CHAPTER 14

Composing Applications

In this chapter, I describe the different ways that components can be combined to create complex features. These composition patterns can be used together, and you will find that most problems can be tackled in several ways, which leaves you free to apply the approach with which you are most comfortable. Table 14-1 puts the chapter in context.

Table 14-1. *Putting Application Composition in Context*

Question	Answer
What is it?	Application composition is the combination of components to create complex features.
Why is it useful?	Composition makes development easier by allowing small and simple components to be written and tested before being combined to work together.
How is it used?	There are different patterns available, but the basic approach is to combine multiple components.
Are there any pitfalls or limitations?	The composition patterns can feel awkward if you are used to deriving functionality from classes, such as in C# or Java development. Many problems can be solved equally well by multiple patterns, which can lead to decision paralysis.
Are there any alternatives?	You could write monolithic components that implement all the features required by an application, although this results in a project that is different to test and maintain.

Table 14-2 summarizes the chapter.

Table 14-2. *Chapter Summary*

Problem	Solution	Listing
Display content received from the parent component	Use the `children` prop	8–9
Manipulate the components received via the `children` prop	Use the `React.children` prop	10, 11
Enhance an existing component	Create a specialized component or a higher-order component	12–18
Combine higher-order components	Chain the function calls together	19, 20
Provide a component with the content it should render	Use a render prop	21–24
Distribute data and functions without threading props	Use a context	25–34
Consume a context without using a render prop	Use the simplified context API for class-based components and the `useContext` hook for functional components	35, 36
Prevent errors from unmounting the application	Define an error boundary	37–39

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 14-1.

■ **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 14-1. Creating the Example Project

```
npx create-react-app composition
```

Run the commands shown in Listing 14-2 to navigate to the `composition` folder and add the Bootstrap package to the project.

Listing 14-2. Adding the Bootstrap CSS Framework

```
cd composition
npm install bootstrap@4.1.2
```

To include the Bootstrap CSS stylesheet in the application, add the statement shown in Listing 14-3 to the index.js file, which can be found in the src folder.

Listing 14-3. 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();
```

Creating the Example Components

Add a file called ActionButton.js to the src folder and add the content shown in Listing 14-4.

Listing 14-4. The Contents of the ActionButton.js File in the src Folder

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

export class ActionButton extends Component {

    render() {
        return (
            <button className={` btn btn-${this.props.theme} m-2` }
                    onClick={ this.props.callback }>
                { this.props.text }
            </button>
        )
    }
}
```

This is a similar button component to the one I used in Chapter 13, which accepts configuration settings via its prop, including a function that is invoked in response to the click event. Next, add a file called Message.js to the src folder and add the content shown in Listing 14-5.

Listing 14-5. The Contents of the Message.js File in the src Folder

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

export class Message extends Component {
    render() {
        return (
            <div className={`h5 bg-${this.props.theme } text-white p-2`}>
                { this.props.message }
            </div>
        )
    }
}
```

This component displays a message received as a prop. The final change is to replace the contents of the `App.js` file with the code shown in Listing 14-6, which renders content that uses the other components and defines the state data that the `Message` component requires.

Listing 14-6. The Contents of the App.js File in the src Folder

```
import React, { Component } from 'react';
import { Message } from "./Message";
import { ActionButton } from './ActionButton';

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            counter: 0
        }
    }

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

    render() {
        return  <div className="m-2 text-center">
                    <Message theme="primary"
                        message={ `Counter: ${this.state.counter}`} />
                    <ActionButton theme="secondary"
                        text="Increment" callback={ this.incrementCounter } />
                </div>
    }
}
```

The content rendered by the `App` component displays the `Message` and `ActionButton` components and configures them so that clicking the button will update the `counter` state data value, which has been passed as a prop to the `Message` component.

Using the command prompt, run the command shown in Listing 14-7 in the `composition` folder to start the development tools.

Listing 14-7. 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 content shown in Figure 14-1.

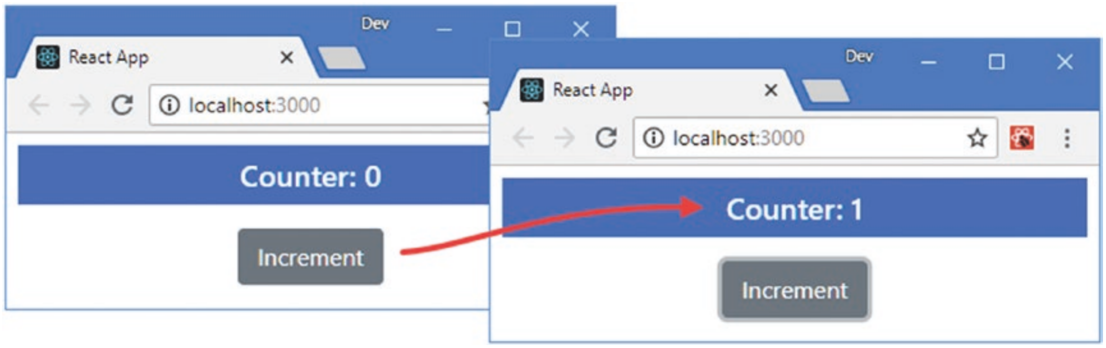


Figure 14-1. *Running the example application*

Understanding the Basic Component Relationship

The components in the example project are simple, but they illustrate the basic relationship that underpins React development: parent components configure children with data props and receive notifications through function props, which leads to state data changes and triggers the update process, as shown in Figure 14-2.

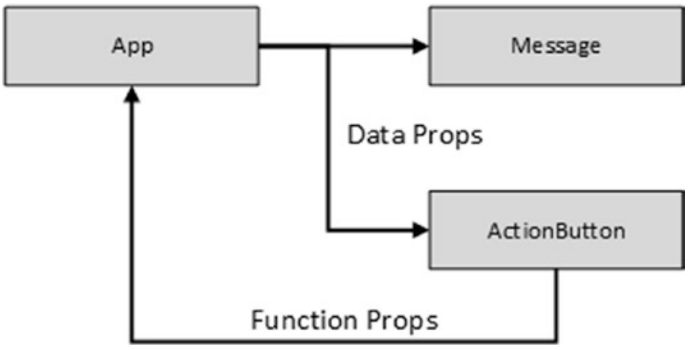


Figure 14-2. *The basic component relationship*

This relationship is the foundation for React development and is the basic pattern used to arrange features in applications. This pattern is easy to understand in a simple example, its use in more complex situations can be less obvious, and it can be hard to know how to locate and distribute the state data, props, and callbacks without duplicating code and data.

Using the Children Prop

React provides a special `children` prop that is used when a component needs to display content provided by its parent but doesn't know what that content will be in advance. This is a useful way of reducing duplication by standardizing features in a container that can be reused across an application. To demonstrate, I created a file called `ThemeSelector.js` in the `src` folder and used it to define the component shown in Listing 14-8.

Listing 14-8. The Contents of the ThemeSelector.js File in the src Folder

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

export class ThemeSelector extends Component {

    render() {
        return (
            <div className="bg-dark p-2">
                <div className="bg-info p-2">
                    { this.props.children }
                </div>
            </div>
        )
    }
}
```

This component renders two `div` elements that contain an expression whose value is the `children` prop. To show how the content for the children prop is provided, Listing 14-9 applies the `ThemeSelector` in the `App` component.

Listing 14-9. Using a Container Component in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { Message } from "./Message";
import { ActionButton } from './ActionButton';
import { ThemeSelector } from './ThemeSelector';

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            counter: 0
        }
    }

    incrementCounter = () => {
        this.setState({ counter: this.state.counter + 1 });
    }
    render() {
        return  <div className="m-2 text-center">
                    <ThemeSelector>
                        <Message theme="primary"
                            message={ `Counter: ${this.state.counter}`} />
                        <ActionButton theme="secondary"
                            text="Increment" callback={ this.incrementCounter } />
                    </ThemeSelector>
                </div>
    }
}
```

The App component provides content for the ThemeSelector component by defining elements between its start and end tags. In this case, the elements apply the Message and ActionButton components. When React processes the content rendered by the App component, the content between the ThemeSelector tags is assigned to the props.children property, producing the result shown in Figure 14-3.

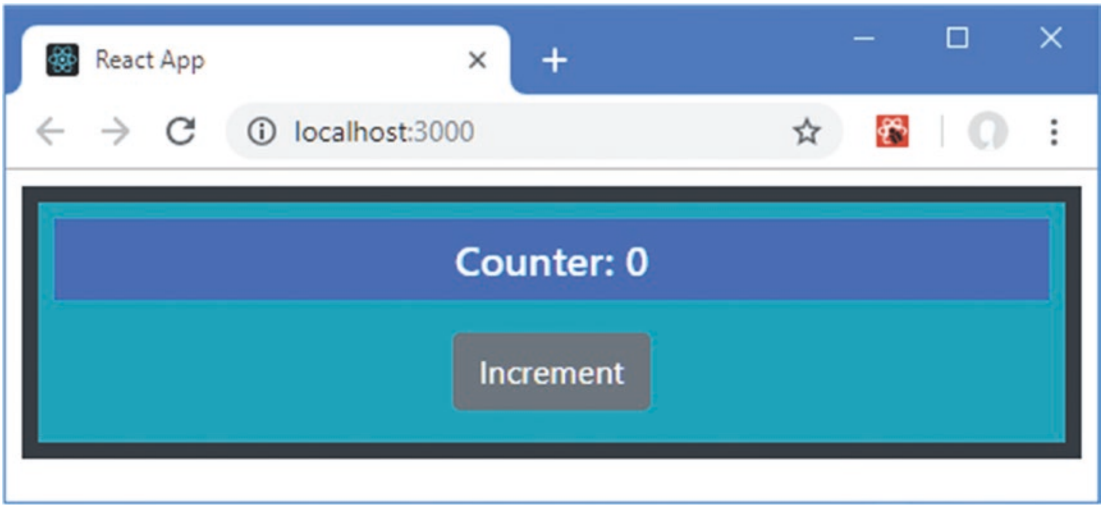


Figure 14-3. *A container component*

The ThemeSelector component doesn't add a lot of value at present, but you can see how it acts as a container for the content provided by the App component.

Manipulating Prop Children

Components that use the children prop are useful only when they are able to provide services to their children, which can be difficult when there is no advanced knowledge of what those children will provide. To help work around this limitation, React provides a number of methods that a container can use to manipulate its children, as described in Table 14-3.

Table 14-3. *The Container Children Methods*

Name	Description
`React.Children.map`	This method invokes a function for each child and returns an array of the function results.
`React.Children.forEach`	This method invokes a function for each child without returning an array.
`React.Children.count`	This method returns the number of children.
`React.Children.only`	This method throws an array if the collection of children it receives contains more than one child.
`React.Children.toArray`	This method returns an array of children, which can be used to reorder or remove elements.
`React.cloneElement`	This method is used to duplicate a child element and allows new props to be added by the container.

Adding Props to Container Children

A component can't manipulate the content it receives from the parent directly, so to provide the components received through the `children` prop with additional data or functions, the `React.Children.map` method is used in conjunction with the `React.cloneElement` method to duplicate the child components and assign additional props.

Listing 14-10 adds a `select` element to the content rendered by the `ThemeSelector` that updates a state data property and allows a user to choose one of the theme colors provided by the Bootstrap CSS framework, which is then passed on to the container's children as a prop.

Listing 14-10. Adding Theme Selection in the ThemeSelector.js File in the src Folder

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

export class ThemeSelector extends Component {

    constructor(props) {
        super(props);
        this.state = {
            theme: "primary"
        }
        this.themes = ["primary", "secondary", "success", "warning", "dark"];
    }

    setTheme = (event) => {
        this.setState({ theme : event.target.value });
    }

    render() {

        let modChildren = React.Children.map(this.props.children,
            (c => React.cloneElement(c, { theme: this.state.theme})));
        return (
            <div className="bg-dark p-2">
                <div className="form-group text-left">
                    <label className="text-white">Theme:</label>
                    <select className="form-control" value={ this.state.theme }
                            onChange={ this.setTheme }>
                        { this.themes.map(theme =>
                            <option key={ theme } value={ theme }>{theme}</option>) }
                    </select>
                </div>

                <div className="bg-info p-2">
                    { modChildren }
                </div>
            </div>
        )
    }
}
```

Because props are read-only, I can't use the `React.Children.forEach` method to simply enumerate the child components and assign a new property to their `props` object. Instead, I used the `map` method to enumerate the children and used the `React.cloneElement` method to duplicate each child with an additional prop.

```
...
let modChildren = React.Children.map(this.props.children,
    (c => React.cloneElement(c, { theme: this.state.theme})));
...
```

The `cloneElement` method accepts a child component and a props object, which is merged with the child component's existing props.

One consequence of using the `map` method to enumerate the child components into an array is that React expects each component to have a key prop and will report a warning in the browser's JavaScript console.

The result is that the props passed to the `Message` and `ActionButton` components are a combination of those defined by the `App` component and those added using the `cloneElement` method by the `ThemeSelector` component. When you choose a theme from the `select` element, an update is performed, and the selected theme is applied to the `Message` and `ActionButton` components, as shown in Figure 14-4.

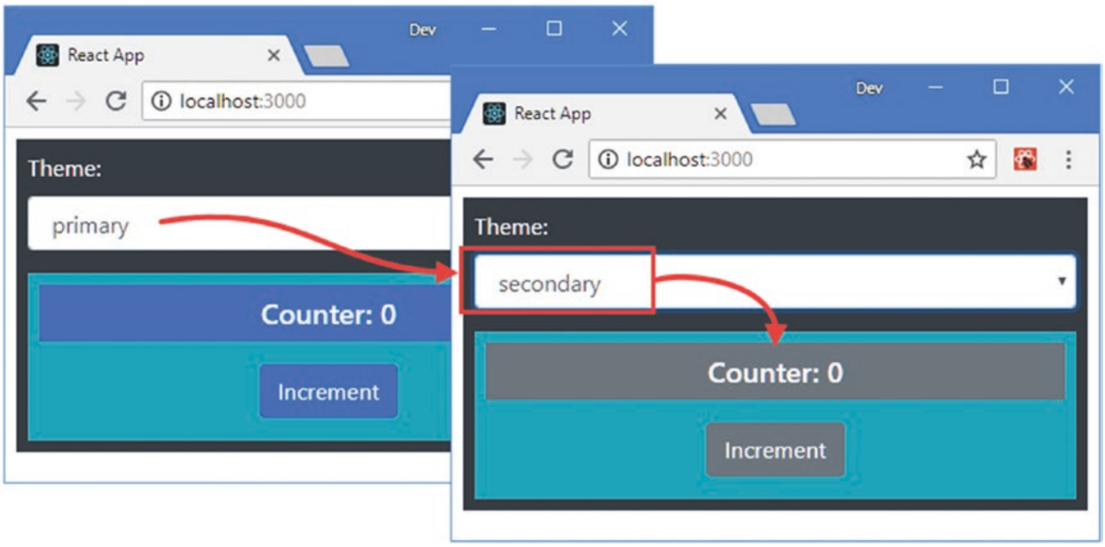


Figure 14-4. *Adding props to contained components*

Ordering or Omitting Components

Although a container doesn't have any advanced knowledge of its children, the `toArray` method described in Table 14-3 can be used to convert the children to an array that can be manipulated using the standard JavaScript features, such as sorting or adding and removing items. This type of operation can also be performed on the result from the `React.Children.map` method, also described in Table 14-3, which returns an array as well.

In Listing 14-11, I have added a button to the `ThemeSelector` component that reverses the order of the children when it is clicked, which I achieve by calling the `reverse` method on the array produced by the `map` method.

Listing 14-11. Reversing Children in the ThemeSelector.js File in the src Folder

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

export class ThemeSelector extends Component {

    constructor(props) {
        super(props);
        this.state = {
            theme: "primary",
            reverseChildren: false
        }
        this.themes = ["primary", "secondary", "success", "warning", "dark"];
    }
    setTheme = (event) => {
        this.setState({ theme : event.target.value });
    }

    toggleReverse = () => {
        this.setState({ reverseChildren: !this.state.reverseChildren});
    }

    render() {

        let modChildren = React.Children.map(this.props.children,
            (c => React.cloneElement(c, { theme: this.state.theme})));

        if (this.state.reverseChildren) {
            modChildren.reverse();
        }

        return (
            <div className="bg-dark p-2">
                <button className="btn btn-primary" onClick={ this.toggleReverse }>
                    Reverse
                </button>
                <div className="form-group text-left">
                    <label className="text-white">Theme:</label>
                    <select className="form-control" value={ this.state.theme }
                            onChange={ this.setTheme }>
                        { this.themes.map(theme =>
                            <option key={ theme } value={ theme }>{theme}</option>) }
                    </select>
                </div>

                <div className="bg-info p-2">
                    { modChildren }
                </div>
            </div>
        )
    }
}
```

This type of operation is more typically used with lists of similar objects, such as products in an online store, but it can be applied to any children, as shown in Figure 14-5.

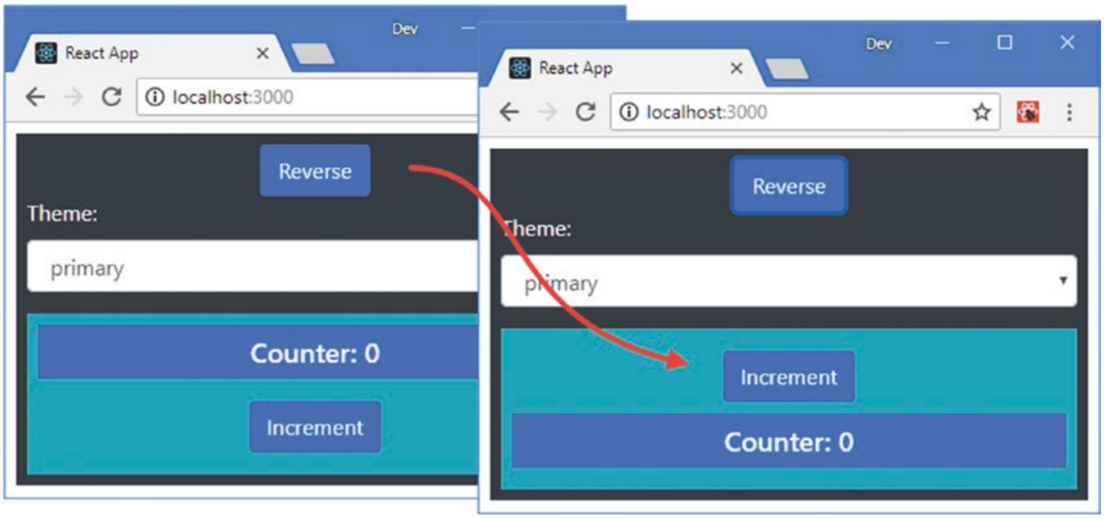


Figure 14-5. Changing the order of child components in a container

Creating a Specialized Component

Some components provide specialized versions of the features provided by another, more general, component. In some frameworks, specialization is handled by using features such as class inheritance, but React relies on the specialized component rendering the more general component and managing its behavior with props. To demonstrate, I added a file called `GeneralList.js` to the `src` folder and used it to define the component shown in Listing 14-12.

■ **Note** If you are used to class-based languages, such as C# or Java, you might expect to create a subclass using the same `extends` keyword that stateful components employ to inherit functionality from the React `Component` class. This is not how React is intended to be used, and you should compose components even though it can feel odd to do so at first.

Listing 14-12. The Contents of the GeneralList.js File in the src Folder

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

export class GeneralList extends Component {

    render() {
        return (
            <div className={`bg-${this.props.theme} text-white p-2`}>
                { this.props.list.map((item, index) =>
                    <div key={ item }>{ index + 1 }: { item }</div>
                )}
            </div>
        )
    }
}
```

This component receives a prop named `list`, which is processed using the array `map` method to render a series of `div` elements. To create a component that receives a list and allows it to be sorted, I can create a more specialized component that builds on the features provided by the `GeneralList`. I added a file called `SortedList.js` to the `src` folder and used it to define the component shown in Listing 14-13.

Listing 14-13. The Contents of the SortedList.js File in the src Folder

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

export class SortedList extends Component {

    constructor(props) {
        super(props);
        this.state = {
            sort: false
        }
    }

    getList() {
        return this.state.sort
            ? [...this.props.list].sort() : this.props.list;
    }

    toggleSort = () => {
        this.setState({ sort : !this.state.sort });
    }

    render() {
        return (
            <div>
                <GeneralList list={ this.getList() } theme="info" />
                <div className="text-center m-2">
                    <ActionButton theme="primary" text="Sort"
                        callback={this.toggleSort} />
                </div>
            </div>
        )
    }
}
```

The `SortedList` renders a `GeneralList` as part of its output and uses the `list` prop to control the presentation of the data, allowing the user to selected a sorted or unsorted list. In Listing 14-14, I have changed the layout of the `App` component to show the general and more specific components side by side.

Listing 14-14. Changing the Component Layout in the App.js File in the src Folder

```
import React, { Component } from 'react';
//import { Message } from "./Message";
//import { ActionButton } from './ActionButton';
//import { ThemeSelector } from './ThemeSelector';
import { GeneralList } from './GeneralList';
import { SortedList } from "./SortedList";

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            // counter: 0
            names: ["Zoe", "Bob", "Alice", "Dora", "Joe"]
        }
    }

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

    render() {
        return  (
            <div className="container-fluid">
                <div className="row">
                    <div className="col-6">
                        <GeneralList list={ this.state.names } theme="primary" />
                    </div>
                    <div className="col-6">
                        <SortedList list={ this.state.names } />
                    </div>
                </div>
            </div>
        )
    }
}
```

The result is that the general list and the sortable list are both presented to the user, as shown in Figure 14-6.

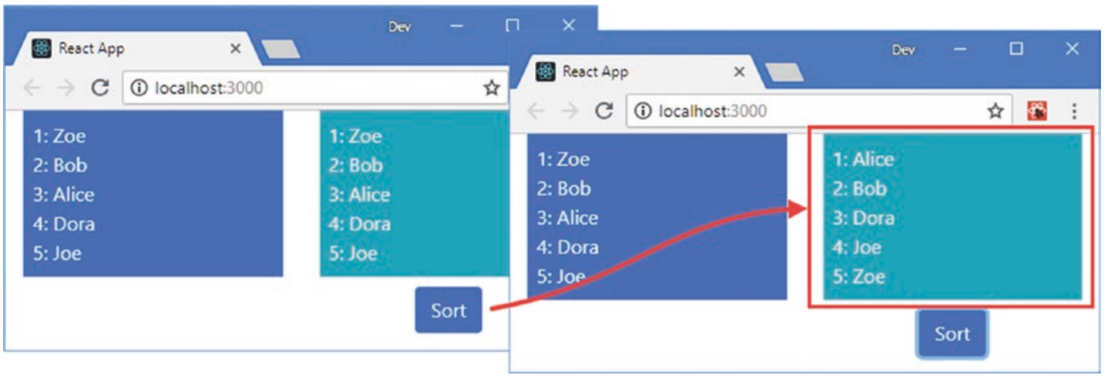


Figure 14-6. *General and more specialized components*

Creating Higher-Order Components

Higher-order components (HOCs) provide an alternative to specialized components and are useful when components require common code but may not render related content. HOCs are often used for *cross-cutting concerns*, a term that refers to tasks that span the entire application and would otherwise lead to the same features being implemented in several places. The most commonly encountered examples of cross-cutting concerns are authorization, logging, and data retrieval. To demonstrate the use of HOCs, I added a file called `ProFeature.js` to the `src` folder and used it to define the component shown in Listing 14-15.

Listing 14-15. The Contents of the ProFeature.js File in the src Folder

```
import React from "react";

export function ProFeature(FeatureComponent) {
    return function(props) {

        if (props.pro) {
            let { pro, ...childProps}  = props;
            return <FeatureComponent {...childProps} />
        } else {
            return (
                <h5 className="bg-warning text-white text-center">
                    This is a Pro Feature
                </h5>
            )
        }
    }
}
```

A HOC is a function that accepts a component and returns a new component that wraps around it to provide additional features. In Listing 14-15, the HOC is a function called `ProFeature`, and it accepts a component that should be presented to the user only when the value of the prop named `pro` is `true`, acting as a simple authorization feature. To display the component, the `render` method uses the component received as the function argument and passes on all of its props, except the one named `pro`.

```
...
let { pro, ...childProps}  = props;
return <FeatureComponent {...childProps} />
...
```

When the pro prop is false, the ProFeature HOC function returns a header element that displays a warning. Listing 14-16 updates the App component to use ProFeature to protect one of its child components.

Listing 14-16. Using an HOC in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { GeneralList } from './GeneralList';
import { SortedList } from "./SortedList";
import { ProFeature } from "./ProFeature";

const ProList = ProFeature(SortedList);

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            names: ["Zoe", "Bob", "Alice", "Dora", "Joe"],
            cities: ["London", "New York", "Paris", "Milan", "Boston"],
            proMode: false
        }
    }

    toggleProMode = () => {
        this.setState({ proMode: !this.state.proMode});
    }

    render() {
        return (
            <div className="container-fluid">
                <div className="row">
                    <div className="col-12 text-center p-2">
                        <div className="form-check">
                            <input type="checkbox" className="form-check-input"
                                value={ this.state.proMode }
                                onChange={ this.toggleProMode } />
                            <label className="form-check-label">Pro Mode</label>
                        </div>
                    </div>
                </div>
                <div className="row">
                    <div className="col-3">
                        <GeneralList list={ this.state.names } theme="primary" />
                    </div>
```

```
                        <div className="col-3">
                            <ProList list={ this.state.names }
                                    pro={ this.state.proMode } />
                        </div>
                        <div className="col-3">
                            <GeneralList list={ this.state.cities } theme="secondary" />
                        </div>
                        <div className="col-3">
                            <ProList list={ this.state.cities }
                                pro={ this.state.proMode } />
                        </div>
                    </div>
                </div>
            )
        }
}
```

HOCs are used to create new components by invoking the function, like this:

```
...
const ProList = ProFeature(SortedList);
...
```

Because HOCs are functions, you can define additional arguments to configure behavior, but in this example, I pass the component that I want to wrap as the only argument. I assign the result from the function to a constant named `ProList`, which I use like any other component in the render method.

```
...
<ProList list={ this.state.cities } pro={ this.state.proMode } />
...
```

I defined the `pro` prop for the HOC and the `list` prop for the `SortedList` component that it wraps. The value of the `pro` prop is set by toggling a checkbox, with the effect shown in Figure 14-7.

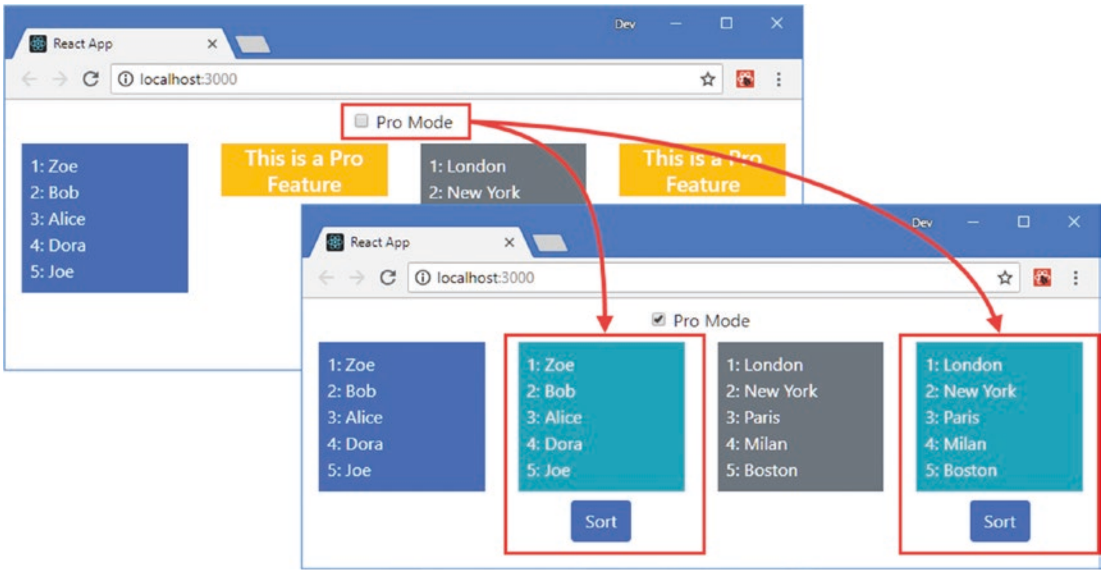


Figure 14-7. *Using higher-order components*

Creating Stateful Higher-Order Components

Higher-order components can be stateful, which allows for more complex features to be added to an application. I added a file called `ProController.js` to the `src` folder and used it to define the HOC shown in Listing 14-17.

Listing 14-17. The Contents of the ProController.js File in the src Folder

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

export function ProController(FeatureComponent) {

    const ProtectedFeature = ProFeature(FeatureComponent);

    return class extends Component {

        constructor(props) {
            super(props);
            this.state = {
                proMode: false
            }
        }

        toggleProMode = () => {
            this.setState({ proMode: !this.state.proMode});
        }
        render() {
            return (
                <div className="container-fluid">
                    <div className="row">
                        <div className="col-12 text-center p-2">
                            <div className="form-check">
                                <input type="checkbox" className="form-check-input"
                                    value={ this.state.proMode }
                                    onChange={ this.toggleProMode } />
                                <label className="form-check-label">Pro Mode</label>
                            </div>
                        </div>
                    </div>
                    <div className="row">
                        <div className="col-12">
                            <ProtectedFeature {...this.props}
                                 pro={ this.state.proMode } />
                        </div>
                    </div>
                </div>
            )
        }
    }
}
```

The HOC function returns a class-based stateful component that presents the checkbox and uses the ProFeature HOC to control visibility of the wrapped component. Listing 14-18 updates App component to use the ProController component.

Listing 14-18. Using a New HOC in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { GeneralList } from './GeneralList';
import { SortedList } from "./SortedList";
//import { ProFeature } from "./ProFeature";
import { ProController } from "./ProController";

const ProList = ProController(SortedList);

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            names: ["Zoe", "Bob", "Alice", "Dora", "Joe"],
            cities: ["London", "New York", "Paris", "Milan", "Boston"],
            //proMode: false
        }
    }
    render() {
        return (
            <div className="container-fluid">
                <div className="row">
                    <div className="col-3">
                        <GeneralList list={ this.state.names } theme="primary" />
                    </div>
                    <div className="col-3">
                        <ProList list={ this.state.names }  />
                    </div>
                    <div className="col-3">
                        <GeneralList list={ this.state.cities } theme="secondary" />
                    </div>
                    <div className="col-3">
                        <ProList list={ this.state.cities }  />
                    </div>
                </div>
            </div>
        )
    }
}
```

Figure 14-8 shows the effect of HOC, which gives each protected component its own checkbox.

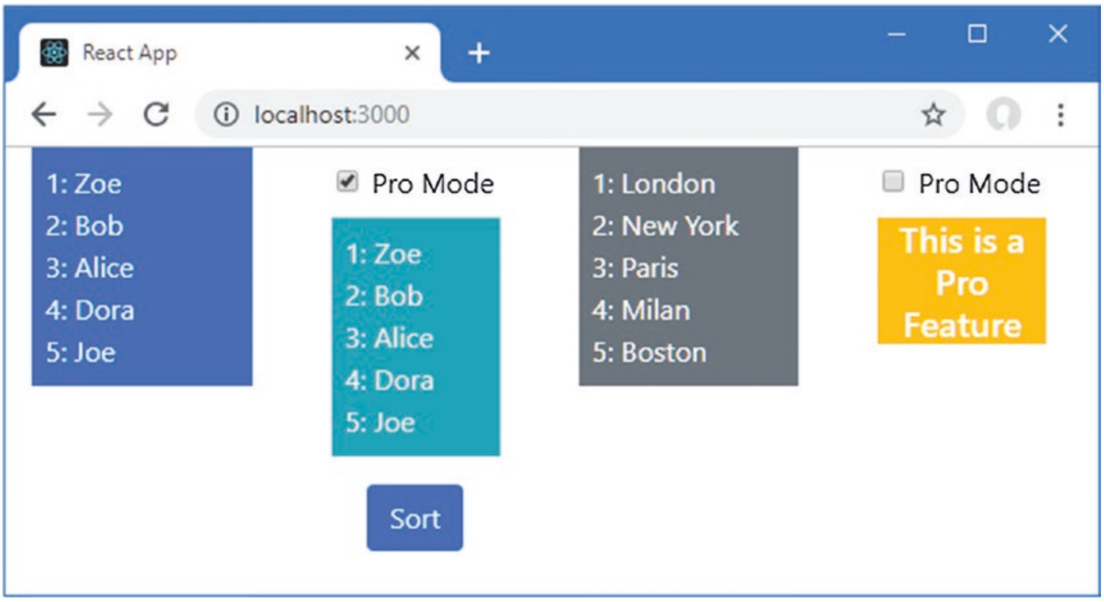


Figure 14-8. *A stateful higher-order component*

Combining Higher-Order Components

A useful feature of HOCs is they can be combined by changing only the function call that creates the wrapped component class. To demonstrate, I added a file called `LogToConsole.js` to the `src` folder and used it to define the HOC shown in Listing 14-19.

Listing 14-19. The Contents of the LogToConsole.js File in the src Folder

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

export function LogToConsole(FeatureComponent, label, logMount, logRender, logUnmount) {
    return class extends Component {

        componentDidMount() {
            if (logMount) {
                console.log(`${label}: mount`);
            }
        }

        componentWillUnmount() {
            if (logUnmount) {
                console.log(`${label}: unmount`);
            }
        }

        render() {
            if (logRender) {
                console.log(`${label}: render`);
            }
            return <FeatureComponent { ...this.props } />
        }
    }
}
```

The HOC function receives the component that will be wrapped, along with a label argument that is used to write messages to the browser's JavaScript console. There are three further arguments that specify whether log messages will be written when the component is mounted, rendered, and unmounted, following the stateful component lifecycle described in Chapter 11. To apply the new HOC, I have changed only the function that creates the wrapped component, as shown in Listing 14-20.

Listing 14-20. Combining HOCs in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { GeneralList } from './GeneralList';
import { SortedList } from "./SortedList";
//import { ProFeature } from "./ProFeature";
import { ProController } from "./ProController";
import { LogToConsole } from "./LogToConsole";

const ProList = ProController(LogToConsole(SortedList, "Sorted", true, true, true));
```

```
export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            names: ["Zoe", "Bob", "Alice", "Dora", "Joe"],
            cities: ["London", "New York", "Paris", "Milan", "Boston"],
            //proMode: false
        }
    }
    render() {
        return (
            <div className="container-fluid">
                <div className="row">
                    <div className="col-3">
                        <GeneralList list={ this.state.names }
                            theme="primary" />
                    </div>
                    <div className="col-3">
                        <ProList list={ this.state.names }  />
                    </div>
                    <div className="col-3">
                        <GeneralList list={ this.state.cities }
                            theme="secondary" />
                    </div>
                    <div className="col-3">
                        <ProList list={ this.state.cities }  />
                    </div>
                </div>
            </div>
        )
    }
}
```

The effect is that the `SortedList` component is wrapped by the `LogToConsole` component, which is in turn wrapped by the `ProFeature` component. As you toggle the Pro Mode checkbox, you will see the following messages displayed in the browser's JavaScript console:

```
...
Sorted: render
Sorted: mount
Sorted: unmount
...
```

Using Render Props

A *render prop* is a function prop that provides a component with the content it should render, providing an alternative model of wrapping one component in another. In Listing 14-21, I have rewritten the `ProFeature` component so it uses a render prop.

Listing 14-21. Using a Render Prop in the ProFeature.js File in the src Folder

```
import React from "react";

export function ProFeature(props) {
    if (props.pro) {
        return props.render();
    } else {
        return (
            <h5 className="bg-warning text-white text-center">
                This is a Pro Feature
            </h5>
        )
    }
}
```

Components that use render props are defined in the normal way. The difference is in the `render` method, where a function prop named `render` is invoked to display content provided by the parent.

```
...
return props.render();
...
```

The parent component provides the function for the `render` prop when it applies the component. In Listing 14-22, I have changed the App component so it provides the `ProFeature` component with the function it requires. (I have also removed some of the content for sake of brevity.)

■ **Tip** The name of the prop that the parent uses to provide the function doesn't have to be `render`, although that is the convention. You can use any name, just as long as it is applied consistently in both the parent and the child components.

Listing 14-22. Using a Render Prop in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { GeneralList } from './GeneralList';
import { SortedList } from "./SortedList";
import { ProFeature } from "./ProFeature";
// import { ProController } from "./ProController";
// import { LogToConsole } from "./LogToConsole";

// const ProList = ProController(LogToConsole(SortedList, "Sorted", true, true));

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            names: ["Zoe", "Bob", "Alice", "Dora", "Joe"],
            cities: ["London", "New York", "Paris", "Milan", "Boston"],
            proMode: false
        }
    }

    toggleProMode = () => {
        this.setState({ proMode: !this.state.proMode});
    }

    render() {
        return (
            <div className="container-fluid">
                <div className="row">
                    <div className="col-12 text-center p-2">
                        <div className="form-check">
                            <input type="checkbox" className="form-check-input"
                                value={ this.state.proMode }
                                onChange={ this.toggleProMode } />
                            <label className="form-check-label">Pro Mode</label>
                        </div>
                    </div>
                </div>
                <div className="row">
                    <div className="col-6">
                        <GeneralList list={ this.state.names }
                            theme="primary" />
                    </div>
                    <div className="col-6">
                        <ProFeature pro={ this.state.proMode }
                            render={ () => <SortedList list={ this.state.names } /> }
                        />
                    </div>
                </div>
            </div>
        )
    }
}
```

The ProFeature component is provided with a pro prop that is used to determine whether a feature is displayed and with a render prop that is set to a function that returns a SortedList element.

```
...
<ProFeature pro={ this.state.proMode }
    render={ () => <SortedList list={ this.state.names } /> } />
...
```

When React renders the application's content, the ProFeature component's render method is invoked, which in turn invokes the render prop function, which leads to the creation of a new SortedList component. Using a render prop achieves the same result as the HOC, as shown in Figure 14-9.

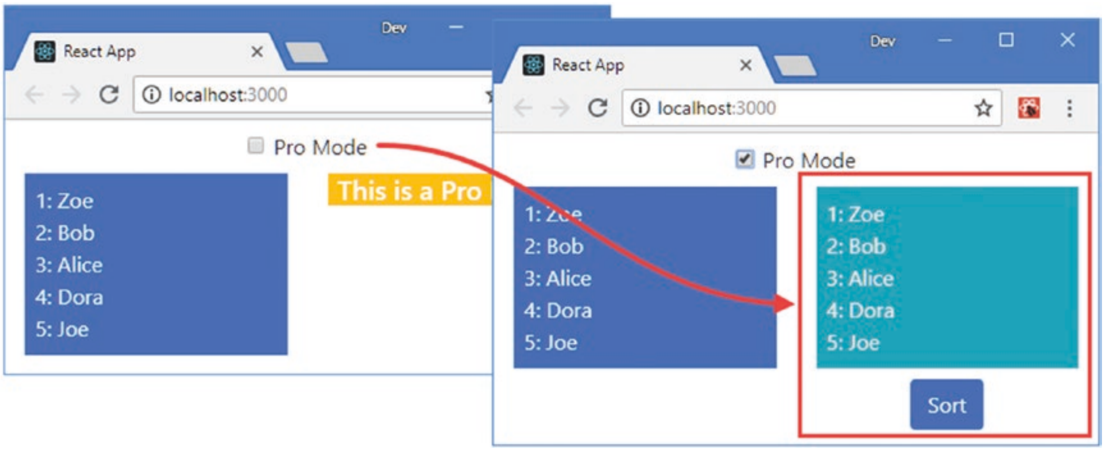


Figure 14-9. *Using a render prop*

Using a Render Prop with an Argument

Render props are regular JavaScript functions, which means they can be invoked with arguments. This can be a useful feature in its own right, but it also helps understand how the context feature works, which I describe in the next section.

Using an argument allows the component that invokes the `render` prop to provide props to the wrapper content. This is a technique that is more readily understood with an example. In Listing 14-23, I changed the `ProFeature` component so that it passes a string argument to the `render` prop function.

Listing 14-23. Adding an Argument in the ProFeature.js File in the src Folder

```
import React from "react";

export function ProFeature(props) {
    if (props.pro) {
        return props.render("Pro Feature");
    } else {
        return (
            <h5 className="bg-warning text-white text-center">
                This is a Pro Feature
            </h5>
        )
    }
}
```

The argument can be received by the component that defines the render prop function and used in the content that it produces, as shown in Listing 14-24.

Listing 14-24. Receiving a Render Prop Argument in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { GeneralList } from './GeneralList';
import { SortedList } from "./SortedList";
import { ProFeature } from "./ProFeature";
```

```
export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            names: ["Zoe", "Bob", "Alice", "Dora", "Joe"],
            cities: ["London", "New York", "Paris", "Milan", "Boston"],
            proMode: false
        }
    }

    toggleProMode = () => {
        this.setState({ proMode: !this.state.proMode});
    }

    render() {
        return (
            <div className="container-fluid">
                <div className="row">
                    <div className="col-12 text-center p-2">
                        <div className="form-check">
                            <input type="checkbox" className="form-check-input"
                                value={ this.state.proMode }
                                onChange={ this.toggleProMode } />
                            <label className="form-check-label">Pro Mode</label>
                        </div>
                    </div>
                </div>

                <div className="row">
                    <div className="col-6">
                        <GeneralList list={ this.state.names }
                            theme="primary" />
                    </div>
                    <div className="col-6">
                        <ProFeature pro={ this.state.proMode }
                            render={ text =>
                                <React.Fragment>
                                    <h4 className="text-center">{ text }</h4>
                                    <SortedList list={ this.state.names } />
                                </React.Fragment>
                            } />
                    </div>
                </div>
            </div>
        )
    }
}
```

The content produced when the checkbox is selected shows how the ProFeature component is able to influence the content produced by the render prop function, as shown in Figure 14-10.

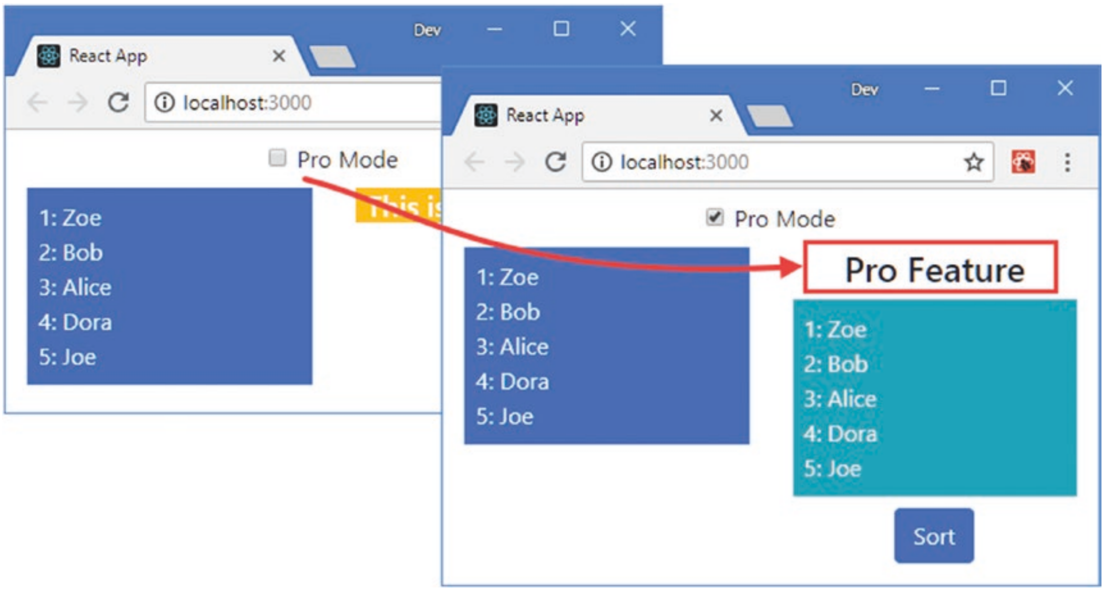


Figure 14-10. *Using a render prop with an argument*

Using Contexts for Global Data

The management of props can become difficult, regardless of how you choose to compose your application. As the complexity of the application increases, so does the number of components that need to cooperate. As the hierarchy of components grows, the state data gets lifted up higher in the application, further away from where that data is used, with the result that every component has to pass on props that it doesn't use directly but that its descendants rely on.

To help avoid this problem, React provides the *context* feature, which allows state data to be passed from where it is defined to where it is needed without having to be relayed through the intermediate components. To demonstrate, I am going to make the Pro Mode in the example application more granular so that it disables the Sort button rather than hides the data list entirely.

In Listing 14-25, I have added a property to the button element rendered by the `ActionButton` component that sets the `disabled` property based on a prop and changes the Bootstrap theme to make it more obvious when the button is disabled.

■ **Tip** The Redux package is often used for more complex projects and can be easier to use in large applications. See Chapters 19 and 20 for details.

Listing 14-25. Disabling a Button in the ActionButton.js File in the src Folder

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

export class ActionButton extends Component {
    render() {
        return (
            <button className={ this.getClasses(this.props.proMode)}
                    disabled={ !this.props.proMode }
                    onClick={ this.props.callback }>
                { this.props.text }
            </button>
        )
    }

    getClasses(proMode) {
        let col = proMode ? this.props.theme : "danger";
        return `btn btn-${col} m-2`;
    }
}
```

The proMode property that the ActionButton depends on is part of the App component's state, which also defines the checkbox that is used to change its value. The result is a chain of components that have to receive the proMode property from their parent and pass it on to their children. Even in the simple example application, this means that the SortedList component has to pass on the proMode data value even though it doesn't use it directly, as shown in Figure 14-11.

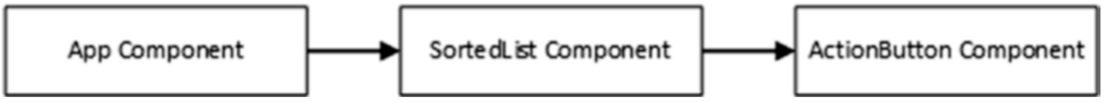


Figure 14-11. *Passing on props in the example application*

This is known as *prop drilling* or *prop threading*, where data values are passed through the component hierarchy to reach the point where they can be used. It is easy to forget to pass on a prop that is required by a descendant, and it can be hard work to figure out where the threading of a prop has missed a step in a complex application. In Listing 14-26, I have updated the App component to remove the ProFeature component from the previous section and to pass on the value of the proMode state property to the SortedList component as a prop, beginning the process of threading the prop.

Listing 14-26. Threading a Prop in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { GeneralList } from './GeneralList';
import { SortedList } from "./SortedList";
//import { ProFeature } from "./ProFeature";

export default class App extends Component {
    constructor(props) {
        super(props);
        this.state = {
            names: ["Zoe", "Bob", "Alice", "Dora", "Joe"],
            cities: ["London", "New York", "Paris", "Milan", "Boston"],
            proMode: false
        }
    }

    toggleProMode = () => {
        this.setState({ proMode: !this.state.proMode});
    }

    render() {
        return (
            <div className="container-fluid">
                <div className="row">
                    <div className="col-12 text-center p-2">
                        <div className="form-check">
                            <input type="checkbox" className="form-check-input"
                                value={ this.state.proMode }
                                onChange={ this.toggleProMode } />
                            <label className="form-check-label">Pro Mode</label>
                        </div>
                    </div>
                </div>

                <div className="row">
                    <div className="col-6">
                        <GeneralList list={ this.state.names }
                            theme="primary" />
                    </div>
                    <div className="col-6">
                        <SortedList proMode={this.state.proMode}
                            list={ this.state.names } />
                    </div>
                </div>
            </div>
        )
    }
}
```

The SortedList component doesn't use the proMode prop directly, but it must be passed on to the ActionButton component, completing the prop threading, as shown in Listing 14-27.

Listing 14-27. Threading a Prop in the SortedList.js File in the src Folder

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

export class SortedList extends Component {
    constructor(props) {
        super(props);
        this.state = {
            sort: false
        }
    }

    getList() {
        return this.state.sort
            ? [...this.props.list].sort() : this.props.list;
    }

    toggleSort = () => {
        this.setState({ sort : !this.state.sort });
    }

    render() {
        return (
            <div>
                <GeneralList list={ this.getList() } theme="info" />
                <div className="text-center m-2">
                    <ActionButton theme="primary" text="Sort"
                        proMode={ this.props.proMode }
                        callback={this.toggleSort} />
                </div>
            </div>
        )
    }
}
```

The result is that the value of the proMode state value is passed from the App component, through the SortedList component, and is received and used by the ActionButton component, as shown in Figure 14-12.

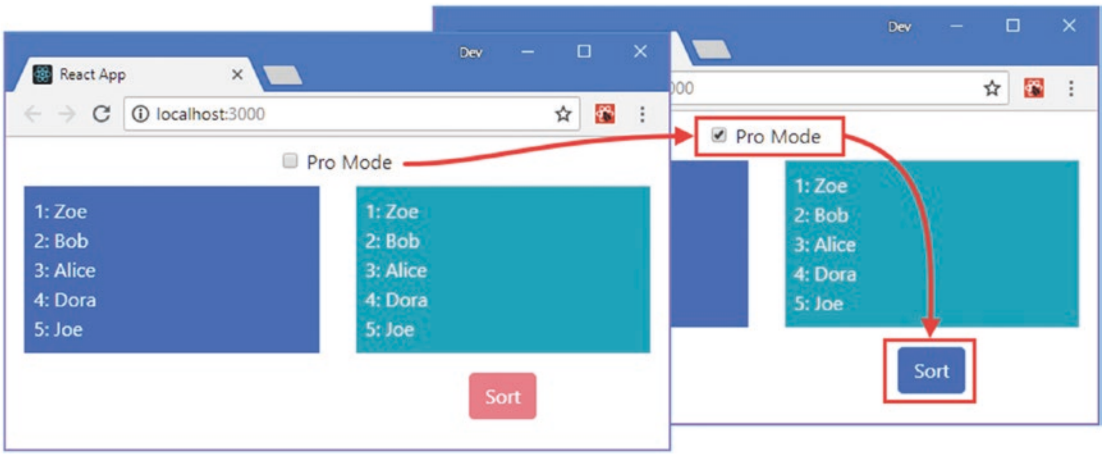


Figure 14-12. *Threading a prop through the application*

It is this problem that the context feature solves, allowing state data to be passed directly to the component that uses it, without needing to be threaded through the intermediate components that separate them in the hierarchy.

Defining the Context

The first step is to define the context, which is the mechanism by which the state data is distributed. Contexts can be defined anywhere in the application. I added a file called `ProModeContext.js` in the `src` folder with the code shown in Listing 14-28.

Listing 14-28. The Contents of the ProModeContext.js in the src Folder

```
import React from "react";

export const ProModeContext = React.createContext({
    proMode: false
})
```

The `React.createContext` method is used to create a new context and is provided with a data object that is used to specify the context's default values, which are overridden when the context is used, which I demonstrate shortly. In the listing, the context that I defined is called `ProModeContext`, and it defines a `proMode` property, which will be `false` by default.

Creating the Context Consumer

The next step is to consume the context where the data value is required, as shown in Listing 14-29.

Listing 14-29. Creating a Context Consumer in the ActionButton.js File in the src Folder

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

export class ActionButton extends Component {

    render() {
        return (
            <ProModeContext.Consumer>
                { contextData =>
                    <button
                        className={ this.getClasses(contextData.proMode)}
                        disabled={ !contextData.proMode }
                        onClick={ this.props.callback }>
                            { this.props.text }
                    </button>
                }
            </ProModeContext.Consumer>
        )
    }
    getClasses(proMode) {
        let col = proMode ? this.props.theme : "danger";
        return `btn btn-${col} m-2`;
    }
}
```

Consuming a context is similar to defining a render prop, with the addition of a custom HTML element to select the context that is required. First comes the HTML element whose tag name is the context name, followed by a period, followed by `Consumer`, like this:

```
...
return <ProModeContext.Consumer>

   // ...context can be consumed here...

</ProModeContext.Consumer>
...
```

Between the start and end tags of the HTML element is a function that receives the context object and renders the content that depends on it.

```
...
<ProModeContext.Consumer>
    { contextData =>
        <button
            className={ this.getClasses(contextData.proMode)}
            disabled={ !contextData.proMode }
            onClick={ this.props.callback }>
                { this.props.text }
        </button>
    }
</ProModeContext.Consumer>
...
```

The component can still access the component's state and prop data, which can be mixed freely with the data provided by the context. In this example, the `callback` prop is still used to handle `click` events, while the `proMode` context property is used to set the value of the `className` and `disabled` attributes.

Creating the Context Provider

The final step is to create a context provider, which associates the source state data with the context, as shown in Listing 14-30.

Listing 14-30. Creating a Context Provider in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { GeneralList } from './GeneralList';
import { SortedList } from "./SortedList";
import { ProModeContext } from './ProModeContext';
```

```
export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            names: ["Zoe", "Bob", "Alice", "Dora", "Joe"],
            cities: ["London", "New York", "Paris", "Milan", "Boston"],
            //proMode: false,
            proContextData: {
                proMode: false
            }
        }
    }

    toggleProMode = () => {
        this.setState(state =>  state.proContextData.proMode
            = !state.proContextData.proMode);
    }

    render() {
        return (
            <div className="container-fluid">
                <div className="row">
                    <div className="col-12 text-center p-2">
                        <div className="form-check">
                            <input type="checkbox" className="form-check-input"
                                value={ this.state.proContextData.proMode }
                                onChange={ this.toggleProMode } />
                            <label className="form-check-label">Pro Mode</label>
                        </div>
                    </div>
                </div>
                <div className="row">
                    <div className="col-6">
                        <GeneralList list={ this.state.names }
                            theme="primary" />
                    </div>
                    <div className="col-6">
                        <ProModeContext.Provider value={ this.state.proContextData }>
                            <SortedList list={ this.state.names } />
                        </ProModeContext.Provider>
                    </div>
                </div>
            </div>
        )
    }
}
```

I don’t want to expose all the App component’s state data to the context consumers, so I have created a nested proContextData state object that has a proMode property. To apply the context, another custom HTML element is used, with the tag name of the context name, followed by a period, followed by Provider.

```
...
<ProModeContext.Provider value={ this.state.proContextData }>
    <SortedList list={ this.state.names } />
</ProModeContext.Provider>
...
```

The value property is used to provide the context with data values that override the defaults defined in Listing 14-28, which in this case is the proContextData state object.

■ **Tip** Use the version of the setState method that accepts a function if you need to update a nested state property, as shown in Listing 14-28. See Chapter 11 for details of the different ways that setState can be used.

The components that are defined between the start and end ProModeContext.Provider tags are able to access the state data directly by using the ProModeContext.Consumer element. In the example application, this means that the App component's proMode state data property is available directly in the ActionButton component without being threaded through the SortedList component, as illustrated in Figure 14-13.

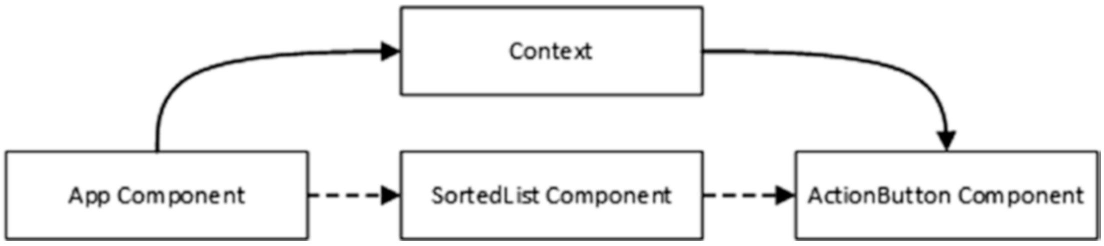


Figure 14-13. *The effect of using a context to distribute a state data property*

Changing Context Data Values in a Consumer

The data values in the context are read-only, but you can include a function in a context object that updates the source state data, creating the equivalent of a function prop. In Listing 14-31, I added a placeholder for the function that will be used if the provider applies the content without using the value property.

Listing 14-31. Adding a Function in the ProModeContext.js file in the src Folder

```
import React from "react";

export const ProModeContext = React.createContext({
    proMode: false,
    toggleProMode: () => {}
})
```

The function has an empty body and is used to prevent errors only if the default data object is received by a consumer. To demonstrate modifying a context data value, I am going to create a component that will render the checkbox used to toggle the pro mode. I added a file called `ProModeToggle.js` to the `src` folder and used it to define the component shown in Listing 14-32.

Listing 14-32. The Contents of the ProModeToggle.js File in the src Folder

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

export class ProModeToggle extends Component {

    render() {
        return <ProModeContext.Consumer>
            { contextData => (
                <div className="form-check">
                    <input type="checkbox" className="form-check-input"
                        value={ contextData.proMode }
                        onChange={ contextData.toggleProMode } />
                    <label className="form-check-label">
                        { this.props.label }
                    </label>
                </div>
                )
            }
        </ProModeContext.Consumer>
    }
}
```

This component is a context consumer and uses the `proMode` property to set the value of a checkbox and invokes the `toggleProMode` function when it changes. The component also uses a prop to set the content of a `label` element, just to show that a component that consumes a context is still able to receive props from its parent. In Listing 14-33, I have updated the `App` component so that it uses the `ProModeToggle` component and provides the context with a function.

■ **Caution** Avoid the temptation to create the object for a context in the provider's `render` method, which can be appealing because it avoids the need to create nested state objects and to assign methods to state properties. Creating a new object each time the `render` method is called undermines the change detection process that React applies to contexts and can lead to additional updates.

Listing 14-33. Expanding the Context in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { GeneralList } from './GeneralList';
import { SortedList } from "./SortedList";
import { ProModeContext } from './ProModeContext';
import { ProModeToggle } from './ProModeToggle';
```

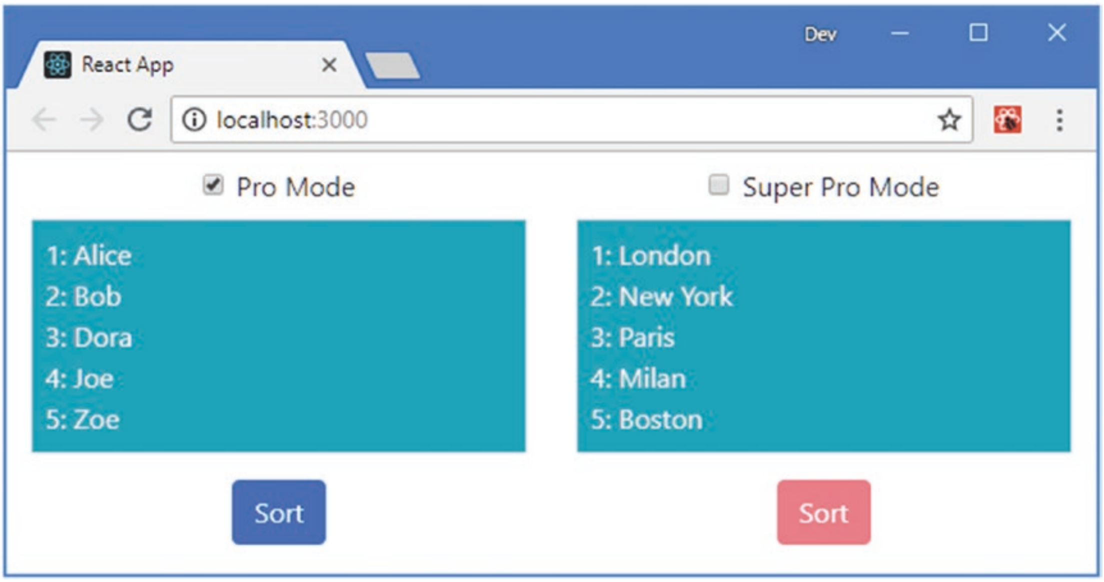
```
export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            names: ["Zoe", "Bob", "Alice", "Dora", "Joe"],
            cities: ["London", "New York", "Paris", "Milan", "Boston"],
            //proMode: false,
            proContextData: {
                proMode: false,
                toggleProMode: this.toggleProMode
            }
        }
    }

    toggleProMode = () => {
        this.setState(state =>  state.proContextData.proMode
            = !state.proContextData.proMode);
    }

    render() {
        return (
            <div className="container-fluid">
                <ProModeContext.Provider value={ this.state.proContextData }>
                    <div className="row">
                        <div className="col-12 text-center p-2">
                            <ProModeToggle label="Pro Mode" />
                        </div>
                    </div>
                    <div className="row">
                        <div className="col-6">
                            <GeneralList list={ this.state.names }
                                theme="primary" />
                        </div>
                        <div className="col-6">
                            <SortedList list={ this.state.names } />
                        </div>
                    </div>
                </ProModeContext.Provider>
            </div>
        )
    }
}
```

To provide an object that has both the state data and a function, I have added a property whose value is the toggleProMode method and that allows the context consumer to change the value of the state data property and, in doing so, trigger an update. Notice that I have lifted up the ProModeContext.Provider element so that the ProModeToggle and the SortedList component are both in scope. This is optional, and I could have given each child component its own context element, just as long as the value attributes used the same object. Using separate elements can be useful when you want to use multiple instances of the context with different groups of components, as shown in Listing 14-34.

Listing 14-34. Using Multiple Contexts in the App.js File in the src Folder

```
import React, { Component } from 'react';
//import { GeneralList } from './GeneralList';
import { SortedList } from "./SortedList";
import { ProModeContext } from './ProModeContext';
import { ProModeToggle } from './ProModeToggle';

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            names: ["Zoe", "Bob", "Alice", "Dora", "Joe"],
            cities: ["London", "New York", "Paris", "Milan", "Boston"],
            proContextData: {
                proMode: false,
                toggleProMode: this.toggleProMode
            },
            superProContextData: {
                proMode: false,
                toggleProMode: this.toggleSuperMode
            }
        }
    }

    toggleProMode = () => {
        this.setState(state =>  state.proContextData.proMode
            = !state.proContextData.proMode);
    }

    toggleSuperMode = () => {
        this.setState(state =>  state.superProContextData.proMode
            = !state.superProContextData.proMode);
    }

    render() {
        return  (
            <div className="container-fluid">
                <div className="row">
                    <div className="col-6 text-center p-2">
                        <ProModeContext.Provider value={ this.state.proContextData }>
                            <ProModeToggle label="Pro Mode" />
                        </ProModeContext.Provider>
                    </div>
                    <div className="col-6 text-center p-2">
                        <ProModeContext.Provider
                                value={ this.state.superProContextData }>
                            <ProModeToggle label="Super Pro Mode" />
                        </ProModeContext.Provider>
                    </div>
                </div>
                <div className="row">
                    <div className="col-6">
                        <ProModeContext.Provider value={ this.state.proContextData }>
                            <SortedList list={ this.state.names } />
                        </ProModeContext.Provider>
                    </div>
                    <div className="col-6">
                        <ProModeContext.Provider
                                value={ this.state.superProContextData }>
                            <SortedList list={ this.state.cities } />
                        </ProModeContext.Provider>
                    </div>
                </div>
            </div>
        )
    }
}
```

The App component uses different contexts to manage two pro levels, as shown in Figure 14-14. Each context has its own data object, and React keeps track of the providers and consumers for each one.



1: Alice
2: Bob
3: Dora
4: Joe
5: Zoe

1: London
2: New York
3: Paris
4: Milan
5: Boston

Figure 14-14. *Using multiple contexts*

Using the Simplified Context Consumer APIs

React offers an alternative means to access a context that can be easier to use than a render prop function, as shown in Listing 14-35.

Listing 14-35. Using the Simpler Context API in the ProModeToggle.js File in the src Folder

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

export class ProModeToggle extends Component {
    static contextType = ProModeContext;

    render() {
        return (
            <div className="form-check">
                <input type="checkbox" className="form-check-input"
                    value={ this.context.proMode }
                    onChange={ this.context.toggleProMode } />
                <label className="form-check-label">
                    { this.props.label }
                </label>
            </div>
        )
    }
}
```

A `static` property named `contextType` is assigned the context, which is then available throughout the component as `this.context`. This is a relatively recent addition to React, but it can be easier to use, with the caveat that a component can consume only one context.

Consuming a Context Using Hooks

The `useContext` hook provides the counterpart to the `contextType` property for functional components. In Listing 14-36, I have rewritten the `ProModeToggle` component to be defined as a function that relies on the `useContext` hook.

Listing 14-36. Using a Hook in the ProModeToggle.js File in the src Folder

```
import React, { useContext } from "react";
import { ProModeContext } from "./ProModeContext";

export function ProModeToggle(props) {

    const context = useContext(ProModeContext);

    return (
        <div className="form-check">
            <input type="checkbox" className="form-check-input"
                value={ context.proMode }
                onChange={ context.toggleProMode } />
            <label className="form-check-label">
                { props.label }
            </label>
        </div>
    )
}
```

The useContext function returns a context object through which the properties and functions can be accessed.

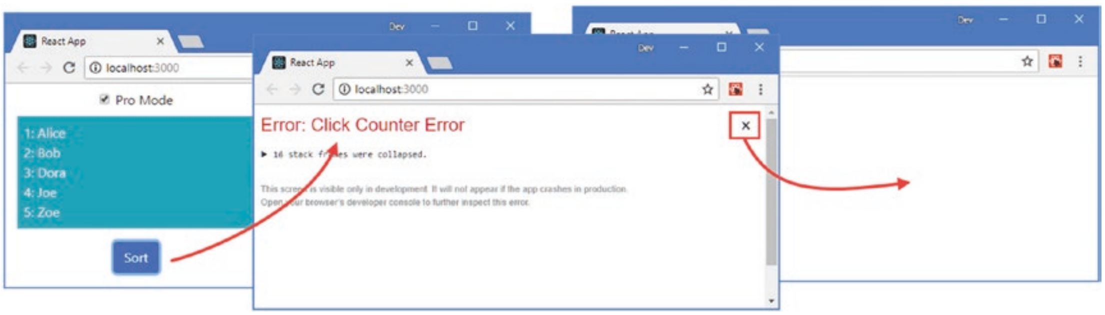
Defining Error Boundaries

When a component generates an error in its render method or in a lifecycle method, it propagates up the component hierarchy until it reaches the top of the application, at which point all of the application's components are unmounted. This means that any error can effectively terminate the application, which is rarely ideal, especially if the error is one that the application can recover from. To demonstrate the default error handling behavior, I changed the ActionButton component so it throws an error the second time the button element is clicked, as shown in Listing 14-37.

Listing 14-37. Throwing an Error in the ActionButton.js File in the src Folder

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

export class ActionButton extends Component {

    constructor(props) {
        super(props);
        this.state = {
            clickCount: 0
        }
    }

    handleClick = () => {
        this.setState({ clickCount: this.state.clickCount + 1});
        this.props.callback();
    }

    render() {
        return (
            <ProModeContext.Consumer>
                { contextData => {
                    if (this.state.clickCount > 1) {
                        throw new Error("Click Counter Error");
                    }
                    return <button
                        className={ this.getClasses(contextData.proMode)}
                        disabled={ !contextData.proMode }
                        onClick={ this.handleClick }>
                                { this.props.text }
                    </button>
                }}
            </ProModeContext.Consumer>
        )
    }
    getClasses(proMode) {
        let col = proMode ? this.props.theme : "danger";
        return `btn btn-${col} m-2`;
    }
}
```

To see the default behavior, enable one of the checkboxes and click the associated button. The order of the list will be changed when you click the first time. When you click again, the error will be thrown, and you will see the response shown in Figure 14-15. This message is shown during development but is disabled when the application is built for deployment, as demonstrated in Chapter 8. Click the close icon in the top right of the browser window, and you will see that all of the application's components have been unmounted, leaving an empty browser window.



Error: Click Counter Error

▶ 16 stack frames were collapsed.

This screen is visible only in development. It will not appear if the app crashes in production.
Open your browser's developer console to further inspect this error.

Figure 14-15. *The default error handling*

The browser's JavaScript console displays a stack trace for the error.

```
...
Uncaught Error: Click Counter Error
    at ActionButton.js:23
    at updateContextConsumer (react-dom.development.js:13799)
    at beginWork (react-dom.development.js:13987)
    at performUnitOfWork (react-dom.development.js:16249)
...
```

Creating the Error Boundary Component

Class-based components can implement the `componentDidCatch` lifecycle method, which is invoked when a child component throws an error. The React convention is to use dedicated error-handling components, known as *error boundaries*, that intercept errors and either recover the application so it can continue execution or display a message to the user to indicate the nature of the problem. I added a file called `ErrorBoundary.js` to the `src` folder and used it to define the error boundary shown in Listing 14-38.

■ **Caution** Error boundaries apply only to errors that are thrown in lifecycle methods and do not respond to errors thrown in event handlers. Error boundaries also cannot be used for asynchronous HTTP requests and a `try/catch` block must be used instead, as shown in Part 3.

Listing 14-38. The Contents of the ErrorBoundary.js File in the src Folder

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

export class ErrorBoundary extends Component {

    constructor(props) {
        super(props);
        this.state = {
            errorThrown: false
        }
    }

    componentDidCatch = (error, info) => this.setState({ errorThrown: true});

    render() {
        return (
            <React.Fragment>
                { this.state.errorThrown &&
                    <h3 className="bg-danger text-white text-center m-2 p-2">
                        Error Detected
                    </h3>
                }
                { this.props.children }
            </React.Fragment>
        )
    }
}
```

The `componentDidCatch` method receives the error object thrown by the problem component and an additional information object that provides the component's stack trace, which can be useful for logging.

When an error boundary is used, React will invoke the `componentDidCatch` method and then call the `render` method. The content rendered by the error boundary is handled using the mounting phase of the component lifecycle, as described in Chapter 13, so that new instances of all the components are created. This sequence allows the error boundary to change the content that is rendered to avoid problems or change the state of the application so that the error will not occur again. For this example, I have taken the third option, which is to render the same content but with a message noting that the error has been detected. This is an approach that can be used when the error has arisen because of a problem outside the scope of the application, such as when data cannot be obtained from a web service. Error boundaries are applied as container components, as shown in Listing 14-39.

Listing 14-39. Applying an Error Boundary in the SortedList.js File in the src Folder

```
import React, { Component } from "react";
import { GeneralList } from "./GeneralList";
import { ActionButton } from "./ActionButton";
import { ErrorBoundary } from "./ErrorBoundary";
```

```
export class SortedList extends Component {

    constructor(props) {
        super(props);
        this.state = {
            sort: false
        }
    }

    getList() {
        return this.state.sort
            ? [...this.props.list].sort() : this.props.list;
    }

    toggleSort = () => {
        this.setState({ sort : !this.state.sort });
    }

    render() {
        return (
            <div>
                <ErrorBoundary>
                    <GeneralList list={ this.getList() } theme="info" />
                    <div className="text-center m-2">
                        <ActionButton theme="primary" text="Sort"
                            proMode={ this.props.proMode }
                            callback={this.toggleSort} />
                    </div>
                </ErrorBoundary>
            </div>
        )
    }
}
```

The error boundary will handle errors thrown by any of the components it contains and any of their descendants. To see the effect, click one of the Sort buttons twice and close the error warning message to see the message indicating the error has been detected, as shown in Figure 14-16.

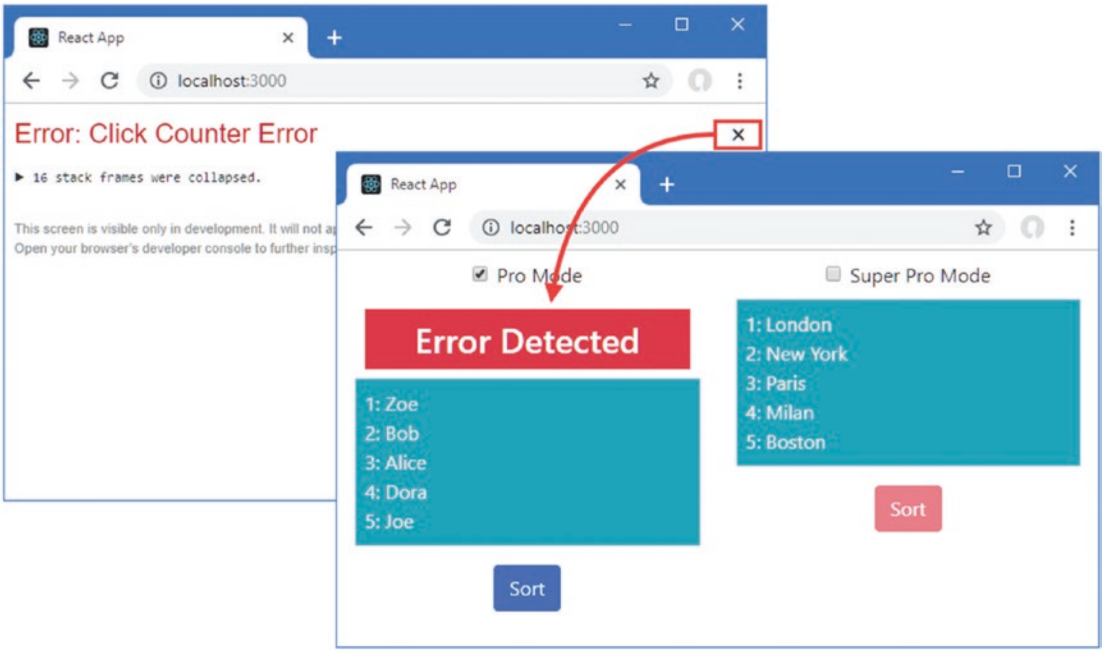

React App
localhost:3000
Error: Click Counter Error
▶ 16 stack frames were collapsed.
This screen is visible only in development. It will not a
Open your browser's developer console to further insp
Pro Mode
Super Pro Mode
Error Detected
1: Zoe
2: Bob
3: Alice
4: Dora
5: Joe
1: London
2: New York
3: Paris
4: Milan
5: Boston
Sort
Sort

Figure 14-16. *The effect of an error boundary*

Summary

In this chapter, I described the different ways that components can be combined to compose applications, including containers, higher-order components, and render props. I also showed you how contexts can be used to distribute global data and avoid prop threading and how error boundaries can be used to handle problems in component lifecycle methods. In the next chapter, I describe the features that React provides for working with forms.