CHAPTER 22

Advanced URL Routing

In this chapter, I describe the advanced features that are available for URL routing with the React-Router package. I show you how to create components that can participate in the routing process, how to navigate programmatically, how to generate routes programmatically, and how to use URL routing in components that are connected to the data store. Table 22-1 puts the advanced URL routing features in context.

Table 22-1. *Putting Advanced URL Routing in Context*

Question	Answer
What is it?	The advanced routing features provide programmatic access to the URL routing system.
Why is it useful?	These features allow components to be aware of the routing system and the currently active route.
How is it used?	Access to the advanced routing features is provided by props.
Are there any pitfalls or limitations?	These are advanced features, and care is required to ensure that they are properly integrated into components.
Are there any alternatives?	These are optional features. Applications can use the standard features described in Chapter 21 or avoid URL routing entirely.

Table 22-2 summarizes the chapter.

Table 22-2. *Chapter Summary*

Problem	Solution	Listing
Receive details of the routing system in a component	Use the props provided by the `Route` component or use the `withRouter` higher-order component	3, 4, 10–12, 19–23
Get details of the current navigation location	Use the `location` prop	5
Get URL segments from the current route	Add parameters to the URL	6–9
Navigate programmatically	Use the methods defined by the `history` prop	13, 14
Prompt the user before navigation	Use the `Prompt` component	15–17

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_22

Preparing for This Chapter

In this chapter, I continue using the productapp project from Chapter 21. To prepare for this chapter, change the router that is used by the application from HashRouter to BrowserRouter, so that the HTML5 History API is used for navigation, and simplified the Link and Router components, as shown in Listing 22-1.

■ **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 22-1. Changing Routers and Routes in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, NavLink, Route, Switch, Redirect }
    from "react-router-dom";
import { ProductDisplay } from "./ProductDisplay";
import { SupplierDisplay } from "./SupplierDisplay";

export class Selector extends Component {

    render() {
        return <Router>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active"
                            to="/products">Products</NavLink>
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active"
                            to="/suppliers">Suppliers</NavLink>
                    </div>
                    <div className="col">
                        <Switch>
                            <Route path="/products" component={ ProductDisplay} />
                            <Route path="/suppliers" component={ SupplierDisplay } />
                            <Redirect to="/products" />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

Open a command prompt, navigate to the productapp folder, and run the command shown in Listing 22-2 to start the development tools.

Listing 22-2. Starting the Development Tools

```
npm start
```

Once the application has been compiled, the development HTTP server will start and display the content shown in Figure 22-1.

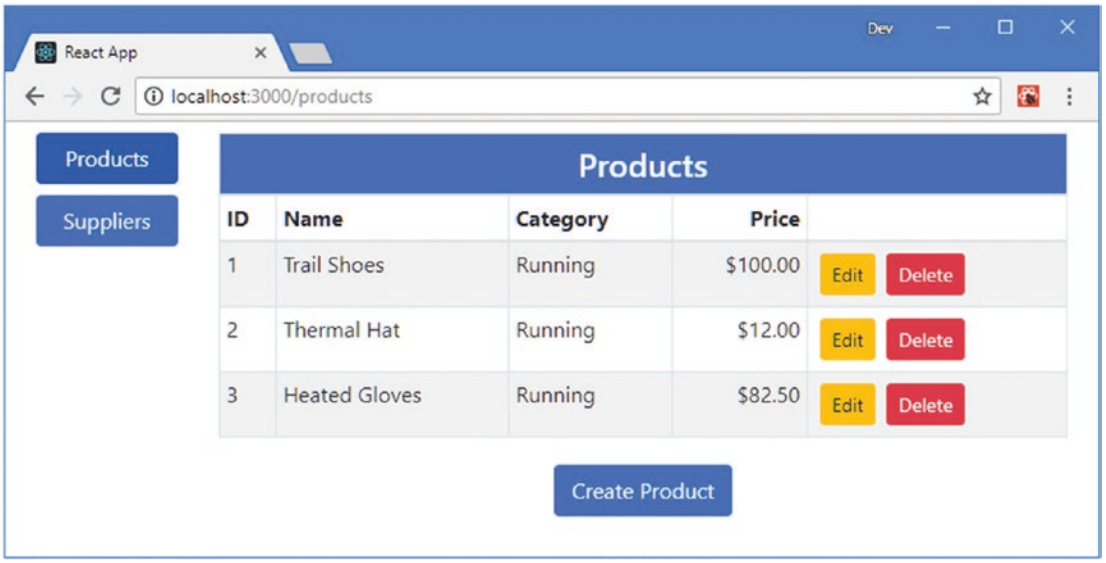

Figure 22-1. *Running the example application*

Creating Routing-Aware Components

When a `Route` displays a component, it provides it with context data that describes the current route and with access to an API that can be used for navigation, allowing components to be aware of the current location and to participate in routing. When the `component` prop is used, the `Route` passes the data and API to the component it displays as props, named `match`, `location`, and `history`. When the `render` prop is used, the render function is passed an object that has `match`, `location`, and `history` properties whose values are the same objects used as `render` props. The `match`, `location`, and `history` objects are described in Table 22-3.

Table 22-3. *The Props Provided by the Route Component*

Name	Description
`match`	This prop provides information about how the Route component matched the current browser URL.
`location`	This prop provides a representation of the current location and can be used for navigation instead of URLs expressed as strings.
`history`	This prop provides an API that can be used for navigation, as demonstrated in the "Navigating Programmatically" section.

Understanding the Match Prop

The match prop provides a component with details of how the parent Route matches the current URL. As I demonstrated in Chapter 21, a single Route can be used to match a range of URLs, and routing-aware components often need details about the current URL, which are available through the properties shown in Table 22-4.

Table 22-4. *The Match Prop Properties*

Name	Description
url	This property returns the URL that the Route has matched.
path	This property returns the path value used to match the URL.
params	This property returns the route params, which allow segments of a URL to be mapped to variables, as described in the "Using URL Parameters" section.
isExact	This property returns true if the route path exactly matches the URL.

To demonstrate the use of the routing props, I created the src/routing folder and added to it a file called RouteInfo.js with the component shown in Listing 22-3, which displays the values of the match prop's properties.

Listing 22-3. The Contents of the RouteInfo.js File in the src/routing Folder

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

export class RouteInfo extends Component {

    renderTable(title, prop, propertyNames) {
        return <React.Fragment>
            <tr><th colSpan="2" className="text-center">{ title }</th></tr>
            { propertyNames.map(p =>
                <tr key={p }>
                    <td>{ p }</td>
                    <td>{ JSON.stringify(prop[p]) }</td>
                </tr>)
            }
        </React.Fragment>
    }

    render() {
        return <div className="bg-info m-2 p-2">
            <h4 className="text-white text-center">Route Info</h4>
            <table className="table table-sm table-striped bg-light">
                <tbody>
                    { this.renderTable("Match", this.props.match,
                        ["url", "path", "params", "isExact"] )}
                </tbody>
            </table>
        </div>
    }
}
```

The RouteInfo component displays the url, path, params, and isExact properties of the match prop in a table and will allow me to easily add additional details from the other routing props later. The properties are serialized because the values are a mix of objects and Booleans that can cause display problems if used literally. In Listing 22-4, I have added a navigation link to the Selector component, along with a Route that displays the RouteInfo component.

Listing 22-4. Adding a Route in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, NavLink, Route, Switch, Redirect }
    from "react-router-dom";
import { ProductDisplay } from "./ProductDisplay";
import { SupplierDisplay } from "./SupplierDisplay";
import { RouteInfo } from "./routing/RouteInfo";

export class Selector extends Component {

    render() {
        return <Router>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active"
                            to="/products">Products</NavLink>
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active"
                            to="/suppliers">Suppliers</NavLink>
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active" to="/info">Route Info</NavLink>
                    </div>
                    <div className="col">
                        <Switch>
                            <Route path="/products" component={ ProductDisplay} />
                            <Route path="/suppliers" component={ SupplierDisplay } />
                            <Route path="/info" component={ RouteInfo } />
                            <Redirect to="/products" />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

Save the changes and click on the Route Info link and you will see the details of the match prop, as shown in Figure 22-2. The values displayed indicate that the path prop of the Route component is /info and it matches the /info URL which the new Link component targets. The information provided by the match prop will become more useful as I introduce more advanced routing features, especially when I introduce URL parameters in the "Using URL Parameters" section.

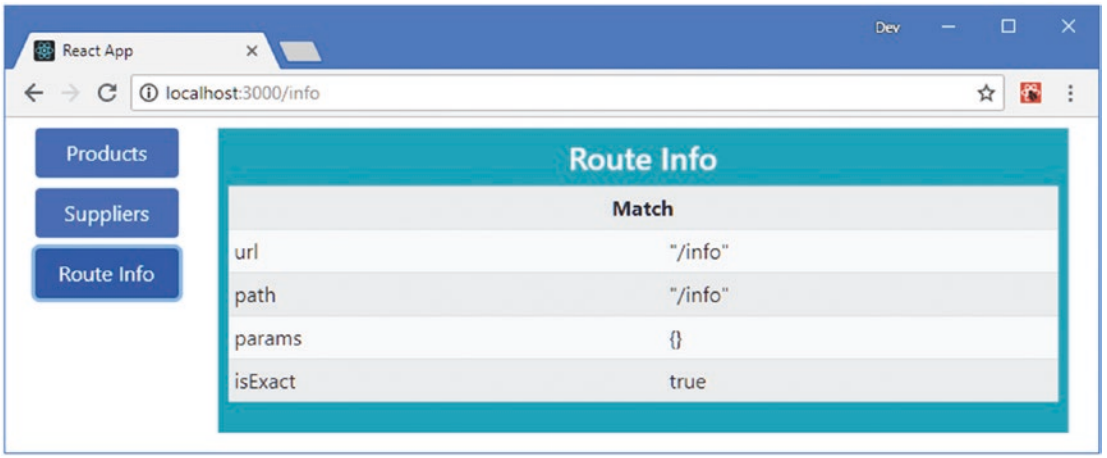

Figure 22-2. *Details provided by the match routing prop*

Understanding the Location Prop

The `location` object is used to describe a navigation location. The `location` object provided as props describes the current location and has the properties described in Table 22-5.

Table 22-5. *The Location Properties*

Name	Description
`key`	This property returns a key that identifies the location.
`pathname`	This property returns the path of the location.
`search`	This property returns the search term of the location URL (the part of the URL that follows the ? character).
`hash`	This property returns the URL fragment of the location URL (the part that follows the # character).
`state`	This property is used to associated arbitrary data with a location.

The `location` properties provide some overlap with the `match` prop, but the idea is that a component can retain a location object and use it to refer to a location instead of using strings as the value for the `to` prop of the `Link`, `NavLink`, and `Redirect` components. In Listing 22-5, I have added the location prop to the data displayed by the `RouteInfo`, along with a `Link` element that uses the `location` object for its navigation target.

Listing 22-5. Using the Location Prop in the RouteInfo.js File in the src/routing Folder

```
import React, { Component } from "react";
import { Link } from "react-router-dom";

export class RouteInfo extends Component {

    renderTable(title, prop, propertyNames) {
        return <React.Fragment>
            <tr><th colSpan="2" className="text-center">{ title }</th></tr>
            { propertyNames.map(p =>
                <tr key={p }>
                    <td>{ p }</td>
                    <td>{ JSON.stringify(prop[p]) }</td>
                </tr>)
            }
        </React.Fragment>
    }

    render() {
        return <div className="bg-info m-2 p-2">
            <h4 className="text-white text-center">Route Info</h4>
            <table className="table table-sm table-striped bg-light">
                <tbody>
                    { this.renderTable("Match", this.props.match,
                        ["url", "path", "params", "isExact"] )}
                    { this.renderTable("Location", this.props.location,
                        ["key", "pathname", "search", "hash", "state"] )}
                </tbody>
            </table>
            <div className="text-center m-2 bg-light">
                <Link className="btn btn-primary m-2"
                    to={ this.props.location }>Location</Link>
            </div>
        </div>
    }
}
```

Figure 22-3 shows the details of the location prop and the new `Link` component.

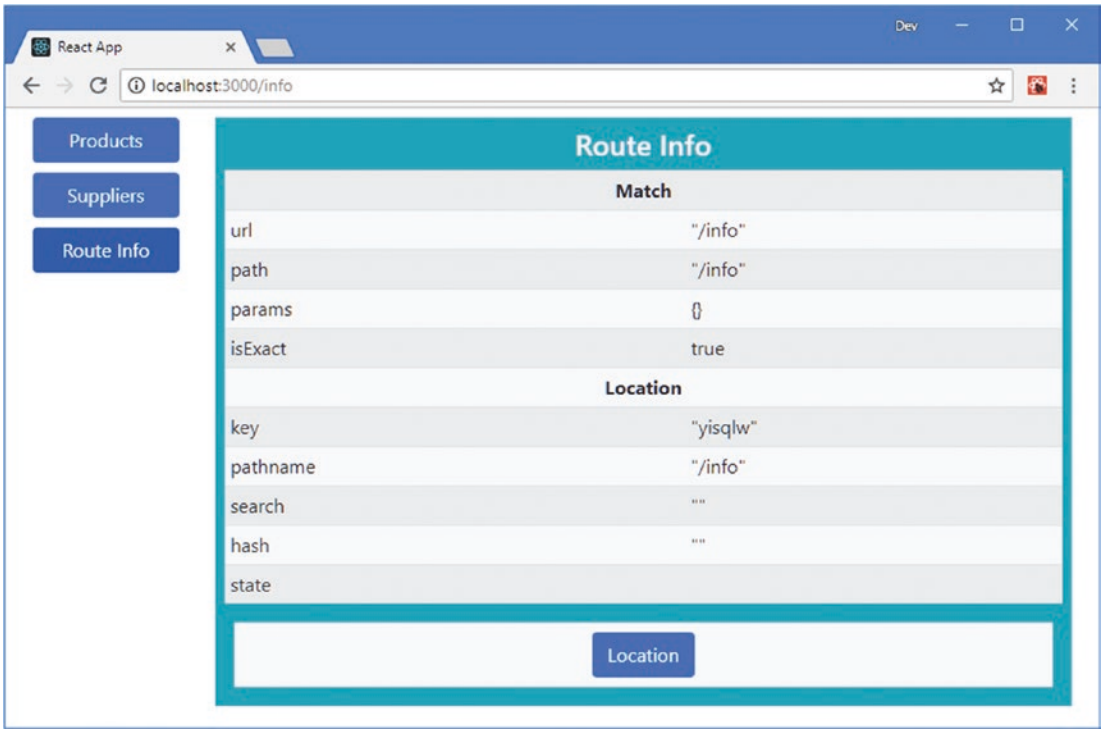

Figure 22-3. *Displaying details of the location routing prop*

Using the `location` prop as the value for a `Link` component's `to` prop isn't especially useful at the moment because it is only able to navigate to the current location. As you will see, components can be used to respond to multiple routes and may receive a series of locations over time, which makes using a `location` object both useful and more convenient than working with URLs expressed as strings.

Using URL Parameters

When a component is aware of the URL routing system, it will often need to adapt its behavior to the current URL. The React-Router package supports URL parameters, which assign the contents of a URL segment to a variable that can be read by components, allowing them to respond to the current location without having to parse the URL or understand its structure. Listing 22-6 shows the addition of a `Route` whose path includes a URL parameter and `Link` components that target it.

Listing 22-6. Defining a URL Parameter in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, NavLink, Route, Switch, Redirect }
    from "react-router-dom";
import { ProductDisplay } from "./ProductDisplay";
import { SupplierDisplay } from "./SupplierDisplay";
```

```
import { RouteInfo } from "./routing/RouteInfo";

export class Selector extends Component {

    render() {
        return <Router>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active"
                            to="/products">Products</NavLink>
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active"
                            to="/suppliers">Suppliers</NavLink>
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active"
                            to="/info/match">Match</NavLink>
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active"
                            to="/info/location">Location</NavLink>
                    </div>
                    <div className="col">
                        <Switch>
                            <Route path="/products" component={ ProductDisplay} />
                            <Route path="/suppliers" component={ SupplierDisplay } />
                            <Route path="/info/:datatype" component={ RouteInfo } />
                            <Redirect to="/products" />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

URL parameters are specified as `path` prop segments that start with a colon (the `:` character). In the example, the `Route` for the `RouteInfo` component has a `path` prop with a URL parameter named `datatype`.

```
...
<Route path="/info/:datatype" component={ RouteInfo } />
...
```

When the `Route` matches a URL, it will assign the value of the second segment to a URL parameter called `datatype`, which will be passed on to the `RouteInfo` component through the `match` prop's `params` property. If you click the navigation links added to the example in Listing 22-6, you will see different values displayed for the `params` property, as shown in Figure 22-4.

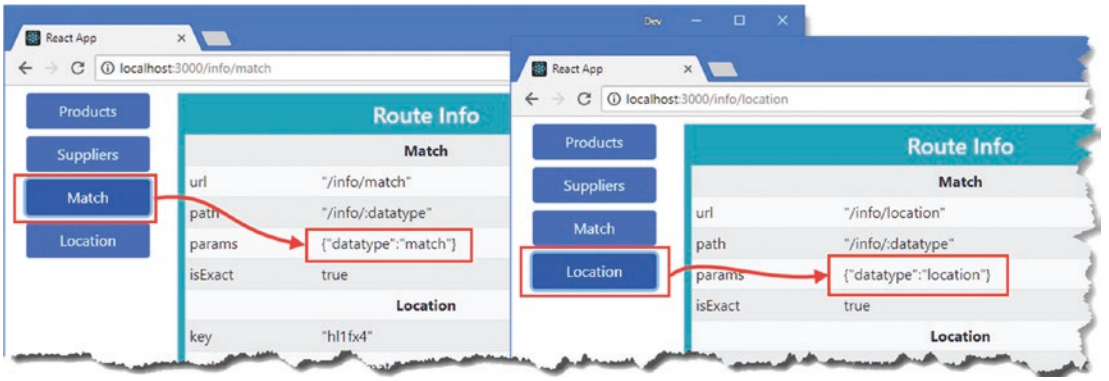

Figure 22-4. *Receiving URL parameters through the match prop*

When the URL is `/info/match`, the value of the `datatype` parameter is `match`. When the URL is `/info/location`, the value of the datatype parameter is `location`. In Listing 22-7, I have updated the `RouteInfo` component to use the `datatype` prop to select the context data to present to the user.

Listing 22-7. Using a URL Parameter Prop in the RouteInfo.js File in the src/routing Folder

```
import React, { Component } from "react";
import { Link } from "react-router-dom";

export class RouteInfo extends Component {

    renderTable(title, prop, propertyNames) {
        return <React.Fragment>
            <tr><th colSpan="2" className="text-center">{ title }</th></tr>
            { propertyNames.map(p =>
                <tr key={p }>
                    <td>{ p }</td>
                    <td>{ JSON.stringify(prop[p]) }</td>
                </tr>)
            }
        </React.Fragment>
    }

    render() {
        return <div className="bg-info m-2 p-2">
            <h4 className="text-white text-center">Route Info</h4>
            <table className="table table-sm table-striped bg-light">
                <tbody>
                    { this.props.match.params.datatype ==="match"
                        && this.renderTable("Match", this.props.match,
                            ["url", "path", "params", "isExact"] )}
                    { this.props.match.params.datatype === "location"
                        &&  this.renderTable("Location", this.props.location,
                            ["key", "pathname", "search", "hash", "state"] )}
                </tbody>
            </table>
```

```
            <div className="text-center m-2 bg-light">
                <Link className="btn btn-primary m-2"
                    to={ this.props.location }>Location</Link>
            </div>
        </div>
    }
}
```

The component receives URL parameters as part of the routing props and uses them just like any other prop. In the listing, the value of the `datatype` URL parameter is used for inline expressions that display the match or location object, as shown in Figure 22-5.

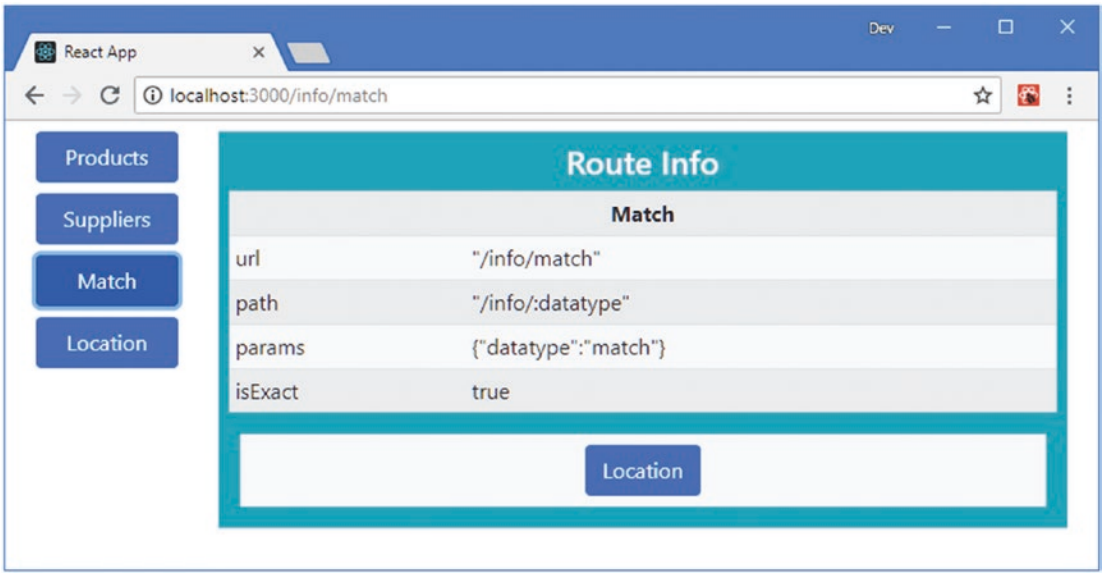

Figure 22-5. *Responding to a URL parameter by selecting content*

UNDERSTANDING OPAQUE URL STRUCTURE

URL parameters are not just a convenient way for a component to receive the contents of a URL segment. They also decouple the structure of the URL from the components that are targeted by it, allowing the structure of the URL to be altered or multiple URLs to target the same content without modifying the component. The component in Listing 22-7, for example, depends on the `datatype` URL parameter but doesn't have any dependency on the part of the URL from which it is obtained. This means that the component will work with a path such as `/info/:datatype` but will can also be matched by a path such as `/diagnostics/routing/:datatype` without requiring changes to the component's code.

The advantage of URL parameters is that the component just needs to know the names of the URL parameters it requires and not the details of where they appear in the URL.

Using Optional URL Parameters

The addition of the URL parameter means that the /info URL will no longer be matched by the Route component. I could solve this by adding another Route, but a more elegant approach is to use an optional parameter, which will allow the URL to match the path even if there is no corresponding segment. In Listing 22-8, I have added a NavLink that navigates to the /info URL and changed the path of the Route component so that the datatype parameter is optional.

Listing 22-8. Using an Optional URL Parameter in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, NavLink, Route, Switch, Redirect }
    from "react-router-dom";
import { ProductDisplay } from "./ProductDisplay";
import { SupplierDisplay } from "./SupplierDisplay";
import { RouteInfo } from "./routing/RouteInfo";

export class Selector extends Component {

    render() {
        return <Router>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active"
                            to="/products">Products</NavLink>
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active"
                            to="/suppliers">Suppliers</NavLink>
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active"
                            to="/info/match">Match</NavLink>
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active"
                            to="/info/location">Location</NavLink>
                        <NavLink className="m-2 btn btn-block btn-primary"
                            activeClassName="active" to="/info">All Info</NavLink>
                    </div>
                    <div className="col">
                        <Switch>
                            <Route path="/products" component={ ProductDisplay} />
                            <Route path="/suppliers" component={ SupplierDisplay } />
                            <Route path="/info/:datatype?" component={ RouteInfo } />
                            <Redirect to="/products" />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

Optional URL parameters are denoted with a question mark (the ? character) after the parameter name, so datatype? indicates an optional parameter that will be given the name datatype if there is a corresponding segment in the URL. If there is no segment, the path will still match, but there will be no datatype value. In Listing 22-9, I have updated the RouteInfo component so that it displays details of both the match and location objects if there is no datatype value.

■ **Tip** For a complete list of the different ways that URL parameters can be specified, see https://github.com/pillarjs/path-to-regexp, which is the GitHub repository for the package that processes URLs.

Listing 22-9. Handling an Optional URL Parameter in the RouteInfo.js File in the src Folder

```
import React, { Component } from "react";
import { Link } from "react-router-dom";

export class RouteInfo extends Component {

    renderTable(title, prop, propertyNames) {
        return <React.Fragment>
            <tr><th colSpan="2" className="text-center">{ title }</th></tr>
            { propertyNames.map(p =>
                <tr key={p }>
                    <td>{ p }</td>
                    <td>{ JSON.stringify(prop[p]) }</td>
                </tr>)
            }
        </React.Fragment>
    }

    render() {
        return <div className="bg-info m-2 p-2">
            <h4 className="text-white text-center">Route Info</h4>
            <table className="table table-sm table-striped bg-light">
                <tbody>
                    { (this.props.match.params.datatype === undefined ||
                            this.props.match.params.datatype ==="match")
                        && this.renderTable("Match", this.props.match,
                            ["url", "path", "params", "isExact"] )}
                    { (this.props.match.params.datatype === undefined ||
                            this.props.match.params.datatype === "location")
                        &&  this.renderTable("Location", this.props.location,
                            ["key", "pathname", "search", "hash", "state"] )}
                </tbody>
            </table>
            <div className="text-center m-2 bg-light">
                <Link className="btn btn-primary m-2"
                    to={ this.props.location }>Location</Link>
            </div>
        </div>
    }
}
```

The value of the datatype parameter will be undefined if no segment in the URL has been matched. The changes in the listing and the addition of the optional URL parameter allow the component to respond to a wider range of URLs without requiring additional Route components to be used.

Accessing Routing Data in Other Components

A Route will add props to the components it displays but can't provide them directly to other components, including the descendants of the components it displays. To avoid prop threading, the React-Router package provides two different approaches for providing access to routing data in descendant components, as described in the following sections.

Accessing Routing Data Directly in a Component

The most direct way to get access to routing data is to use a Route in the render method. To demonstrate, I added a file called ToggleLink.js to the src/routing folder and used it to define the component shown in Listing 22-10.

■ **Tip** This is the same component I used to highlight the active route in the SportsStore application in Part 1.

Listing 22-10. The Contents of the ToggleLink.js File in the src/routing Folder

```
import React, { Component } from "react";
import { Route, Link } from "react-router-dom";

export class ToggleLink extends Component {

    render() {
        return <Route path={ this.props.to } exact={ this.props.exact }
                children={ routeProps => {

            const baseClasses = this.props.className || "m-2 btn btn-block";
            const activeClass = this.props.activeClass || "btn-primary";
            const inActiveClass = this.props.inActiveClass || "btn-secondary"

            const combinedClasses =
                `${baseClasses} ${routeProps.match ? activeClass : inActiveClass}`

            return <Link to={ this.props.to } className={ combinedClasses }>
                        { this.props.children }
                    </Link>
        }} />
    }
}
```

The Route component's children prop is used to render content regardless of the current URL and is assigned a function that receives the routing context data. The path prop is used to indicate interest in a URL, and when the current URL matches the path, the routeProps object passed to the children function includes a match object that defines the properties described in Table 22-4.

The ToggleLink component allows me to solve a minor niggle that arises between the NavLink component and the Bootstrap CSS framework. The NavLink works by adding a class to the anchor element it renders when a path is matched and removing it the rest of the time. This causes a problem for some combinations of Bootstrap classes because the order in which they are defined in the CSS stylesheet means that some classes, such as btn-primary, won't take effect until a related class, such as btn-secondary, are removed.

The ToggleLink component fixes this problem by adding an active class when there is a match object and adding an inactive class when there is no match.

```
...
const combinedClasses =
    `${baseClasses} ${routeProps.match ? activeClass : inActiveClass}`
...
```

A Link is still used to generate the navigation element and respond to clicks but is styled by the ToggleLink component such that I make free use of the Bootstrap CSS classes. In Listing 22-11, I have replaced each NavLink with a ToggleLink.

Listing 22-11. Replacing Navigation Components in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch, Redirect }
    from "react-router-dom";
import { ProductDisplay } from "./ProductDisplay";
import { SupplierDisplay } from "./SupplierDisplay";
import { RouteInfo } from "./routing/RouteInfo";
import { ToggleLink } from "./routing/ToggleLink";

export class Selector extends Component {

    render() {
        return <Router>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        <ToggleLink to="/products">Products</ToggleLink>
                        <ToggleLink to="/suppliers">Suppliers</ToggleLink>
                        <ToggleLink to="/info/match">Match</ToggleLink>
                        <ToggleLink to="/info/location">Location</ToggleLink>
                        <ToggleLink to="/info" exact={ true }>All Info</ToggleLink>
                    </div>
                    <div className="col">
                        <Switch>
                            <Route path="/products" component={ ProductDisplay} />
                            <Route path="/suppliers" component={ SupplierDisplay } />
                            <Route path="/info/:datatype?" component={ RouteInfo } />
                            <Redirect to="/products" />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

I have relied on the default classes specified in Listing 22-10, with the result that the navigation buttons are added to the Bootstrap `btn-primary` class when they are active and to the `btn-secondary` class when they are inactive, as shown in Figure 22-6.

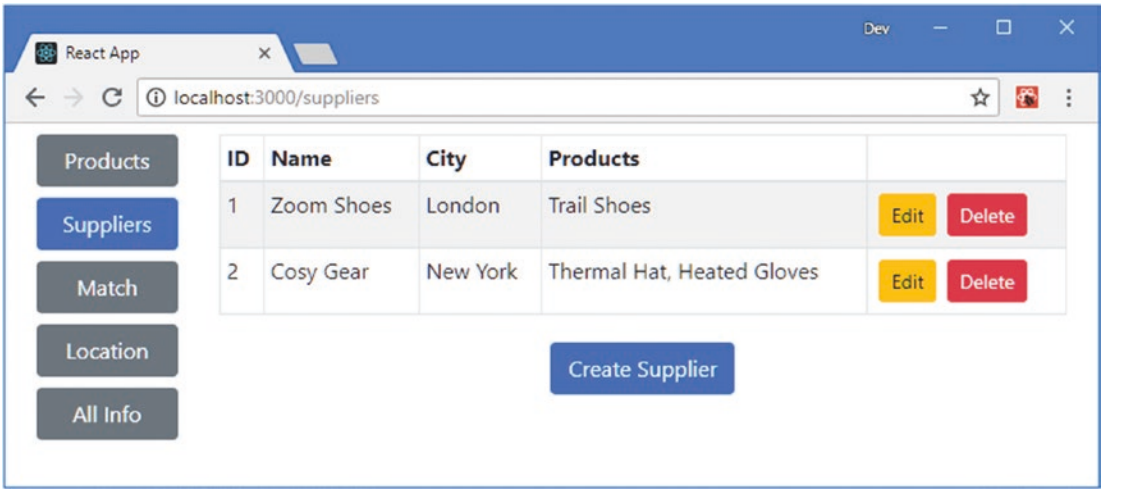

Figure 22-6. *Accessing routing data directly in a component*

Accessing Routing Data Using a Higher-Order Component

The `withRouter` function is a higher-order component that provides access to the routing system without directly using a `Route` (although that is the technique used inside the `withRouter` function). When a component is passed to `withRouter`, it receives the `match`, `location`, and `history` objects as props, just as though it had been rendered directly by a `Route` using the `component` prop. This can be a convenient alternative to writing components that render a `Route`. In Listing 22-12, I have used the `withRouter` function to allow the `RouteInfo` component to be used outside of a `Route`.

Listing 22-12. Creating a Routing HOC in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch, Redirect, withRouter }
    from "react-router-dom";
import { ProductDisplay } from "./ProductDisplay";
import { SupplierDisplay } from "./SupplierDisplay";
import { RouteInfo } from "./routing/RouteInfo";
import { ToggleLink } from "./routing/ToggleLink";

const RouteInfoHOC = withRouter(RouteInfo)

export class Selector extends Component {

    render() {
        return <Router>
            <div className="container-fluid">
```

```
                <div className="row">
                    <div className="col-2">
                        <ToggleLink to="/products">Products</ToggleLink>
                        <ToggleLink to="/suppliers">Suppliers</ToggleLink>
                        <ToggleLink to="/info/match">Match</ToggleLink>
                        <ToggleLink to="/info/location">Location</ToggleLink>
                        <ToggleLink to="/info" exact={ true }>All Info</ToggleLink>
                    </div>
                    <div className="col">
                        <RouteInfoHOC />
                        <Switch>
                            <Route path="/products" component={ ProductDisplay} />
                            <Route path="/suppliers" component={ SupplierDisplay } />
                            <Route path="/info/:datatype?" component={ RouteInfo } />
                            <Redirect to="/products" />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

The `withRouter` function is used to provide the `RouteInfo` component with the data it requires even when it is not displayed by a `Route`. The result is that details of the `match` and `location` objects are always displayed, as shown in Figure 22-7.

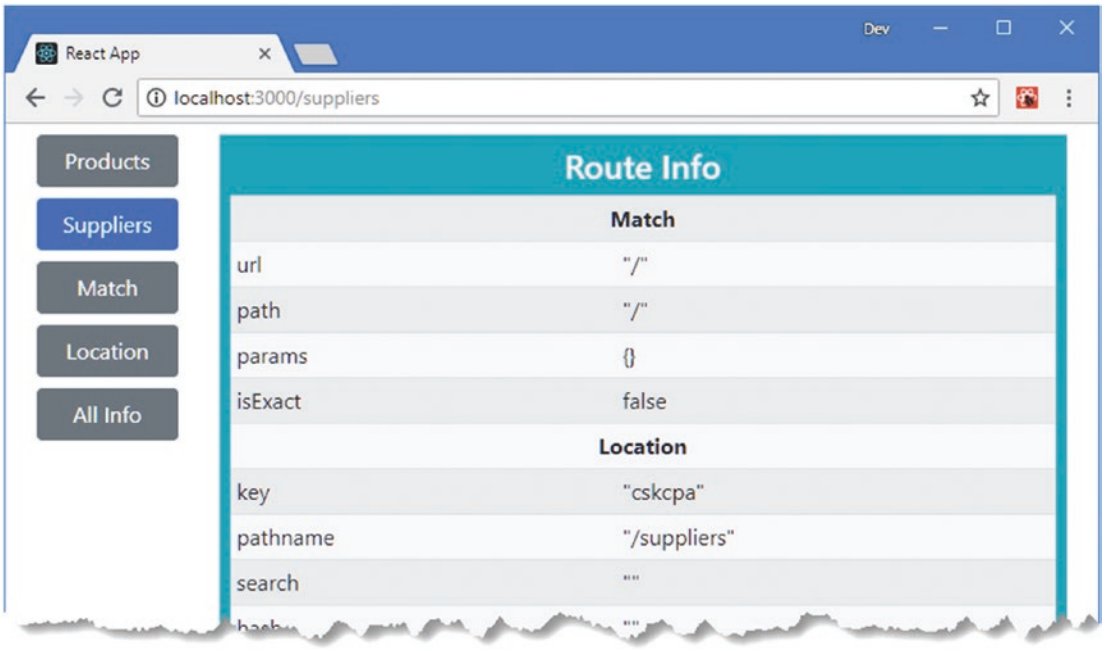

Figure 22-7. *Using the withrouter HOC*

The withRouter function doesn't provide support for matching paths, which means that the match object is of little use. The location object, however, provide details of the application's current location, and the history object can be used for programmatic navigation, as described in the next section.

Navigating Programmatically

Not all navigation can be handled using Link or NavLink components, especially where the application needs to perform some internal action in response to an event and only then perform navigation. The history object that is provided to components provides an API that allows programmatic access to the routing system, using the methods described in Table 22-6. The history object provides a consistent interface for navigation regardless of whether the application uses the HTML5 History API or URL fragments.

Table 22-6. *The history Methods*

Name	Description
push(path)	This method navigates to the specified path and adds a new entry in the browser's history. An optional state property can be provided that is available through the location.state property.
replace(path)	This method navigates to the specified path and replaces the current location in the browser's history. An optional state property can be provided that is available through the location.state property.
goBack()	This method navigates to the previous location in the browser's history.
goForward()	This method navigates to the next location in the browser's history.
go(n)	This method navigates to the history location n places from the current location. Use positive values to move forward and negative values to move backward.
block(prompt)	This method blocks navigation until the user responds to a prompt, as described in the "Prompting the User Before Navigation" section.

In Listing 22-13, I replaced the Link in the ToggleLink component with a button whose event handler navigates programmatically.

Listing 22-13. Navigating Programmatically in the ToggleLink.js File in the src/router Folder

```
import React, { Component } from "react";
import { Route } from "react-router-dom";

export class ToggleLink extends Component {

    handleClick = (history) => {
        history.push(this.props.to);
    }

    render() {
        return <Route path={ this.props.to } exact={ this.props.exact }
                children={ routeProps => {
```

```
            const baseClasses = this.props.className || "m-2 btn btn-block";
            const activeClass = this.props.activeClass || "btn-primary";
            const inActiveClass = this.props.inActiveClass || "btn-secondary"

            const combinedClasses =
                `${baseClasses} ${routeProps.match ? activeClass : inActiveClass}`

            return  <button className={ combinedClasses }
                        onClick={ () => this.handleClick(routeProps.history) }>
                        {this.props.children}
                    </button>
        }} />
    }
}
```

The `onClick` handler passes the `history` object received from the `Route` component to the `handleClick` method, which uses the `push` method to navigate to the location specified by the `to` prop. There is no visible difference because the anchor elements rendered by the `Link` components were already styled to appear as buttons, but the `ToggleLink` component now handles its navigation directly.

Navigating Programmatically Using Components

An alternative to using the `history` object is to render components that perform navigation. In Listing 22-14, I have changed the `ToggleLink` component so that clicking the `button` element updates state data that causes a `Redirect` to be rendered instead.

Listing 22-14. Navigating Using Components in the ToggleLink.js File in the src/router Folder

```
import React, { Component } from "react";
import { Route, Redirect } from "react-router-dom";

export class ToggleLink extends Component {

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

    handleClick = () => {
        this.setState({ doRedirect: true},
            () => this.setState({ doRedirect: false }));
    }

    render() {
        return <Route path={ this.props.to } exact={ this.props.exact }
                children={ routeProps => {

            const baseClasses = this.props.className || "m-2 btn btn-block";
            const activeClass = this.props.activeClass || "btn-primary";
```

```
            const inActiveClass = this.props.inActiveClass || "btn-secondary"

            const combinedClasses =
                `${baseClasses} ${routeProps.match ? activeClass : inActiveClass}`

            return  <React.Fragment>
                { this.state.doRedirect && <Redirect to={ this.props.to } /> }
                <button className={ combinedClasses } onClick={ this.handleClick }>
                    {this.props.children}
                </button>
            </React.Fragment>
        }} />
    }
}
```

Clicking the button sets the `doRedirect` property to `true`, which triggers an update that renders the `Redirect` component. The `doRedirect` property is set back to `false` automatically so that the component's normal content is rendered again. The result is the same as Listing 22-13, and choosing an approach is a matter of preference and personal style.

Prompting the User Before Navigation

Navigation can be delayed by rendering a `Prompt`, which allows the user to confirm or cancel navigation and which is often used to avoid accidentally abandoning form data. The `Prompt` component supports the props described in Table 22-7.

Table 22-7. *The Prompt Component Props*

Name	Description
`message`	This prop defines the message displayed to the user. It can be expressed as a string or as a function that accepts a `location` object and returns a string.
`when`	This prop will prompt the user only when its value evaluates to `true` and can be used to conditionally block navigation.

Only a single `Prompt` is used, but it doesn't matter where it is rendered because it doesn't perform any action until the application changes to a new location, at which point the user will be asked to confirm navigation. In Listing 22-15, I added a `Prompt` to the `Selector` component.

■ **Tip** Only one `Prompt` is needed, and you should not render additional `Prompt` instances in the components that perform navigation, such as the `ToggleLink` component in the example application. You will receive a warning in the JavaScript console if you render multiple `Prompt` components.

Listing 22-15. Prompting the User in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch, Redirect, withRouter, Prompt }
    from "react-router-dom";
import { ProductDisplay } from "./ProductDisplay";
import { SupplierDisplay } from "./SupplierDisplay";
import { RouteInfo } from "./routing/RouteInfo";
import { ToggleLink } from "./routing/ToggleLink";

const RouteInfoHOC = withRouter(RouteInfo)

export class Selector extends Component {

    render() {
        return <Router>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        <ToggleLink to="/products">Products</ToggleLink>
                        <ToggleLink to="/suppliers">Suppliers</ToggleLink>
                        <ToggleLink to="/info/match">Match</ToggleLink>
                        <ToggleLink to="/info/location">Location</ToggleLink>
                        <ToggleLink to="/info" exact={ true }>All Info</ToggleLink>
                    </div>
                    <div className="col">
                        <Prompt message={ loc =>
                            `Do you want to navigate to ${loc.pathname}`} />
                        <RouteInfoHOC />
                        <Switch>
                            <Route path="/products" component={ ProductDisplay} />
                            <Route path="/suppliers" component={ SupplierDisplay } />
                            <Route path="/info/:datatype?" component={ RouteInfo } />
                            <Redirect to="/products" />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

To see the effect of the `Prompt`, click one of the button elements rendered by the `ToggleLink` components. You will be asked to confirm navigation, as shown in Figure 22-8.

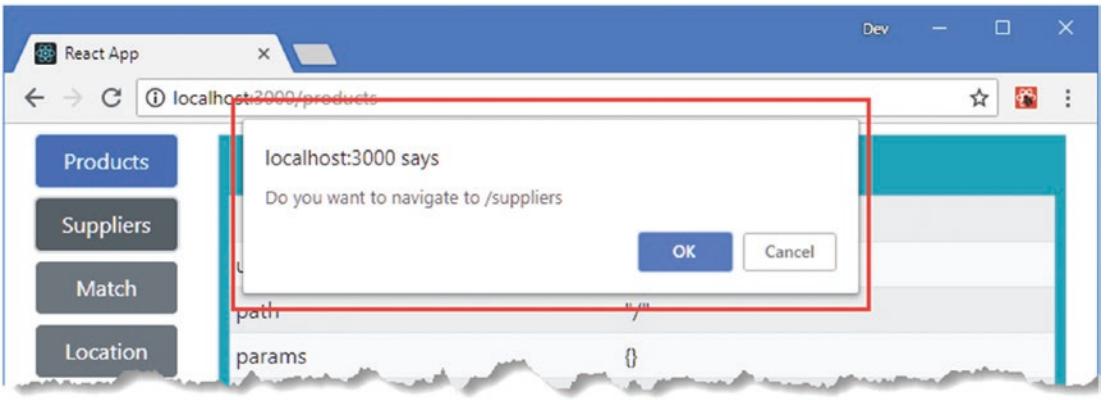

Figure 22-8. *Prompting the user before navigating*

■ **Tip** If you prefer using the `history` object for navigation, the `block` method can be used to set up a prompt that will be presented to the user, as demonstrated in the next section.

Presenting a Custom Navigation Prompt

The `BrowserRouter` and `HashRouter` components provide a `getUserConfirmation` prop that is used to replace the default prompt with a custom function. To present a prompt to the user that is inline with the rest of the application's content, I added a file called `CustomPrompt.js` to the `src/routing` folder and used it to define the component shown in Listing 22-16.

Listing 22-16. The Contents of the CustomPrompt.js File in the src/routing Folder

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

export class CustomPrompt extends Component {

    render() {
        if (this.props.show) {
            return <div className="alert alert-warning m-2 text-center">
                <h4 className="alert-heading">Navigation Warning</h4>
                    { this.props.message }
                <div className="p-1">
                    <button className="btn btn-primary m-1"
                        onClick={ () => this.props.callback(true) }>
                            Yes
                    </button>
                    <button className="btn btn-secondary m-1"
                        onClick={ () => this.props.callback(false )}>
                            No
                    </button>
```

```
                </div>
            </div>
        }
        return null;
    }
}
```

The CustomPrompt component is responsible for displaying a message to the user and presenting Yes and No buttons that invoke a callback function that will confirm or block navigation. In Listing 22-17, I have applied the CustomPrompt in the Selector component, along with the state data required to manage the prompting process.

Listing 22-17. Appling a Custom Prompt in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch, Redirect, withRouter, Prompt }
    from "react-router-dom";
import { ProductDisplay } from "./ProductDisplay";
import { SupplierDisplay } from "./SupplierDisplay";
import { RouteInfo } from "./routing/RouteInfo";
import { ToggleLink } from "./routing/ToggleLink";
import { CustomPrompt } from "./routing/CustomPrompt";

const RouteInfoHOC = withRouter(RouteInfo)

export class Selector extends Component {

    constructor(props) {
        super(props);
        this.state = {
            showPrompt: false,
            message: "",
            callback: () => {}
        }
    }

    customGetUserConfirmation = (message, navCallback) => {
        this.setState({
            showPrompt: true, message: message,
            callback: (allow) => { navCallback(allow);
                this.setState({ showPrompt: false}) }
        });
    }

    render() {
        return <Router getUserConfirmation={ this.customGetUserConfirmation }>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        <ToggleLink to="/products">Products</ToggleLink>
                        <ToggleLink to="/suppliers">Suppliers</ToggleLink>
                        <ToggleLink to="/info/match">Match</ToggleLink>
```

```
                    <ToggleLink to="/info/location">Location</ToggleLink>
                    <ToggleLink to="/info" exact={ true }>All Info</ToggleLink>
                </div>
                <div className="col">
                    <CustomPrompt show={ this.state.showPrompt }
                        message={ this.state.message }
                        callback={ this.state.callback } />
                    <Prompt message={ loc =>
                        `Do you want to navigate to ${loc.pathname}?` } />
                    <RouteInfoHOC />
                    <Switch>
                        <Route path="/products" component={ ProductDisplay} />
                        <Route path="/suppliers" component={ SupplierDisplay } />
                        <Route path="/info/:datatype?" component={ RouteInfo } />
                        <Redirect to="/products" />
                    </Switch>
                </div>
            </div>
        </div>
    </Router>
  }
}
```

The `getUserConfirmation` prop supported by `BrowserRouter` and `HashRouter` is assigned a function that receives a message to display to the user and a callback that is invoked by the user's decision: `true` to processed with navigation and false to `block` it. In the listing, the `getUserConfirmation` prop will invoke the `customGetUserConfirmation` method, which updates the state data used for the `CustomPrompt` props, with the result that the user is prompted, as shown in Figure 22-9.

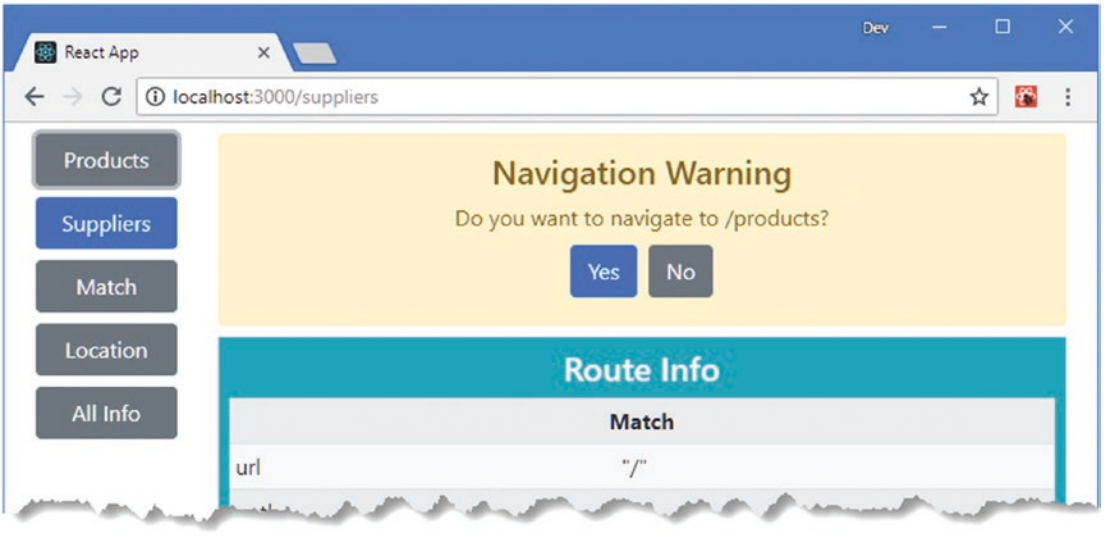

Figure 22-9. *Using a custom prompt*

■ **Tip** Notice that I still need to use a `Prompt`, which is responsible for triggering the process that displays the `CustomPrompt`.

Generating Routes Programmatically

The `Selector` component uses the `ToggleLink` and `Route` components to set up the mappings between the URLs that the application supports and the content they relate to, but this wasn't the way that the application worked before I added support for URL routing. Instead, the `App` component treated the `Selector` as a container and provided it with children to display, like this:

```
import React, { Component } from "react";
import { Provider } from "react-redux";
import dataStore from "./store";
import { Selector } from "./Selector";
import { ProductDisplay } from "./ProductDisplay";
import { SupplierDisplay } from "./SupplierDisplay";

export default class App extends Component {

    render() {
        return  <Provider store={ dataStore }>
                    <Selector>
                        <ProductDisplay name="Products" />
                        <SupplierDisplay name="Suppliers" />
                    </Selector>
                </Provider>
    }
}
```

The use of container components that provide services without hard-coded knowledge of their children is important in React development and can be easily applied when using React-Router because routes are defined and handled using components. In Listing 22-18, I have revised the `Selector` component to remove the locally defined routes and generate them from the `children` props instead.

Listing 22-18. Generating Routes from Children in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch, Redirect, Prompt }
    from "react-router-dom";
// import { ProductDisplay } from "./ProductDisplay";
// import { SupplierDisplay } from "./SupplierDisplay";
//import { RouteInfo } from "./routing/RouteInfo";
import { ToggleLink } from "./routing/ToggleLink";
import { CustomPrompt } from "./routing/CustomPrompt";

//const RouteInfoHOC = withRouter(RouteInfo)

export class Selector extends Component {
```

```
    constructor(props) {
        super(props);
        this.state = {
            showPrompt: false,
            message: "",
            callback: () => {}
        }
    }

    customGetUserConfirmation = (message, navCallback) => {
        this.setState({
            showPrompt: true, message: message,
            callback: (allow) => { navCallback(allow);
                this.setState({ showPrompt: false}) }
        });
    }

    render() {

        const routes = React.Children.map(this.props.children, child => ({
            component: child,
            name: child.props.name,
            url: `/${child.props.name.toLowerCase()}`
        }));

        return <Router getUserConfirmation={ this.customGetUserConfirmation }>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        { routes.map(r => <ToggleLink key={ r.url } to={ r.url }>
                                                { r.name }
                                            </ToggleLink>)}
                    </div>
                    <div className="col">
                        <CustomPrompt show={ this.state.showPrompt }
                            message={ this.state.message }
                            callback={ this.state.callback } />
                        <Prompt message={ loc =>
                            `Do you want to navigate to ${loc.pathname}?`} />
                        <Switch>
                            { routes.map( r => <Route key={ r.url } path={ r.url }
                                    render={ () => r.component } />)}
                            <Redirect to={ routes[0].url } />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

The `Selector` processes its children to build up the mappings between URLs and components and generates the required `ToggleLink` and `Route` components, which I have supplemented with a `Redirect` component, producing the result shown in Figure 22-10.

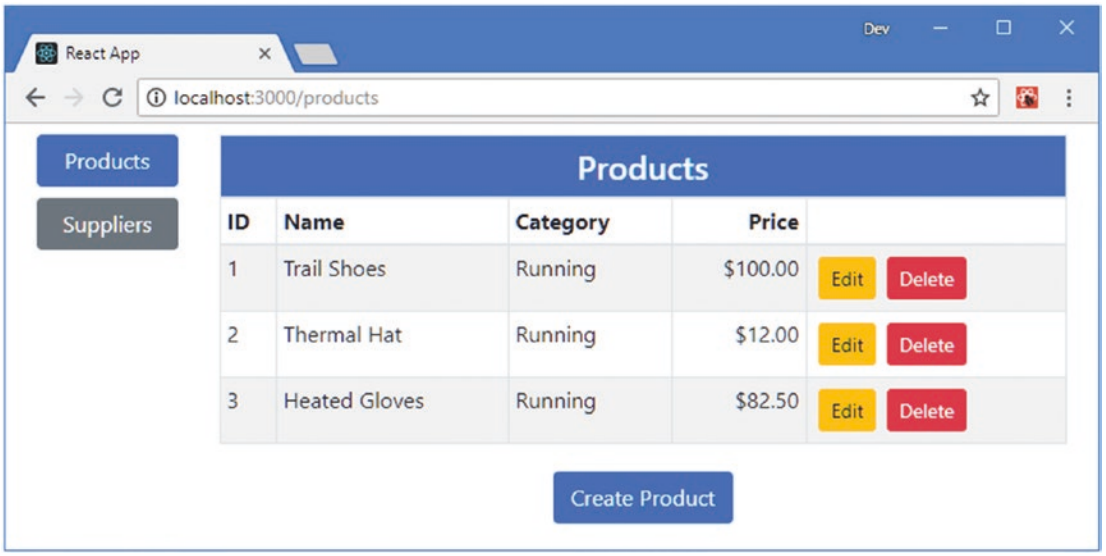

Figure 22-10. *Generating routes programmatically*

Using Routing with Connected Data Store Components

To complete the adoption of routing in the example application, I am going to move the remaining state data that coordinates components out of the data store and manage it with the set of URLs described in Table 22-8.

Table 22-8. *The URLs for the Example Application*

Name	Description
`/products/table`	This URL will display the table of products.
`/products/create`	This URL will display the editor to allow a new product to be created.
`/products/edit/4`	This URL will display the editor to allow an existing product to be edited, where the last URL segment identifies the product to change.
`/suppliers/table`	This URL will display the table of suppliers.
`/suppliers/create`	This URL will display the editor to allow a new supplier to be created.
`/suppliers/edit/4`	This URL will display the editor to allow an existing supplier to be edited, where the last URL segment identifies the supplier to change.

The URLs that the application requires can be handled with a single path with URL parameters, as follows:

```
...
/:datatype/:mode?/:id?
...
```

In the sections that follow, I will update the components in the application so that the data store is used only for the model data, while the details of which content should be displayed to the user is represented in the URL. (This kind of hard separation is only one approach, and you can take a softer line if it suits your project so that some state data is handled in the data store and some through URLs. As with so much in React development, there is no absolute correct approach.)

Replacing the Display Components

The `ProductDisplay` and `SupplierDisplay` components have been responsible for deciding whether the table or editor is displayed for a specific data type. The differences between these components have been reduced as features have been added to the example application, and the introduction of URL routing means that a single component can easily handle the content selection for both types of data. I added a file called `RoutedDisplay.js` to the `src/routing` folder and used it to define the component shown in Listing 22-19.

Listing 22-19. The Contents of the RoutedDisplay.js File in the src/routing Folder

```
import React, { Component } from "react";
import { ProductTable } from "../ProductTable"
import { ProductEditor } from "../ProductEditor";
import { EditorConnector } from "../store/EditorConnector";
import { PRODUCTS } from "../store/dataTypes";
import { TableConnector } from "../store/TableConnector";
import { Link } from "react-router-dom";
import { SupplierEditor } from "../SupplierEditor";
import { SupplierTable } from "../SupplierTable";

export const RoutedDisplay = (dataType) => {

    const ConnectedEditor = EditorConnector(dataType, dataType === PRODUCTS
        ? ProductEditor: SupplierEditor);
    const ConnectedTable = TableConnector(dataType, dataType === PRODUCTS
        ? ProductTable : SupplierTable);

    return class extends Component {
        render() {
            const modeParam = this.props.match.params.mode;
            if (modeParam === "edit" || modeParam === "create") {
                return <ConnectedEditor key={ this.props.match.params.id || -1 } />
            } else {
                return <div className="m-2">
                    <ConnectedTable />
                    <div className="text-center">
                        <Link to={`/${dataType}/create`}
                                className="btn btn-primary m-1">
                            Create
                        </Link>
                    </div>
                </div>
            }
        }
    }
}
```

This component performs the same task as the ProductDisplay and SupplierDisplay components but receives the data type it is responsible for as an argument, which allows the EditorConnector and TableConnector components to be created.

Updating the Connected Editor Component

The EditorConnector component is responsible for creating a ProductEditor or SupplierEditor that is connected to the Redux data store. In Listing 22-20, I have used the withRouter function to create a component that is provided with routing data but also remains connected to the data store.

Listing 22-20. Using Routing in the EditorConnector.js File in the src/store Folder

```
import { connect } from "react-redux";
//import { endEditing } from "./stateActions";
import { PRODUCTS, SUPPLIERS  } from "./dataTypes";
import { saveAndEndEditing } from "./multiActionCreators";
import { withRouter } from "react-router-dom";

export const EditorConnector = (dataType, presentationComponent) => {

    const mapStateToProps = (storeData, ownProps) => {
        const mode = ownProps.match.params.mode;
        const id = Number(ownProps.match.params.id);
        return {
            editing: mode === "edit" || mode === "create",
            product: (storeData.modelData[PRODUCTS].find(p => p.id === id)) || {},
            supplier:(storeData.modelData[SUPPLIERS].find(s => s.id === id)) || {}
        }
    }

    const mapDispatchToProps = {
        //cancelCallback: endEditing,
        saveCallback: (data) => saveAndEndEditing(data, dataType)
    }

    const mergeProps = (dataProps, functionProps, ownProps) => {
        let routedDispatchers = {
            cancelCallback: () => ownProps.history.push(`/${dataType}`),
            saveCallback: (data) => {
                functionProps.saveCallback(data);
                ownProps.history.push(`/${dataType}`);
            }
        }
        return Object.assign({}, dataProps, routedDispatchers, ownProps);
    }

    return withRouter(connect(mapStateToProps,
        mapDispatchToProps, mergeProps)(presentationComponent));
}
```

The component no longer uses the data store to work out whether the user is editing or creating an object and gets this information from the URL, along with the `id` value when an object is being edited.

■ **Tip** Notice that I use `Number` to parse the `id` URL parameter, which is presented as a string. I need the `id` value to be a number in order to locate objects.

I have used the ability to merge props, described in Chapter 20, to create wrappers around the data store action creator so that data is saved to the store and then the `history` object is used for navigation. The cancel action is no longer required and can be handled directly by navigating away from the current location.

AVOIDING BLOCKED UPDATES

The `withRouter` and `connect` functions both produce components that try to minimize updates using the `shouldComponentUpdate` method, which is described in Chapter 13. When the `withRouter` and `connect` functions are used together, the result can be a component that doesn't always update because the React-Router and React-Redux packages perform simple comparisons on props and don't realize that a change has occurred. To avoid this problem, simplify the props structure to allow changes to be more easily detected.

Updating the Connected Table Component

The same process must be performed on the component that connects the tables that display objects to the data store, as shown in Listing 22-21.

Listing 22-21. Using Routing in the TableConnector.js File in the src/store Folder

```
import { connect } from "react-redux";
//import { startEditingProduct, startEditingSupplier } from "./stateActions";
import { deleteProduct, deleteSupplier } from "./modelActionCreators";
import { PRODUCTS, SUPPLIERS } from "./dataTypes";
import { withRouter } from "react-router-dom";

export const TableConnector = (dataType, presentationComponent) => {

    const mapStateToProps = (storeData, ownProps) => {
        if (dataType === PRODUCTS) {
            return { products: storeData.modelData[PRODUCTS] };
        } else {
            return {
                suppliers: storeData.modelData[SUPPLIERS].map(supp => ({
                    ...supp,
                    products: supp.products.map(id =>
                        storeData.modelData[PRODUCTS]
                            .find(p => p.id === Number(id)) || id)
                            .map(val => val.name || val)
                    }))
```

```
            }
        }
    }

    const mapDispatchToProps = (dispatch, ownProps) => {
        if (dataType === PRODUCTS) {
            return {
                //editCallback: (...args) => dispatch(startEditingProduct(...args)),
                deleteCallback: (...args) => dispatch(deleteProduct(...args))
            }
        } else {
            return {
                //editCallback: (...args) => dispatch(startEditingSupplier(...args)),
                deleteCallback: (...args) => dispatch(deleteSupplier(...args))
            }
        }
    }

    const mergeProps = (dataProps, functionProps, ownProps) => {
        let routedDispatchers = {
            editCallback: (target) => {
                ownProps.history.push(`/${dataType}/edit/${target.id}`);
            },
            deleteCallback: functionProps.deleteCallback
        }
        return Object.assign({}, dataProps, routedDispatchers, ownProps);
    }

    return withRouter(connect(mapStateToProps,
        mapDispatchToProps, mergeProps)(presentationComponent));
}
```

Once again, I have used the `withRouter` and `connect` functions to produce a component that has access to the routing data and the data store. The editing function is handled by navigating to a URL that indicates the data type and `id` value. Deleting data is a task handled entirely by the data store and requires no navigation.

Completing the Routing Configuration

The final step is to update the routing configuration to support the URLs defined in Table 22-8. In Listing 22-22, I updated the `Selector` component so that it applies the `RoutedDisplay` component in its `render` function. (I also removed the navigation prompt components and code for brevity.)

Listing 22-22. Changing the Routing Configuration in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch, Redirect }
    from "react-router-dom";
import { ToggleLink } from "./routing/ToggleLink";
//import { CustomPrompt } from "./routing/CustomPrompt";
import { RoutedDisplay } from "./routing/RoutedDisplay";
```

```
export class Selector extends Component {

    render() {

        const routes = React.Children.map(this.props.children, child => ({
            component: child,
            name: child.props.name,
            url: `/${child.props.name.toLowerCase()}`,
            datatype: child.props.datatype
        }));

        return <Router getUserConfirmation={ this.customGetUserConfirmation }>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        { routes.map(r => <ToggleLink key={ r.url } to={ r.url }>
                                                { r.name }
                                            </ToggleLink>)}
                    </div>
                    <div className="col">
                        <Switch>
                            { routes.map(r =>
                               <Route key={ r.url }
                                   path={ `/:datatype(${r.datatype})/:mode?/:id?` }
                                   component={ RoutedDisplay(r.datatype)} />
                            )}
                            <Redirect to={ routes[0].url } />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

The children provided by the parent component are no longer components and exist only to provide prop values to the Selector so that it can set up the Route components. In Listing 22-23, I have reflected this change in the App component, which now uses a custom HTML element to configure the Selector rather than using the data-specific components directly.

Listing 22-23. Completing the Routing Configuration in the App.js File in the src Folder

```
import React, { Component } from "react";
import { Provider } from "react-redux";
import dataStore from "./store";
import { Selector } from "./Selector";
// import { ProductDisplay } from "./ProductDisplay";
// import { SupplierDisplay } from "./SupplierDisplay";
import { PRODUCTS, SUPPLIERS } from "./store/dataTypes";

export default class App extends Component {
```

```
    render() {
        return  <Provider store={ dataStore }>
                    <Selector>
                        <data name="Products" datatype={ PRODUCTS } />
                        <data name="Suppliers" datatype ={ SUPPLIERS } />
                    </Selector>
                </Provider>
    }
}
```

The result is that the data store is no longer used for coordination between components, which is now handled entirely through the URL, as shown in Figure 22-11.

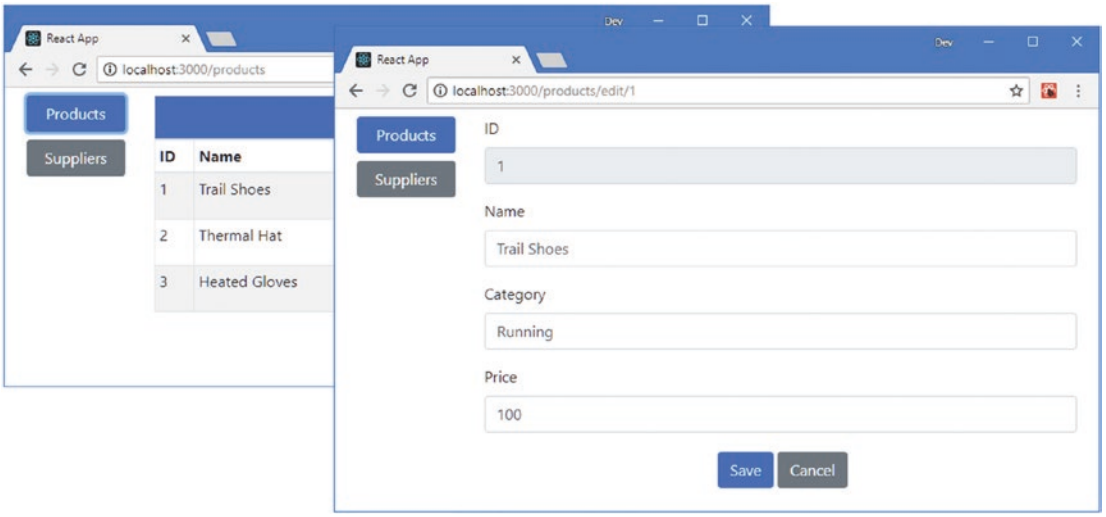

Figure 22-11. *Using URL routing to coordinate components*

Summary

In this chapter, I showed you how to use the advanced features provided by the React-Router package. I demonstrated how to create components that are aware of the routing system, how to use URL parameters to provide components with easy access to data from the current route, and how to use the routing features programmatically. I also demonstrated how components can participate in the routing system while also being connected to Redux, allowing state data to be handled via URLs while the application's model data is managed by a data store. In the next chapter, I show you how to consume a RESTful web service.