.js` file that moves a specified name to the front of the array and updated the `App` component so it passes the function to its children using a prop.

**Listing 10-33.** Adding a Function in the `App.js` File in the `src` Folder

```
import React from "react";
import { Summary } from "../Summary";
import ReactDOM from "react-dom";

let names = ["Bob", "Alice", "Dora"]

function reverseNames() {
  names.reverse();
  ReactDOM.render(<App />, document.getElementById('root'));
}

function promoteName(name) {
  names = [name, ...names.filter(val => val !== name)];
  ReactDOM.render(<App />, document.getElementById('root'));
}

export default function App() {
  return (
    <table className="table table-sm table-striped">
      <thead>
        <tr><th>#</th><th>Name</th><th>Letters</th></tr>
      </thead>
      <tbody>
        { names.map((name, index) =>
          <tr key={ name }>
            <Summary index={index} name={name}
              reverseCallback={reverseNames}
              promoteCallback={promoteName} />
          </tr>
        )}
      </tbody>
    </table>
  )
}
```

The new function receives the name that should be moved to the start of the array as its parameter. In Listing 10-34, I added another button element to the content rendered by the `Summary` component and used the `onClick` property to invoke the new function prop.

**Listing 10-34.** Invoking a Function Prop in the Summary.js File in the src Folder

```
import React from "react";

export function Summary(props) {
  return (
    <React.Fragment>
      <td>{ props.index + 1} </td>
      <td>{ props.name } </td>
      <td>{ props.name.length } </td>
      <td>
        <button className="btn btn-primary btn-sm"
          onClick={ props.reverseCallback }>
          Change
        </button>
        <button className="btn btn-info btn-sm m-1"
          onClick={ () => props.promoteCallback(props.name) }>
          Promote
        </button>
      </td>
    </React.Fragment>
  )
}
```

Instead of making the App component work out which name has been selected, the function prop is invoked with an argument.

```
...
<button className="btn btn-info btn-sm m-1"
  onClick={ () => props.promoteCallback(props.name) }>
  Promote
</button>
...
```

The `onClick` expression is a fat arrow function that calls the function prop when it is invoked. It is important that you define a function like this, and if you simply specify the function prop directly in the expression, you won't get the results you expect, as described in the sidebar. Clicking one of the Promote buttons will move the corresponding name to the first position in the array so that it is displayed at the top of the table, as shown in Figure 10-15.

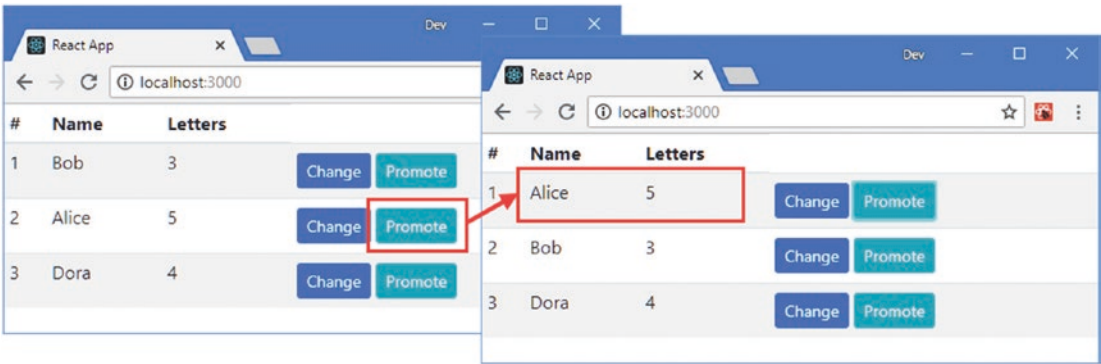


Figure 10-15. Invoking a function prop with an argument

AVOIDING THE PREMATURE INVOCATION PITFALL

When you need to invoke a function prop with an argument, you should always specify a fat arrow function that invokes the prop, like this:

```
...
<button onClick={ ( ) => props.promoteCallback(props.name) }>
  Promote
</button>
...
```

You will almost certainly forget to do this at least once and call the function prop directly in the expression, like this:

```
...
<button onClick={ props.promoteCallback(props.name) }>
  Promote
</button>
...
```

React will evaluate the expression when the component renders its content, which will invoke the prop even though the user hasn't clicked the button element. This is rarely the intended effect and can cause unexpected behaviors or produce an error, depending on what the prop does when it is invoked. In the case of the component in Listing 10-34, for example, the effect is to create a "Maximum Update Depth Exceeded" error, which occurs because the function prop asks React to re-render the components, which causes the Summary component to render content, which invokes the prop again. This continues until React halts execution and reports an error.

## Passing on Props to Child Components

React applications are created by combining components, creating a series of parent-child relationships. This arrangement often requires a component to receive a data value or callback function from its parent and pass it on to its children. To demonstrate how a prop is passed on, I added a file called `CallbackButton.js` to the `src` folder and used it to define the component shown in Listing 10-35.

**Listing 10-35.** The Contents of the `CallbackButton.js` File in the `src` Folder

```
import React from "react";

export function CallbackButton(props) {
  return (
    <button className={`btn btn-${props.theme} btn-sm m-1`}
      onClick={ props.callback }>
      { props.text }
    </button>
  )
}
```

This component renders a button element whose text content is set using a prop named `text` and that invokes a function provided through the prop named `callback` when clicked. There is also a `theme` prop that is used to select the Bootstrap CSS style for the button element.

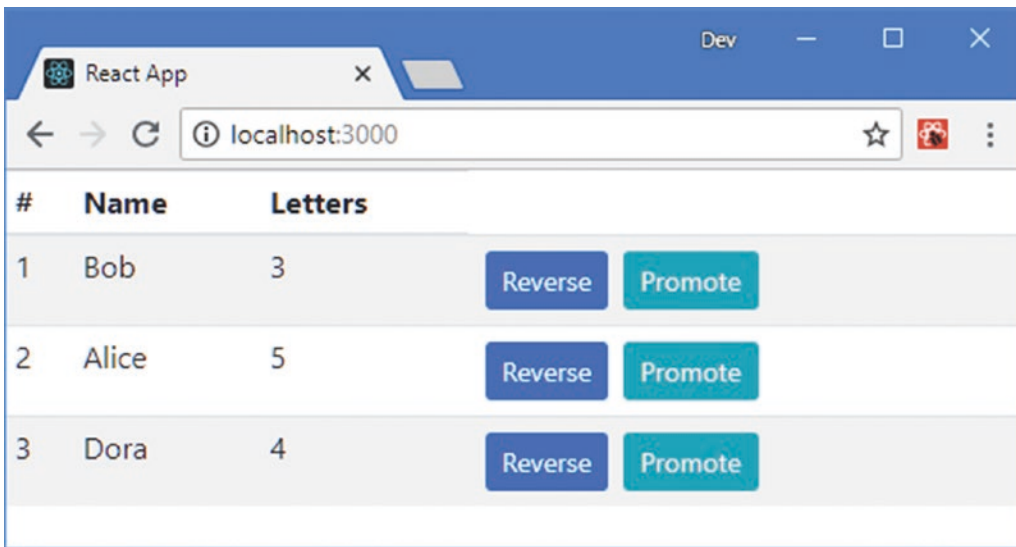
In Listing 10-36, I have updated the `Summary` component to use the `CallbackButton` component, which it configures by passing on props from its parent and adding additional props of its own.

**Listing 10-36.** Adding a Component in the `Summary.js` File in the `src` Folder

```
import React from "react";
import { CallbackButton } from "../CallbackButton";

export function Summary(props) {
  return (
    <React.Fragment>
      <td>{ props.index + 1 } </td>
      <td>{ props.name } </td>
      <td>{ props.name.length } </td>
      <td>
        <CallbackButton theme="primary"
          text="Reverse" callback={ props.reverseCallback } />
        <CallbackButton theme="info" text="Promote"
          callback={ () => props.promoteCallback(props.name) } />
      </td>
    </React.Fragment>
  )
}
```

The component that receives the props doesn't know—or care—where they originated, and they are received through the same props argument, producing the result shown in Figure 10-16.



**Figure 10-16.** *Passing on props*

## Passing On All Props to Child Components

The destructuring operator can be used if a component's parent provides props that have the same names as the props expected by the component's child. To demonstrate, I added a file called `SimpleButton.js` to the `src` folder and used it to define the component shown in Listing 10-37.

**Listing 10-37.** The Contents of the `SimpleButton.js` File in the `src` Folder

```
import React from "react";

export function SimpleButton(props) {
  return (
    <button onClick={ props.callback } className={props.className}>
      { props.text }
    </button>
  )
}
```

The `SimpleButton` component expects `callback`, `className`, and `text` props. When the `SimpleButton` component is applied by the `CallbackButton` component, there is overlap between the props provided by the parent, which means that the destructuring operator can be used to pass on props, as shown in Listing 10-38.

**Listing 10-38.** Passing on Props in the `CallbackButton.js` File in the `src` Folder

```
import React from "react";
import { SimpleButton } from "./SimpleButton";
```



```
export function CallbackButton(props) {
  return (
    <SimpleButton {...props} className={`btn btn-${props.theme} btn-sm m-1`} />
  )
}
```

The `{...props}` expression passes on all of the props received from the parent component, which are supplemented by the `className` prop. If a component wants to withhold specific props from its children, then a slightly different approach can be used, as shown in Listing 10-39.

**Listing 10-39.** Selectively Passing on Props in the `CallbackButton.js` File in the `src` Folder

```
import React from "react";
import { SimpleButton } from "../SimpleButton";

export function CallbackButton(props) {
  let { theme, ...childProps } = props;
  return (
    <SimpleButton { ...childProps }
      className={`btn btn-${props.theme} btn-sm m-1`} />
  )
}
```

The rest operator is used in a statement that creates a `childProps` object that contains all of the parent's props except `theme`. The destructuring operator is used to pass the props from the `childProps` object to the child component.

## Providing Default Prop Values

As the number of props used in an application grows, you may find yourself repeating the same set of prop values, even though the values are the same each time. An alternative approach is to define a set of defaults and override only them when you need to use a different value. In Listing 10-40, I defined a set of default prop values for the `CallbackButton` component.

**Listing 10-40.** Defining Default Values in the `CallbackButton.js` File in the `src` Folder

```
import React from "react";
import { SimpleButton } from "../SimpleButton";

export function CallbackButton(props) {
  let { theme, ...childProps } = props;
  return (
    <SimpleButton {...childProps}
      className={`btn btn-${props.theme} btn-sm m-1`} />
  )
}

CallbackButton.defaultProps = {
  text: "Default Text",
  theme: "warning"
}
```

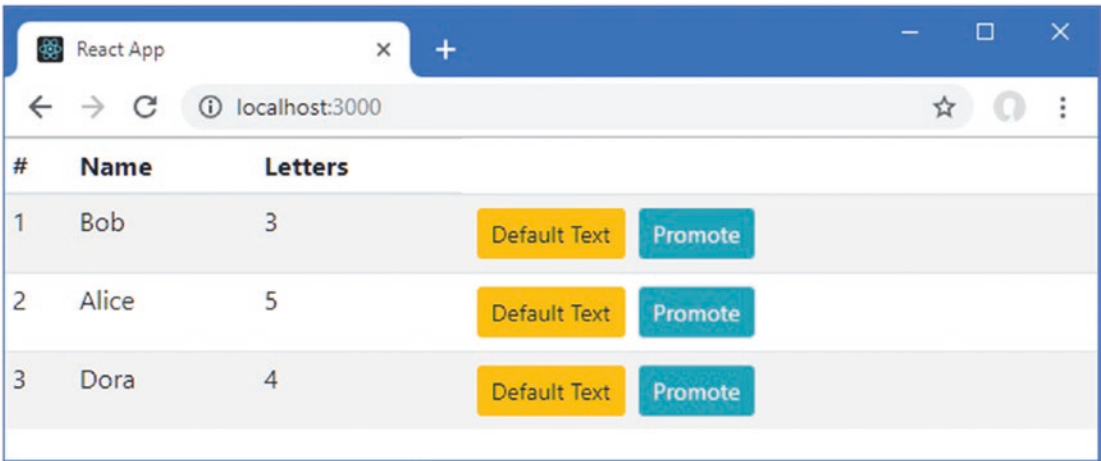
A property called `defaultProps` is added to the component and assigned an object that provides default values for props that are used if the parent component doesn't provide a value. In Listing 10-41, I changed the `Summary` component so that it relies on the default props for one `CallbackButton` element but provides values for the other.

**Listing 10-41.** Relying on Prop Defaults in the `Summary.js` File in the `src` Folder

```
import React from "react";
import { CallbackButton } from "../CallbackButton";

export function Summary(props) {
  return (
    <React.Fragment>
      <td>{ props.index + 1} </td>
      <td>{ props.name } </td>
      <td>{ props.name.length } </td>
      <td>
        <CallbackButton callback={props.reverseCallback} />
        <CallbackButton theme="info" text="Promote"
          callback={ () => props.promoteCallback(props.name)} />
      </td>
    </React.Fragment>
  )
}
```

The first `CallbackButton` element relies on the default values, producing the result shown in Figure 10-17.



**Figure 10-17.** Using default prop values

## Type Checking Prop Values

Props are unable to indicate what data types they are expecting to receive and have no way to signal to their ancestor components when they are unable to use a data value received as a prop. To help avoid these problems, React allows a component to declare the types it expects for its props, as shown in Listing 10-42.

**Listing 10-42.** Declaring Prop Types in the SimpleButton.js File in the src Folder

```
import React from "react";
import PropTypes from "prop-types";

export function SimpleButton(props) {
  return (
    <button onClick={ props.callback } className={props.className}>
      { props.text}
    </button>
  )
}

SimpleButton.defaultProps = {
  disabled: false
}

SimpleButton.propTypes = {
  text: PropTypes.string,
  theme: PropTypes.string,
  callback: PropTypes.func,
  disabled: PropTypes.bool
}
```

A `propTypes` property is added to the component and assigned an object whose property names correspond to prop names and whose values specify the type that the component expects. Types are specified using `PropTypes` values, which are imported from the `prop-types` package, and the most useful `PropTypes` values are described in Table 10-3.

---

■ **Tip** You can combine any of the types in Table 10-3 with `isRequired` to generate a warning if a value for that prop isn't supplied by the parent component: `PropTypes.bool.isRequired`.

---

**Table 10-3.** Useful `PropTypes` Values

Name	Description
array	This value specifies that a prop should be an array.
bool	This value specifies that a prop should be a bool.
func	This value specifies that a prop should be a function.
number	This value specifies that a prop should be a number value.
object	This value specifies that a prop should be an object.
string	This value specifies that a prop should be a string.

To demonstrate how types are checked, in Listing 10-43, I have added a value to the `CallbackButton` element for the `disabled` prop, using a string value rather than the `bool` specified in Listing 10-42.

**Listing 10-43.** Providing the Wrong Type in the `Summary.js` File in the `src` Folder

```
import React from "react";
import { CallbackButton } from "../CallbackButton";

export function Summary(props) {
  return (
    <React.Fragment>
      <td>{ props.index + 1 } </td>
      <td>{ props.name } </td>
      <td>{ props.name.length } </td>
      <td>
        <CallbackButton callback={props.reverseCallback} />
        <CallbackButton theme="info" text="Promote"
callback={ () => props.promoteCallback(props.name)}
disabled="true" />
      </td>
    </React.Fragment>
  )
}
```

This is a common error, where a string literal value is used where a `bool` or number is expected. It can be hard to figure out where the problem is, especially since the prop is defined by an ancestor of the component where the problem occurs. Using a prop type makes the problem obvious. When you save the changes, the browser will reload, and you will see the following message displayed in the browser's JavaScript console:

```
...
index.js:2178 Warning: Failed prop type: Invalid prop `disabled` of type `string` supplied
to `SimpleButton`, expected `boolean`.
...
```

To resolve the problem, I could change the prop value so that it sends the expected type to the component. An alternative approach is to make the component more flexible so that it is able to deal with both `Boolean` and `string` values for the `disabled` prop. Given how common it is to create `string` prop values when `Boolean` values are required, this is a good idea, especially if you are writing components that are going to be used by other development teams. In Listing 10-44, I have added support to the `SimpleButton` component for dealing with both types and updated its `propTypes` configuration to reflect the change.

---

■ **Note** The prop type checks are performed only during development and are disabled when the application is prepared for deployment. See Chapter 8 for an example of preparing an application for deployment.

---

**Listing 10-44.** Accepting Multiple Prop Types in the `SimpleButton.js` File in the `src` Folder

```
import React from "react";
import PropTypes from "prop-types";
```

```

export function SimpleButton(props) {
  return (
    <button onClick={ props.callback } className={props.className}
      disabled={ props.disabled === "true" || props.disabled === true }
      { props.text}
    </button>
  )
}

SimpleButton.defaultProps = {
  disabled: false
}

SimpleButton.propTypes = {
  text: PropTypes.string,
  theme: PropTypes.string,
  callback: PropTypes.func,
  disabled: PropTypes.oneOfType([PropTypes.bool, PropTypes.string])
}

```

There are two useful `PropTypes` methods that can be used to specify multiple types or specific values, as described in Table 10-4.

**Table 10-4.** *Useful `PropTypes` Methods*

Name	Description
<code>oneOfType</code>	This method accepts an array of <code>PropTypes</code> values that the component is willing to receive.
<code>oneOf</code>	This method accepts an array of values that the component is willing to receive.

In Listing 10-44, I used the `oneOfType` method to tell React that the `disabled` property can accept both Boolean and string values. The component is able to process the value I provided for the `disabled` property in Listing 10-43, which disables the button elements, as shown in Figure 10-18.

---

■ **Tip** The alternative approach would have been to change the prop value to a Boolean when applying the component, which can be done using an expression for the `disabled` property: `disabled={ true }`.

---

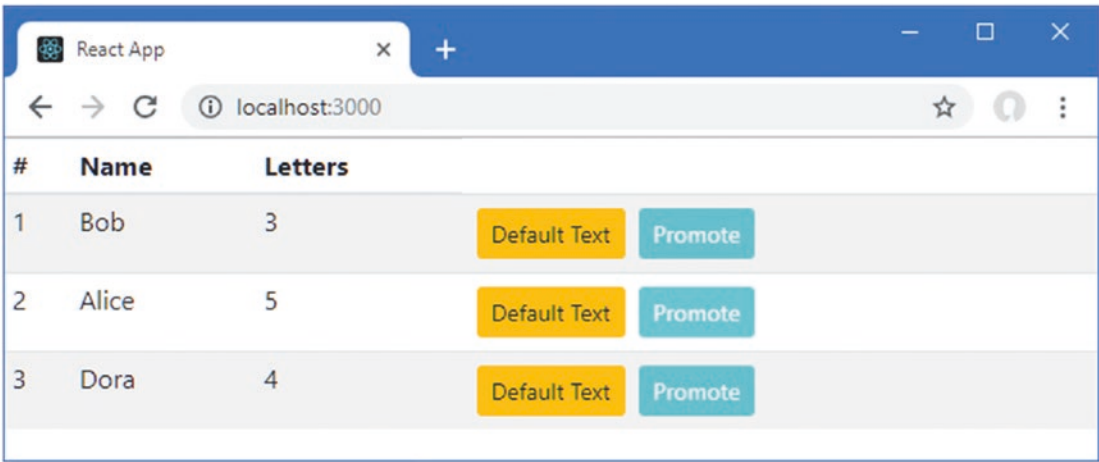


Figure 10-18. Accepting multiple prop types

## Summary

In this chapter, I introduced stateless components, which are the simplest version of the key building block in React applications. I demonstrated how stateless components are defined, how they render content, and how components can be combined to create more complex features. I also explained how a parent component is able to pass on data to its children using props and showed you how props can also be used for functions, which provides the basic features required for communication between components. I finished this chapter by showing you the features that define default values and types for props. In the next chapter, I explain how to create components that have state data.