12

MULTIPAGE APPS WITH REACT ROUTER

LEARNING OBJECTIVES

By the end of this chapter, you will be able to do the following:

- Build multipage single-page applications (and understand why this is not an oxymoron).
- Use the React Router package to load different React components for different URL paths.
- Create static and dynamic routes (and understand what routes are in the first place).
- Navigate the website user via both links and programmatic commands.
- Build nested page layouts.

INTRODUCTION

Having worked through the first eleven chapters of this book, you should now know how to build React components and web apps, as well as how to manage components and app-wide state, and how to share data between components (via props or context).

But even though you know how to compose a React website from multiple components, all these components are on the same single website page. Sure, you can display components and content conditionally, but users will never switch to a different page. This means that the URL path will never change; users will always stay on **`your-domain.com`**. Also, at this point in time, your React apps don't support any paths such as **`your-domain.com/products`** or **`your-domain.com/blog/latest`**.

> **NOTE**
>
> **Uniform Resource Locators** (**URLs**) are references to web resources. For example, https://academind.com/courses is a URL that points to a specific page of the author's website. In this example, **`academind.com`** is the **domain name** of the website and **`/courses`** is the **path** to a specific website page.

For React apps, it might make sense that the path of the loaded website never changes. After all, in *Chapter 1*, *React – What and Why*, you learned that you build **single-page applications** (**SPAs**) with React.

But even though it might make sense, it's also quite a serious limitation.

ONE PAGE IS NOT ENOUGH

Having just a single page means that complex websites that would typically consist of multiple pages (e.g., an online shop with pages for products, orders, and more) become quite difficult to build with React. Without multiple pages, you have to fall back to state and conditional values to display different content on the screen.

But without changing URL paths, your website visitors can't share links to anything but the starting page of your website. Also, any conditionally loaded content will be lost when a new visitor visits that starting page. That will also be the case if users simply reload the page they're currently on. A reload fetches a new version of the page, and so any state (and therefore user interface) changes are lost.

For these reasons, you absolutely need a way of including multiple pages (with different URL paths) in a single React app for most React websites. Thanks to modern browser features and a highly popular third-party package, that is indeed possible (and the default for most React apps).

Via the **React Router** package, your React app can listen to URL path changes and display different components for different paths. For example, you could define the following path-component mappings:

- **`<domain>/`** => **`<Home />`** component is loaded.
- **`<domain>/products`** => **`<ProductList />`** component is loaded.
- **`<domain>/products/p1`** => **`<ProductDetail />`** component is loaded.
- **`<domain>/about`** => **`<AboutUs />`** component is loaded.
- Technically, it's still an SPA because there's still only one HTML page being sent to website users. But in that single-page React app, different components are rendered conditionally by the React Router package based on the specific URL paths that are being visited. As the developer of the app, you don't have to manually manage this kind of state or render content conditionally. In addition, your website is able to handle different URL paths, and therefore, individual pages can be shared or reloaded.

GETTING STARTED WITH REACT ROUTER AND DEFINING ROUTES

React Router is a third-party React library that can be installed in any React project. Once installed, you can use various components in your code to enable the aforementioned features.

Inside your React project, the package is installed via this command:

```
npm install react-router-dom
```

Once installed, you can import and use various components (and Hooks) from that library.

To start supporting multiple pages in your React app, you need to set up **routing** by going through the following steps:

1. Create different components for your different pages (e.g., **`Dashboard`** and **`Orders`** components).
2. Use the **`BrowserRouter`**, **`Routes`**, and **`Route`** components from the React Router library to enable routing and define the **routes** that should be supported by the React app.

In this context, the term **routing** refers to the React app being able to load different components for different URL paths (e.g., different components for the **`/`** and **`/orders`** paths). A route is a definition that's added to the React app that defines the URL path for which a predefined JSX snippet should be rendered (e.g., the **`Orders`** component should be loaded for the **`/orders`** path).

In an example React app that contains **`Dashboard`** and **`Orders`** components, and wherein the React Router library was installed via **`npm install`**, you can enable routing and navigation between these two components by editing the **`App`** component (in **`src/App.js`**) like this:

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';

import Dashboard from './routes/Dashboard';
import Orders from './routes/Orders';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Dashboard />} />
        <Route path="/orders" element={<Orders />} />
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

> **NOTE**
>
> You can find the complete example code on GitHub at https://packt.link/uX1mb.

In the preceding code snippet, three components are used from the **`react-router-dom`** package:

- **`BrowserRouter`**, which enables all routing-related features (e.g., it sets up a listener that detects and analyzes URL changes). This component must only be used once in the entire application (typically in your root component, such as the **`App`** component).
- **`Routes`**, which contains your route definitions. You can use this component multiple times to define multiple, independent groups of routes. Whenever the URL path changes, **`Routes`** checks whether any of its route definitions (via **`Route`**) matches that URL path and should therefore be activated.
- **`Route`**, which is used to define an individual route. The **`path`** prop is used to define the URL path that should activate this route. Only one route can be active per **`Routes`** group. Once activated, the content defined via the **`elements`** prop is rendered.

The placement of the **`Routes`** and **`Route`** components also defines where the conditional page content should be displayed. You can think of the **`<Route />`** element being replaced with the content defined via the **`element`** prop once the route becomes active. Therefore, the positioning of the **`Route`** components matter.

If you run the provided example React app (via **`npm start`**), you'll see the following output on the screen:

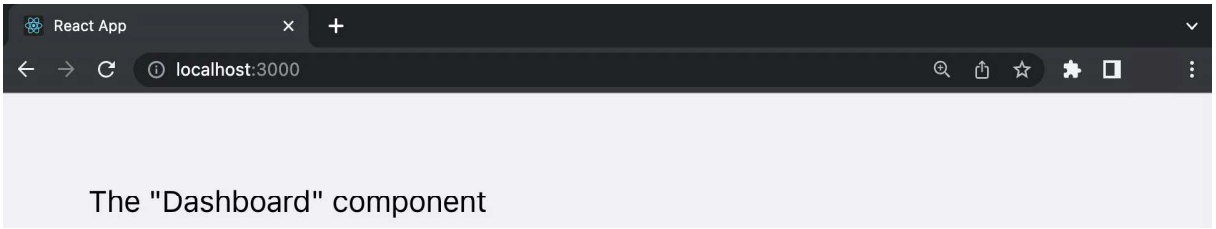

Figure 12.1: The Dashboard component content is loaded for localhost:3000

The content of the **Dashboard** component is displayed on the screen. Please note that this content is not defined in the **App** component (in the code snippet shared previously). Instead, there, only two route definitions were added: one for the **/** path (i.e., for **localhost.3000/**) and one for the **/orders** path (**localhost:3000/orders**).

> **NOTE**
>
> **localhost** is a local address that's typically used for development. When you deploy your React app (i.e., you upload it to a web server), you will receive a different domain—or assign a custom domain. Either way, it will not be **localhost** after deployment.
>
> The part after **localhost** (**:3000**) defines the network port to which the request will be sent. Without the additional port information, ports **80** or **443** (as the default HTTP(S) ports) are used automatically. During development, however, these are not the ports you want. Instead, you would typically use ports such as **3000**, **8000**, or **8080** as these are normally unoccupied by any other system processes and hence can be used safely. Projects created via **create-react-app** use port **3000**.

Since **localhost:3000** is loaded by default (when running **npm start**), the first route definition (**<Route path="/" element={<Dashboard />} />**) becomes active. This route is active because its path (**"/"**) matches the path of **localhost:3000** (since this is the same as **localhost:3000/**).

As a result, the JSX code defined via **element** is rendered on the screen. In this case, this means that the content of the **Dashboard** component is displayed because the **element** prop value of this route definition is **<Dashboard />**. It is quite common to use single components (such as **<Dashboard />**, in this example), but you could render any JSX content.

In the preceding example, no complex page is displayed. Instead, only some text shows up on the screen. This will change later in this chapter, though.

But it gets interesting if you manually change the URL from just **localhost:3000** to **localhost:3000/orders** in the browser address bar. In any of the previous chapters, this would not have changed the page content. But now, with routing enabled and the appropriate routes being defined, the page content does change, as shown:

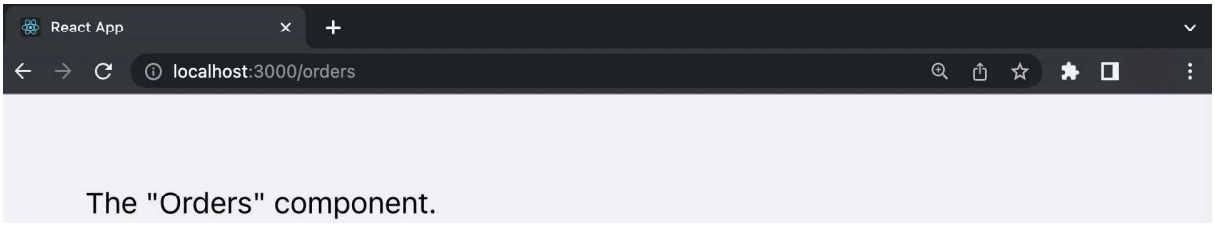

Figure 12.2: For /orders, the content of the Orders component is displayed

Once the URL changes, the content of the **`Orders`** component is displayed on the screen. It's again just some basic text in this first example, but it shows that different code is rendered for different URL paths.

However, this basic example has a major flaw (besides the quite boring page content). Right now, users must enter URLs manually. But, of course, that's not how you typically use websites.

ADDING PAGE NAVIGATION

To allow users to switch between different website pages without editing the browser address bar manually, websites normally contain links. A link is typically added via the **`<a>`** HTML element (the anchor element), like this:

```
<a href="/orders">All Orders</a>
```

For the previous example, on-page navigation could therefore be added by modifying the **`App`** component code like this:

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';

import Dashboard from './routes/Dashboard';
import Orders from './routes/Orders';

function App() {
  return (
    <BrowserRouter>
    <nav>
      <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/orders">All Orders</a></li>
  </ul>
    </nav>
      <Routes>
        <Route path="/" element={<Dashboard />} />
```

```
        <Route path="/orders" element={<Orders />} />
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

The **<nav>**, **<ul>**, and **<li>** elements are optional. It's just an approach for structuring the main navigation of a page in a semantically correct way. But with the two **<a>** elements added, website visitors see two navigation options that they can click to switch between pages:

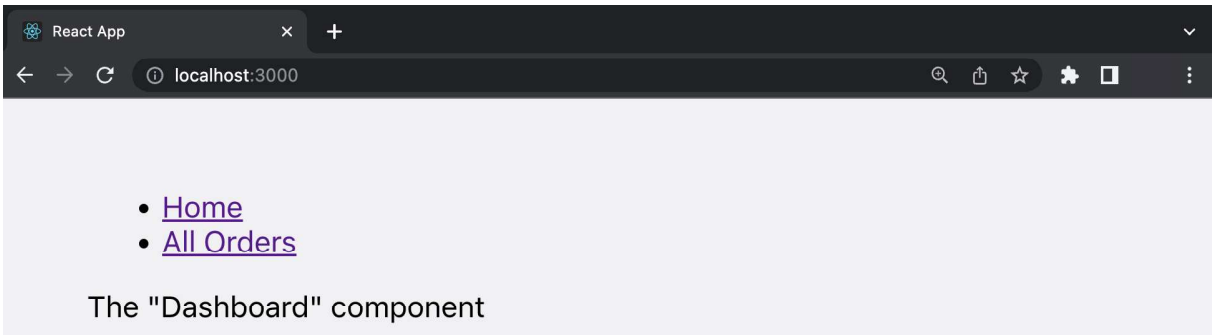

Figure 12.3: Two navigation links are presented to the user

Of course, you would typically add more CSS styling, but these two links allow users to navigate to either the **Dashboard** or **Orders** page (i.e., render the respective component).

> **NOTE**
>
> The code for this example can be found at https://packt.link/8H5yI.

This approach works but has a major flaw: the website is reloaded every time a user clicks one of those links. You can tell that it's reloaded because the browser's refresh icon changes to a cross (briefly) whenever you click a link.

This happens because the browser sends a new HTTP request to the server whenever a link is clicked. Even though the server always returns the same single HTML page, the page is reloaded during that process (because of the new HTTP request that was sent).

While that's not a problem on this simple demo page, it would be an issue if you had a state (e.g., an app-wide state managed via context) that should not be reset during a page change.

The following, slightly adjusted, example app illustrates this problem:

```
function App() {
  const [counter, setCounter] = useState(0);

  function incCounterHandler() {
    setCounter((prevCounter) => prevCounter + 1);
  }

  return (
    <BrowserRouter>
      <nav>
        <ul>
          <li>
            <a href="/">Home</a>
          </li>
          <li>
            <a href="/orders">All Orders</a>
          </li>
        </ul>
      </nav>
      <p>
        {counter} - <button onClick={incCounterHandler}>Increase</button>
      </p>
      <Routes>
        <Route path="/" element={<Dashboard />} />
        <Route path="/orders" element={<Orders />} />
      </Routes>
    </BrowserRouter>
  );
}
```

> **NOTE**
>
> The code for this example can be found at https://packt.link/ZZgPu.

In this example, a simple counter was added to the **`App`** component. Since the two route definitions (the **`<Route />`** elements) are inside the **`App`** component, that component should not be replaced when a user visits a different page.

At least, that's the theory. As you can see in the following screenshot, the **`counter`** state is lost whenever any link is clicked:

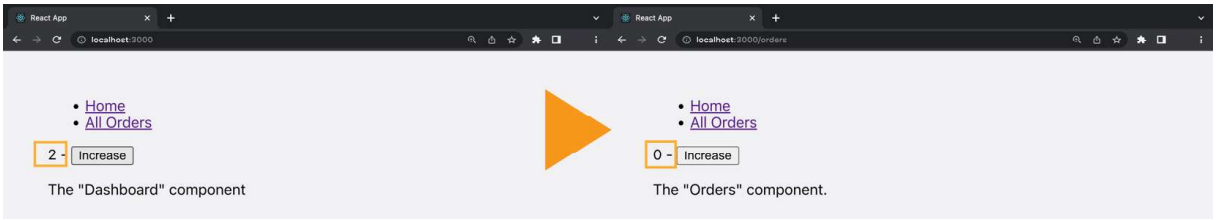

Figure 12.4: The counter state is reset when switching the page

In the screenshot, you can see that the counter is initially set to **2** (because the button was clicked twice). After navigating from **`Dashboard`** to the **`Orders`** page (via the appropriate navigation link), the counter changes to **0**.

That happens because the page is reloaded due to the HTTP request that's sent by the browser.

To work around this issue and avoid this unintended page reload, you must prevent the browser's default behavior. Instead of sending a new HTTP request, the browser should simply update the URL (from **`localhost:3000`** to **`localhost:3000/orders`**). The React Router library can then react to this URL change and update the screen as needed. Therefore, to the website user, it seems as if a different page was loaded. But behind the scenes, it's just the page document (the DOM) that was updated.

Thankfully, you don't have to implement the logic for this on your own. Instead, the React Router library exposes a special **`Link`** component that should be used for rendering links that behave as described previously.

To use this new component, the code in **`src/App.js`** must be adjusted like this:

```
import { useState } from 'react';
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';
```

```
import Dashboard from './routes/Dashboard';
import Orders from './routes/Orders';

function App() {
  const [counter, setCounter] = useState(0);

  function incCounterHandler() {
    setCounter((prevCounter) => prevCounter + 1);
  }

  return (
    <BrowserRouter>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/orders">All Orders</Link>
          </li>
        </ul>
      </nav>
      <p>
        {counter} - <button onClick={incCounterHandler}>Increase</button>
      </p>
      <Routes>
        <Route path="/" element={<Dashboard />} />
        <Route path="/orders" element={<Orders />} />
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

> **NOTE**
>
> The code for this example can be found at https://packt.link/kPXCL.

Inside the **`<li>`** elements, the new **`Link`** component is used. That component requires the **`to`** prop, which is used to define the URL path that should be loaded.

By using this component in place of the **`<a>`** anchor element, the counter state is no longer reset. This is because React Router now prevents the browser's default behavior (i.e., the unintended page reload described above) and displays the correct page content.

The **`Link`** component is therefore the default component that should be used for adding links to React websites. However, there are the following two exceptions:

- **External links**: Links that lead to external resources and websites
- **Navigation links**: Links that should (possibly) change their appearance upon route changes

External links are links that lead to any resource that's not one of your React app routes. For example, you could have a link that leads to a Wikipedia page. Since that's not part of your React app, the **`Link`** component is not the correct choice. Instead, you should use the **`<a>`** element for those kinds of links:

```
<a href="https://wikipedia.org">Look it up!</a>
```

But the second exception, navigation links, is worth a closer look.

FROM LINK TO NAVLINK

You can, in theory, always use the **`Link`** component for internal links. But you will sometimes have links on your page that should change their appearance (e.g., their color) when they lead to the currently active route. Links placed in the main navigation bar often need that behavior in order to reflect which page is active at the moment.

On many websites, the main navigation bar signals to the user which page they are currently on—simply by highlighting the navigation item that leads to the current page in the navigation bar.

Consider this code snippet, which is the example from previously without the state but with some extra styling and a **`<header>`** element (also only added for styling and semantic purposes):

```
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';

import Dashboard from './routes/Dashboard';
import Orders from './routes/Orders';
```

```
function App() {
  return (
    <BrowserRouter>
      <header>
        <nav>
          <ul>
            <li>
              <Link to="/">Dashboard</Link>
            </li>
            <li>
              <Link to="/orders">All Orders</Link>
            </li>
          </ul>
        </nav>
      </header>
      <Routes>
        <Route path="/" element={<Dashboard />} />
        <Route path="/orders" element={<Orders />} />
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

This code snippet does not yet have any new functionality or behavior. It just adds a more realistic-looking main navigation bar:

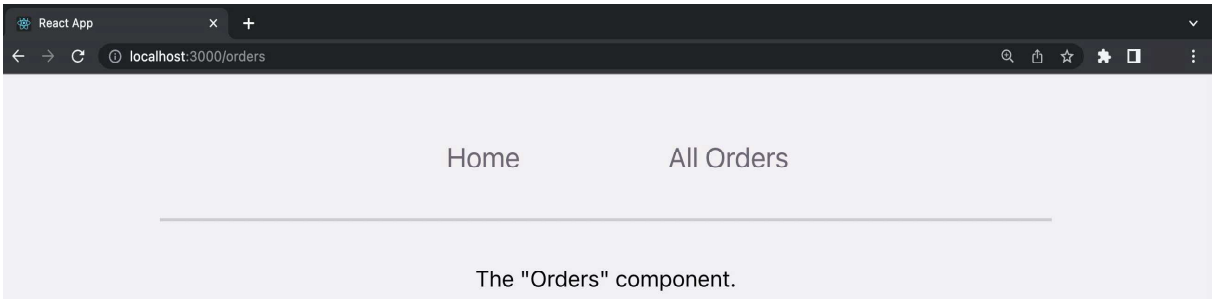

Figure 12.5: The styled navigation bar is above the main route content

In this screenshot, the navigation bar received some styling. It now looks more like a proper website.

But you might notice that it's not always immediately obvious which page the user is currently on. Sure, the dummy page content currently displays the page name (**`"Orders"`** or **`"Dashboard"`**), and the URL path also is **`/orders`** or **`/`**. But, when navigating a website, the most obvious clue users are accustomed to is a highlighted element in the main navigation bar.

To prove this point, compare the previous screenshot to the following one:

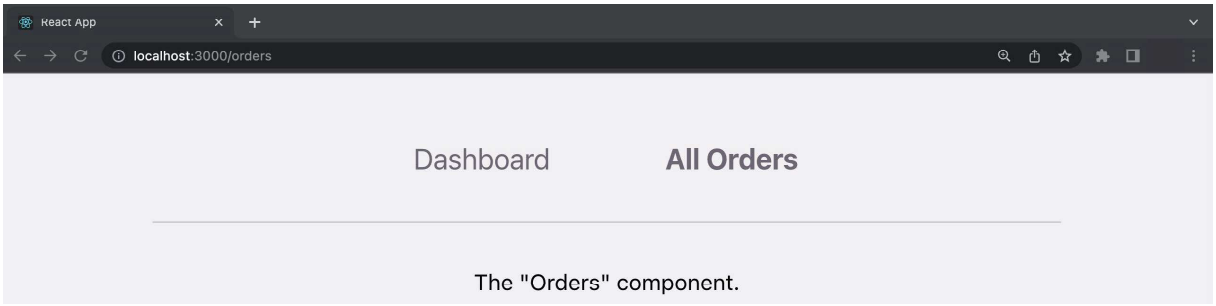

Figure 12.6: The highlighted All Orders navigation link appears bold

In this version of the website, it's immediately clear that the user is on the **`"Orders"`** page since the **`All Orders`** navigation link is highlighted (in this case by making it bold). It's subtle things such as this that make websites more usable and can ultimately lead to higher user engagement.

But how can this be achieved?

To do this, you would not use the **`Link`** component but instead, a special alternative component, offered by **`react-router-dom`**: the **`NavLink`** component.

The **`NavLink`** component is used pretty much like the **`Link`** component. You wrap it around some text (the link's caption), and you define the target path via the **`to`** prop. But the **`NavLink`** component offers two special props:

- The **`style`** prop
- The **`className`** prop

To be precise, the **`Link`** component also supports **`style`** and **`className`**, which are standard HTML element props, after all. **`Link`** supports these props to allow developers to style the link elements.

But when using **`NavLink`**, **`style`** and **`className`** work slightly differently. Instead of requiring the standard **`style`** and **`className`** prop values (see *Chapter 6*, *Styling React Apps*), the two props now accept functions.

The function passed to either **`style`** or **`className`** automatically receives an argument (provided by the React Router library). This will be an object with a Boolean **`isActive`** property that indicates whether this link is leading to the currently active route.

This **`isActive`** value can then be used to conditionally return different **`style`** or **`className`** values based on the current state of the navigation link.

The adjusted **`App.js`** code that leads to the user interface shown in the previous screenshot looks like this:

```
import { BrowserRouter, Routes, Route, NavLink } from 'react-router-dom';

import Dashboard from './routes/Dashboard';
import Orders from './routes/Orders';

function App() {
  return (
    <BrowserRouter>
      <header>
        <nav>
          <ul>
            <li>
              <NavLink
                to="/"
                style={({ isActive }) =>
                  isActive ? { fontWeight: 'bold' } : undefined
                }
              >
                Dashboard
              </NavLink>
            </li>
            <li>
              <NavLink
                to="/orders"
                style={({ isActive }) =>
                  isActive ? { fontWeight: 'bold' } : undefined
                }
```

```
              >
                All Orders
              </NavLink>
            </li>
          </ul>
        </nav>
      </header>
      <Routes>
        <Route path="/" element={<Dashboard />} />
        <Route path="/orders" element={<Orders />} />
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

In this code snippet, the **`NavLink`** component replaces the **`Link`** component. The **`style`** prop is added to both **`NavLink`** components, and for both links, an arrow function is passed to **`style`** (though you can also use a normal function). The **`isActive`** property is extracted from the received object via object destructuring. In the function body, a ternary expression is used to either return a style value of **`{ fontWeight: 'bold' }`** or **`undefined`** (in which case no special styling is added).

> **NOTE**
>
> You can find the finished code for this example on GitHub at https://packt.link/FcM7A.

One important note is that **`NavLink`** will consider a route to be active if its path matches the current URL path *or* if its path starts with the current URL path. For example, if you had a **`/blog/all-posts`** route, a **`NavLink`** component that points at just **`/blog`** would be considered active if the current route is **`/blog/all-posts`** (because that route path starts with **`/blog`**). If you don't want this behavior, you can add the special **`end`** prop to the **`NavLink`** component, as follows:

```
<NavLink
  to="/blog"
  style={({ isActive }) => isActive ? { color: 'red' } : undefined}
```

```
  end>
    Blog
</NavLink>
```

With this special prop added, this NavLink would only be considered active if the current route is exactly **`/blog`—for `/blog/all-posts`**, the link would not be active.

`NavLink` is always the preferred choice when the styling of a link depends on the currently active route. For all other internal links, use **`Link`**. For external links, **`<a>`** is the element of choice.

ROUTE COMPONENTS VERSUS "NORMAL" COMPONENTS

It's worth mentioning and noting that, in the previous examples, the **`Dashboard`** and **`Orders`** components were regular React components. You could use these components anywhere in your React app—not just as values for the **`element`** prop of the **`Route`** component.

The two components are special in that both are stored in the **`src/routes`** folder in the project directory. They are not stored in the **`src/components`** folder, which was used for components throughout this book.

That's not something you have to do, though. Indeed, the folder names are entirely up to you. These two components could be stored in **`src/components`**. You could also store them in an **`src/elements`** folder. It really is up to you. But using **`src/routes`** is quite common for components that are exclusively used for routing. Popular alternatives are **`src/screens`**, **`src/views`**, and **`src/pages`** (again, it is up to you).

If your app includes any other components that are not used as routing elements, you would still store those in **`src/components`** (i.e., in a different path). This is not a hard rule or a technical requirement, but it does help with keeping your React projects manageable. Splitting your components across multiple folders makes it easier to quickly understand which components fulfill which purpose in the project.

In the example project mentioned previously, you can, for example, refactor the code such that the navigation code is stored in a separate component (e.g., a **`MainNavigation`** component, stored in **`src/components/shared/MainNavigation.js`**). The component code looks like this:

```
import { NavLink } from 'react-router-dom';

import classes from './MainNavigation.module.css';

function MainNavigation() {
  return (
    <header className={classes.header}>
      <nav>
        <ul className={classes.links}>
          <li>
            <NavLink
              to="/"
              className={({ isActive }) =>
                isActive ? classes.active : undefined
              }
              end
            >
              Dashboard
            </NavLink>
          </li>
          <li>
            <NavLink
              to="/orders"
              className={({ isActive }) =>
                isActive ? classes.active : undefined
              }
            >
              All Orders
            </NavLink>
          </li>
        </ul>
      </nav>
    </header>
```

```
  );
}

export default MainNavigation;
```

In this code snippet, the **`NavLink`** component is adjusted to assign a CSS class named **`active`** to any link that belongs to the currently active route. In general, CSS classes are added to various elements (with the help of CSS Modules, as discussed in *Chapter 6*, *Styling React Apps*). Besides that, it's essentially the same navigation menu code as that used earlier in this chapter.

This **`MainNavigation`** component can then be imported and used in the **`App.js`** file like this:

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';

import MainNavigation from './components/shared/MainNavigation';
import Dashboard from './routes/Dashboard';
import Orders from './routes/Orders';

function App() {
  return (
    <BrowserRouter>
      <MainNavigation />
      <Routes>
        <Route path="/" element={<Dashboard />} />
        <Route path="/orders" element={<Orders />} />
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

Importing and using the **`MainNavigation`** component leads to a leaner **`App`** component and yet preserves the same functionality as before.

These changes show how you can combine routing, components that are only used for routing (**`Dashboard`** and **`Orders`**), and components that are used outside of routing (**`MainNavigation`**).

But you can also take it a step further. Semantically, it makes sense to wrap your route component content (i.e., the JSX code returned by **`Dashboard`** and **`Orders`**) with the **`<main>`** element. This default HTML element is commonly used to wrap the main content of the page.

Since both components would therefore wrap their returned values with **`<main>`**, you can create a separate **`Layout`** component that's wrapped around the entire block of route definitions (i.e., around the **`Routes`** component) and looks like this:

```
import MainNavigation from './MainNavigation';

function Layout({ children }) {
  return (
    <>
      <MainNavigation />
      <main>{children}</main>
    </>
  );
}

export default Layout;
```

This component uses the special **`children`** prop (see *Chapter 3*, *Components and Props*) to wrap the **`<main>`** element around whatever JSX content is placed between the **`<Layout>`** tags. In addition, it renders the **`MainNavigation`** component above the **`<main>`** element.

The **`Layout`** component (which could be stored in **`src/components/shared/Layout.js`**) can then be imported and used in **`App.js`**, as follows:

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';

import Layout from './components/shared/Layout';
import Dashboard from './routes/Dashboard';
import Orders from './routes/Orders';

function App() {
  return (
    <BrowserRouter>
      <Layout>
        <Routes>
          <Route path="/" element={<Dashboard />} />
```

```
          <Route path="/orders" element={<Orders />} />
        </Routes>
      </Layout>
    </BrowserRouter>
  );
}

export default App;
```

<Layout> is wrapped around the **<Routes>** element, which contains the various route definitions. Therefore, no matter which route is active, its content is wrapped with the **<main>** element and **MainNavigation** is placed above it.

As a result, with some additional CSS styling added, the website looks like this:

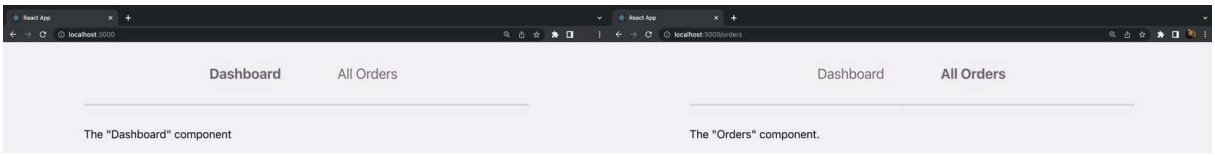

Figure 12.7: Both route components share the same layout

It's essentially the same look as before, but it's now achieved via a leaner **App** component and by splitting application responsibilities (such as providing navigation, managing routes, etc.) into different components.

> **NOTE**
>
> You can find the finished code for this example on GitHub at https://packt.link/TkNsc.

But even with this split into multiple components, the demo application still suffers from an important problem: it only supports static, predefined routes. But, for most websites, those kinds of routes are not enough.

FROM STATIC TO DYNAMIC ROUTES

Thus far, all examples have had two routes: **/** for the **Dashboard** component and **/orders** for the **Orders** component. But you can, of course, add as many routes as needed. If your website consists of 20 different pages, you can (and should) add 20 route definitions (i.e., 20 **Route** components) to your **App** component.

On most websites, however, you will also have some routes that can't be defined manually—because not all routes are known in advance.

Consider the example from before, enriched with additional components and some realistic dummy data:

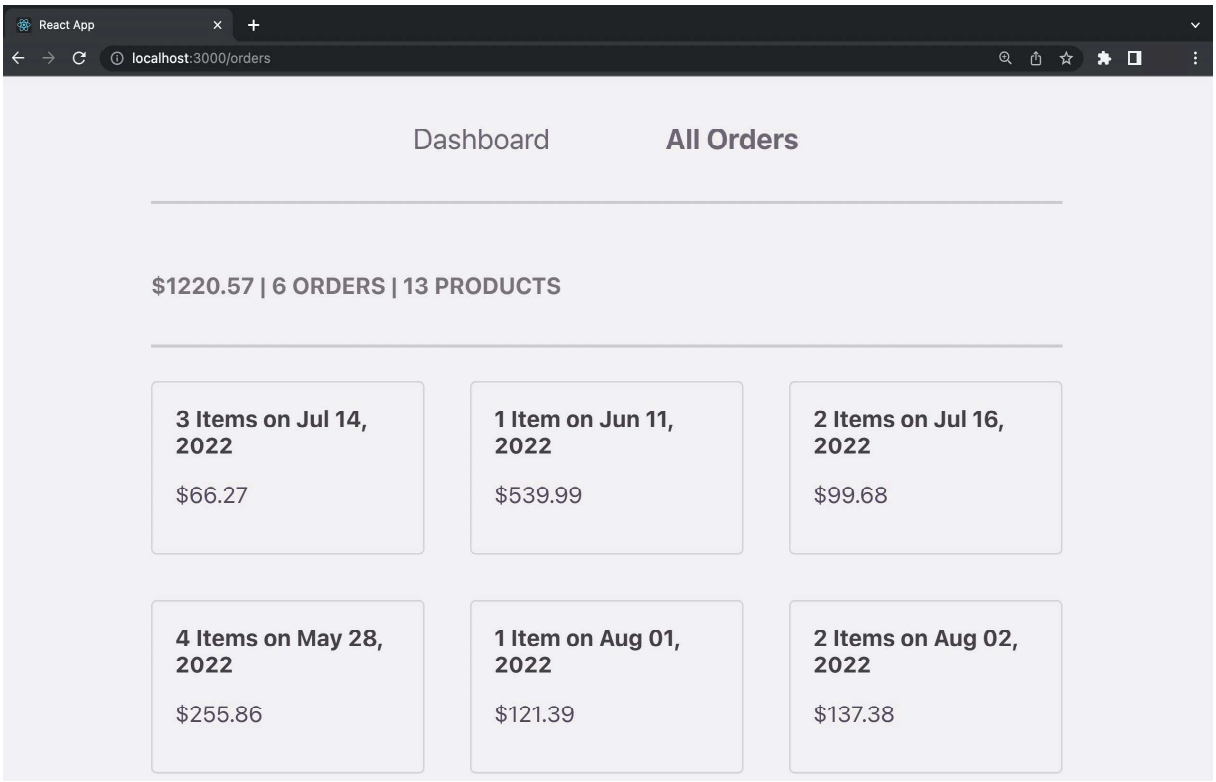

Figure 12.8: A list of order items

> **NOTE**
>
> You can find the code for this example on GitHub at https://packt.link/KcDA6. In the code, you'll notice that many new components and style files were added. The code does not use any new features, though. It's just used to display a more realistic user interface and output some dummy data.

In the preceding screenshot, *Figure 12.8*, you can see a list of order items being output on the **`All Orders`** page (i.e., by the **`Orders`** component).

In the underlying code, every order item is wrapped with a **`Link`** component so that a separate page with more details will be loaded for each item:

```
function OrdersList() {
  return (
    <ul className={classes.list}>
      {orders.map((order) => (
        <li key={order.id}>
          <Link to='/orders'><OrderItem order={order} /></Link>
        </li>
      ))}
    </ul>
  );
}
```

In this code snippet, the path for the **`Link`** component is set to **`/orders`**. However, that's not the final value that should be assigned. Instead, this example highlights an important problem: while it's the same component that should be loaded for every order item (i.e., some component that displays detailed data about the selected order), the exact content of the component depends on which order item was selected. It's the same component with different data.

Outside of routing, you would use props to reuse the same component with different data. But with routing, it's not just about the component. You also must support different paths—because the detailed data for different orders should be loaded via different paths. Otherwise, you would again end up with URLs that are not shareable or reloadable.

Therefore, the path must include not only some static identifier (such as **`/orders`**) but also a dynamic value that's different for every order item. For three order items with **`id`** values **`o1`**, **`o2`**, and **`o3`**, the goal could be to support the **`/orders/o1`**, **`/orders/o2`**, and **`/order/o3`** paths.

For this reason, the following three route definitions could be added:

```
<Route path="/orders/o1" element={ <OrderDetail id="o1" /> } />
<Route path="/orders/o2" element={ <OrderDetail id="o2" /> } />
<Route path="/orders/o3" element={ <OrderDetail id="o3" /> } />
```

But this solution has a major flaw. Adding all these routes manually is a huge amount of work. And that's not even the biggest problem. You typically don't even know all values in advance. In this example, when a new order is placed, a new route would have to be added. But you can't adjust the source code of your website every time a visitor places an order.

Clearly, then, a better solution is needed. React Router offers that better solution as it supports *dynamic routes*.

Dynamic routes are defined just like other routes, except that, when defining their **`path`** values, you will need to include one or more placeholder values with identifiers of your choice.

The **`OrderDetail`** route definition therefore looks like this:

```
<Route path="/orders/:id" element={ <OrderDetail /> } />
```

The following three key things have changed:

- It's just one route definition instead of a (possibly) infinite list of definitions.
- **`path`** contains a dynamic route segment (**`:id`**).
- **`OrderDetail`** no longer receives an **`id`** prop.

The **`:id`** syntax is a special syntax supported by React Router. Whenever a segment of a path starts with a colon, React Router treats it as a dynamic segment. That means that it will be replaced with a different value in the actual URL path. For the **`/orders/:id`** route path, the **`/orders/o1`**, **`/orders/o2`**, and **`/orders/abc`** paths would all match and therefore activate the route.

Of course, you don't have to use **`:id`**. You can use any identifier of your choice. For the preceding example, **`:orderId`**, **`:order`**, or **`:oid`** could also make sense.

The identifier will help your app to load the correct data inside the page component (inside **`OrderDetail`** in this case). That's why the **`id`** prop was removed from **`OrderDetail`**. Since only one route is defined, only one specific **`id`** value can be passed via props. That won't help. Therefore, a different way of loading order-specific data must be used.

Extracting Route Parameters

In the previous example, when a website user visits **`/orders/o1`** or **`/orders/o2`** (or the same path for any other order ID), the **`OrderDetail`** component is loaded. This component should then output more information about the specific order that was selected (i.e., the order whose **`id`** is encoded in the URL path).

By the way, that's not just the case for this example; you can think of many other types of websites as well. You could also have, for example, an online shop with routes for products (**`/products/p1`**, **`/products/p2`**, etc.), or a travel blog where users can visit individual blog posts (**`/blog/post1`**, **`/blog/post2`**, etc.).

In all these cases, the question is, how do you get access to the data that should be loaded for the specific identifier (e.g., the ID) that's included in the URL path? Since it's always the same component that's loaded, you need a way of dynamically identifying the order, product, or blog post for which the detail data should be fetched.

One possible solution would be the usage of props. Whenever you build a component that should be reusable yet configurable and dynamic, you can use props to accept different values. For example, the **`OrderDetail`** component could accept an **`id`** prop and then, inside the component function body, load the data for that specific order ID.

However, as mentioned in the previous section, this is not a possible solution when loading the component via routing. Keep in mind that the **`OrderDetail`** component is created when defining the route:

```
<Route path="/orders/:id" element={ <OrderDetail />} />
```

Since the component is created when defining the route in the **`App`** component, you can't pass in any dynamic, ID-specific prop values.

Fortunately, though, that's not necessary. React Router gives you a solution that allows you to extract the data encoded in the URL path from inside the component that's displayed on the screen (when the route becomes active): the **`useParams()`** Hook.

This Hook can be used to get access to the route parameters of the currently active route. Route parameters are simply the dynamic values encoded in the URL path—**`id`**, in the case of this **`OrderDetail`** example.

Inside the **`OrderDetail`** component, **`useParams()`** can therefore be used to extract the specific order **`id`** and load the appropriate order data, as follows:

```
import { useParams } from 'react-router-dom';

import Details from '../components/orders/Details';
import { getOrderById } from '../data/orders';

function OrderDetail() {
  const params = useParams();
```

```
  const orderId = params.id; // in this example, orderId is "o1" or "o2"
etc.
  const order = getOrderById(orderId);

  return <Details order={order} />;
}

export default OrderDetail;
```

As you can see in this snippet, **`useParams()`** returns an object that contains all route parameters of the currently active route as properties. Since the route path was defined as **`/orders/:id`**, the **`params`** object contains an **`id`** property. The value of that property is then the actual value encoded in the URL path (e.g., **`o1`**). If you choose a different identifier name in the route definition (e.g., **`/orders/:orderId`** instead of **`/orders/:id`**), that property name must be used to access the value in the **`params`** object (i.e., access **`params.orderId`**).

> **NOTE**
>
> You find the complete code on GitHub at https://packt.link/YKmjL.

By using route parameters, you can thus easily create dynamic routes that lead to different data being loaded. But, of course, defining routes and handling route activation are not that helpful if you do not have links leading to the dynamic routes.

CREATING DYNAMIC LINKS

As mentioned earlier in this chapter (in the *Adding Page Navigation* section), website visitors should be able to click on links that should then take them to the different pages that make up the overall website—meaning, those links should activate the various routes defined with the help of React Router.

As explained in the *Adding Page Navigation* and *From Link to NavLink* sections, for internal links (i.e., links leading to routes defined inside the React app), the **`Link`** or **`NavLink`** components are used.

So, for static routes such as **`/orders`**, links are created like this:

```
<Link to="/orders">All Orders</Link> // or use <NavLink> instead
```

When building a link to a dynamic route such as **`/orders/:id`**, you can therefore simply create a link like this:

```
<Link to="/orders/o1">All Orders</Link>
```

This specific link loads the **`OrderDetails`** component for the order with the ID **`o1`**.

Building the link as follows would be incorrect:

```
<Link to="/orders/:id">All Orders</Link>
```

The dynamic path segment (**`:id`**) is only used when defining the route—not when creating a link. The link has to lead to a specific resource (a specific order, in this case).

But creating links to specific orders, as shown previously, is not very practical. Just as it wouldn't make sense to define all dynamic routes individually (see the *From Static to Dynamic Routes* section), it doesn't make sense to create the respective links manually.

Sticking to the orders example, there is also no need to create links like that as you already have a list of orders that's output on one page (the **`Orders`** component, in this case). Similarly, you could have a list of products in an online shop. In all these cases, the individual items (orders, products, etc.) should be clickable and lead to details pages with more information.

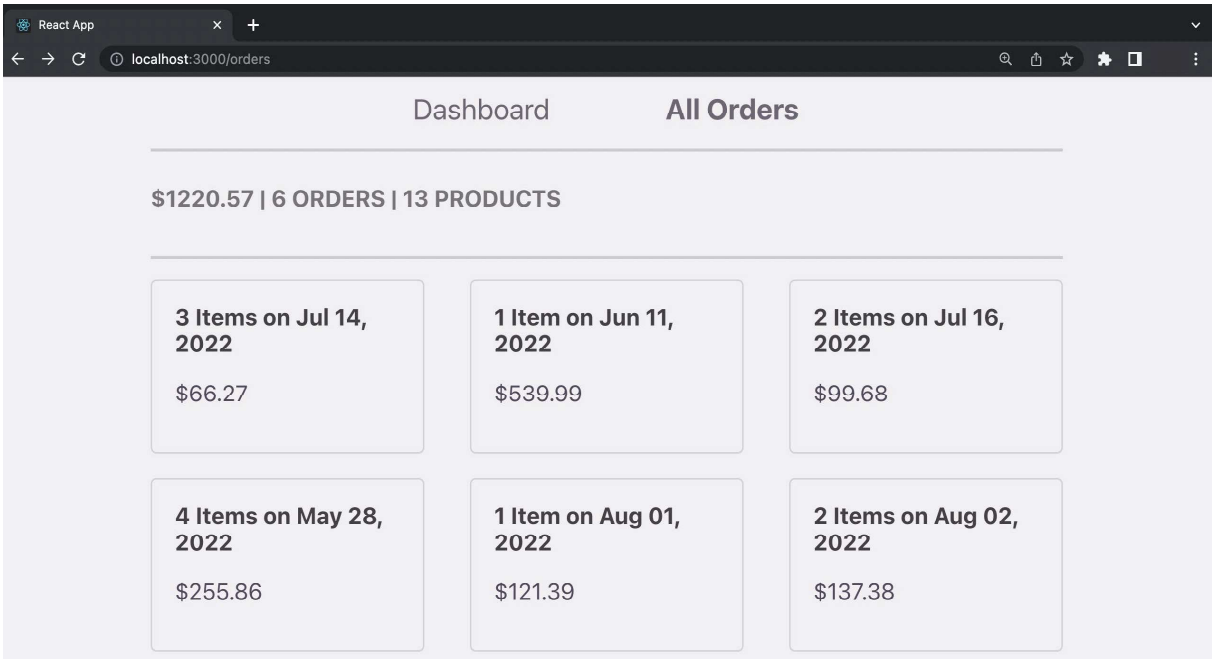

Figure 12.9: A list of clickable order items

Therefore, the links can be generated dynamically when rendering the list of JSX elements. In the case of the orders example, the code looks like this:

```
function OrdersList() {
  return (
    <ul className={classes.list}>
      {orders.map((order) => (
        <li key={order.id}>
          <Link to={`/orders/${order.id}`}><OrderItem order={order} /></
Link>
        </li>
      ))}
    </ul>
  );
}
```

In this code example, the value of the **`to`** prop is set equal to a string that includes the dynamic **`order.id`** value. Therefore, every list item receives a unique link that leads to a different details page. Or, to be precise, the link always leads to the same component but with a different order **`id`** value, hence loading different order data.

> **NOTE**
>
> In this code snippet (which can be found at https://packt.link/iDObH, the string is created as a **template literal**. That's a default JavaScript feature that simplifies the creation of strings that include dynamic values.
>
> You can learn more about template literals on MDN at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals.

NAVIGATING PROGRAMMATICALLY

In the previous section, as well as earlier in this chapter, user navigation was enabled by adding links to the website. Indeed, links are the default way of adding navigation to a website. But there are scenarios where programmatic navigation is required instead.

Programmatic navigation means that a new page should be loaded via JavaScript code (rather than using a link). This kind of navigation is typically required if the active page changes in response to some action—e.g., upon form submission.

If you take the example of form submission, you will normally want to extract and save the submitted data. But thereafter, the user will sometimes need to be redirected to a different page. For example, it makes no sense to keep the user on a **`Checkout`** page after processing the entered credit card details. You might want to redirect the user to a **`Success`** page instead.

In the example discussed throughout this chapter, the **`All Orders`** page could include an input field that allows users to directly enter an order **`id`** and load the respective orders data after clicking the **`Find`** button.

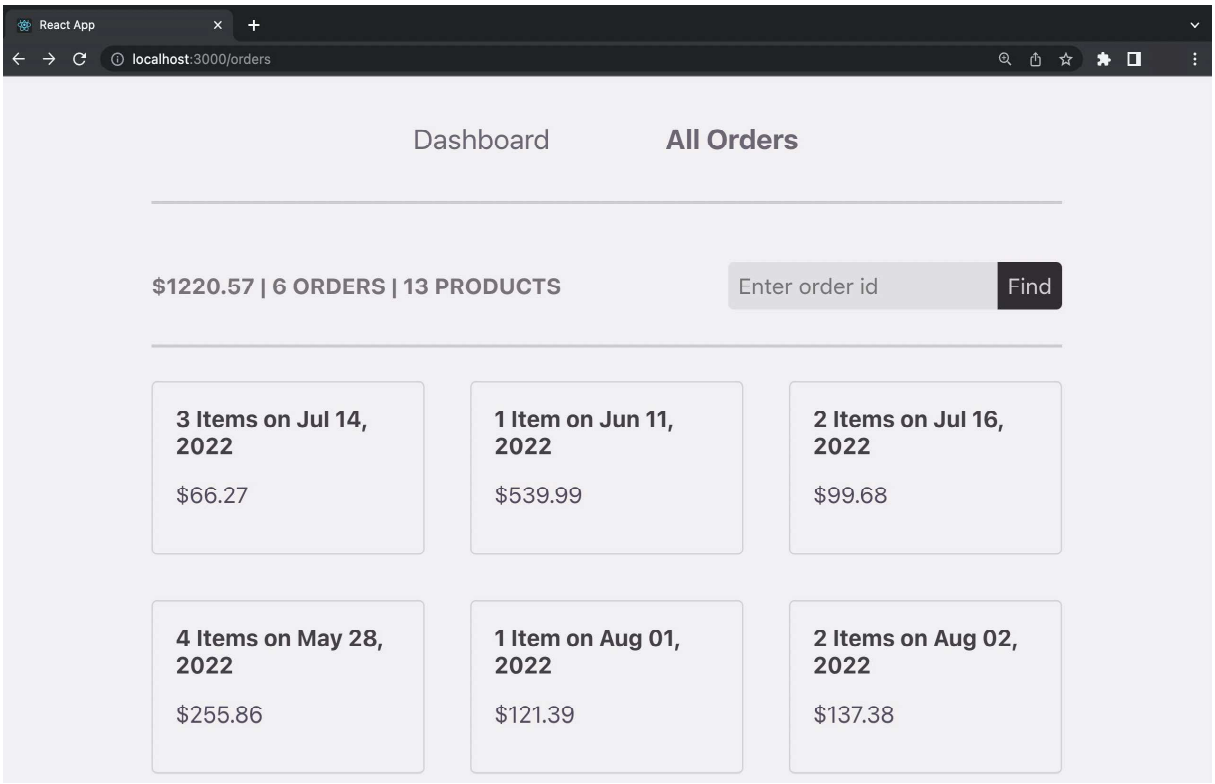

Figure 12.10: An input field that can be used to quickly load a specific order

In this example, the entered order **`id`** is first processed and validated before the user is sent to the respective details page. If the provided **`id`** is invalid, an error message is shown instead. The code looks like this:

```
function OrdersSummary() {
  const orderIdInputRef = useRef();

  const { quantity, total } = getOrdersSummaryData();

  function findOrderHandler(event) {
    event.preventDefault();
```

```
    const orderId = orderIdInputRef.current.value;
    const orderExists = orders.some((order) => order.id === orderId);

    if (!orderExists) {
      alert('Could not find an order for the entered id.');
      return;
    }
    // should navigate the user here
  }

  return (
    <div className={classes.row}>
      <p className={classes.summary}>
        ${total.toFixed(2)} | {orders.length} Orders | {quantity}
Products
      </p>
      <form className={classes.form} onSubmit={findOrderHandler}>
        <input
          type="text"
          placeholder="Enter order id"
          aria-label="Find an order by id."
          ref={orderIdInputRef}
        />
        <button>Find</button>
      </form>
    </div>
  );
}
```

The code snippet does not yet include the code that will actually trigger the page change, but it does show how the user input is read and validated.

Therefore, this is a perfect scenario for the use of programmatic navigation. A link can't be used here since it would immediately trigger a page change—without allowing you to validate the user input first (at least not after the link was clicked).

The React Router library also supports programmatic navigation for cases like this. You can import and use the special **`useNavigate()`** Hook to gain access to a navigation function that can be used to trigger a navigation action (i.e., a page change):

```
import { useNavigate } from 'react-router-dom';

const navigate = useNavigate();
navigate('/orders'); // programmatic alternative to <Link to="/orders">
```

Hence, the **`OrdersSummary`** component from previously can be adjusted like this to use this new Hook:

```
function OrdersSummary() {
  const orderIdInputRef = useRef();
  const navigate = useNavigate();

  const { quantity, total } = getOrdersSummaryData();

  function findOrderHandler(event) {
    event.preventDefault();
    const orderId = orderIdInputRef.current.value;
    const orderExists = orders.some((order) => order.id === orderId);

    if (!orderExists) {
      alert('Could not find an order for the entered id.');
      return;
    }

    navigate(`/orders/${orderId}`);
  }

  // returned JSX code did not change, hence omitted
}
```

It's worth noting that the value passed to **`navigate()`** is a dynamically constructed string. Programmatic navigation supports both static and dynamic paths.

> **NOTE**
>
> The code for this example can be found at https://packt.link/9cdSZ.

No matter whether you need link-based or programmatic navigation, the two approaches also support another important feature: redirecting users.

REDIRECTING

Thus far, all the explored navigation options (links and programmatic navigation) forward a user to a specific page.

In most cases, that's the intended behavior. But in some cases, the goal is to redirect a user instead of forwarding them.

The difference is subtle but important. When a user is forwarded, they can use the browser's navigation buttons (**`Back`** and **`Forward`**) to go back to the previous page or forward to the page they came from. For redirects, that's not possible. Whenever a user is redirected to a specific page (rather than forwarded), they can't use the **`Back`** button to return to the previous page.

Redirecting users can, for example, be useful for ensuring that users can't go back to a login page after authenticating successfully.

When using React Router, the default behavior is to forward users. But you can easily switch to redirecting by adding the special **`replace`** prop to the **`Link`** (or **`NavLink`**) components, as follows:

```
<Link to="/success" replace>Confirm Checkout</Link>
```

When using programmatic navigation, you can pass a second, optional argument to the **`navigate()`** function. That second parameter value must be an object that can contain a **`replace`** property that should be set to **`true`**, if you want to redirect users:

```
navigate('/dashboard', { replace: true });
```

Being able to redirect or forward users allows you to build highly user-friendly web applications that offer the best possible user experience for different scenarios.

NESTED ROUTES

Another core feature offered by React Router that has not yet been covered is its support for nested routes.

Nested routes are routes that are descendants of other routes. Just as you can build a tree of components, you can also build a tree of route definitions. Though, typically, you don't need very deep levels of route nesting.

This feature can be tricky to understand at first, however, so consider the following example **App** component:

```
function App() {
  return (
    <BrowserRouter>
      <Layout>
        <Routes>
          <Route path="/" element={<Dashboard />} />
          <Route path="/orders" element={<OrdersRoot />}>
            <Route element={<Orders />} index />
            <Route path=":id" element={<OrderDetail />} />
          </Route>
        </Routes>
      </Layout>
    </BrowserRouter>
  );
}
```

> **NOTE**
>
> You can find the complete code on GitHub at https://packt.link/PKFAv.

As you can see in this example, the route definition for the **/orders** route contains two child route definitions:

- **<Route element={<Orders />} index />**
- **<Route path=":id" element={<OrderDetail />} />**

The first interesting thing about these two route definitions is that they are children of another route definition (that is, another **Route** component). Thus far, all route definitions were only direct children of **Routes** (and therefore siblings to each other).

Besides being nested inside another route definition, these two route definitions also have some strange props. The second nested route has a path of **":id"** instead of **"/:id"**, and the first nested route has no path at all but a special **index** prop instead.

The **index** prop will be explained further shortly.

Regarding the paths, paths starting with a slash (**/**) are absolute paths. They're always appended directly after the domain name. On the other hand, paths that do not start with a slash are relative paths; they are appended after the currently active path. Hence, the nested **:id** path yields an overall path of **/orders/:id** if activated, while the **/orders** route is active. This switch from absolute to relative paths is needed when building nested routes because the inner routes will actually be connected to the outer route.

But what's the idea behind nested routes?

You might notice that the parent route of the two nested routes will render an **<OrdersRoot />** element when activated (due to its **element** prop value). So, whenever a user visits **/orders**, that component gets rendered.

But it's actually not just that component that will be rendered. The first nested component (which displays **<Orders />**) will also be displayed because of its **index** prop. The **index** prop "tells" React Router that this route should also be active if its parent route is displayed and no other nested route was activated.

The idea of nested routes is that multiple routes can be active simultaneously to render a hierarchical component structure. In the preceding example, when a user visits **/orders**, both the **OrdersRoot** and **Orders** components would be rendered. For **/orders/o1**, **OrdersRoot** and **OrderDetail** would be shown on the screen.

But these components would not only show up on the screen at the same time (as they would were they simply rendered as siblings), but are also nested into each other. The final structure for **/orders/o1** would be similar to the following component composition:

```
<App>
  <OrdersRoot>
    <OrderDetail />
  </OrdersRoot>
</App>
```

But this still doesn't explain why you might want to use nested routes. What's the advantage of producing such nested component structures via routing?

It can be helpful if multiple routes have some shared user interface.

For example, both the **`OrderDetail`** and **`Orders`** components might need to display the order search input field like this:

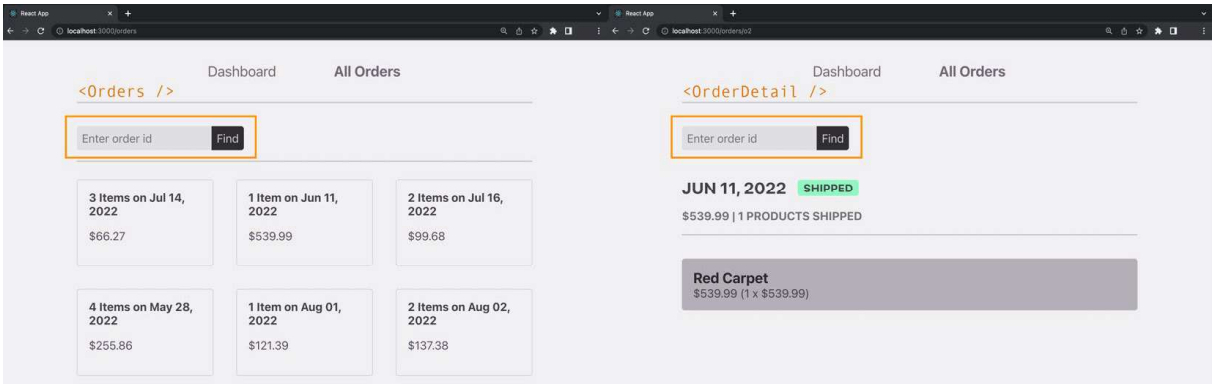

Figure 12.11: Both components have a shared order-specific user interface element

In this screenshot, the **`Orders`** and **`OrderDetail`** components share the order search input. But this input is not part of the overall layout (that is, the one stored in the **`Layout`** component, which is wrapped around the entire set of route definitions). It's not included in the general page layout because it should not affect all routes—only the order-specific ones. For example, the **`/`** route (the dashboard route) should not display this order ID search field.

Thanks to nested routes, this can be achieved. This piece of the user interface can be shared by including it in the **`OrdersRoot`** component. Since that's the component loaded for the parent route of the two nested routes, it and its content will be included for both child routes.

The code for the **`OrdersRoot`** component looks like this:

```
import { Outlet } from 'react-router-dom';
import OrderSearch from '../components/orders/OrderSearch';

function OrdersRoot() {
  return (
    <>
      <OrderSearch />
      <Outlet />
    </>
  );
}
```

It's quite a simple component, but it does include that shared **`OrderSearch`** component (which renders and controls the search input field you see in the preceding screenshot).

However, the **`OrdersRoot`** component also renders another interesting component: the **`Outlet`** component.

`Outlet` is a special component provided by React Router. After importing it from **`react-router-dom`**, it can be used to denote the location where child route components should be rendered.

React Router does not guess where to render any child routes you might have defined. Instead, you have to "tell" it by adding the **`<Outlet />`** element in exactly that location where React Router should create and display the child route elements. For the preceding example, **`Orders`** and **`OrderDetail`** (which are the child route components) are therefore rendered into this place.

Nested routes are a feature that's not necessarily needed all the time. But they can be very useful for building more complex user interfaces where certain elements should be shared among different routes. Especially if components must be shared across some but not all routes, nested routes can be a great solution that's easy to implement. On the other hand, if some component is needed on all pages (e.g., the main navigation bar), creating a **`Layout`** component that wraps all route definitions might be more feasible. That said, if you prefer to, you could also use nested routes in that case.

HANDLING UNDEFINED ROUTES

Previous sections in this chapter have all assumed that you have predefined routes that should be reachable by website visitors. But what if a visitor enters a URL that's simply not supported?

For example, the demo website used throughout this chapter supports the **`/`**, **`/orders`**, and **`/orders/<some-id>`** paths. But it does not support **`/home`**, **`/products/p1`**, **`/abc`**, or any other path that's not one of the defined route paths.

To show a custom *"Not Found"* page, you can define a **`"Catch All"`** route with a special path—the **`*`** path:

```
<Route path="*" element={ <NotFound /> } />
```

When adding this route to the list of route definitions in the **`App`** component, the **`NotFound`** component will be displayed on the screen when no other route matches the entered or generated URL path.

LAZY LOADING

In *Chapter 9*, *Behind the Scenes of React and Optimization Opportunities*, you learned about lazy loading—a technique that can be used to load certain pieces of the React application code only when needed.

Code splitting makes a lot of sense if some components will be loaded conditionally and may not be needed at all. Hence, routing is a perfect scenario for lazy loading. When applications have multiple routes, some routes may never be visited by a user. Even if all routes are visited, not all the code for all app routes (i.e., for their components) must be downloaded right at the start of the application. Instead, it makes sense to only download code for individual routes when they actually become active.

Thankfully, lazy loading with routing works just as explained in *Chapter 9*; you use React's **`lazy()`** function, the dynamic **`import()`** function, and React's **`Suspense`** component to split your code into multiple bundles.

Since the goal is to split code by route, all these features are used in the place where your route definitions are. In most cases, that will be the **`App`** component.

For the example application discussed throughout this chapter, lazy loading is implemented like this:

```
import { lazy, Suspense } from 'react';
import { BrowserRouter, Routes, Route } from 'react-router-dom';

import Layout from './components/shared/Layout';
import Dashboard from './routes/Dashboard';

const OrdersRoot = lazy(() => import('./routes/OrdersRoot'));
const Orders = lazy(() => import('./routes/Orders'));
const OrderDetail = lazy(() => import('./routes/OrderDetail'));

function App() {
  return (
    <BrowserRouter>
      <Layout>
        <Suspense fallback={<p>Loading...</p>}>
          <Routes>
            <Route path="/" element={<Dashboard />} />
            <Route path="/orders" element={<OrdersRoot />}>
              <Route element={<Orders />} index />
```

```
              <Route path=":id" element={<OrderDetail />} />
            </Route>
          </Routes>
        </Suspense>
      </Layout>
    </BrowserRouter>
  );
}

export default App;
```

In this code example, **`lazy`** and **`Suspense`** are imported from React. The **`OrdersRoot`**, **`Orders`**, and **`OrderDetail`** components are no longer directly imported but are instead created via the **`lazy()`** function, which uses the dynamic **`import()`** function to dynamically load the component code when needed.

> **NOTE**
>
> The code for this example can be found at https://packt.link/NIfVW.

Finally, the route definitions are wrapped with the **`Suspense`** component so that React can show some fallback content (**`<p>Loading...</p>`**, in this case) if downloading the code takes a bit longer.

As explained in *Chapter 9*, *Behind the Scenes of React and Optimization Opportunities*, adding lazy loading can improve your React application's performance considerably. You should always consider using lazy loading, but you should not use it for every route. It would be especially illogical for routes that are guaranteed to be loaded early, for instance. In the previous example, it would not make too much sense to lazy load the **`Dashboard`** component since that's the default route (with a path of **`/`**).

But routes that are not guaranteed to be visited at all (or at least not immediately after the website is loaded) are great candidates for lazy loading.

SUMMARY AND KEY TAKEAWAYS

- Routing is a key feature for many React apps.
- With routing, users can visit multiple pages despite being on an **SPA**.
- The most common package that helps with routing is the React Router library (**`react-router-dom`**).

- Routes are defined with the help of the **`Routes`** and **`Route`** components (typically in the **`App`** component, but you can do it anywhere).
- The **`Route`** component takes a **`path`** (for which the route should become active) and an **`element`** (the content that should be displayed) prop.
- Users can navigate between routes by manually changing the URL path, by clicking links or because of programmatic navigation.
- Internal links (i.e., links leading to application routes defined by you) should be created via the **`Link`** or **`NavLink`** components, while links to external resources use the standard **`<a>`** element.
- Programmatic navigation is triggered via the **`navigate()`** function, which is yielded by the **`useNavigate()`** Hook.
- You can define static and dynamic routes: static routes are the default, while dynamic routes are routes where the path (in the route definition) contains a dynamic segment (denoted by a colon, e.g., **`:id`**).
- The actual values for dynamic path segments can be extracted via the **`useParams()`** Hook.
- React Router also supports nested routes, which helps with sharing user interface elements between routes.
- You can use lazy loading to load route-specific code only when the route is actually visited by the user.

WHAT'S NEXT?

Routing is a feature that's not supported by React out of the box but still matters for most React applications. That's why it's included in this book and why the React Router library exists. Routing is a crucial concept that completes your knowledge about the most essential React ideas and concepts, allowing you to build both simple and complex React applications.

This chapter also concludes the list of core React features you must know as a React developer. Of course, you can always dive deeper to explore more patterns and third-party libraries. The next (and last) chapter will share some resources and possible next steps you could dive into after finishing this book.

TEST YOUR KNOWLEDGE!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to the examples that can be found at https://packt.link/xQKlJ:

1. How is routing different from loading content conditionally?
2. What's the purpose of the `Routes` and `Route` components?
3. How should you add links to different routes to your pages?
4. How can dynamic routes (e.g., details for one of many products) be added to your app?
5. How can dynamic route parameter values be extracted (e.g., to load product data)?
6. What's the purpose of nested routes?

APPLY WHAT YOU LEARNED

Apply your knowledge about routing to the following activities.

ACTIVITY 12.1: CREATING A BASIC THREE-PAGE WEBSITE

In this activity, your task is to create a basic first version for a brand-new online shop website. The website must support three main pages:

- A welcome page
- A products overview page that shows a list of available products
- A product details page, which allows users to explore product details

Final website styling, content, and data will be added by other teams, but you should provide some dummy data and default styling. You must also add a main navigation bar at the top of each website page.

The finished pages should look like this:

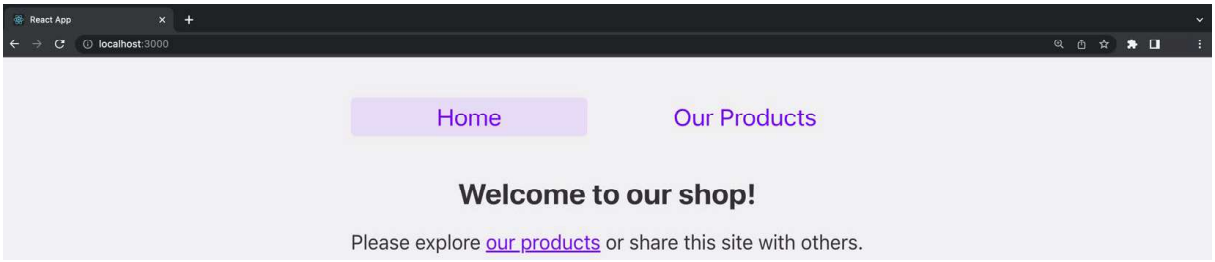

Figure 12.12: The final welcome page

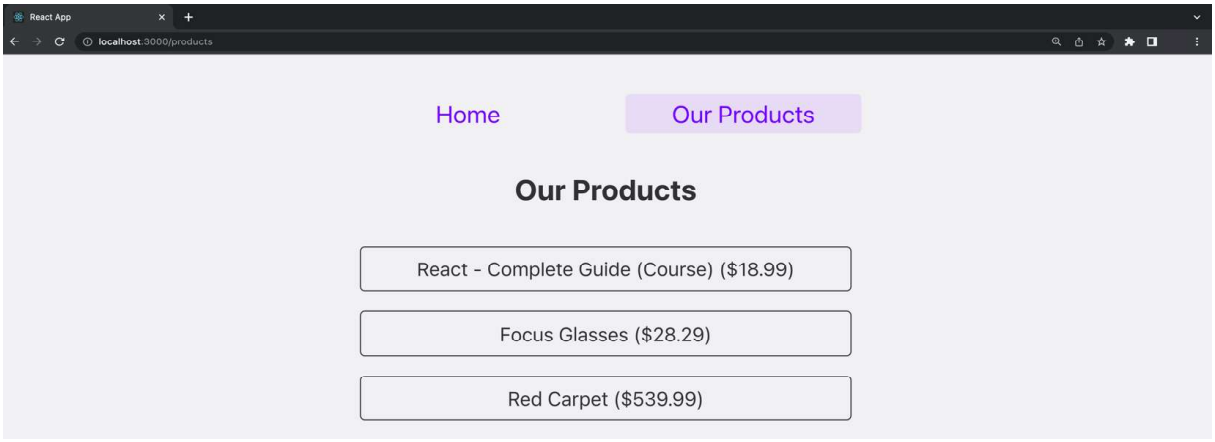

Figure 12.13: The final products page

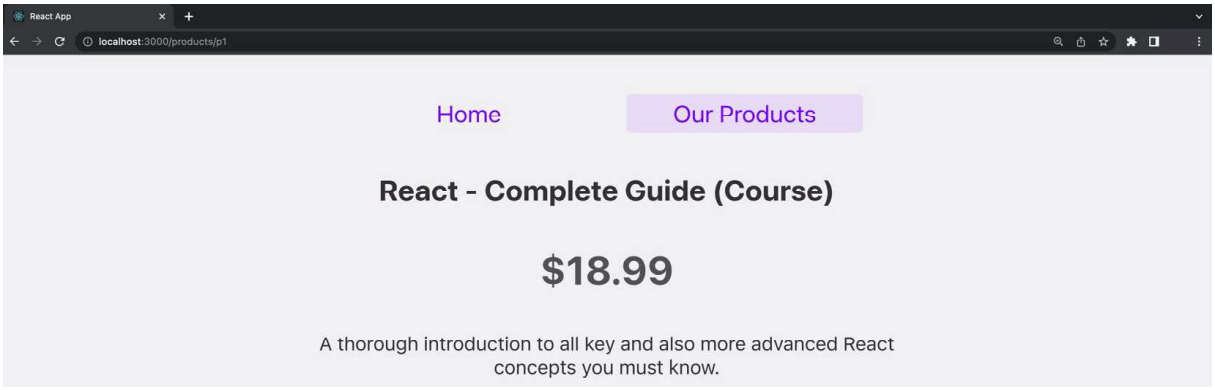

Figure 12.14: The final product details page

> **NOTE**
>
> For this activity, you can, of course, write all CSS styles on your own. But if you want to focus on the React and JavaScript logic, you can also use the finished CSS file from the solution at https://packt.link/Am59v.
>
> If you use that file, explore it carefully to ensure you understand which IDs or CSS classes might need to be added to certain JSX elements of your solution. You can also use the solution's dummy data instead of creating your own dummy product data. You will find the data for this at https://packt.link/xLCx1.

To complete the activity, the solution steps are as follows:

1. Create a new React project and install the React Router package.
2. Create components (with the content shown in the preceding screenshot) that will be loaded for the three required pages.
3. Enable routing and add the route definitions for the three pages.
4. Add a main navigation bar that's visible for all pages.
5. Add all required links and ensure that the navigation bar links reflect whether or not a page is active.

> **NOTE**
>
> The solution to this activity can be found via this link.

ACTIVITY 12.2: ENHANCING THE BASIC WEBSITE

In this activity, your task is to enhance the basic website you built in the previous activity. In addition to all the features added there, this website should also use a main layout that wraps all pages. Your job is to implement this layout with the help of a feature provided by React Router.

You must also improve the initial loading time of the website by conditionally loading code that's not needed immediately once it is required.

Last but not least, you have been asked to implement a *"Not Found"* page that should be displayed for all URL paths that are not supported by the website.

> **NOTE**
>
> If you skipped the previous activity or need a refresher, you can use the solution provided for it as a starting point for this activity. You will find this in the GitHub repository at http://packt.link/wPcgN.

The *"Not Found"* page should look like this:

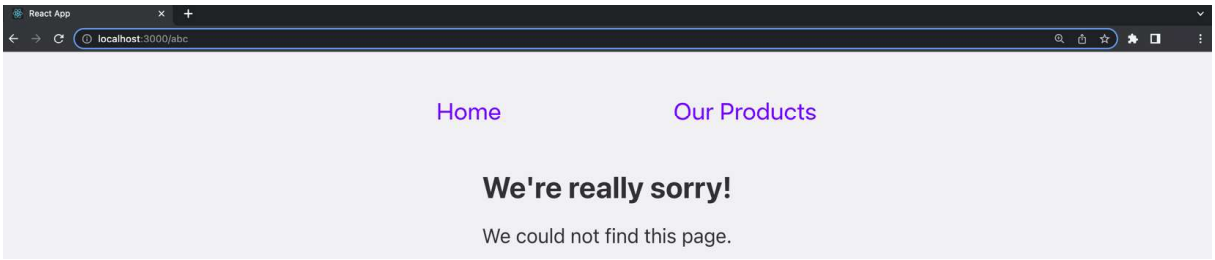

Figure 12.15: The final "Not Found" page

To complete the activity, the solution steps are as follows:

1. Create a new layout component (which includes the main navigation) and use it with the help of React Router's nested routes feature.
2. Add code splitting to the website and load all routes, for which it makes sense, lazily.
3. Add a new Not Found page component to the website. Also, add the respective route definition and consider loading it lazily.

> **NOTE**
>
> The solution to this activity can be found via this link.

13

MANAGING DATA WITH REACT ROUTER

LEARNING OBJECTIVES

By the end of this chapter, you will be able to do the following:

- Use React Router to fetch or send data without using `useEffect()` or `useState()`
- Share data between different routes without using React's context feature
- Update the UI based on the current data submission status
- Create page and action routes
- Improve the user experience by deferring the loading of non-critical data

INTRODUCTION

In the preceding chapter, you learned how to use React Router to load different components for different URL paths. This is an important feature as it allows you to build multipage websites while still using React.

Routing is a crucial feature for many web applications, and React Router is therefore a very important package. But just as most websites need routing, almost all websites need to fetch and manipulate data. For example, HTTP requests in most websites are sent to load data (such as a list of products or blog posts) or to mutate data (for example, to create a product or a blog post).

In *Chapter 8*, *Handling Side Effects*, you learned that you can use the **`useEffect()`** Hook and various other React features to send HTTP requests from inside a React application. But if you're using React Router (specifically, version 6.4. or higher), you get some new, even more powerful tools for working with data.

This chapter will explore which new features are added by React Router 6.4 and how they may be used to simplify the process of fetching or sending data.

DATA FETCHING AND ROUTING ARE TIGHTLY COUPLED

As mentioned previously, most websites do need to fetch (or send) data and most websites do need more than one page. But it's important to realize that these two concepts are typically closely related.

Whenever a user visits a new page (such as **`/posts`**), it's likely that some data will need to be fetched. In the case of a **`/posts`** page, the required data is probably a list of blog posts that is retrieved from a backend server. The rendered React component (such as **`Posts`**) must therefore send an HTTP request to the backend server, wait for the response, handle the response (as well as potential errors) and, ultimately, display the fetched data.

Of course, not all pages need to fetch data. Landing pages, "About Us" pages, or "Terms & Use" pages probably don't need to fetch data when a user visits them. Instead, data on those pages is likely to be static. It might even be included in the source code as it doesn't change frequently.

But many pages do need to get data from a backend every time they're loaded—for instance, "Products", "News", "Events" pages, or other infrequently updated pages like the "User Profile".

And data fetching isn't everything. Most websites also contain features that require data submission—be it a blog post that can be created or updated, product data that's administered, or a user comment that can be added. Hence, sending data to a backend is also a very common use case.

And beyond requests, components might also need to interact with other browser APIs, such as **`localStorage`**. For example, user settings might need to be fetched from storage as a certain page loads.

Naturally, all these interactions happen on pages. But it might not be immediately obvious how tightly data fetching and submission are coupled to routing.

Most of the time, data is fetched when a route becomes active, i.e., when a component (the page component) is rendered for the first time. Sure, users might also be able to click a button to refresh the data, but while this is optional, data fetching upon initial page load is almost always required.

And when it comes to sending data, there is also a close connection to routing. At first sight, it's not clear how it's related because, while it makes sense to fetch data upon page load, it's less likely that you will need to send some data immediately (except perhaps tracking or analytics data).

But it's very likely that *after* sending data, you will want to navigate to a different page, meaning that it's actually the other way around and instead of initiating data fetching as a page loads, you want to load a different page after sending some data. For example, after an administrator entered some product data and submitted the form, they should typically be redirected to a different page (for example, from **`/products/new`** to the **`/products`** page).

The connection between data fetching, submission, and routing can therefore be summarized by the following points:

- **Data fetching** often should be initiated when a route becomes active (if that page needs data)
- After **submitting data** the user should often be redirected to another route

Because these concepts are tightly coupled, React Router (since version 6.4) provides extra features that vastly simplify the process of working with data.

SENDING HTTP REQUESTS WITHOUT REACT ROUTER

Working with data is not just about sending HTTP requests. As mentioned in the previous section, you may also need to store or retrieve data via **`localStorage`** or perform some other operation as a page gets loaded. But sending HTTP requests is an especially common scenario and will therefore be the main use case considered for the majority of this chapter. Nonetheless, it's vital to keep in mind that what you learn in this chapter is not limited to sending HTTP requests.

As you will learn in this chapter, React Router (6.4 or higher) provides various features that help with sending HTTP requests (or using other data fetching and manipulation APIs) and routing, but you can also send HTTP requests (or interact with **`localStorage`** or other APIs) without these features. Indeed, *Chapter 8*, *Handling Side Effects*, already taught you how HTTP requests can be sent from inside React components.

The following snippet exemplifies how a given list of blog posts could be fetched and displayed:

```
import { useState, useEffect } from 'react';

function Posts() {
  const [loadedPosts, setLoadedPosts] = useState();
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState();

  useEffect(() => {
    async function fetchPosts() {
      setIsLoading(true);

      try {
        const response = await fetch(
          'https://jsonplaceholder.typicode.com/posts'
        );

        if (!response.ok) {
          throw new Error('Fetching posts failed.');
        }

        const resData = await response.json();
        setLoadedPosts(resData);
      } catch (error) {
```

```
        setError(error.message);
      }

      setIsLoading(false);
    }

    fetchPosts();
  }, []);

  let content = <p>No posts found.</p>;

  if (isLoading) {
    content = <p>Loading...</p>;
  }

  if (error) {
    content = <p>{error}</p>;
  }

  if (loadedPosts) {
    content = (
      <ul className="posts">
        {loadedPosts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    );
  }

  return (
    <main>
      <h1>Your Posts</h1>
      {content}
    </main>
  );
}
```

This example component uses the same dummy backend server (returning dummy data) that was used in *Chapter 8*, *Handling Side Effects*: **`https://jsonplaceholder.typicode.com`**.

> **NOTE**
>
> You can find the complete code on GitHub at https://packt.link/hlXz6.

This backend returns a list of blog posts, which are then displayed as list items by the **`Posts`** component. However, these items are only displayed if the following statements are **`true`**:

- The HTTP request is done (i.e., **`isLoading`** is **`false`**)
- There is no error (i.e., **`error`** is **`undefined`** or **`null`**)

The request is sent via **`useEffect()`**. The **`fetchPosts()`** function defined in **`useEffect()`** is called when the component renders for the first time. That's the case because **`useEffect()`** has an empty list of dependencies. It therefore only executes once.

This example contains no new concepts. Instead, all these topics (and a very similar example) were covered in *Chapter 8*, *Handling Side Effects*. But even though it's a relatively basic example, where only one request is sent and the response data can be used without any further transformations, quite a bit of code is required to implement this functionality.

And that's where React Router (6.4. or higher) comes in.

LOADING DATA WITH REACT ROUTER

With React Router, the example from the previous chapter can be simplified down to this code snippet:

```
import { useLoaderData } from 'react-router-dom';

function Posts() {
  const loadedPosts = useLoaderData();

  return (
    <main>
      <h1>Your Posts</h1>
```

```
      <ul className="posts">
        {loadedPosts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </main>
  );
}

export default Posts;

export async function loader() {
  const response = await fetch('https://jsonplaceholder.typicode.com/
posts');
  if (!response.ok) {
    throw new Error('Could not fetch posts');
  }
  return response;
}
```

Believe it or not, it really *is* that much less code than in the previous example. Though, to be fair, the content that should be displayed in case of an error is missing here. It's in a separate file (which will be shown later), but it would only add three extra lines of code.

> **NOTE**
>
> Don't try pasting the preceding example into an existing React app—even if that app has React Router 6.4 installed. A special route definition syntax (introduced later in this chapter) is required to enable these new React Router features.

In the preceding code snippet, you see a couple of new features that haven't been covered yet in the book. The **`loader()`** function and the **`useLoaderData()`** Hook are added by React Router. These features, along with many others that will be explored throughout this chapter, have been available since version 6.4 of the React Router package.

With that version (or a more recent one) installed, you can set an extra **`loader`** prop on your route definitions. This prop accepts a function that will be executed by React Router whenever this route is activated:

```
<Route path="/posts" element={<Posts />} loader={() => {...}} />
```

This function can be used to perform any data fetching or other tasks required to successfully display the page component. The logic for getting that required data can therefore be extracted from the component and moved into a separate function.

Since many websites have dozens or even hundreds of routes, adding these loader functions inline on the **`<Route />`** element can quickly lead to complex and confusing route definitions. For this reason, you will typically add (and export) the **`loader()`** function in the same file that contains the component that needs the data.

When setting up the route definitions, you can then import the component and its loader function and use it like this:

```
import Posts, { loader as postsLoader } from './components/Posts';

// … other code …

<Route path="/posts" element={<Posts />} loader={postsLoader} />
```

Assigning an alias (**`postsLoader`**, in this example) to the imported loader function is optional but recommended since you most likely have multiple loader functions from different components, which would otherwise lead to name clashes.

With this **`loader`** defined, React Router will execute the **`loader()`** function whenever a route is activated. To be precise, the **`loader()`** function is called **before** the component function is executed (that is, before the component is rendered).

Figure 13.1: The Posts component is rendered after the loader is executed

This also explains why the **`Posts`** component example at the beginning of this section contained no code that handled any loading state. This is simply because there *was* no loading state since a component function is only executed after its loader has finished (and the data is available). React Router won't finish the page transition until the **`loader()`** function has finished its job (though, as you will learn towards the end of this chapter, there is a way of changing this behavior).

The **`loader()`** function can perform any operation of your choice (such as sending an HTTP request, or reaching out to browser storage via the **`localStorage`** API). Inside that function, you should return the data that should be exposed to the component function. It's also worth noting that the **`loader()`** function can return any kind of data. It may also return a **`Promise`** object that then resolves to any kind of data. In that case, React Router will automatically wait for the **`Promise`** to be fulfilled before executing the related route component function. The **`loader()`** function can thus perform both asynchronous and synchronous tasks.

> **NOTE**
>
> It's important to understand that the **`loader()`** function, like all the other code that makes up your React app, executes on the client side (that is, in the browser of a website visitor). Therefore, you may perform any action that could be performed anywhere else (for example, inside **`useEffect()`**) in your React app as well.
>
> You must not try to run code that belongs to the server side. Directly reaching out to a database, writing to the file system, or performing any other server-side tasks will fail or introduce security risks, meaning that you might accidentally expose database credentials on the client side.

Of course, the component that belongs to a **`loader`** (that is, the component that's part of the same **`<Route />`** definition) needs the data returned by the **`loader`**. This is why React Router offers a new Hook for accessing that data: the **`useLoaderData()`** Hook.

When called inside a component function, this Hook yields the data returned by the **`loader`** that belongs to the component. If that returned data is a **`Promise`**, React Router (as mentioned earlier) will automatically wait for that **`Promise`** to resolve and provide the resolved data when **`useLoaderData()`** is called.

The **`loader()`** function may also return an HTTP response object (or a **`Promise`** resolving to a response). This is the case in the preceding example because the **`fetch()`** function yields a **`Promise`** that resolves to a response. In that instance, **`useLoaderData()`** automatically extracts the response body and provides direct access to the data that was attached to the response.

> **NOTE**
>
> If a response should be returned, the returned object must adhere to the standard **`Response`** interface, as defined here: https://developer.mozilla.org/en-US/docs/Web/API/Response.
>
> Returning responses might be strange at first. After all, the **`loader()`** code is still executed inside the browser (not on a server). Therefore, technically, no request was sent, and no response should be required (since the entire code is executed in the same environment, that is, the browser).
>
> For that reason, you don't have to return a response; you may return any kind of value. But Remix supports the usage of a response object (as a "data vehicle") as an alternative.

`useLoaderData()` can be called in any component rendered by the currently active route component. That may be the route component itself (**`Posts`**, in the preceding example), but it may also be any nested component.

For example, **`useLoaderData()`** can also be used in a **`PostsList`** component that's included in the **`Posts`** component (which has a **`loader`** added to its route definition):

```
import { useLoaderData } from 'react-router-dom';

function PostsList() {
  const loadedPosts = useLoaderData();

  return (
    <main>
      <h1>Your Posts</h1>
      <ul className="posts">
        {loadedPosts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
```

```
      </ul>
    </main>
  );
}
export default PostsList;
```

For this example, the **`Posts`** component file looks like this:

```
import PostsList from '../components/PostsList';

function Posts() {
  return (
    <main>
      <h1>Your Posts</h1>
      <PostsList />
    </main>
  );
}

export default Posts;

export async function loader() {
  const response = await fetch('https://jsonplaceholder.typicode.com/
posts');
  if (!response.ok) {
    throw new Error('Could not fetch posts');
  }
  return response;
}
```

This means that **`useLoaderData()`** can be used in exactly the place where you need the data. The **`loader()`** function can also be defined wherever you want but it must be added to the route where the data is required.

You can't use **`useLoaderData()`** in components where no loader is defined for the currently active route.

ENABLING THESE EXTRA ROUTER FEATURES

If you want to use these data-related React Router features, it's not enough to have version 6.4 or higher installed. This is an important prerequisite, but you also must tweak your route definition code a little bit.

In *Chapter 12*, *Multipage Apps with React Router*, you learned that routes can be defined as follows:

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';

import Posts from './pages/Posts';
import Welcome from './pages/Welcome';

function App() {
  return (
    <BrowserRouter>
      <Routes>
          <Route index element={<Welcome />} />
          <Route path="/posts" element={<Posts />} />
      </Routes>
    </BrowserRouter>
  );
}
```

You can still do that with version 6.4, but you won't be able to use React Router's new data-related features. Instead, to enable these features, the preceding route definition code must be changed to look like this:

```
import {
  createBrowserRouter,
  createRoutesFromElements,
  Route,
  RouterProvider,
} from 'react-router-dom';

import Posts, { loader as postsLoader } from './pages/Posts';
import Welcome from './pages/Welcome';

const router = createBrowserRouter(
  createRoutesFromElements(
    <>
      <Route path="/" element={<Welcome />} />
      <Route path="/posts" element={<Posts />} loader={postsLoader} />
    </>
  )
);
```

```
function App() {
  return <RouterProvider router={router} />;
}
```

Instead of returning **<BrowserRouter>** (which includes the route definitions, wrapped by **<Routes>**), you must now create a **router** object by calling **createBrowserRouter()**.

This function then accepts an array of route definition objects. You can create this array on your own (shown later) or get a valid list of objects by executing **createRoutesFromElements()** and passing your route definition JSX code to that function.

Note that **createRoutesFromElements()** only takes a single element, such as a fragment (see *Chapter 2, Understanding React Components and JSX*), as used in this example, or a **<Route>** element. This single element is wrapped around all other **<Route>** elements.

The created **router** object is then passed as a value for the **router** prop to the **<RouterProvider>** component.

As an alternative to **createRoutesFromElements()**, you can also define your route objects manually, like this:

```
const router = createBrowserRouter([
  { path: '/', element: <Welcome /> },
  { path: '/posts', element: <Posts />, loader: postsLoader },
]);
```

This approach is a bit more concise, though it may be unfamiliar. Essentially, instead of using React components (such as **<Route>**) to define routes, you would use plain JavaScript objects (grouped into an array) for this method.

You can use whichever approach you prefer. Conceptually and feature-wise there is no difference between the two.

LOADING DATA FOR DYNAMIC ROUTES

For most websites, it's unlikely that static, pre-defined routes alone will be sufficient to meet your needs. For instance, if you created a blogging site with exclusively static routes, you would be limited to a simple list of blog posts on **/posts**. To add more details about a selected blog post on routes such as **/posts/1** or **/posts/2** (for posts with different **id** values) you would need to include dynamic routes.

As you learned in the previous chapter, dynamic routes can be defined like this:

```
<Route path="/posts/:id" element={<PostDetails />} />
```

This code still works (when using **`createRoutesFromElements()`**), though you can also use the alternative approach of defining route objects, as mentioned earlier, and define a dynamic route (via **`createBrowserRouter()`**), as shown here:

```
const router = createBrowserRouter([
  // … other routes
  { path: '/posts/:id', element: <PostDetails /> }
]);
```

Of course, React Router also supports data fetching with help of the **`loader()`** function for dynamic routes.

The **`PostDetails`** component can be implemented like this:

```
import { useLoaderData } from 'react-router-dom';

function PostDetails() {
  const post = useLoaderData();
  return (
    <main id="post-details">
      <h1>{post.title}</h1>
      <p>{post.body}</p>
    </main>
  );
}

export default PostDetails;

export async function loader({ params }) {
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts/' + params.id
  );
  if (!response.ok) {
    throw new Error('Could not fetch post for id ' + params.id);
  }
  return response;
}
```

If it looks very similar to the **`Posts`** component in the "*Loading Data with React Router*" section, that's no coincidence. Because the **`loader()`** function works in exactly the same way, there is just one extra feature being used to get hold of the dynamic path segment value: a **`params`** object that's made available by React Router.

When adding a **`loader()`** function to a route definition, React Router calls that function whenever the route becomes active, right before the component is rendered. When executing that function, React Router passes an object that contains extra information as an argument to **`loader()`**.

This object passed to **`loader()`** includes two main properties:

- A **`request`** property that contains an object with more details about the request that led to the route activation
- A **`params`** property that yields an object containing a key-value map of all dynamic route parameters for the active route

The **`request`** object doesn't matter for this example and will be discussed in the next section. But the **`params`** object contains an **`id`** property that carries the **`id`** value of the post for which the route is loaded. The property is named **`id`** because, in the route definition, **`/posts/:id`** was chosen as a path. If a different placeholder name had been chosen, a property with that name would have been available on **`params`** (for example, for **`/posts/:postId`**, this would be **`params.postId`**). This behavior is similar to the **`params`** object yielded by **`useParams()`**, as explained in *Chapter 12, Multipage Apps with React Router*.

With help of the **`params`** object and the post **`id`**, the appropriate post **`id`** can be included in the outgoing request URL (for the **`fetch()`** request), and hence the correct post data can be loaded from the backend API. Once the data arrives, React Router will render the **`PostDetails`** component and expose the loaded post via the **`useLoaderData()`** Hook.

LOADERS, REQUESTS, AND CLIENT-SIDE CODE

In the preceding section, you learned about a **`request`** object being provided to the **`loader()`** function. Getting such a **`request`** object might be confusing because React Router is a client-side library—all the code executes in the browser, not on a server. Therefore, no request should reach the React app (as HTTP requests are sent from the client to the server, not between JavaScript functions on the client side).

And, indeed, there is no request being sent via HTTP. Instead, React Router creates a request object via the browser's built-in **`Request`** interface to use it as a "data vehicle." This request is not sent via HTTP, but it's used as a value for the **`request`** property on the data object that is passed to your **`loader()`** function.

> **NOTE**
>
> For more information on the built-in **`Request`** interface, visit https://developer.mozilla.org/en-US/docs/Web/API/Request.

This **`request`** object will be unnecessary in many **`loader`** functions, but there are occasional scenarios in which you can extract useful information from that object—information that might be needed in the **`loader`** to fetch the right data.

For example, you can use the **`request`** object and its **`url`** property to get access to any search parameters (query parameters) that may be included in the currently active page's URL:

```
export async function loader({ request }) {
  // e.g. for localhost:3000/posts?sort=desc
  const sortDirection = new URL(request.url).searchParams.get('sort');

  // Example: Fetch sorted posts, based on local 'sort' query param value
  const response = await fetch(
    'https://example.com/posts?sorting=' + sortDirection
  );
  return response;
}
```

In this code snippet, the **`request`** value is used to get hold of a query parameter value that's used in the React app URL. That value is then used in an outgoing request.

However, it is vital that you keep in mind that the code inside your **`loader()`** function, just like all your other React code, always executes on the client side (at least, as long as you don't combine React with any other frameworks like NextJS or Remix).

> **NOTE**
>
> These frameworks are beyond the scope of this book. Both NextJS and Remix build up on top of React and, for example, add server-side rendering of React components. Visit https://nextjs.org or https://remix.run for more information. In addition, the author also offers courses for both frameworks. For NextJS, visit https://acad.link/nextjs and, for Remix, visit https://acad.link/remix.

LAYOUTS REVISITED

React Router supports the concept of layout routes. These are routes that contain other routes and render those other routes as nested children, and as you may recall, this concept was introduced in *Chapter 12, Multipage Apps with React Router*.

With React Router 6.4, a layout route can be defined like this:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    children: [
      { index: true, element: <Welcome /> },
    ]
  }
]);
```

The **`index`** route is a child route of the **`/`** route, which in turn is the layout route in this example. The **`Root`** component could look like this:

```
function Root() {
  return (
    <>
      <header>
        <MainNavigation />
      </header>
```

```
      <Outlet />
    </>
  );
}
```

As mentioned, layout routes were introduced in the previous chapter. But when using the extra data capabilities offered by React Router, there are two noteworthy changes:

- Unlike with **`<BrowserRouter />`**, if you need some shared layout, you can't wrap a React component around your route definitions. **`createBrowserRouter()`** only accepts React fragments and **`<Route />`** elements—no other components. For that reason, you must use a layout route as shown in the previous example instead.
- Layout routes can also be used to share data across routes without using React's context feature.

Because of the first point, you'll typically use more layout routes than you did prior to the release of React Router 6.4. Before that version, you could simply wrap any component you wanted around your route definitions. For example, the following code works without issue:

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';

import Layout from './components/shared/Layout';
import Dashboard from './routes/Dashboard';
import Orders from './routes/Orders';

function App() {
  return (
    <BrowserRouter>
      <Layout>
        <Routes>
          <Route path="/" element={<Dashboard />} />
          <Route path="/orders" element={<Orders />} />
        </Routes>
      </Layout>
    </BrowserRouter>
  );
}

export default App;
```

Since **`BrowserRouter`** does not support the extra data capabilities (such as the **`loader()`** function) you must use **`createBrowserRouter()`** instead. But **`createBrowserRouter()`** does not accept anything but route definitions. That's why shared layouts must be implemented via layout routes instead of wrapper components.

The second point is the more interesting one, though. Layout routes can be used for sharing data across routes.

Consider this example website:

Figure 13.2: A website with a header, a sidebar, and some main content

This website has a header with a navigation bar, a sidebar showing a list of available posts, and a main area that displays the currently selected blog post.

> **NOTE**
>
> The dummy data shown in *Figure 13.2* is fetched from https://jsonplaceholder.typicode.com/posts. You will find the complete source code for this example at https://packt.link/Ef8LS.

This example includes two layout routes:

- The root layout route, which includes the top navigation bar that is shared across all pages
- A posts layout route, which includes the sidebar and the main content of its child routes (for example, the details for a selected post)

The route definitions code looks like this:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />, // main layout, adds navigation bar
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        element: <PostsLayout />, // posts layout, adds posts sidebar
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          { path: ':id', element: <PostDetails />, loader:
postDetailLoader },
        ],
      },
    ],
  },
]);
```

With this setup, both the **`<Posts />`** and the **`<PostDetails />`** components are rendered next to the sidebar (since the sidebar is part of the **`<PostsLayout />`** element).

The interesting part is that the **`/posts`** route (i.e., the layout route) loads the post data, as it has the **`postsLoader`** assigned to it, and so the **`PostsLayout`** component file looks like this:

```
import { Outlet, useLoaderData } from 'react-router-dom';

import PostsList from '../components/PostsList';

function PostsLayout() {
  const loadedPosts = useLoaderData();
  return (
    <div id="posts-layout">
      <nav>
        <PostsList posts={loadedPosts} />
      </nav>
      <main>
        <Outlet />
      </main>
    </div>
  );
}

export default PostsLayout;

export async function loader() {
  const response = await fetch('https://jsonplaceholder.typicode.com/
posts');
  if (!response.ok) {
    throw new Error('Could not fetch posts');
  }
  return response;
}
```

Since layout routes are also regular routes, you can add **`loader()`** functions and **`useLoaderData()`** just as you could in any other route. But because layout routes are activated for multiple child routes, their data is also displayed for different routes. In the preceding example, the list of blog posts is always displayed on the left side of the screen, no matter if a user visits **`/posts`** or **`/posts/10`**:

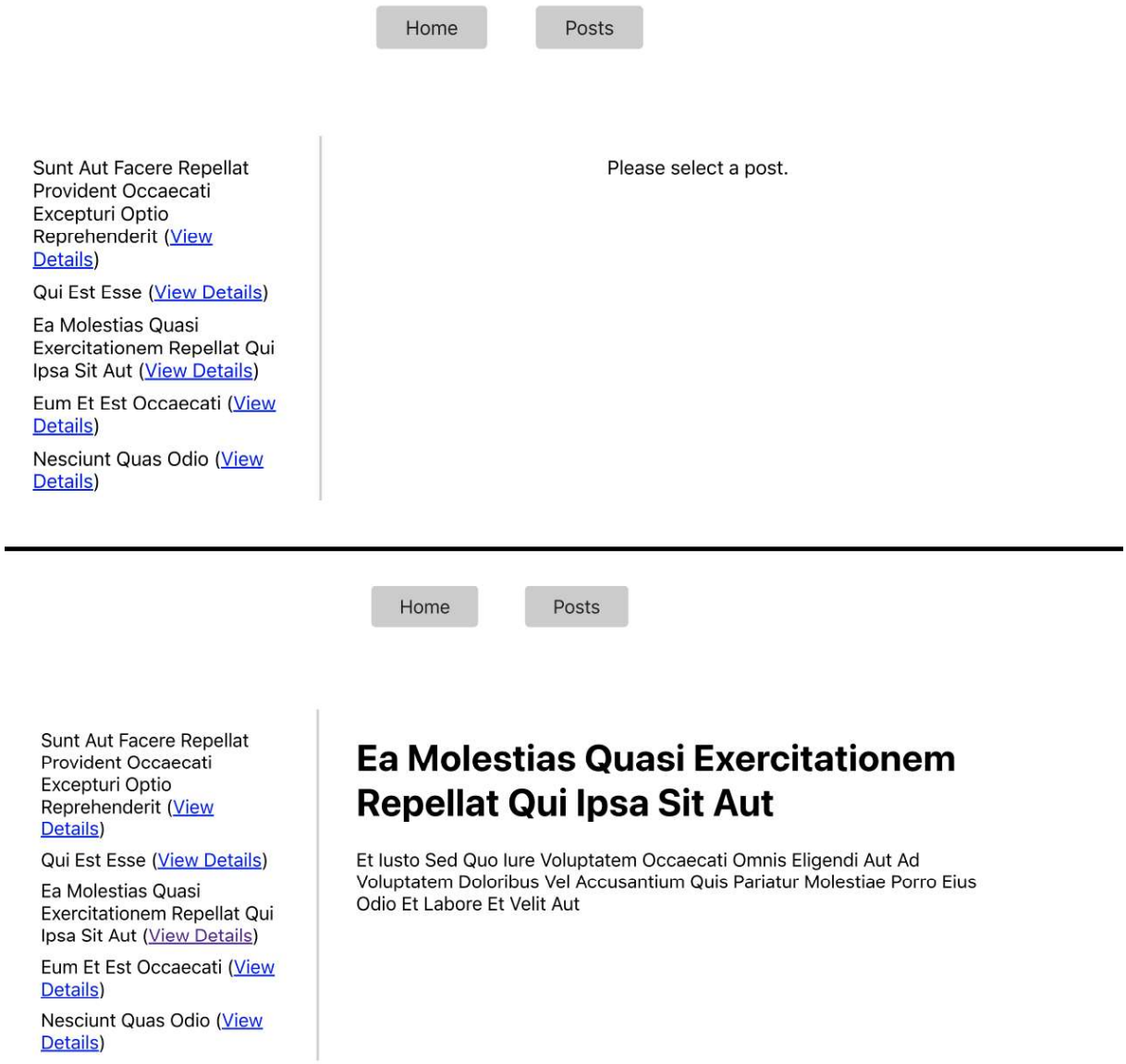

Figure 13.3: The same layout and data are used for different child routes

In this screenshot, the layout and data used do not change as different child routes are activated. React Router also avoids unnecessary data refetching (for the blog posts list data) as you switch between child routes. It's smart enough to realize that the surrounding layout hasn't changed.

REUSING DATA ACROSS ROUTES

Layouts do not just help you share data by sharing components that use data (such as the sidebar in the previous example). They also allow you to load data in a layout route and use it in a child route.

For example, the **`PostDetails`** component (that is, the component that's rendered for the **`/posts/:id`** route) needs the data for a single post, and that data can be retrieved via a **`loader`** attached to the **`/posts/:id`** route:

```
export async function loader({ params }) {
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts/' + params.id
  );
  if (!response.ok) {
    throw new Error('Could not fetch post for id ' + params.id);
  }
  return response;
}
```

This example was discussed earlier in this chapter in the *Loading Data for Dynamic Routes* section. This approach is fine, but in some situations, this extra HTTP request can be avoided.

In the example from the previous section, the **`PostsLayout`** route already fetched a list of all posts. That layout component is also active for the **`PostDetails`** route. In such a scenario, fetching a single post is unnecessary, since all the data has already been fetched for the list of posts. Of course, refetching would be required if the request for the list of posts didn't yield all the data required by **`PostDetails`**.

But if all the data is available, React Router allows you to tap into the loader data of a parent route component via the **`useRouteLoaderData()`** Hook.

This Hook can be used like this:

```
const posts = useRouteLoaderData('posts');
```

`useRouteLoaderData()` requires a route identifier as an argument. It requires an identifier assigned to the ancestor route that contains the data that should be reused. You can assign such an identifier via the **`id`** property to your routes as part of the route definitions code:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />, // main layout, adds navigation bar
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        id: 'posts', // the id value is up to you
        element: <PostsLayout />, // posts layout, adds posts sidebar
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          { path: ':id', element: <PostDetails />, loader:
postDetailLoader },
        ],
      },
    ],
  },
]);
```

The **`useRouteLoaderData()`** Hook then returns the same data **`useLoaderData()`** yields in that route to which you added the **`id`**. In this example, it would provide a list of blog posts.

In **`PostDetails`**, this list can be used like this:

```
import { useParams, useRouteLoaderData } from 'react-router-dom';

function PostDetails() {
  const params = useParams();
  const posts = useRouteLoaderData('posts');
  const post = posts.find((post) => post.id.toString() === params.id);
  return (
    <main id="post-details">
      <h1>{post.title}</h1>
      <p>{post.body}</p>
```

```
    </main>
  );
}

export default PostDetails;
```

The **`useParams()`** Hook is used to get access to the dynamic route parameter value, and the **`find()`** method is used on the list of posts to identify a single post with a fitting **`id`** property. In this example, you would thus avoid sending an unnecessary HTTP request by reusing data that's already available.

HANDLING ERRORS

In the first example at the very beginning of this chapter (where the HTTP request was sent with help of **`useEffect()`**), the code did not just handle the success case but also possible errors. In all the React Router-based examples since then, error handling was omitted. Error handling was not discussed up to this point because while React Router plays an important role in error handling, it's vital to first gain a solid understanding of how React Router 6.4 works in general and how it helps with data fetching. But, of course, errors can't always be avoided and definitely should not be ignored.

Thankfully, handling errors is also very straightforward and easy when using React Router's data capabilities. You can set an **`errorElement`** property on your route definitions and define the element that should be rendered when an error occurs:

```
// ... other imports
import Error from './components/Error';

const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    errorElement: <Error />,
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        id: 'posts',
        element: <PostsLayout />,
        loader: postsLoader,
        children: [
```

```
          { index: true, element: <Posts /> },
          { path: ':id', element: <PostDetails /> },
        ],
      },
    ],
  },
]);
```

This **`errorElement`** property can be set on any route definition of your choice, or even multiple route definitions simultaneously. React Router will render the **`errorElement`** of the route closest to the place where the error was thrown.

In the preceding snippet, no matter which route produced an error, it would always be the root route's **`errorElement`** that was displayed (since that's the only route definition with an **`errorElement`**). But if you also added an **`errorElement`** to the **`/posts`** route, and the **`:id`** route produced an error, it would be the **`errorElement`** of the **`/posts`** route that was shown on the screen, as follows:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    errorElement: <Error />, // used for any errors not handled by nested
routes
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        id: 'posts',
        element: <PostsLayout />,
        // used if /posts or /posts/:id throws an error
        errorElement: <PostsError />,
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          { path: ':id', element: <PostDetails /> },
        ],
      },
    ],
  },
]);
```

This allows you, the developer, to set up fine-grained error handling.

Inside the component used as a value for the **`errorElement`**, you can get access to the error that was thrown via the **`useRouteError()`** Hook:

```
import { useRouteError } from 'react-router-dom';

function Error() {
  const error = useRouteError();

  return (
    <>
      <h1>Oh no!</h1>
      <p>An error occurred</p>
      <p>{error.message}</p>
    </>
  );
}

export default Error;
```

With this simple yet effective error-handling solution, React Router allows you to avoid managing error states yourself. Instead, you simply define a standard React element (via the **`element`** prop) that should be displayed when things go right and an **`errorElement`** to be displayed if things go wrong.

ONWARD TO DATA SUBMISSION

Thus far, you've learned a lot about data fetching. But as mentioned earlier in this chapter, React Router also helps with data submission.

Consider the following example component:

```
function NewPost() {
  return (
    <form id="post-form">
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" name="title" />
      </p>
      <p>
        <label htmlFor="text">Text</label>
        <textarea id="text" name="text" rows={3} />
```

```
      </p>
      <button>Save Post</button>
    </form>
  );
}

export default NewPost;
```

This component renders a **`<form>`** element that allows users to enter the details for a new post. Due to the following route configuration, the component is displayed whenever the **`/posts/new`** route becomes active:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    errorElement: <Error />,
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        id: 'posts',
        element: <PostsLayout />,
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          { path: ':id', element: <PostDetails /> },
          { path: 'new', element: <NewPost /> },
        ],
      },
    ],
  },
]);
```

Without React Router's data-related features, you would typically handle form submission like this:

```

function NewPost() {
  const titleInput = useRef();
  const textInput = useRef();
  const navigate = useNavigate();
```

```
  async function submitHandler(event) {
    event.preventDefault(); // prevent the browser from sending a HTTP
request
    const enteredTitle = titleInput.current.value;
    const enteredText = textInput.current.value;
    const postData = {
      title: enteredTitle,
      text: enteredText
    };

    await fetch('https://jsonplaceholder.typicode.com/posts', {
      method: 'POST',
      body: JSON.stringify(postData),
      headers: {'Content-Type': 'application/json'}
    });
    navigate('/posts');
  }

  return (
    <form onSubmit={submitHandler}>
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" ref={titleInput} />
      </p>
      <p>
        <label htmlFor="text">Text</label>
        <textarea id="text" rows={3} ref={textInput} />
      </p>
      <button>Save Post</button>

    </form>
  );
}
```

Just as before when fetching data, this requires a lot of code and logic added to the component. You must manually handle the form submission, input data extraction, sending the HTTP request, and transitioning to a different page after sending the HTTP request. All these things happen inside the component. In addition, you might also need to manage loading state and potential errors (excluded in the preceding example).

Again, React Router offers some help. Where a **`loader()`** function can be added to handle data loading, an **`action()`** function can be defined to handle data submission.

When using the new **`action()`** function, the preceding example component looks like this:

```
import { Form, redirect } from 'react-router-dom';

function NewPost() {
  return (
    <Form method="post" id="post-form">
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" name="title" />
      </p>
      <p>
        <label htmlFor="text">Text</label>
        <textarea id="text" rows={3} name="text" />
      </p>
      <button>Save Post</button>
    </Form>
  );
}

export default NewPost;

export async function action({ request }) {
  const formData = await request.formData();
  const postData = Object.fromEntries(formData);
  await fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify(postData),
    headers: { 'Content-Type': 'application/json' },
  });
  return redirect('/posts');
}
```

This code is shorter and, most importantly, simpler (even though it might not look simpler yet, since it includes a couple of new features).

Besides the addition of the **`action()`** function, the example code snippet includes the following important changes and features:

- A **`<Form>`** component that's used instead of **`<form>`**.
- The **`method`** prop is set on the **`<Form>`** (to **`"post"`**), and the **`onSubmit`** event handler was removed.
- The form input elements have names assigned to them (via the **`name`** prop), and the refs were removed.
- In the action, the entered data is extracted via a **`formData()`** method (combined with **`Object.fromEntries()`**).
- The user is redirected via a newly added **`redirect()`** function (instead of **`useNavigate()`** and **`navigate()`**).

But what are these elements about?

WORKING WITH ACTION() AND FORM DATA

Just like **`loader()`**, **`action()`** is a special function that can be added to route definitions, as follows:

```
import NewPost, { action as newPostAction } from './components/NewPost';

// ...

{ path: 'new', element: <NewPost />, action: newPostAction },
```

With the **`action`** prop set on a route definition, the assigned function is automatically called whenever a **`<Form>`** targeting this route is submitted. **`Form`** is a component provided by React Router that should be used instead of the default **`<form>`** element.

Internally, **`Form`** uses the default **`<form>`** element but prevents the browser default of creating and sending an HTTP request upon form submission. Instead, React Router creates a **`FormData`** object and calls the **`action()`** function defined for the route that's targeted by the **`<Form>`**, passing a request object, based on the built-in **`Request`** interface, to it. The passed request object contains the form data generated by React Router.

> **NOTE**
>
> For further reading on React Router's **`FormData`** object, visit https://developer.mozilla.org/en-US/docs/Web/API/FormData.

The form data object that is created by React Router includes all form input values entered into the submitted form. To be registered, an input element such as **`<input>`**, **`<select>`**, or **`<textarea>`** must have the **`name`** attribute assigned to it. The values set for those **`name`** attributes can later be used to extract the entered data.

The **`Form`** component also sets the HTTP method of the request object to the value assigned to the **`method`** prop. It's important to understand that the request is not sent via HTTP since **`action()`**, just like **`loader()`** or the component function, still executes in the browser rather than the server. React Router simply uses this request object as a "data vehicle" for passing information (such as the form data or chosen method) to the **`action()`** function.

The **`action()`** function then receives an object with a **`request`** property that contains the created request object with the included form data:

```
export function action({ request }) {
 // do something with 'request' (e.g., extract data)
}
```

The **`method`** property could be used inside the **`action()`** function to perform different actions for different forms. For example, one form could generate and pass a **`POST`** request via **`<Form method="post">`**, whereas another form might yield a **`DELETE`** request via **`<Form method="delete">`**. The same action could handle both form submissions and perform different tasks based on the value of the **`method`** property:

```
export function action({ request }) {
  if (request.method === 'DELETE') {
```

```
    // do something, e.g., send a "delete post" request to backend API
  }

  if (request.method === 'POST') {
    // do something, e.g., send a "create new post" request to backend
API
  }
}
```

But while using the **`method`** for performing different tasks for different forms can be very useful, the form data that's attached to the **`request`** object is often even more important. The **`request`** object can be used to extract the values entered into the form input fields like this:

```
export async function action({ request }) {
  const formData = await request.formData();
  const postData = Object.fromEntries(formData);

  // ...
}
```

The built-in **`formData()`** method yields a **`Promise`** that resolves to an object that offers a **`get()`** method that could be used to get an entered value by its identifier (that is, by the **`name`** attribute value set on the input element). For example, the value entered into **`<input name="title">`** could be retrieved via **`formData.get('title')`**.

Alternatively, you can follow the approach chosen in the preceding code snippet and convert the **`formData`** object to a simple key-value object via **`Object.fromEntries(formData)`**. This object (**`postData`**, in the preceding example) contains the names set on the form input elements as properties and the entered values as values for those properties (meaning that **`postData.title`** would yield the value entered in **`<input name="title">`**).

The extracted data can then be used for any operations of your choice. That could be an extra validation step or an HTTP request sent to some backend API, where the data may get stored in a database or file:

```
export async function action({ request }) {
  const formData = await request.formData();
  const postData = Object.fromEntries(formData);
  await fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify(postData),
```

```
    headers: { 'Content-Type': 'application/json' },
  });
  return redirect('/posts');
}
```

Finally, once all intended steps were performed, the **`action()`** function should return a value—any value of any type. Though, as with the **`loader()`** function, you may also return a response.

Indeed, for actions, it's highly likely that you will want to navigate to a different page once the action has been performed (that is, once the HTTP request to an API has been sent). This may be required to navigate the user away from the data input page to a page that displays all available data entries (for example, from **`/posts/new`** to **`/posts`**).

To simplify this common pattern, React Router provides a **`redirect()`** function that yields a response object that causes React Router to switch to a different route. You can therefore return the result of calling **`redirect()`** in your **`action()`** function to ensure that the user is navigated to a different page. It's the equivalent of calling **`navigate()`** (via **`useNavigate()`**) when manually handling form submissions.

RETURNING DATA INSTEAD OF REDIRECTING

As mentioned, your **`action()`** functions may return anything. You don't have to return a response object. And while it is quite common to return a redirect response, you may occasionally want to return some raw data instead.

One scenario in which you might **`not`** want to redirect the user is after validating the user's input. Inside the **`action()`** function, before sending the entered data to some API, you may wish to validate the provided values first. If an invalid value (such as an empty title) is detected, a great user experience is typically achieved by keeping the user on the route with the **`<Form>`**. The values entered by the user shouldn't be cleared and lost; instead, the form should be updated to present useful validation error information to the user. This information can be passed from the **`action()`** to the component function so that it can be displayed there (for example, next to the form input fields).

In situations like this, you can return a "normal" value (that is, not a redirect response) from your **`action()`** function:

```
export async function action({ request }) {
  const formData = await request.formData();
  const postData = Object.fromEntries(formData);
```

```
  let validationErrors = [];

  if (postData.title.trim().length === 0) {
    validationErrors.push('Invalid post title provided.')
  }
  if (postData.text.trim().length === 0) {
    validationErrors.push('Invalid post text provided.')
  }

  if (validationErrors.length > 0) {
    return validationErrors;
  }

  await fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify(postData),
    headers: { 'Content-Type': 'application/json' },
  });
  return redirect('/posts');
}
```

In this example, a **`validationErrors`** array is returned if the entered **`title`** or **`text`** values are empty.

Data returned by an **`action()`** function can be used in the route component (or any other nested component) via the **`useActionData()`** Hook:

```
import { Form, redirect, useActionData } from 'react-router-dom';

function NewPost() {
  const validationErrors = useActionData();

  return (
    <Form method="post" id="post-form">
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" name="title" />
      </p>
      <p>
        <label htmlFor="text">Text</label>
        <textarea id="text" name="text" rows={3} />
      </p>
```

```
      <ul>
        {validationErrors &&
          validationErrors.map((err) => <li key={err}>{err}</li>)}
      </ul>
      <button>Save Post</button>
    </Form>
  );
}
```

`useActionData()` works a lot like **`useLoaderData()`**, but unlike **`useLoaderData()`**, it's not guaranteed to yield any data. This is because while **`loader()`** functions always get called before the route component is rendered, the **`action()`** function only gets called once the **`<Form>`** is submitted.

In this example, **`useActionData()`** is used to get access to the **`validationErrors`** returned by **`action()`**. If **`validationErrors`** is truthy (that is, is not **`undefined`**), the array will be mapped to a list of error items that are displayed to the user:

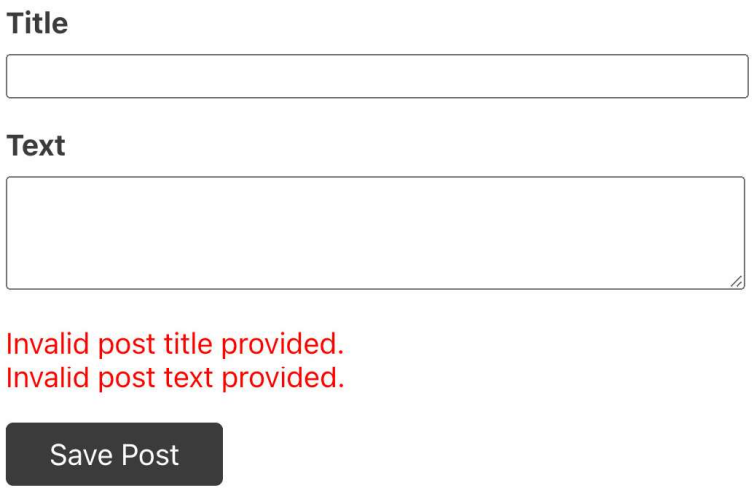

Figure 13.4: Validation errors are output below the input fields

The **`action()`** function is therefore quite versatile in that you can use it to perform an action and redirect away as well as to conduct more than one operation and return different values for different use cases.

CONTROLLING WHICH <FORM> TRIGGERS WHICH ACTION

In *Chapter 12, Multipage Apps with React Router*, you learned that when **`<Form>`** is used instead of **`<form>`**, React Router will execute the targeted **`action()`** function. But which **`action()`** function is targeted by **`<Form>`**?

By default, it's the **`action()`** function assigned to the route that also renders the form. Consider this route definition:

```
{ path: '/posts/new', element: <NewPost />, action: newPostAction }
```

With this definition, the **`newPostAction()`** function would be triggered whenever any **`<Form>`** inside of the **`NewPost`** component (or any nested component) is submitted.

In many cases, this default behavior is exactly what you want. But you can also target **`action()`** functions defined on other routes, and these can be triggered by setting the **`action`** prop on **`<Form>`** to the path of the route that contains the **`action()`** that should be executed:

```
// form rendered in a component that belongs to /posts
<Form method="post" action="/save-data">
  ...
</Form>
```

This form would lead to the **`action`** belonging to the **`/save-data`** route to be executed—even though the **`<Form>`** component is rendered as part of a component that belongs to a different route (e.g., **`/posts`**).

It is worth noting, though, that targeting a different route will lead to a page transition to that route's path, even if your action does not return a redirect response. In a later section of this chapter, entitled *"Behind-the-Scenes Data Fetching and Submission"*, you will learn how that behavior could be avoided.

REFLECTING THE CURRENT NAVIGATION STATUS

After submitting a form, the **`action()`** function that's triggered may need some time to perform all intended operations. Sending HTTP requests to APIs in particular can take up to a few seconds.

Of course, it's not a great user experience if the user doesn't get any feedback about the current data submission status. It's not immediately clear if anything happened at all after the submit button was clicked.

For that reason, you might want to show a loading spinner or update the button caption while the **`action()`** function is running. Indeed, one common way of providing user feedback is to disable the submit button and change its caption like this:

Figure 13.5: The submit button is grayed out

You can get the current React Router status (that is, whether it's currently transitioning to another route or executing an **`action()`** function) via the **`useNavigation()`** Hook. This Hook provides a navigation object that contains various pieces of routing-related information.

Most importantly, this object has a **`state`** property that yields a string describing the current navigation status. This property is set to one of the following three possible values:

- **`submitting`**: If an **`action()`** function is currently executing
- **`loading`**: If a **`loader()`** function is currently executing (for example, because of a **`redirect()`** response)
- **`idle`**: If no **`action()`** or **`loader()`** functions are currently being executed

You can therefore use this **`state`** property to find out whether React Router is currently navigating to a different page or executing an **`action()`**. Hence, the submit button can be updated as shown in the preceding screenshot via this code:

```
import { Form, redirect, useActionData, useNavigation } from 'react-
router-dom';

function NewPost() {
  const validationErrors = useActionData();

  const navigation = useNavigation();
```

```
  const isSubmitting = navigation.state !== 'idle';

  return (
    <Form method="post" id="post-form">
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" name="title" />
      </p>
      <p>
        <label htmlFor="text">Text</label>
        <textarea id="text" name="text" rows={3} />
      </p>
      <ul>
        {validationErrors &&
          validationErrors.map((err) => <li key={err}>{err}</li>)}
      </ul>
      <button disabled={isSubmitting}>
        {isSubmitting ? 'Saving...' : 'Save Post'}
      </button>
    </Form>
  );
}
```

In this example, the **`isSubmitting`** constant is **`true`** if the current navigation state is anything but **`'idle'`**. This constant is then used to disable the submit button (via the **`disabled`** attribute) and adjust the button's caption.

SUBMITTING FORMS PROGRAMMATICALLY

In some cases, you won't want to instantly trigger an **`action()`** when a form is submitted—for example, if you need to ask the user for confirmation first such as when triggering actions that delete or update data.

For such scenarios, React Router allows you to submit a form (and therefore trigger an **`action()`** function) programmatically. Instead of using the **`Form`** component provided by React Router, you handle the form submission manually using the default **`<form>`** element. As part of your code, you can then use a **`submit()`** function provided by React Router's **`useSubmit()`** Hook to trigger the **`action()`** manually once you're ready for it.

Consider this example:

```
import {
  redirect,
  useParams,
  useRouteLoaderData,
  useSubmit,
} from 'react-router-dom';

function PostDetails() {
  const params = useParams();
  const posts = useRouteLoaderData('posts');
  const post = posts.find((post) => post.id.toString() === params.id);

  const submit = useSubmit();

  function submitHandler(event) {
    event.preventDefault();

    const proceed = window.confirm('Are you sure?');

    if (proceed) {
      submit(
        { message: 'Your data, if needed' },
        {
          method: 'delete',
        }
      );
    }
  }

  return (
    <main id="post-details">
      <h1>{post.title}</h1>
      <p>{post.body}</p>
```

```
      <form onSubmit={submitHandler}>
        <button>Delete</button>
      </form>
    </main>
  );
}

export default PostDetails;

// action must be added to route definition!
export async function action({ request }) {
  const formData = await request.formData();
  console.log(formData.get('message'));
  console.log(request.method);
  return redirect('/posts');
}
```

In this example, the **`action()`** is manually triggered by programmatically submitting data via the **`submit()`** function provided by **`useSubmit()`**. This approach is required as it would otherwise be impossible to ask the user for confirmation (via the browser's **`window.confirm()`** method).

Because data is submitted programmatically, the default **`<form>`** element should be used and the **`submit`** event handled manually. As part of this process, the browser's default behavior of sending an HTTP request must also be prevented manually.

Typically, using **`<Form>`** instead of programmatic submission is preferable. But in certain situations, such as the preceding example, being able to control form submission manually can be useful.

BEHIND-THE-SCENES DATA FETCHING AND SUBMISSION

There also are situations in which you may need to trigger an action or load data without causing a page transition.

A "Like" button would be an example. When it's clicked, a process should be triggered in the background (such as storing information about the user and the liked post), but the user should not be directed to a different page:

Figure 13.6: A like button below a post

To achieve this behavior, you could wrap the button into a **`<Form>`** and, at the end of the **`action()`** function, simply redirect back to the page that is already active.

But technically, this would still lead to an extra navigation action. Therefore, **`loader()`** functions would be executed and other possible side-effects might occur (the current scroll position could be lost, for example). For that reason, you might want to avoid this kind of behavior.

Thankfully, React Router offers a solution: the **`useFetcher()`** Hook, which yields an object that contains a **`submit()`** method. Unlike the **`submit()`** function provided by **`useSubmit()`**, the **`submit()`** method yielded by **`useFetcher()`** is meant for triggering actions (or **`loader()`** functions) without starting a page transition.

A "Like" button, as described previously, can be implemented like this (with help of **`useFetcher()`**):

```
import {
  // ... other imports
  useFetcher,
} from 'react-router-dom';
```

```
import { FaHeart } from 'react-icons/fa';

function PostDetails() {
  // ... other code & logic

  const fetcher = useFetcher();

  function likePostHandler() {
    fetcher.submit(null, {
      method: 'post',
      action: `/posts/${post.id}/like`, // targeting an action on another
route
    });
  }

  return (
    <main id="post-details">
      <h1>{post.title}</h1>
      <p>{post.body}</p>
      <p>
        <button className="icon-btn" onClick={likePostHandler}>
          <FaHeart />
          <span>Like this post</span>
        </button>
      </p>
      <form onSubmit={submitHandler}>
        <button>Delete</button>
      </form>
    </main>
  );
}
```

The **`fetcher`** object returned by **`useFetcher()`** has various properties. For example, it also contains properties that provide information about the current status of the triggered action or loader (including any data that may have been returned).

But this object also includes two important methods:

- **`load()`**: To trigger the **`loader()`** function of a route (e.g., **`fetcher.load('/route-path')`**)
- **`submit()`**: To trigger an **`action()`** function with the provided data and configuration

In the code snippet above, the **`submit()`** method is called to trigger the action defined on the **`/posts/<post-id>/like`** route. Without **`useFetcher()`** (i.e., when using **`useSubmit()`** or **`<Form>`**), React Router would switch to the selected route path when triggering its action. With **`useFetcher()`**, this is avoided, and the action of that route can be called from inside another route (meaning the action defined for **`/posts/<post-id>/like`** is called while the **`/posts/<post-id>`** route is active).

This also allows you to define routes that don't render any element (that is, in which there is no page component) and instead only contain a **`loader()`** or **`action()`** function. For example, the **`/posts/<post-id>/like`** route file looks like this:

```
export function action({ params }) {
  console.log('Triggered like action.');
  console.log(`Liking post with id ${params.id}.`);
  // Do anything else
  // May return data or response, including redirect() if needed
}
```

It's registered as a route as follows:

```
import { action as likeAction } from './pages/like';
// ...
{ path: ':id/like', action: likeAction },
```

This works because this **`action()`** is only triggered via the **`submit()`** method provided by **`useFetcher()`**. **`<Form>`** and the **`submit()`** function yielded by **`useSubmit()`** would instead initiate a route transition to **`/posts/<post-id>/like`**. Without the **`element`** property being set on the route definition, this transition would lead to an empty page, as shown here:

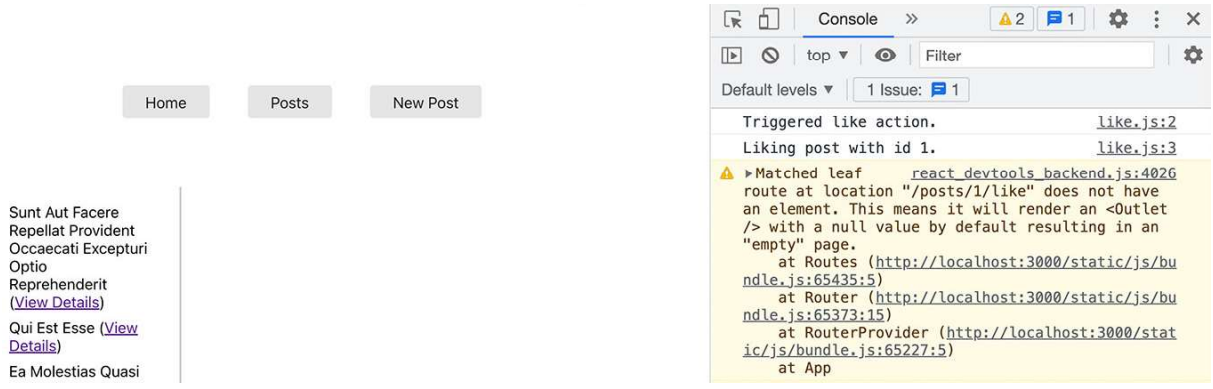

Figure 13.7: An empty (nested) page is displayed, along with a warning message

Because of the extra flexibility it offers, **`useFetcher()`** can be very useful when building highly interactive user interfaces. It's not meant as a replacement for **`useSubmit()`** or **`<Form>`** but rather as an additional tool for situations where no route transition is required or wanted.

DEFERRING DATA LOADING

Up to this point in the chapter, all data-fetching examples have assumed that a page should only be displayed once all its data has been fetched. That's why there was never any loading state that would have been managed (and hence no loading spinner that would have been displayed).

In many situations, this is exactly the behavior you want as it does not often make sense to show a loading spinner for a fraction of a second just to then replace it with the actual page data.

But there are also situations in which the opposite behavior might be desirable—for example, if you know that a certain page will take quite a while to load its data (possibly due to a complex database query that must be executed on the backend) or if you have a page that loads different pieces of data and some pieces are much slower than others.

In such scenarios, it may make sense to render the page component even though some data is still missing. React Router also supports this use case by allowing you to defer data loading, which in turn enables the page component to be rendered before the data is available.

Deferring data loading is as simple as using the **`defer()`** function provided by React Router like this:

```
import { defer } from 'react-router-dom';
// ... other imports
export async function loader() {
  return defer({
    posts: getPosts(),
  });
}
```

In this example, **`getPosts()`** is a function that returns a (slow) **`Promise`**:

```
async function getPosts() {
  const response = await fetch('https://jsonplaceholder.typicode.com/
posts');
  await wait(3); // utility function, simulating a slow response
  if (!response.ok) {
    throw new Error('Could not fetch posts');
  }
  const data = await response.json();
  return data;
}
```

React Router's **`defer()`** function takes an object as an argument. This object contains one key-value pair for every data fetching operation that's part of the **`loader()`** function. The values in this object are of type **`Promise`**; otherwise, there wouldn't be anything to defer.

Inside the component function where **`useLoaderData()`** is used, you must also use a new component provided by React Router: the **`Await`** component. It's used like this:

```
import { Suspense } from 'react';
import { Await } from 'react-router-dom';
// ... other imports

function PostsLayout() {
  const data = useLoaderData();

  return (
    <div id="posts-layout">
      <nav>
        <Suspense fallback={<p>Loading posts...</p>}>
          <Await resolve={data.posts}>
            {(loadedPosts) => <PostsList posts={loadedPosts} />}
          </Await>
        </Suspense>
      </nav>
      <main>
        <Outlet />
      </main>
    </div>
```

```
  );
}
```

The **<Await>** element takes a **resolve** prop that receives a value from the loader data. It's wrapped by the **< Suspense>** component provided by React.

The value passed to resolve is a **Promise**. It's one of the values stored in the object that was passed to **defer()**. For that reason, you use the key names defined in the object that was passed to **defer()** to access the data in the component function (meaning that **data.posts** is used because of **defer({posts: getPosts()})**).

Await automatically waits for the **Promise** to resolve before then calling the function that's passed to **<Await>** as a child (that is, the function passed between the **<Await>** opening and closing tags). This function is executed by React Router once the data of the deferred operation is available. Therefore, inside that function, **loadedPosts** is received as a parameter, and the final user interface elements can be rendered.

The **Suspense** component that's used as a wrapper around **<Await>** defines some fallback content that is rendered as long as the deferred data is not yet available. In the *"Lazy Loading"* section of the previous chapter, the **Suspense** component was used to show some fallback content until the missing code was downloaded. Now, it's used to bridge the time until the required data is available.

When using **defer()** (and **<Await>**) like this, you would still load other parts of the website while waiting for the posts data:

Figure 13.8: Post details are already visible while the list of posts is loading

Another big advantage of **`defer()`** is that you can easily combine multiple fetching processes and control which processes should be deferred and which ones should not. For example, a route might be fetching different pieces of data. If only one process tends to be slow, you could defer only the slow one like this:

```
export async function loader() {
  return defer(
    {
      posts: getPosts(), // slow operation => deferred
      userData: await getUserData() // fast operation => NOT deferred
    }
  );
}
```

In this example, **`getUserData()`** is not deferred because the **`await`** keyword is added in front of it. Therefore, JavaScript waits for that **`Promise`** (the **`Promise`** returned by **`getUserData()`**) to resolve before returning from **`loader()`**. Hence, the route component is rendered once **`getUserData()`** finishes but before **`getPosts()`** is done.

SUMMARY AND KEY TAKEAWAYS

- React Router can help you with data fetching and submission.
- You can register **`loader()`** functions for your routes, causing data fetching to be initialized as a route becomes active.
- **`loader()`** functions return data (or responses, wrapping data) that can be accessed via **`useLoaderData()`** in your component functions.
- **`loader()`** data can be used across components via **`useRouteLoaderData()`**.
- You can also register **`action()`** functions on your routes that are triggered upon form submissions.
- To trigger **`action()`** functions, you must use React Router's **`<Form>`** component or submit data programmatically via **`useSubmit()`** or **`useFetcher()`**.

- **`useFetcher()`** can be used to load or submit data without initiating a route transition.
- When fetching slow data, you can use **`defer()`** to defer loading some or all of a route's data.

WHAT'S NEXT?

Fetching and submitting data are extremely common tasks, especially when building more complex React applications.

Typically, those tasks are closely connected to route transitions, and React Router is the perfect tool for handling this kind of operation. With the release of version 6.4, the React Router package offers powerful data management capabilities that vastly simplify these processes.

In this chapter, you learned how React Router assists you with fetching or submitting data and which advanced features help you handle both basic and more complex data manipulation scenarios.

This chapter also concludes the list of core React features you must know as a React developer. Of course, you can always dive deeper to explore more patterns and third-party libraries. The next (and last) chapter will share some resources and possible next steps you could dive into after finishing this book.

TEST YOUR KNOWLEDGE!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to the examples found at https://packt.link/cbDjn:

1. How are data fetching and submission related to routing?
2. What is the purpose of **`loader()`** functions?
3. What is the purpose of **`action()`** functions?
4. What is the difference between **`<Form>`** and **`<form>`**?
5. What is the difference between **`useSubmit()`** and **`useFetcher()`**?
6. What is the idea behind **`defer()`**?

APPLY WHAT YOU LEARNED

Apply your knowledge about routing, combined with data manipulation, to the following activity.

ACTIVITY 13.1: A TO-DOS APP

In this activity, your task is to create a basic to-do list web app that allows users to manage their daily to-do tasks. The finished page must allow users to add to-do items, update to-do items, delete to-do items and view a list of to-do items.

The following paths must be supported:

- **`/`**: The main page, responsible for loading and displaying a list of to-do items
- **`/new`**: A page, opened as a modal above the main page, allowing users to add a new to-do item
- **`/:id`**: A page, also opened as a modal above the main page, allowing users to update or delete a selected to-do item

If no to-do items exist yet, a fitting info message should be shown on the **`/`** page. If users try to visit **`/:id`** with an invalid to-do ID, an error modal should be displayed.

> **NOTE**
>
> For this activity, there is no backend API you could use. Instead, use **`localStorage`** to manage the to-dos data. Keep in mind that the **`loader()`** and **`action()`** functions are executed on the client side and can therefore use any browser APIs, including **`localStorage`**.
>
> You will find example implementations for adding, updating, deleting, and getting to-do items from **`localStorage`** at https://packt.link/XCbu0.
>
> Also, don't be confused by the pages that open as modals above other pages. Ultimately, these are simply nested pages, styled as modal overlays. In case you get stuck, you can use the example **`Modal`** wrapper component found at https://packt.link/qPIvp.
>
> For this activity, you can write all CSS styles on your own if you so choose. But if you want to focus on the React and JavaScript logic, you can also use the finished CSS file from the solution at https://packt.link/G0IKW.
>
> If you use that file, explore it carefully to ensure you understand which IDs or CSS classes might need to be added to certain JSX elements of your solution.

To complete the activity, perform the following steps:

1. Create a new React project and install the React Router package (make sure it's version 6.4 or later).
2. Create components (with the content shown in the preceding screenshots) that will be loaded for the three required pages. Also add links (or programmatic navigation) between these pages.
3. Enable routing and add the route definitions for the three pages.
4. Create **`loader()`** functions to load (and use) all the data needed by the individual pages.
5. Add **`action()`** functions for adding, updating, and deleting to-dos.
6. Add error handling in case data loading or saving fails.

The finished pages should look like this:

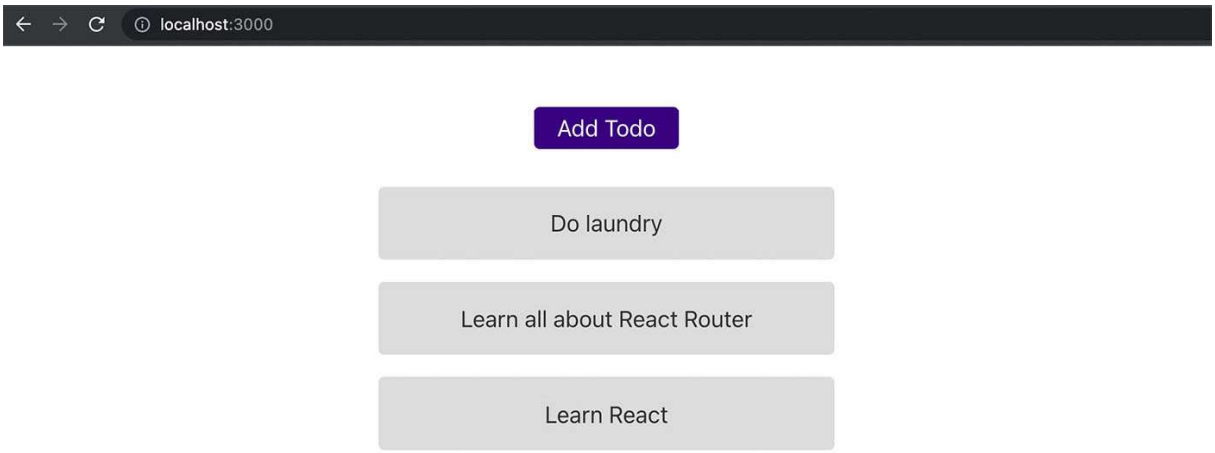

Figure 13.9: The main page displaying a list of to-dos

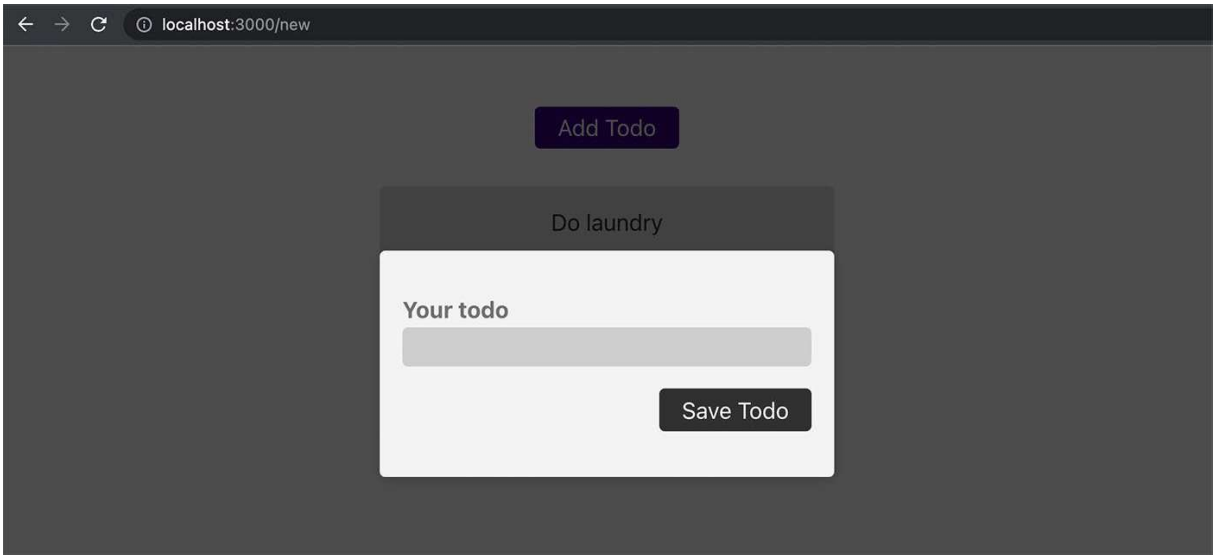

Figure 13.10: The /new page, opened as a modal, allowing users to add a new to-do

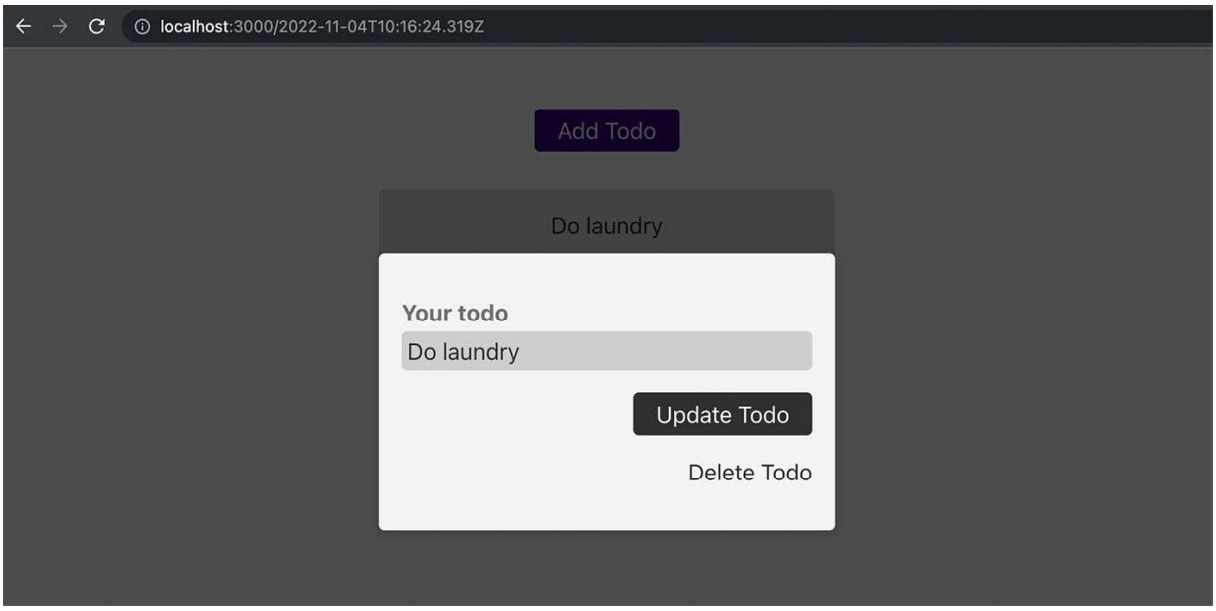

Figure 13.11: The /:id page, also opened as a modal, allowing users to edit or delete a to-do

14

NEXT STEPS AND FURTHER RESOURCES

LEARNING OBJECTIVES

By the end of this chapter, you will know the following:

- How to best practice what you've learned throughout this book
- Which React topics you can explore next
- Which popular third-party React packages might be worth a closer look

INTRODUCTION

With this book, you've gotten a thorough (re-)introduction to the key React concepts you must know in order to work with React successfully, providing both theoretical and practical guidance for components, props, state, context, React Hooks, routing, and many other crucial concepts.

But React is more than just a collection of concepts and ideas. It powers an entire ecosystem of third-party libraries that help with many common React-specific problems. There is also a huge React community that shares solutions for common problems or popular patterns.

In this last, brief chapter, you'll learn about some of the most important and popular third-party libraries you might want to explore. You will also be introduced to other great resources that help with learning React. In addition, this chapter will share some recommendations on how best to proceed and continue to grow as a React developer after finishing this book.

HOW SHOULD YOU PROCEED?

In working through this book, you've learned a lot about React. But it's always challenging to then proceed and apply that knowledge to real projects.

So, how should you proceed? How do you best apply your knowledge and, hence, continue to grow as a React developer?

The most important factor is that you use your knowledge. Don't just read a book. Instead, use your newly gained skills to build some demo projects.

You don't have to build the next Amazon or TikTok. There's a reason why applications like these are built by huge teams. But you should build small demo projects that focus on a couple of core problems. You could, for example, build a very basic website that allows users to store and view their daily goals, or build a basic meetups page where visitors can organize and join meetup events.

To put it simply: practice is king. You must apply what you've learned and build stuff. Because by building demo projects, you'll automatically encounter problems that you'll have to solve without a solution at hand. You'll have to try out different approaches and search the internet for possible (partial) solutions. Ultimately, this is how you learn the most and how you develop your problem-solving skills.

You won't find a solution for all problems in this book, but this book does give you the basic tools and building blocks that will help you with those problems. Solutions are then built by combining these building blocks and by building upon the knowledge gathered throughout this book.

INTERESTING PROBLEMS TO EXPLORE

So, which problems and demo apps could you explore and try to build?

In general, you can try to build (simplified) clones of popular web apps (such as a highly simplified version of Amazon). Ultimately, your imagination is the limit, but in the following sections, you will find details and advice for three project ideas and the challenges that come with them.

BUILD A SHOPPING CART

A very common type of website is an online shop. You can find online shops for all kinds of products—ranging from physical goods such as books, clothing, or furniture to digital products such as video games or movies—and building such an online shop would be an interesting project idea and challenge.

Online shops do come with many features that can't be built with React. For example, the whole payment process is mostly a backend task where requests must be handled by servers. Inventory management would be another feature that takes place in databases and on servers, and not in the browsers of your website visitors. But there are also features that require interactive user interfaces (and that therefore benefit from using React), such as different pages that show lists of available products, product details, or the current status of an order, as you learned in *Chapter 12*, *Multipage Apps with React Router*. You also typically have shopping carts on websites. Building such a cart, combined with the functionality of adding and removing items, would similarly utilize several React features—for example, state management, as explained in *Chapter 4*, *Working with Events and State*.

It all starts with having a couple of pages (routes) for dummy products, product details, and the shopping cart itself. The shopping cart displays items that need to be managed via app-wide state (e.g., via context as covered in *Chapter 10*, *Working with Complex State*), as website visitors must be able to add items to the cart from the product detail page. You will also need a broad variety of React components—many of which must be reusable (e.g., the individual shopping cart items that are displayed). Your knowledge of React components and props from *Chapter 2*, *Understanding React Components and JSX*, and *Chapter 3*, *Components and Props*, will help with that.

The shopping cart state is also a non-trivial state. A simple list of products typically won't do the trick—though you can, of course, at least apply your knowledge from *Chapter 5*, *Rendering Lists and Conditional Content*. Instead, you must check whether an item is already part of the cart or if it's added for the first time. If it's part of the cart already, you must update the quantity of the cart item. Of course, you'll also need to ensure users are able to remove items from the cart or reduce the quantity of an item. And if you want to get even fancier, you can even simulate price changes that must be factored in when updating the shopping cart state.

As you can see, this extremely simple dummy online shop already offers quite a bit of complexity. Of course, you could also add backend functionality and store dummy products in a database. This can be done but is not required to practice working with React. Instead, you can use dummy data that's stored in variables or constants in your React code.

Build an Application's Authentication System (User Signup and Login)

A lot of websites allow users to sign up or log in. For many websites, user authentication is required before performing certain tasks. For example, you must create a Google account before uploading videos to YouTube or using Gmail (and many other Google services). Similarly, an account is typically needed before taking paid online courses or buying (digital) video games online. You also can't perform online banking without being logged in. And that's just a short list; many more examples could be added, but you get the idea. User authentication is required for a broad variety of reasons on many websites.

And on even more websites, it's optionally available. For example, you might be able to order products as a guest, but you benefit from extra advantages when creating an account (e.g., you may track your order history or collect reward points).

Of course, building your own version of YouTube is much too challenging to be a good practice project. There's a reason why Google has thousands of developers on its payroll. But you can identify and clone individual features—such as user authentication.

Build your own user authentication system with React. Make sure that users can sign up and log in. Add a few example pages (routes) to your website and find a way of making some pages only available to logged-in users. These targets might not sound like much, but you will actually face quite a lot of challenges along the way—challenges that force you to find solutions for brand-new problems.

In addition, you can get even more advanced and dive into backend development as well (in addition to the frontend development done with React). While you could just use some dummy logic in your React app code to simulate HTTP requests that are sent to your servers behind the scenes, you could also add a real demo backend instead. That backend would need to store user accounts in a database, validate login requests, and send back authentication tokens that inform the React frontend about the current authentication status of a user. In your React app, these HTTP requests would be treated as side effects, as covered in *Chapter 8*, *Handling Side Effects*.

The backend would be built with a different technology than React though (such as with Node.js and the Express library) and is therefore quite a bit more advanced. Alternatively, you can keep things simple by instead having some app-wide "user is authenticated" state in your React app. In that case, you could still get a bit more advanced by storing that state in the browser (e.g., via **`localStorage`**) to restore it after page reloads.

As you can tell, this "simple" project idea (or, rather, feature idea) presents a lot of challenges and will require you to build on your React knowledge and find solutions for a broad variety of problems.

BUILD AN EVENT MANAGEMENT WEBSITE

If you first were to build your own shopping cart system and get started with user authentication, you could then take it a step further and build a more complex website that combines these features (and offers new, additional features).

One such project idea would be an event management site. This is a website on which users can create accounts and, once they're logged in, events. All visitors can then browse these events and register for them. It would be up to you whether registration as guests (without creating an account first) is possible or not.

It's also your choice whether you want to add backend logic (that is, a server that handles requests and stores users and events in a database) or you will simply store all data in your React application (via app-wide state). If you don't add a backend, all data will be lost whenever the page is reloaded, and you can't see the events created by other users on other machines, but you can still practice all these key React features.

And there are many React features that are needed for this kind of dummy website: reusable components, pages (routes), component-specific and app-wide state, handling and validating user input, displaying conditional and list data, and much more.

Again, this is clearly not an exhaustive list of examples. You can build whatever you want. Be creative and experiment, because you'll only master React if you use it to solve problems.

COMMON AND POPULAR REACT LIBRARIES

No matter which kind of React app you're building, you'll encounter many problems and challenges along the way. From handling and validating user input to sending HTTP requests, complex applications come with many challenges.

You can solve all challenges on your own and even write all the (React) code that's needed on your own. And for practicing, this might indeed be a good idea. But as you're building more and more complex apps, it might make sense to outsource certain problems.

Thankfully, React features a rich and vibrant ecosystem that offers third-party packages that solve all kinds of common problems. Here's a brief, non-exhaustive list of popular third-party libraries that might be helpful:

- **React Hook Form**: A library that simplifies the process of handling and validating user input (https://react-hook-form.com/).
- **Formik**: Another popular library that helps with form input handling and validation (https://formik.org/).
- **Axios**: A general JavaScript library that simplifies the process of sending HTTP requests and handling responses (https://axios-http.com/).
- **SWR**: A React-specific alternative to Axios, also aiming to simplify the process of sending HTTP requests and using response data (https://swr.vercel.app/).
- **Redux**: In the past, this was an essential React library. Nowadays, it can still be important as it can greatly simplify the management of (complex) cross-component or app-wide state (https://redux.js.org/).

This is just a short list of some helpful and popular libraries. Since there's an endless number of potential challenges, you could also compile an infinite list of libraries. Search engines and Stack Overflow (a message board for developers) are your friends when it comes to finding more libraries that solve other problems.

OTHER RESOURCES

As mentioned, React does have a highly vibrant ecosystem—and not just when it comes to third-party libraries. You'll also find hundreds of thousands of blog posts, discussing all kinds of best practices, patterns, ideas, and solutions to possible problems. Searching for the right keywords (such as "React form validation with Hooks") will almost always yield interesting articles or helpful libraries.

You'll also find plenty of paid online courses, such as the "React – The Complete Guide" course at https://www.udemy.com/course/react-the-complete-guide-incl-redux/?couponCode=D_0922, and free tutorials on YouTube.

The official documentation is another great place to explore as it contains deep dives into core topics as well as more tutorial articles: https://reactjs.org/.

BEYOND REACT FOR WEB APPLICATIONS

This book focused on using React to build websites. This was for a couple of reasons. The first is that React, historically, was created to simplify the process of building complex web user interfaces, and React is powering more and more websites every day. It's one of the most widely used client-side web development libraries and more popular than ever before.

But it also makes sense to learn how to use React for web development because you need no extra tools—only a text editor and a browser.

That said, React can be used to build user interfaces outside the browser and websites as well. With React Native and Ionic for React, you have two very popular projects and libraries that use React to build native mobile apps for iOS and Android.

Therefore, after learning all these React essentials, it makes a lot of sense to also explore these projects. Pick up some React Native or Ionic courses (or use the official documentation) to learn how you can use all the React concepts covered in this book to build real native mobile apps that can be distributed through the platform app stores.

React can be used to build all kinds of interactive user interfaces for various platforms. Now that you've finished this book, you have the tools you need to build your next project with React—no matter which platform it targets.

FINAL WORDS

With all the concepts discussed throughout this book, as well as the extra resources and starting points to dive deeper, you are well prepared to build feature-rich and highly user-friendly web applications with React.

No matter if it's a simple blog or a complex Software-as-a-Service solution, you now know the key React concepts you need in order to build a React-driven web app your users will love.

I hope you got a lot out of this book. Definitely share any feedback you have, for example, via Twitter (**`@maxedapps`**) or by sending an email to customercare@packt.com.

CHAPTER 2: UNDERSTANDING REACT COMPONENTS & JSX

ACTIVITY 2.1: CREATING A REACT APP TO PRESENT YOURSELF

Solution:

Perform the following steps to complete this activity:

1. Create a new React project by running npx create-react-app my-app. You can replace my-app with any name of your choice, and you can run this command in any fitting place on your system (e.g., on your desktop). Start the development web server by running npm start inside the created project folder.
2. Open the project with any code editor of your choice—for example, with Visual Studio Code (https://code.visualstudio.com/).
3. Open the App.js file and replace the existing JSX code that is returned with JSX code that structures and contains the information about yourself that you want to output.

```
function App() {
  return (
    <>
      <h2>Hi, this is me - Max!</h2>
      <p>Right now, I am 32 years old and I live in Munich.</p>
      <p>
        My full name is Maximilian Schwarzmüller and I am a web
developer as
        well as top-rated, bestselling online course instructor.
      </p>
    </>
  );
}
```

4. The final output should look like the following:

Figure 2.5: The final activity result—some user information being output on the screen.

> **NOTE**
>
> You will find all code files for this solution at https://packt.link/DJh7X.

ACTIVITY 2.2: CREATING A REACT APP TO LOG YOUR GOALS FOR THIS BOOK

Solution:

Perform the following steps to complete this activity:

1. Create a new React project by running npx create-react-app my-app. You can replace my-app with any name of your choice, and you can run this command in any fitting place on your system (e.g., on your desktop). Start the development web server by running npm start inside the created project folder.
2. Create a new /src/components folder in the project
3. In the /src/components folder, create multiple component files—for example, FirstGoal.js, SecondGoal.js, ThirdGoal.js, GoalList.js and Header.js

   Your project folder should now look something like this:

Figure 2.6: React project with a "components" folder and multiple component files added.

4. Edit the individual goal component files (FirstGoal.js, etc.) and define and export component functions inside of them. Every component function should return a list item with any JSX markup of your choice and the goal title and text as main content. Here's an example for the first goal:

```
function FirstGoal() {
  return (
    <li>
      <article>
        <h2>Teach React in a highly-understandable way</h2>
        <p>
          I want to ensure, that you get the most out of this book
and you
          learn all about React!
        </p>
      </article>
    </li>
  );
}

export default FirstGoal;
```

5. In the GoalList.js file, define and export a GoalList component function and import the individual components. Thereafter, return JSX code that renders an unordered list (<ul>) with the custom goal components as list items:

```
import FirstGoal from './FirstGoal';
import SecondGoal from './SecondGoal';
import ThirdGoal from './ThirdGoal';

function GoalList() {
  return (
    <ul>
      <FirstGoal />
      <SecondGoal />
      <ThirdGoal />
```

```
      </ul>
    );
  }

  export default GoalList;
```

6. In the Header.js file, define and export a Header component and return some header JSX markup:

```
  function Header() {
    return (
      <header>
        <h1>My Goals For This Book</h1>
      </header>
    );
  }

  export default Header;
```

7. Import the GoalList and Header components into the App.js file and replace the default JSX code with your own JSX code that renders these two components:

```
  import GoalList from './components/GoalList';
  import Header from './components/Header';

  function App() {
    return (
      <>
        <Header />
        <GoalList />
      </>
    );
  }

  export default App;
```

The final output should look like the following:

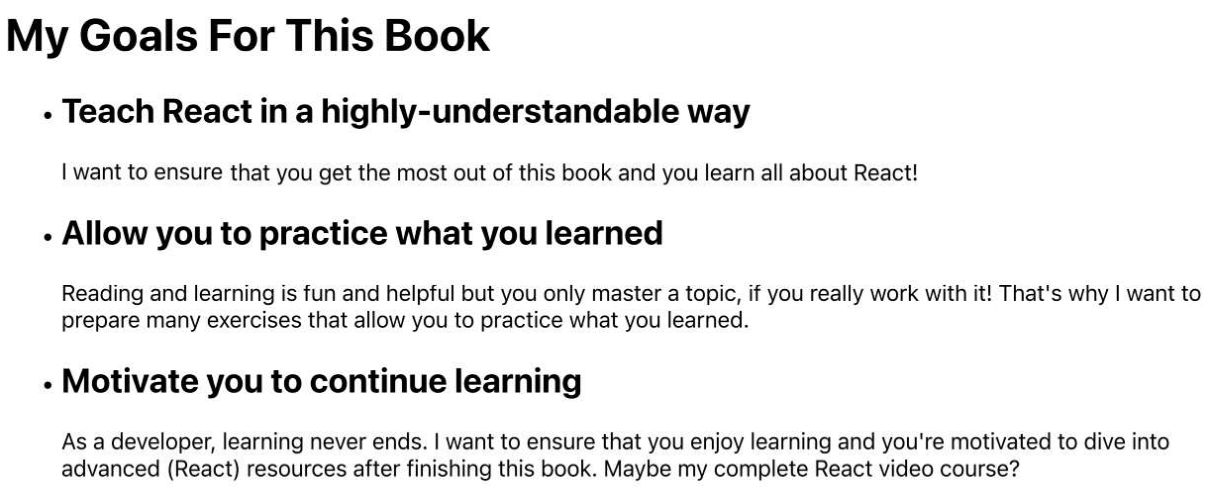

Figure 2.7: The final page output, showing a list of goals.

> **NOTE**
>
> You will find all code files for this solution at https://packt.link/8tvm6.

CHAPTER 3: COMPONENTS & PROPS

ACTIVITY 3.1: CREATING AN APP TO OUTPUT YOUR GOALS FOR THIS BOOK

Solution:

1. Finish *Activity 2.2* from the previous chapter.
2. Add a new component to the **`src/components`** folder, a component function named **`GoalItem`**, in a new **`GoalItem.js`** file.
3. Copy the component function (including the returned JSX code) from **`FirstGoal.js`** and add a new **`props`** parameter to the function. Remove the title and description text from the JSX code:

```
function GoalItem(props) {
  return (
    <li>
      <article>
        <h2></h2>
        <p>

        </p>
      </article>
    </li>
  );
}

export default GoalItem;
```

4. Output the title and description in the **`GoalItem`** component via props—for example, by using **`props.title`** and **`props.children`** (in the fitting places in the JSX code, in other words, between the **`<h2>`** and **`<p>`** tags).
5. In the **`GoalList`** component, remove the **`FirstGoal`**, **`SecondGoal`**, and so on components (imports and JSX code) and import and use the new **`GoalItem`** component instead. Output **`<GoalItem>`** once for every goal that should be displayed, and pass the **`title`** prop and a value for the **`children`** prop to these components:

```
import GoalItem from './GoalItem';

function GoalList() {
  return (
```

```
      <ul>
        <GoalItem title="Teach React in a highly-understandable way">
          Some goal text…
        </GoalItem>
        <GoalItem title="Allow you to practice what you learned">
          Some goal text…
        </GoalItem>
        <GoalItem title="Motivate you to continue learning">
          Some goal text…
        </GoalItem>
      </ul>
    );
  }

  export default GoalList;
```

6. Delete the redundant **`FirstGoal.js`**, **`SecondGoal.js`**, etc. files.

The final user interface could look like this:

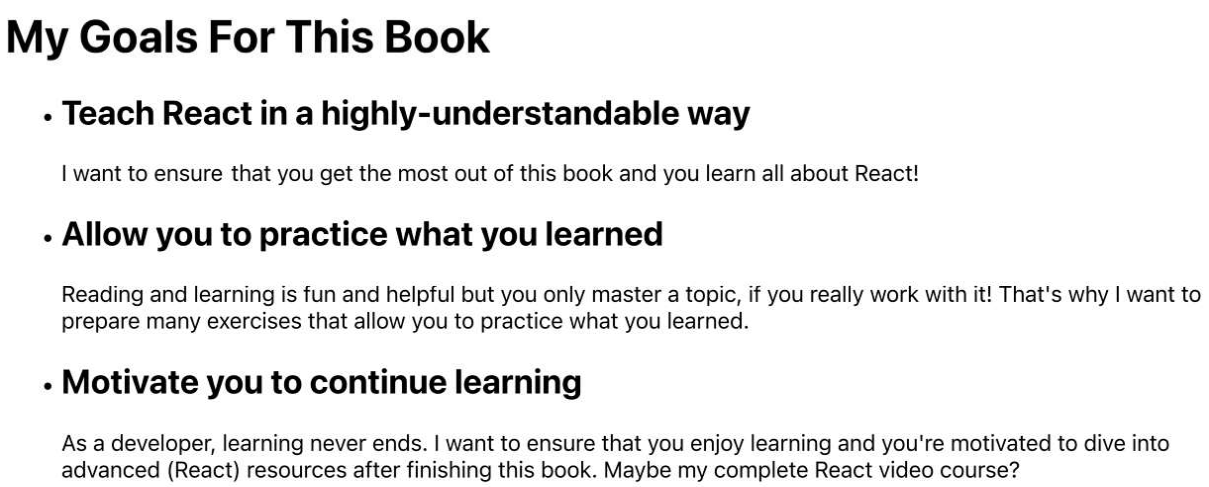

Figure 3.2: The final result: Multiple goals output below each other

> **NOTE**
>
> You will all code files for this solution at https://packt.link/2R4Xo.

CHAPTER 4: WORKING WITH EVENTS & STATE

ACTIVITY 4.1: BUILDING A SIMPLE CALCULATOR

Solution:

Perform the following steps to complete this activity:

1. Add four new components into an **`src/components`** folder in a new React project: **`Add.js`**, **`Subtract.js`**, **`Divide.js`**, and **`Multiply.js`** (also add appropriately named component functions inside the component files).
2. Add the following code to **`Add.js`**:

```
function Add() {

  function changeFirstNumberHandler(event) {

  }

  function changeSecondNumberHandler(event) {

  }

  return (
    <p>
      <input type="number" onChange={changeFirstNumberHandler} /> +
      <input type="number" onChange={changeSecondNumberHandler} /> =
...
    </p>
  );
}

export default Add;
```

This component outputs a paragraph that contains two input elements (for the two numbers) and the result of the calculation. The input elements use the **`onChange`** prop to listen to the change event. Upon this event, the **`changeFirstNumberHandler`** and **`changeSecondNumberHandler`** functions are executed.

3. In order to make the component dynamic and derive the result based on the actual user input, state must be added. Import the **`useState`** Hook from React and initialize an object that contains a property for each of the two numbers. Alternatively, you could also use two individual state slices. Update the state(s) inside the two functions that are connected to the **`change`** event and set the state to the entered user value.

   Make sure you convert the entered value to a number by adding a **+** in front of the value. Otherwise, string values will be stored, which will lead to incorrect results when adding the numbers.

   The updated **`Add.js`** component should look like this:

```
import { useState } from 'react';

function Add() {
  const [enteredNumbers, setEnteredNumbers] = useState({
    first: 0, second: 0
  });

  function changeFirstNumberHandler(event) {
    setEnteredNumbers((prevNumbers) => ({
      first: +event.target.value, // "+" converts strings to numbers!
      second: prevNumbers.second,
    }));
  }

  function changeSecondNumberHandler(event) {
    setEnteredNumbers((prevNumbers) => ({
      first: prevNumbers.first,
      second: +event.target.value,
    }));
  }


  return (
    <p>
      <input type="number" onChange={changeFirstNumberHandler} /> +
      <input type="number" onChange={changeSecondNumberHandler} /> =
...
    </p>
  );
```

```
}

export default Add;
```

4. Next, derive the actual result of the mathematical operation. For this, a new **`result`** variable or constant can be added. Set it to the result of adding the two numbers that are stored in state.

   The finished **`Add.js`** file looks like this:

```
import { useState } from 'react';

function Add() {
  const [enteredNumbers, setEnteredNumbers] = useState({
    first: 0, second: 0
  });

  function changeFirstNumberHandler(event) {
    setEnteredNumbers((prevNumbers) => ({
      first: +event.target.value,
      second: prevNumbers.second,
    }));
  }

  function changeSecondNumberHandler(event) {
    setEnteredNumbers((prevNumbers) => ({
      first: prevNumbers.first,
      second: +event.target.value,
    }));
  }

  const result = enteredNumbers.first + enteredNumbers.second;

  return (
    <p>
      <input type="number" onChange={changeFirstNumberHandler} /> +{'
'}
      <input type="number" onChange={changeSecondNumberHandler} /> =
{result}
    </p>
  );
```

```
}

export default Add;
```

5. Finally, copy the same code into the three other component files (**`Subtract.js`**, **`Multiply.js`**, and **`Divide.js`**). Just make sure to replace the component function name (also in the **`export`** statement) and to update the mathematical operation.

The final result and UI of the calculator should look like this:

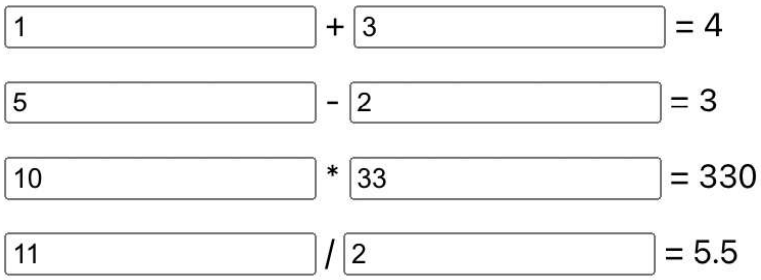

Figure 4.5: Calculator user interface

> **NOTE**
>
> You'll find all code files for this solution at https://packt.link/kARx8.

ACTIVITY 4.2: ENHANCING THE CALCULATOR

Solution:

Perform the following steps to complete this activity:

1. Remove three of the four components from the previous activity and rename the remaining one to **`Calculation.js`** (also rename the function in the component file).
2. Add a **`<select>`** drop-down (between the two inputs) to the **`Calculation`** component and add the four math operations as options (**`<option>`** elements) to it. You might want to give each option a clear identifier (such as **`'add'`**, **`'subtract'`**, and so on) via the built-in **`value`** prop. Remove the result.

The finished JSX code of the **`Calculation`** component should look like this:

```
  return (
    <p>
      <input type="number" onChange={changeFirstNumberHandler} />
      <select>
        <option value="add">+</option>
        <option value="subtract">-</option>
        <option value="multiply">*</option>
        <option value="divide">/</option>
      </select>
      <input type="number" onChange={changeSecondNumberHandler} />
    </p>
```

3. Next, add a **`Result.js`** file with a **`Result`** component in the **`src/components`** folder. In that component, output the result of the calculation (for the moment, output some dummy number):

```
function Result() {
  return <p>Result: 5000</p>;
}

export default Result;
```

4. The problem now is that the inputs are in a different component than the result. The solution is to *lift the state up* to a common ancestor component. In this simple app, that would again be the **`App`** component. That component should manage the entered numbers and the chosen math operation states. It should also derive the result—dynamically, based on the chosen operation and the entered numbers. For this, an if statement can be used in the **`component`** function:

```
import { useState } from 'react';

import Calculation from './components/Calculation';
import Result from './components/Result';

function App() {
  const [enteredNumbers, setEnteredNumbers] = useState({
    first: 0, second: 0
  });
  const [chosenOperation, setChosenOperation] = useState('add');
```

```
  // valid state values: 'add', 'subtract', 'multiply', 'divide'

  function changeFirstNumberHandler(event) {
    setEnteredNumbers((prevNumbers) => ({
      first: +event.target.value,
      second: prevNumbers.second,
    }));
  }

  function changeSecondNumberHandler(event) {
    setEnteredNumbers((prevNumbers) => ({
      first: prevNumbers.first,
      second: +event.target.value,
    }));
  }

  function updateOperationHandler(event) {
    setChosenOperation(event.target.value);
  }

  let result;

  if (chosenOperation === 'add') {
    result = enteredNumbers.first + enteredNumbers.second;
  } else if (chosenOperation === 'subtract') {
    result = enteredNumbers.first - enteredNumbers.second;
  } else if (chosenOperation === 'multiply') {
    result = enteredNumbers.first * enteredNumbers.second;
  } else {
    result = enteredNumbers.first / enteredNumbers.second;
  }

  // return statement omitted, will be defined in the next step
}

export default App;
```

Since the component function will be re-executed by React whenever some state changes, **`result`** will be recalculated upon every state change.

5. Finally, include the two other components (**`Calculation`** and **`Result`**) in the returned JSX code of the **`App`** component. Use props to pass the event handler functions (**`changeFirstNumberHandler`**, **`changeSecondNumberHandler`**, and **`updateOperationHandler`**) to the **`Calculation`** component. Similarly, pass the derived **`result`** to the **`Result`** component. For the event handler functions, the props can be named **`onXYZ`** to indicate that functions are provided as values and that those functions will be used as event handler functions.

   Therefore, the returned JSX code of the **`App`** component should look like this:

```
  return (
    <>
      <Calculation
        onFirstNumberChanged={changeFirstNumberHandler}
        onSecondNumberChanged={changeSecondNumberHandler}
        onOperationChanged={updateOperationHandler}
      />
      <Result calculationResult={result} />
    </>
  );
```

   The **`Calculation`** component receives and uses the three **`onXYZ`** props like this:

```
function Calculation({
  onFirstNumberChanged,
  onSecondNumberChanged,
  onOperationChanged,
}) {
  return (
    <p>
      <input type="number" onChange={onFirstNumberChanged} />
      <select onChange={onOperationChanged}>
        <option value="add">+</option>
        <option value="subtract">-</option>
        <option value="multiply">*</option>
        <option value="divide">/</option>
      </select>
      <input type="number" onChange={onSecondNumberChanged} />
    </p>
  );
```

```
}

export default Calculation;
```

The **`Result`** component receives **`calculationResult`** and uses it like this:

```
function Result({ calculationResult }) {
  return <p>Result: {calculationResult}</p>;
}

export default Result;
```

The final result and UI of the calculator should look like this:

Result: 15

Figure 4.6: User interface of the enhanced calculator

> **NOTE**
>
> You'll find all code files for this solution at https://packt.link/0tdpg.

CHAPTER 5: RENDERING LISTS & CONDITIONAL CONTENT

ACTIVITY 5.1: SHOWING A CONDITIONAL ERROR MESSAGE

Solution:

Perform the following steps to complete this activity:

1. Create a new React project and remove the default JSX code returned by the **App** component. Instead, return a **<form>** element that contains an **<input>** of **type="text"** (for the purpose of this activity, it should not be **type="email"** to make entering incorrect email addresses easier). Also add a **<label>** for the **<input>** and a **<button>** that submits the form. The final JSX code returned by **App** should look something like this:

```
function App() {
  return (
      <form>
        <label htmlFor="email">Your email</label>
        <input type="text" id="email"/>
        <button>Submit</button>
      </form>
  );
}

export default App;
```

2. Register **change** events on the **<input>** element to store and update the entered email address via state:

```
import { useState } from 'react';

function App() {
  const [enteredEmail, setEnteredEmail] = useState();

  function emailChangeHandler(event) {
    setEnteredEmail(event.target.value);
    // enteredEmail is then not used hered, hence you could get a
    // warning related to this. You can ignore it for this example
  }

  return (
      <form>
```

```
        <label htmlFor="email">Your email</label>
        <input type="text" id="email" onChange={emailChangeHandler}
/>
        <button>Submit</button>
      </form>
  );
}

export default App;
```

3. Add a **`submit`** event handler function that is triggered every time the form is submitted. Prevent the browser default (of sending an HTTP request) by calling **`event.preventDefault()`** inside the **`submit`** event handler function. Also add logic to determine whether an email address is valid (contains an `@` sign) or not (no `@` sign):

```
import { useState } from 'react';

function App() {
  const [enteredEmail, setEnteredEmail] = useState();

  function emailChangeHandler(event) {
    setEnteredEmail(event.target.value);
  }

  function submitFormHandler(event) {
    event.preventDefault();
    const emailIsValid = enteredEmail.includes('@');
    // emailIsValid is then not used hered, hence you could get a
    // warning related to this. You can ignore it for this example
  }

  return (
      <form onSubmit={submitFormHandler}>
        <label htmlFor="email">Your email</label>
        <input type="text" id="email" onChange={emailChangeHandler}
/>
        <button>Submit</button>
      </form>
  );
```

```
}

export default App;
```

4. Add a new state slice (the following example has been named **`inputIsInvalid`** and set to **`false`** as a default) that stores the email validity information. Update the **`inputIsInvalid`** state based on the **`emailIsValid`** constant defined in **`submitFormHandler`**. Use the state to show an error message (inside a **`<p>`**) conditionally:

```
import { useState } from 'react';

function App() {
  const [enteredEmail, setEnteredEmail] = useState();
  const [inputIsInvalid, setInputIsInvalid] = useState(false);

  function emailChangeHandler(event) {
    setEnteredEmail(event.target.value);
  }

  function submitFormHandler(event) {
    event.preventDefault();
    const emailIsValid = enteredEmail.includes('@');
    setInputIsInvalid(!emailIsValid);
  }

  return (
    <section>
      <form onSubmit={submitFormHandler}>
        <label htmlFor="email">Your email</label>
        <input type="text" id="email" onChange={emailChangeHandler}
/>
        <button>Submit</button>
      </form>
      {inputIsInvalid && <p>Invalid email address entered!</p>}
    </section>
  );
}

export default App;
```

5. The **`&&`** operator is used in this example, but you could also use **`if`** statements, ternary expressions, or any other possible approach. It's also up to you whether you prefer to create and output conditional JSX elements *inline* (that is, directly inside of the returned JSX code) or with the help of a separate variable or constant.

The final user interface should look and work as shown here:

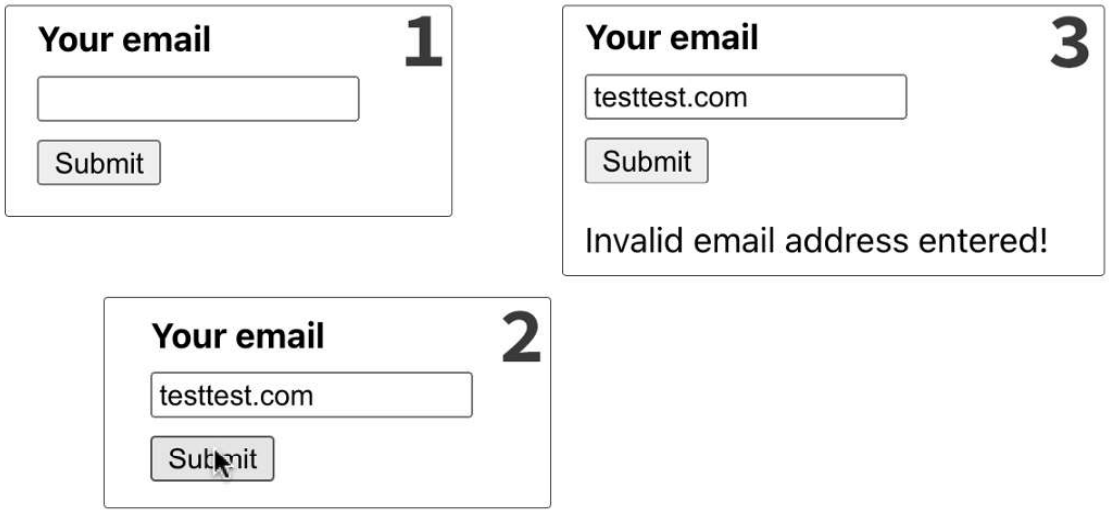

Figure 5.9: The final user interface of this activity

> **NOTE**
>
> You can find all code files for this solution at https://packt.link/GgEPO.

ACTIVITY 5.2: OUTPUTTING A LIST OF PRODUCTS

Solution:

Perform the following steps to complete this activity:

1. Create a new React project and remove the default JSX code returned by the **App** component. Instead, return a **<section>** that contains both a **<button>** (which will later be used to add a new product) and an (empty) **<ul>** element:

```
import { useState } from 'react';

function App() {
  return (
    <section>
      <button>Add Product</button>
      <ul></ul>
    </section>
  );
}

export default App;
```

2. Add an array of initial dummy products to the **App** component. Use this array as the initial value for the **products** state that must be added to the **App** component:

```
import { useState } from 'react';

const INITIAL_PRODUCTS = [
  { id: 'p1', title: 'React - The Complete Guide [Course]', price:
19.99 },
  { id: 'p2', title: 'Stylish Chair', price: 329.49 },
  { id: 'p3', title: 'Ergonomic Chair', price: 269.99 },
  { id: 'p4', title: 'History Video Game Collection', price: 99.99 },
];

function App() {
  const [products, setProducts] = useState(INITIAL_PRODUCTS);

  return (
    <section>
      <button>Add Product</button>
```

```
        <ul></ul>
      </section>
    );
  }

  export default App;
```

3. Output the list of products as part of the returned JSX code:

```
  import { useState } from 'react';

  const INITIAL_PRODUCTS = [
    { id: 'p1', title: 'React - The Complete Guide [Course]', price:
  19.99 },
    { id: 'p2', title: 'Stylish Chair', price: 329.49 },
    { id: 'p3', title: 'Ergonomic Chair', price: 269.99 },
    { id: 'p4', title: 'History Video Game Collection', price: 99.99 },
  ];

  function App() {
    const [products, setProducts] = useState(INITIAL_PRODUCTS);

    return (
      <section>
        <button>Add Product</button>
        <ul>
          {products.map((product) => (
            <li key={product.id}>
              {product.title} (${product.price})
            </li>
          ))}
        </ul>
      </section>
    );
  }

  export default App;
```

4. This example uses the **`map()`** method, but you could also use a **`for`** loop to populate an array with JSX elements, and then output that array as part of the JSX code. It's also up to you whether you create and output the list of JSX elements *inline* (that is, directly inside of the returned JSX code) or with the help of a separate variable or constant.

5. Add a **`click`** event handler to the **`<button>`**. Also, add a new function that is triggered upon **`click`** events on the button. Inside the function, update the **`products`** state such that a new (dummy) product is added. For the **`id`** value of that product, you can generate a pseudo-unique **`id`** by using a new **`Date().toString()`**:

```
import { useState } from 'react';

const INITIAL_PRODUCTS = [
  { id: 'p1', title: 'React - The Complete Guide [Course]', price:
19.99 },
  { id: 'p2', title: 'Stylish Chair', price: 329.49 },
  { id: 'p3', title: 'Ergonomic Chair', price: 269.99 },
  { id: 'p4', title: 'History Video Game Collection', price: 99.99 },
];

function App() {
  const [products, setProducts] = useState(INITIAL_PRODUCTS);

  function addProductHandler() {
    setProducts((curProducts) =>
      curProducts.concat({
        id: new Date().toString(),
        title: 'Another new product',
        price: 15.99,
      })
    );
  }

  return (
    <section>
      <button onClick={addProductHandler}>Add Product</button>
      <ul>
        {products.map((product) => (
          <li key={product.id}>
```

```
              {product.title} (${product.price})
            </li>
          ))}
        </ul>
      </section>
    );
  }

  export default App;
```

The final user interface should look and work as shown here:

Figure 5.10: The final user interface of this activity

> **NOTE**
>
> You can find all code files for this solution at https://packt.link/XaL84.

CHAPTER 6: STYLING REACT APPS

ACTIVITY 6.1: PROVIDING INPUT VALIDITY FEEDBACK UPON FORM SUBMISSION

Solution:

Perform the following steps to complete this activity:

1. Create a new React project and add a **`Form`** component function in a **`components/Form.js`** file in the project.
2. Export the component and import it into **`App.js`**.
3. Output the **`<Form />`** component as part of App's JSX code.
4. In the component, output a **`<form>`** that contains two **`<input>`** elements as well as **`<label>`** elements that belong to those input elements—one input for entering an email address and another input for entering a password.
5. Add a **`<button>`** element that submits the form:

```
function Form() {
  return (
    <form>
      <div>
        <label htmlFor="email">
          Your email
        </label>
        <input
          id="email"
          type="email"
        />
      </div>
      <div>
        <label htmlFor="password">
          Your password
        </label>
        <input
          id="password"
          type="password"
        />
      </div>
      <button>Submit</button>
    </form>
```

```
    );
}
export default Form;
```

6. Add state and event handler functions to register and store entered **`email`** and **`password`** values:

```
import { useState } from 'react';
function Form() {
  const [enteredEmail, setEnteredEmail] = useState('');
  const [enteredPassword, setEnteredPassword] = useState('');
  function changeEmailHandler(event) {
    setEnteredEmail(event.target.value);
  }
  function changePasswordHandler(event) {
    setEnteredPassword(event.target.value);
  }
  return (
    <form>
      <div>
        <label htmlFor="email">
          Your email
        </label>
        <input
          id="email"
          type="email"
          onChange={changeEmailHandler}
        />
      </div>
      <div>
        <label htmlFor="password">
          Your password
        </label>
        <input
          id="password"
          type="password"
          onChange={changePasswordHandler}
        />
      </div>
      <button>Submit</button>
    </form>
```

```
    );
}
export default Form;
```

7. Add a form submission handler function to **`Form`**.
8. Inside that function, validate the entered **`email`** and **`password`** values (with any validation logic of your choosing).

   The validation results (**`true`** or **`false`**) for the two input fields are then stored in two new state slices (one for the email's validity and one for the password's validity):

```
import { useState } from 'react';
function Form() {
  const [enteredEmail, setEnteredEmail] = useState('');
  const [emailIsValid, setEmailIsValid] = useState(true);
  const [enteredPassword, setEnteredPassword] = useState('');
  const [passwordIsValid, setPasswordIsValid] = useState(true);
  function changeEmailHandler(event) {
    setEnteredEmail(event.target.value);
  }
  function changePasswordHandler(event) {
    setEnteredPassword(event.target.value);
  }
  function submitFormHandler(event) {
    event.preventDefault();
    const emailIsValid = enteredEmail.includes('@');
    const passwordIsValid = enteredPassword.trim().length >= 6;
    setEmailIsValid(emailIsValid);
    setPasswordIsValid(passwordIsValid);
    if (!emailIsValid || !passwordIsValid) {
      return;
    }
    // do something...
    console.log('Inputs are valid, form submitted!');
  }
  return (
    <form onSubmit={submitFormHandler}>
      <div>
        <label htmlFor="email">
          Your email
```

```
          </label>
          <input
            id="email"
            type="email"
            onChange={changeEmailHandler}
          />
        </div>
        <div>
          <label htmlFor="password">
            Your password
          </label>
          <input
            id="password"
            type="password"
            onChange={changePasswordHandler}
          />
        </div>
        <button>Submit</button>
      </form>
    );
}
export default Form;
```

9. Use the validation result state values to conditionally add the **`invalid`** CSS class (defined in **`index.css`**) to the **`<label>`** and **`<input>`** elements:

```
function Form() {
  // ... code didn't change, hence omitted ...
  return (
    <form onSubmit={submitFormHandler}>
      <div>
        <label htmlFor="email" className={!emailIsValid &&
'invalid'}>
          Your email
        </label>
        <input
          id="email"
          type="email"
          onChange={changeEmailHandler}
          className={!emailIsValid && 'invalid'}
        />
      </div>
```

```
        <div>
          <label htmlFor="password" className={!passwordIsValid &&
'invalid'}>
            Your password
          </label>
          <input
            id="password"
            type="password"
            onChange={changePasswordHandler}
            className={!passwordIsValid && 'invalid'}
          />
        </div>
        <button>Submit</button>
      </form>
    );
}
```

This example does not use CSS Modules; hence, the CSS classes are added as string values and no special CSS import syntax is used here.

The final user interface should look like this:

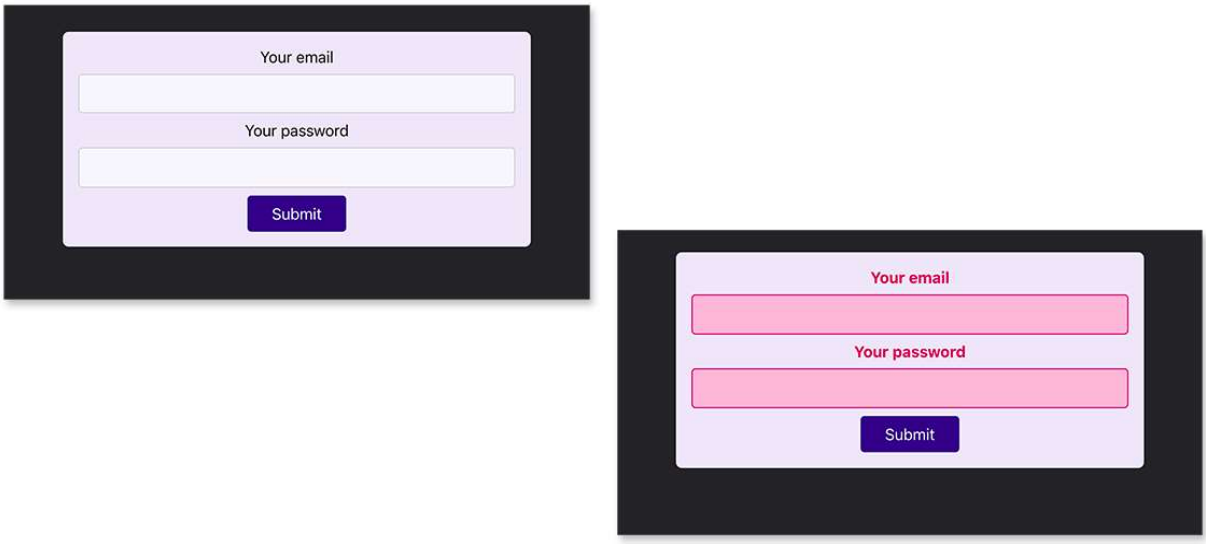

Figure 6.7: The final user interface with invalid input values highlighted in red

> **NOTE**
>
> You will find all code files used for this solution at https://packt.link/1iAe9.

ACTIVITY 6.2: USING CSS MODULES FOR STYLE SCOPING

Solution:

Perform the following steps to complete this activity:

1. Finish *Activity 6.01* or use the provided code on GitHub.
2. Identify all **`Form`**-related CSS rules defined in **`index.css`**. The relevant rules are as follows:

Form-related CSS rules	Relevance
`form { ... }`	This rule sets the general styling for the form.
`input { ... }`	This rule sets the general `<input>` styles.
`label { ... }`	This rule sets the general `<label>` styles.
`button { ... }` and `button:hover { ... }`	These rules set the `<button>` styles.
`label.invalid { ... }` and `input.invalid { ... }`	These rules control the conditionally added styles.

Figure 6.8: Table for Form-related CSS rules

   You could, of course, argue that the **`form`**, **`input`**, **`label`**, and **`button`** styles should be global styles because you might have multiple forms (and inputs, labels, and buttons) in your app—not just in the **`Form`** component. This would be a valid argument, especially for bigger apps. However, in this demo, all these styles are only used in the **`Form`** component and therefore should be migrated.

3. Cut the identified styles from **`index.css`** and add them into a newly created **`components/Form.module.css`** file. The file must end with **`.module.css`** to enable the CSS Modules feature. Since CSS Modules need class selectors, all selectors that don't use classes right now must be changed so that **`form { ... }`** becomes **`.form { ... }`**, **`input { ... }`** becomes **`.input { ... }`**, and so on.

   The leading dots (.) matter because the dot is what defines a CSS class selector. The class names are up to you and don't have to be **`.form`**, **`.input`**, and so on. Use any class names of your choice, but you must migrate all non-class selectors to class selectors. For consistency, **`input.invalid`** is also changed to **`.input.invalid`** and **`label.invalid`** is changed to **`.label.invalid`**.

The finished **`Form.module.css`** file looks like this:

```
.form {
  max-width: 30rem;
  margin: 2rem auto;
  padding: 1rem;
  border-radius: 6px;
  text-align: center;
  background-color: #eee8f8;
}
.input {
  display: block;
  width: 100%;
  font: inherit;
  padding: 0.5rem;
  margin-bottom: 0.5rem;
  font-size: 1.25rem;
  border-radius: 4px;
  background-color: #f9f6fc;
  border: 1px solid #ccc;
}
.label {
  display: block;
  margin-bottom: 0.5rem;
}
.button {
  cursor: pointer;
  font: inherit;
  padding: 0.5rem 1.5rem;
  background-color: #310485;
  color: white;
  border: 1px solid #310485;
  border-radius: 4px;
}
.button:hover {
  background-color: #250364;
}
.label.invalid {
  font-weight: bold;
  color: #ce0653;
```

```
}
.input.invalid {
  background-color: #fcbed6;
  border-color: #ce0653;
}
```

4. Import **`Form.module.css`** into the **`Form.js`** file as follows:

```
import classes from './Form.module.css';.
```

5. Assign the defined and imported classes to the appropriate JSX elements. Keep in mind that you must use the imported **`classes`** object to access the class names since the final class names are unknown to you (they transformed during the build process). For example, **`<button>`** receives its class (**`.button`**) like this: **`<button className={classes.button}>`**. For the conditional classes (the **`.invalid`** classes), the final code looks as follows:

```
<label
  htmlFor="email"
  className={
      !emailIsValid ? `${classes.label} ${classes.invalid}` :
classes.label
    }
  >
  Your email
</label>
```

6. Work with extra variables that store the ternary expressions (or replace them using **`if`** statements) to reduce the amount of logic injected into the JSX code.

   The final **`Form.js`** file looks like this:

```
import { useState } from 'react';
import classes from './Form.module.css';
function Form() {
  // ... other code - did not change ...
  return (
    <form className={classes.form} onSubmit={submitFormHandler}>
      <div>
        <label
          htmlFor="email"
          className={
            !emailIsValid
              ? `${classes.label} ${classes.invalid}`
```

```
              : classes.label
            }
          >
            Your email
          </label>
          <input
            id="email"
            type="email"
            onChange={changeEmailHandler}
            className={
              !emailIsValid
                ? `${classes.input} ${classes.invalid}`
                : classes.input
            }
          />
        </div>
        <div>
          <label
            htmlFor="password"
            className={
              !passwordIsValid
                ? `${classes.label} ${classes.invalid}`
                : classes.label
            }
          >
            Your password
          </label>
          <input
            id="password"
            type="password"
            onChange={changePasswordHandler}
            className={
              !passwordIsValid
                ? `${classes.input} ${classes.invalid}`
                : classes.input
            }
          />
        </div>
        <button className={classes.button}>Submit</button>
      </form>
```

```
    );
  }
  export default Form;
```

As explained above, the final user interface will look the same on the surface as that of the previous activity:

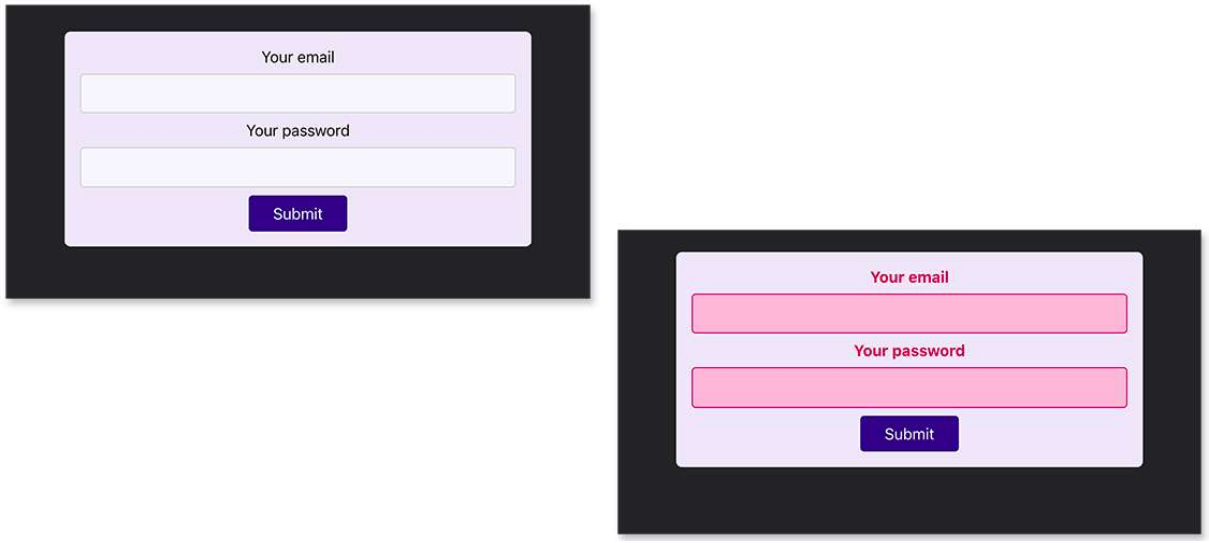

Figure 6.9: The final user interface with invalid input values highlighted in red

Test your new interface to see CSS scoped styles in action.

> **NOTE**
>
> You will find all code files used for this solution at https://packt.link/fPvg1.

CHAPTER 7: PORTALS & REFS

ACTIVITY 7.1: EXTRACT USER INPUT VALUES

Solution:

After downloading the code and running **`npm install`** in the project folder (to install all the required dependencies), the solution steps are as follows:

1. Inside the **`Form`** component, create two ref objects via **`useRef()`**. Make sure that you don't forget to run **`import { useRef } from 'react'`**:

```
import { useRef } from 'react';

function Form() {
  const nameRef = useRef();
  const programRef = useRef();

  // ... other code ...
}
```

2. Still in the **`Form`** component, connect the ref objects to their respective JSX elements (**`<input>`** and **`<select>`**) via the special **`ref`** prop:

```
return (
  <form onSubmit={formSubmitHandler}>
    <div className="form-control">
      <label htmlFor="name">Your name</label>
      <input type="text" id="name" ref={nameRef} />
    </div>
    <div className="form-control">
      <label htmlFor="program">Choose your program</label>
      <select id="program" ref={programRef}>
        <option value="basics">The Basics</option>
        <option value="advanced">Advanced Concepts</option>
        <option value="mastery">Mastery</option>
      </select>
    </div>
    <button>Submit</button>
  </form>
);
```

3. In the **`formSubmitHandler`** function, extract the entered values by accessing the connected ("stored") DOM elements via the special **`current`** property on the ref object. Also output the extracted values to the console via **`console.log()`**:

```
function formSubmitHandler(event) {
  event.preventDefault();

  const enteredName = nameRef.current.value;
  const selectedProgram = programRef.current.value;

  console.log('Entered Name: ' + enteredName);
  console.log('Selected Program: ' + selectedProgram);
}
```

The expected result (user interface) should look like this:

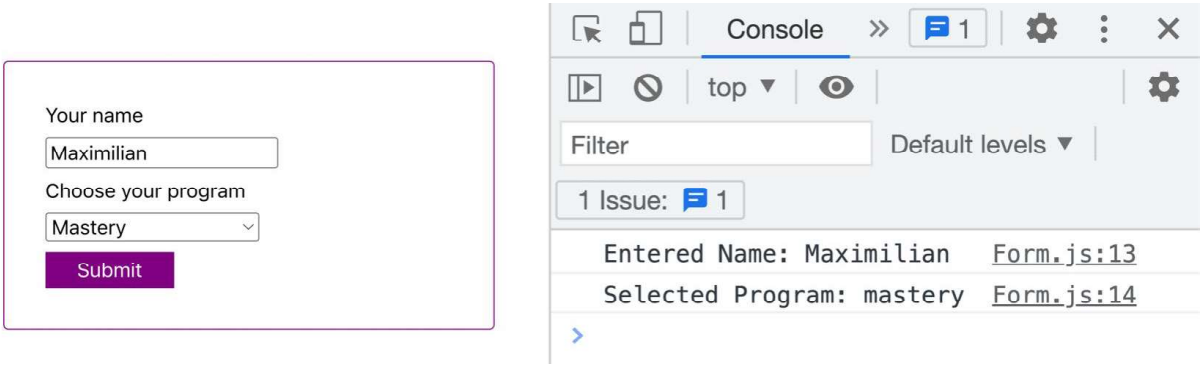

Figure 7.11: The browser developer tools console outputs the selected values

> **NOTE**
>
> You will find all code files used for this solution at https://packt.link/RnnoT.

ACTIVITY 7.2: ADD A SIDE-DRAWER

Solution:

After downloading the code and running **`npm install`** to install all the required dependencies, the solution steps are as follows:

1. Add the conditional rendering logic by adding a **`drawerIsOpen`** state (via **`useState()`**) to the **`MainNavigation`** component. Set the state to **`false`** initially and in a function that should later be executed whenever the backdrop is clicked, while setting the state to **`true`** in a function that is executed upon a click of the menu button:

```
import { useState } from 'react';

import SideDrawer from './SideDrawer';
import classes from './MainNavigation.module.css';

function MainNavigation() {
  const [drawerIsOpen, setDrawerIsOpen] = useState(false);

  function openDrawerHandler() {
    setDrawerIsOpen(true);
  }

  function closeDrawerHandler() {
    setDrawerIsOpen(false);
  }

  return (
    <>
      <header className={classes.header}>
        <h1>Demo App</h1>
        <button className={classes.btn} onClick={openDrawerHandler}>
          <div />
          <div />
          <div />
        </button>
```

```
        </header>
        {drawerIsOpen && <SideDrawer />}
      </>
    );
  }

  export default MainNavigation;
```

2. Pass a pointer to the **`closeDrawerHandler`** function to the **`SideDrawer`** component (here, this is via a prop called **`onClose`**: **`<SideDrawer onClose={closeDrawerHandler} />`**) and execute that function inside the **`SideDrawer`** component whenever the **`<div>`** backdrop is clicked:

```
import classes from './SideDrawer.module.css';

function SideDrawer({ onClose }) {
  return (
    <>
      <div className={classes.backdrop} onClick={onClose} />
      <aside className={classes.drawer}>
        <nav>
          <ul>
            <li>
              <a href="/">Dashboard</a>
            </li>
            <li>
              <a href="/products">All Products</a>
            </li>
            <li>
              <a href="/profile">Your Profile</a>
            </li>
          </ul>
        </nav>
      </aside>
    </>
  );
}

export default SideDrawer;
```

3. To control where the side drawer JSX elements are inserted into the DOM, use React's portal feature. As a first step, add an *"injection hook"* to the **`public/index.html`** file:

```
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
  <div id="drawer"></div>
</bod>
```

4. In this case, **`<div id="drawer">`** was inserted at the end of the **`<body>`** element to make sure that it would be positioned (visually) above any other overlays that might exist.

5. Use the newly added hook (**`<div id="drawer">`**) and the **`createPortal()`** function of **`react-dom`** inside the **`SideDrawer`** component to instruct React to render the component's JSX code in this specific place in the DOM:

```
import { createPortal } from 'react-dom';

import classes from './SideDrawer.module.css';

function SideDrawer({ onClose }) {
  return createPortal(
    <>
      <div className={classes.backdrop} onClick={onClose} />
      <aside className={classes.drawer}>
        <nav>
          <ul>
            <li>
              <a href="/">Dashboard</a>
            </li>
            <li>
              <a href="/products">All Products</a>
            </li>
            <li>
              <a href="/profile">Your Profile</a>
            </li>
          </ul>
        </nav>
      </aside>
    </>,
```

```
        document.getElementById('drawer')
      );
    }

    export default SideDrawer;
```

The final user interface should look and behave like this:

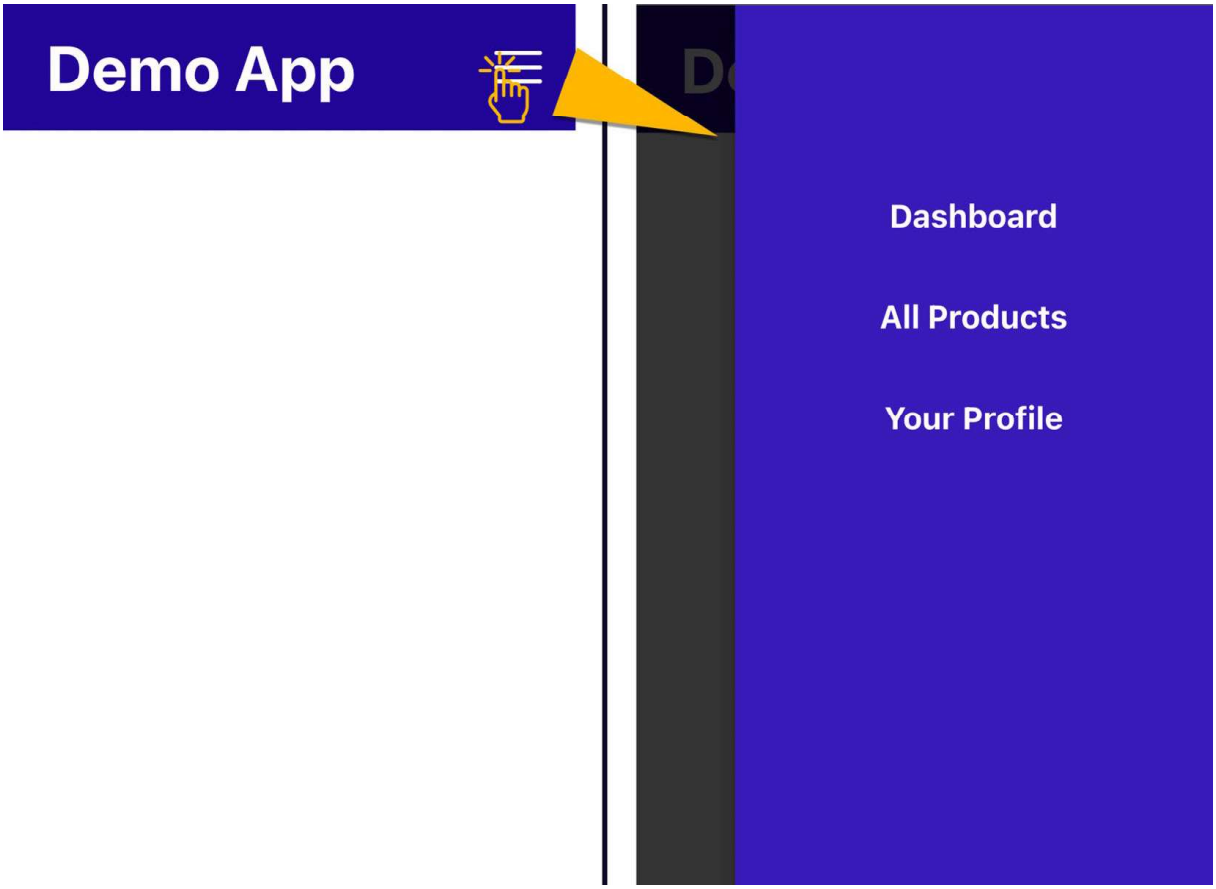

Figure 7.12: A click on the menu button opens the side drawer

Upon clicking on the menu button, the side drawer opens. If the backdrop behind the side drawer is clicked, it should close again.

The final DOM structure (with the side drawer opened) should look like this:

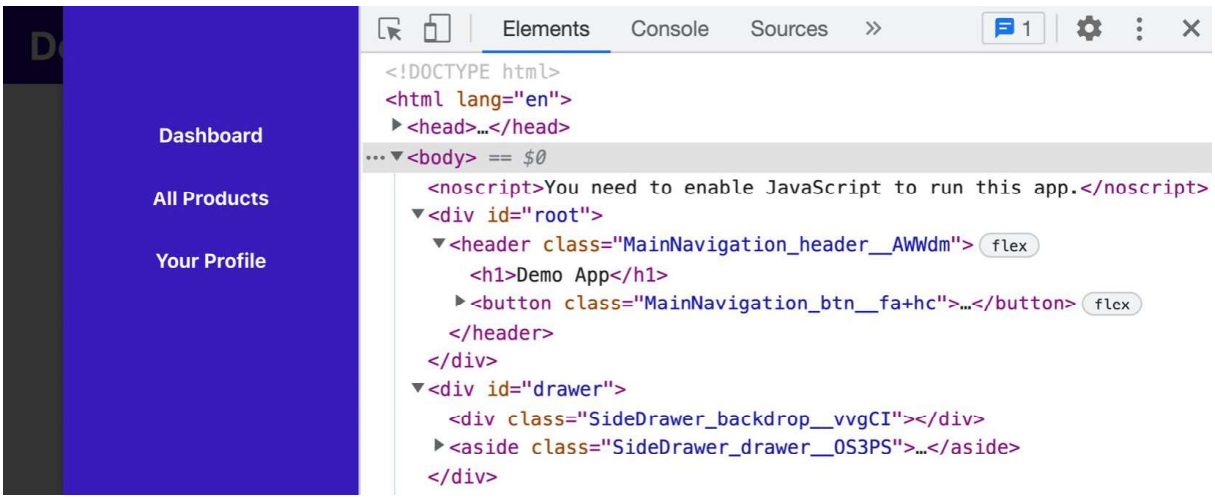

Figure 7.13: The drawer-related elements are inserted in a separate place in the DOM

The side drawer-related DOM elements (the **`<div>`** backdrop and **`<aside>`**) are inserted into a separate DOM node (**`<div id="drawer">`**).

> **NOTE**
>
> You will find all code files used for this solution at https://packt.link/APvHg.

CHAPTER 8: HANDLING SIDE EFFECTS

ACTIVITY 8.1: BUILDING A BASIC BLOG

Solution:

After downloading the code and running **`npm install`** in the project folder to install all required dependencies, the solution steps are as follows:

1. Inside the **`NewPost`** component, in the **`submitHandler`** function, use the **`fetch()`** function to send a **`POST`** request to https://jsonplaceholder.typicode.com/posts:

```
function submitHandler(event) {
  event.preventDefault();
  fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify({ title: enteredTitle }),
  });
}
```

   You can confirm that everything works as intended and view the network request in the browser developer tools via the **`Network`** tab:

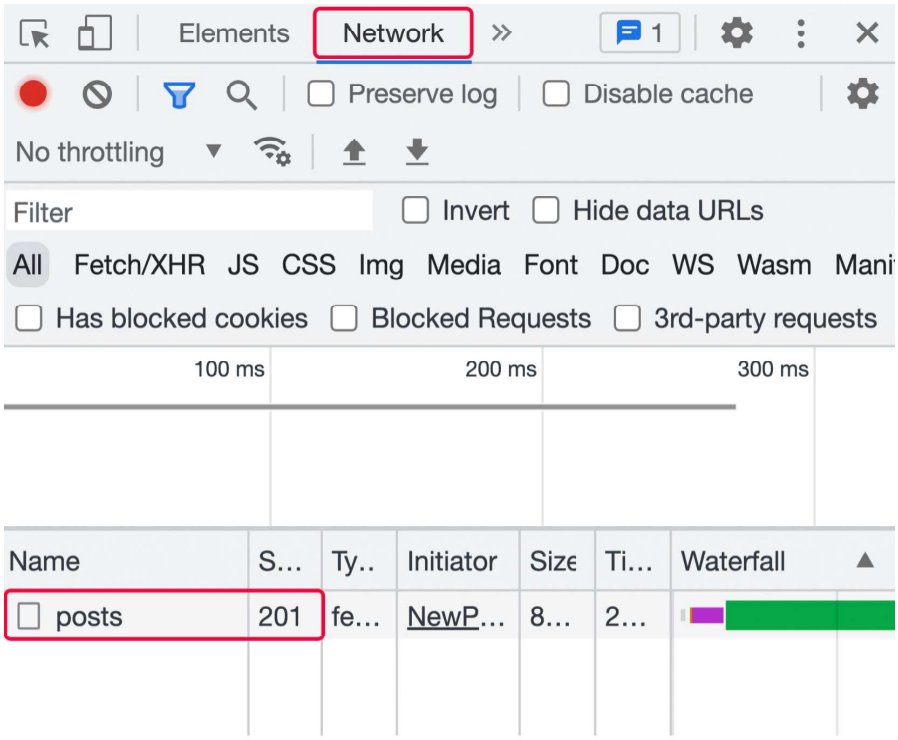

Figure 8.10: The network request is sent successfully

`useEffect()` is not required for this outgoing request because the HTTP request is not triggered when the component function is invoked but rather when the form is submitted. Indeed, if you tried to use **`useEffect()`** for this request, you'd have to get creative as using **`useEffect()`** inside of **`submitHandler`** violates the rules of Hooks.

2. To fetch blog posts when the app is rendered, you should import **`useEffect`** from react in **`App.js`**. Thereafter, inside the **`App`** component, call **`useEffect()`** directly in the component function and pass an effect function (an empty function for now) and an empty dependencies array to the effect:

```
import { useEffect } from 'react';
import BlogPosts from './components/BlogPosts';
import NewPost from './components/NewPost';
function App() {
  useEffect(function () {}, []);
  return (
    <>
      <NewPost />
      <BlogPosts />
    </>
  );
}
export default App;
```

3. Inside the effect function (in the **`App`** component), send a **`GET`** request via the **`fetch()`** function to https://jsonplaceholder.typicode.com/posts. To extract the response and response data in a convenient way, wrap the HTTP request code in a separate **`async function loadPosts`** (defined as part of the effect function), which is called in the effect function. **`await`** both the response as well as the data extracted from the response (via the **`json()`** method):

```
useEffect(function () {
  async function loadPosts() {
    const response = await fetch(
      'https://jsonplaceholder.typicode.com/posts'
    );
```

```
    const blogPosts = await response.json();
  }

  loadPosts();
}, []);
```

Keep in mind that the effect function itself must not be turned into an **`async`** function since it must not return a promise.

4. Add a state value (named **`loadedPosts`** here) to the **`App`** component and set the state value from inside **`loadPosts()`** (the function in the effect function) to the fetched **`blogPosts`** value. Pass the **`loadedPosts`** state value to **`<BlogPosts>`** via props (for example, via a **`posts`** prop):

```
import { useState, useEffect } from 'react';

import BlogPosts from './components/BlogPosts';
import NewPost from './components/NewPost';

function App() {
  const [loadedPosts, setLoadedPosts] = useState([]);

  useEffect(function () {
    async function loadPosts() {
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );

      const blogPosts = await response.json();
      setLoadedPosts(blogPosts);
    }

    loadPosts();
  }, []);

  return (
    <>
      <NewPost />
      <BlogPosts posts={loadedPosts} />
    </>
  );
```

```
}

export default App;
```

5. Inside of **`BlogPosts`**, render the list of blog posts received via props by mapping all blog post list items to **`<li>`** elements. Output the blog post titles in the list:

```
function BlogPosts({ posts }) {
  return (
    <ul className={classes.posts}>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}
```

6. For the bonus task, inside of the **`NewPost`** component, add a new state value called **`isSendingRequest`** via **`useState()`**. Set the state to **`true`** right before the **`POST`** HTTP request is sent and to **`false`** thereafter. Wait for the request to complete by turning **`submitHandler`** into an **`async`** function and await the **`fetch()`** function call:

```
import { useState } from 'react';

import classes from './NewPost.module.css';

function NewPost() {
  const [enteredTitle, setEnteredTitle] = useState('');
  const [isSendingRequest, setIsSendingRequest] = useState(false);

  function updateTitleHandler(event) {
    setEnteredTitle(event.target.value);
  }

  async function submitHandler(event) {
    event.preventDefault();

    setIsSendingRequest(true);

    await fetch('https://jsonplaceholder.typicode.com/posts', {
```

```
        method: 'POST',
        body: JSON.stringify({ title: enteredTitle }),
      });

      setIsSendingRequest(false);
      setEnteredTitle('');
    }

    // JSX code didn't change…
  }

  export default NewPost;
```

7. Still inside **`NewPost`**, set the **`<button>`** caption conditionally based on **`isSendingRequest`** to either show "Saving..." (if **`isSendingRequest`** is **`true`**) or "Save":

```
  return (
    <form onSubmit={submitHandler} className={classes.form}>
      <div>
        <label>Title</label>
        <input
          type="text"
          onChange={updateTitleHandler}
          value={enteredTitle} />
      </div>
      <button disabled={isSendingRequest}>
        {isSendingRequest ? 'Saving...' : 'Save'}
      </button>
    </form>
  );
```

The expected result should be a user interface that looks like this:

Figure 8.11: The final user interface

> **NOTE**
>
> All code files used for this solution can be found at https://packt.link/NoEZz.

CHAPTER 9: BEHIND THE SCENES OF REACT & OPTIMIZATION OPPORTUNITIES

ACTIVITY 9.1: OPTIMIZE AN EXISTING APP

Solution:

Naturally, for an activity like this, chances are high that every developer comes up with different ideas for optimizing the app. And indeed, there will be more than one correct solution. Below, you will find solution steps describing the optimizations I would implement. Definitely feel free to go beyond those steps. For example, in the solution provided below, the code structure will not be changed (that is, components will not be broken apart, etc.). You could absolutely consider such measures as well though.

Four areas of possible improvement were identified for this solution:

- In the **`Authentication`** component, the app allows for switching between the **`Login`** and **`Signup`** forms. While the **`Login`** form is always loaded (it's the first thing every page visitor sees), the **`Signup`** area will not always be needed—existing users will very likely not create a new account after all. Therefore, loading the code for **`Signup`** lazily makes sense, especially since the **`Signup`** component also **internally** uses the **`Validation`** component, which is rather complex and even includes a third-party package (**`react-icons`**). Being able to load all that code only when it's needed ensures that the initially loaded code bundle is kept lean.
- In the **`Validation`** component, both the email addresses and the password are validated. While the email validation is relatively simple and straightforward, validating the password involves various regular expressions (that is, matching for text patterns). Avoiding unnecessary execution of the password validation code therefore makes sense.
- In addition, the **`Validation`** component function will be re-executed whenever the **`Signup`** component function is invoked. Since **`Signup`** also includes state for conditionally showing or hiding extra information (via the **`More Information`** button), ensuring that the **`Validation`** component is not re-evaluated unnecessarily is another important step.

- The last identified area of improvement can be found in the **`Login`** component. There, the **`ResetPassword`** component code should only be loaded when it's needed. Typically, resetting a password involves quite a bit of code and logic (for example, asking a security question, checking for bots, etc.), therefore only loading that code when it's really needed makes sense, especially since most users will not need that feature for most of their visits.

As mentioned, you could identify other areas of improvement as well. However, as explained throughout this chapter, you should be careful not to overoptimize.

Below are the solution steps to tackle the four problems described above:

1. To load the **`Signup`** component lazily (in the **`Authentication`** component), you must import two things from react: the **`lazy()`** function and the **`Suspense`** component:

```
import { useState, lazy, Suspense } from 'react';
```

2. As a next step, the **`Signup`** component import (**`import Signup from './Signup/Signup'`**) should be removed from **`Authentication.js`**. Instead, add code to store the lazy-loaded component in a variable or constant named **`Signup`**:

```
const Signup = lazy(() => import('./Signup/Signup'));
```

3. It's important that the variable or constant is called **`Signup`**, not **`signup`**. It must start with a capital character since it's used as a JSX element.

4. This constant can now be used like a component in your JSX code. However, since it's loaded lazily, the component must be wrapped with the imported **`Suspense`** component (in **`Authentication`**'s returned JSX code). The **`Suspense`** component also needs a fallback JSX element (for example, **`<p>Loading...</p>`**, passed to **`Suspense`** via its **`fallback`** prop).
The final **`Authentication`** component looks like this:

```
import { useState, lazy, Suspense } from 'react';

import Login from './Login/Login';
import classes from './Authentication.module.css';

const Signup = lazy(() => import('./Signup/Signup'));
```

```
function Authentication() {
  const [mode, setMode] = useState('login');

  function switchAuthModeHandler() {
    setMode((prevMode) => (prevMode === 'login' ? 'signup' :
'login'));
  }

  let authElement = <Login />;
  let switchBtnCaption = 'Create a new account';

  if (mode !== 'login') {
    authElement = <Signup />;
    switchBtnCaption = 'Login instead';
  }

  return (
    <div className={classes.auth}>
      <h1>You must authenticate yourself first!</h1>
      <Suspense fallback={<p>Loading...</p>}>{authElement}</Suspense>
      <button
        className={classes.btn}
        onClick={switchAuthModeHandler}>
          {switchBtnCaption}
      </button>
    </div>
  );
}

export default Authentication;
```

5. Next, in order to avoid unnecessary code execution in the **`Validation`** component, start by importing the **`useMemo()`** Hook from **`react`** (in the **`Validation.js`** file):

```
import { useMemo } from 'react';
```

6. Use the **`useMemo()`** Hook to wrap the password validation code (the code between lines 12 to 20 in **`Validation.js`**) with it. Extract that code into an anonymous function that is passed as a first argument to **`useMemo()`**. Make sure to return an object that groups the three-password validation Booleans together. Also, pass a second argument to **`useMemo()`**. It's the dependencies array and should contain the **`password`** variable, since changes to **`password`** should cause the code to execute again. As a last step, store the value returned by **`useMemo()`** in the **`passwordValidityData`** constant. The final code looks like this:

```
const passwordValidityData = useMemo(() => {
  const pwHasMinLength = password.length >= 8;
  const pwHasMinSpecChars = specCharsRegex.test(password);
  const pwHasMinNumbers = numberRegex.test(password);
  return {
    length: pwHasMinLength,
    specChars: pwHasMinSpecChars,
    numbers: pwHasMinNumbers,
  };
}, [password]);
```

7. To ensure that the **`Validation`** component function itself is not executed unnecessarily, wrap it with React's **`memo()`** function. To do this, as a first step, import **`memo`** from **`react`** (still in **`Validation.js`**):

```
import { useMemo, memo } from 'react';
```

8. Thereafter, wrap the exported **`Validation`** function with **`memo()`**:

```
export default memo(Validation);
```

9. To improve the code in the **`Login`** component, add lazy loading for the **`ResetPassword`** component. As a first step, import both **`lazy`** and **`Suspense`** from **`react`** (in **`Login.js`**):

```
import { useState, lazy, Suspense } from 'react';
```

10. Next, replace the **`ResetPassword`** import with a constant or variable that stores the result of calling **`lazy()`** and passing a dynamic import function to it:

```
const ResetPassword = lazy(() => import('./ResetPassword));
```

11. Finally, wrap **`ResetPassword`** in your JSX code with React's **`Suspense`** component and pass an appropriate fallback element (e.g., **`<p>Loading...</p>`**) to **`Suspense`**. The final **`Login`** component function looks like this:

```
import { useState, lazy, Suspense } from 'react';

const ResetPassword = lazy(() => import('./ResetPassword'));

function Login() {
  const [isResetting, setIsResetting] = useState();

  function loginHandler(event) {
    event.preventDefault();
  }

  function startResetPasswordHandler() {
    setIsResetting(true);
  }

  function finishResetPasswordHandler() {
    setIsResetting(false);
  }

  return (
    <>
      <form onSubmit={loginHandler}>
        <div className="form-control">
          <label htmlFor="email">Email</label>
          <input id="email" type="email" />
        </div>
        <div className="form-control">
          <label htmlFor="password">Password</label>
          <input id="password" type="password" />
        </div>
        <button className="main-btn">Login</button>
      </form>
      <button className="alt-btn" onClick={startResetPasswordHandler}>
        Reset password
```

```
          </button>
          <Suspense fallback={<p>Loading...</p>}>
            {isResetting && <ResetPassword
                              onFinish={finishResetPasswordHandler} />}
          </Suspense>
        </>
      );
    }

    export default Login;
```

You can tell that you came up with a good solution and sensible adjustments if you can see extra code fetching network requests (in the **`Network`** tab of your browser developer tools) for clicking on the **`Reset password`** or **`Create a new account`** buttons:

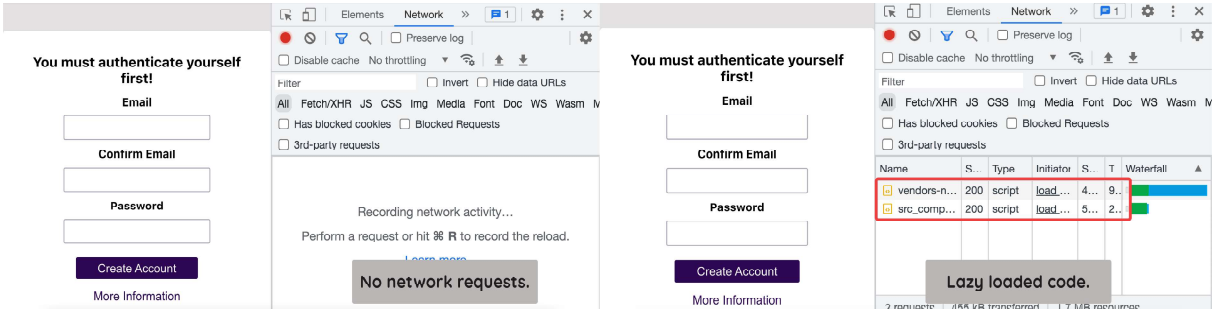

Figure 9.22: In the final solution, some code is lazy loaded

In addition, you should see no **`Validated`** password. console message when typing into the email input fields (**`Email`** and **`Confirm Email`**) of the signup form (that is, the form you switch to when clicking **`Create a new account`**):

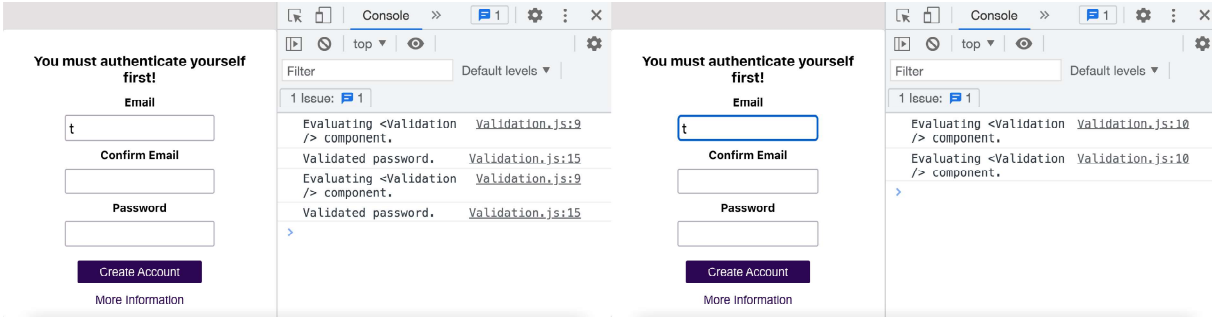

Figure 9.23: No "Validated password." output in the console

You also shouldn't get any console outputs when clicking the **`More Information`** button:

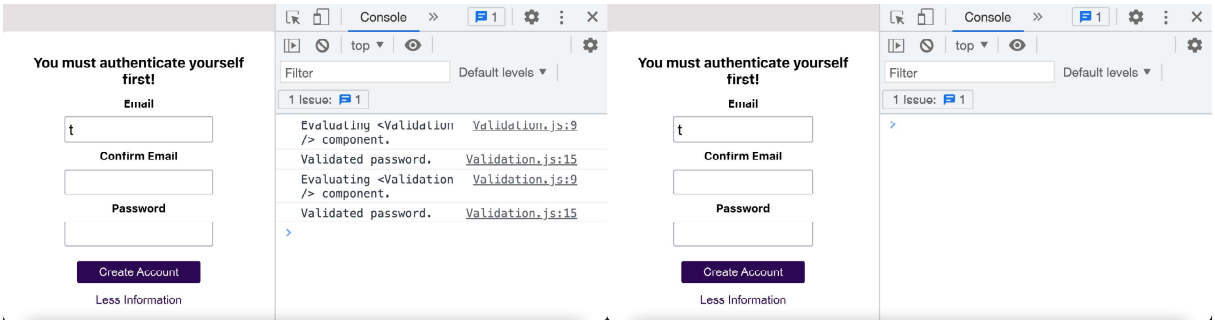

Figure 9.24: No console messages when clicking "More Information"

> **NOTE**
>
> All code files used for this solution can be found at https://packt.link/gBNHd.

CHAPTER 10: WORKING WITH COMPLEX STATE

ACTIVITY 10.1: MIGRATING AN APP TO THE CONTEXT API

Solution:

Perform the following solution steps to complete this activity:

1. As a first step, create a **`store/cart-context.js`** file. This file will hold the context object and the **`Provider`** component that contains the cross-component state logic (which is exposed via the **`Provider`** component).
2. In the newly added **`cart-context.js`** file, create a new **`CartContext`** via React's **`createContext()`** function. To get better IDE auto-completion, the initial context value can be set to an object that has an **`items`** property (an empty array) and two methods: an (empty) **`addItem`** and an (empty) **`removeItem`** method. The created context also must be exported, so that it can be referenced by other project files.

   This is the final cart-context.js file content (for the moment):

```
import { createContext } from 'react';

const CartContext = createContext({
  items: [],
  addItem: () => {},
  removeItem: () => {},
});

export default CartContext;
```

3. To provide the context, the **`CartContext.Provider`** component will be used. To manage the cross-component state in a central place (and not bloat any other component with that logic), you should add an extra React component to the **`cart-context.js`** file: a **`CartContextProvider`** component. That component returns its **`children`** (i.e., content passed between its opening and closing tags), wrapped by **`CartContext.Provider`**:

```
export function CartContextProvider({ children }) {
  return (
    <CartContext.Provider>
      {children}
    </CartContext.Provider>
  );
}
```

4. The **`Provider`** component requires a **`value`** prop. That **`value`** prop contains the actual value that is distributed to all components that subscribe to the context. In this project, the **`value`** prop is set equal to an object that contains an **`items`** property and the two methods defined in the initial context value previously (in *Step 2*). The two methods (**`addItem`** and **`removeItem`**) are empty named functions for the moment but will be populated with logic over the next steps, as follows:

```
export function CartContextProvider({ children }) {
  function addItemHandler(item) {
    // to be added ...
  }

  function removeItemHandler(itemId) {
    // to be added ...
  }

  const contextVal = {
    items: cartItems,
    addItem: addItemHandler,
    removeItem: removeItemHandler
  }

  return (
    <CartContext.Provider value={contextVal}>
      {children}
```

```
      </CartContext.Provider>
    );
  }
```

5. To make the context value dynamic, the **`CartContextProvider`** component must start managing state (via **`useState()`**) and distribute that state via the context value. In order to trigger state updates, appropriate logic must be added to **`addItemHandler`** and **`removeItemHandler`**. The final **`cart-context.js`** file therefore contains the following code:

```
import { createContext, useState } from 'react';

const CartContext = createContext({
  items: [],
  addItem: () => {},
  removeItem: () => {},
});

export function CartContextProvider({ children }) {
    const [cartItems, setCartItems] = useState([]);

  function addItemHandler(item) {
    setCartItems((prevItems) => [...prevItems, item]);
  }

  function removeItemHandler(itemId) {
    setCartItems(
      (prevItems) => prevItems.filter((item) => item.id !== itemId)
    );
  }

  const contextVal = {
    items: cartItems,
    addItem: addItemHandler,
    removeItem: removeItemHandler
  }

  return (
    <CartContext.Provider value={contextVal}>
      {children}
    </CartContext.Provider>
```

```
    );
  }

  export default CartContext;
```

6. Now that all cross-component state management logic has been moved into the **`CartContextProvider`** component, that code must be removed from the **`App`** component (where cross-component state was managed before, when it was lifted up).

   You also must remove all props (and their usage) that were used for passing cart-item-related state down to other components. For the moment, the App component looks like this:

```
import Events from './components/Events/Events';
import MainHeader from './components/MainHeader/MainHeader';

function App() {
  return (
    <>
      <MainHeader />
      <main>
        <Events />
      </main>
    </>
  );
}

export default App;
```

7. The **`CartContextProvider`** component must be wrapped around that part of the overall application component tree that needs access to the context. In this example app, that means that all JSX code in the **`App.js`** file should be wrapped (since both **`MainHeader`** and the **`Events`** component need access to the context value):

```
import Events from './components/Events/Events';
import MainHeader from './components/MainHeader/MainHeader';
import { CartContextProvider } from './store/cart-context';

function App() {
  return (
    <>
```

```
        <CartContextProvider>
          <MainHeader />
          <main>
            <Events />
          </main>
        </CartContextProvider>
      </>
    );
  }

  export default App;
```

8. Now, all components that need context access (either for reading the context value or for calling one of the exposed context value methods) can subscribe to the context via React's **`useContext()`** Hook. In this example project, the **`MainHeader`**, **`EventItem`**, and **`Cart`** components need access. The **`MainHeader`** component therefore uses the **`useContext()`** Hook (and the received context value) like this:

```
import { useContext, useState } from 'react';

import CartContext from '../../store/cart-context';
import Cart from '../Cart/Cart';
import classes from './MainHeader.module.css';

function MainHeader({ cartItems }) {
  const cartCtx = useContext(CartContext);
  const [modalIsOpen, setModalIsOpen] = useState();

  function openCartModalHandler() {
    setModalIsOpen(true);
  }

  function closeCartModalHandler() {
    setModalIsOpen(false);
  }

  const numCartItems = cartCtx.items.length;

  return (
    <>
```

```
      <header className={classes.header}>
        <h1>StateEvents Shop</h1>
        <button onClick={openCartModalHandler}>Cart
({numCartItems})</button>
      </header>
      {modalIsOpen && <Cart onClose={closeCartModalHandler} />}
    </>
  );
}

export default MainHeader;
```

9. Ensure the **`EventItem`** component looks like this:

```
import { useContext } from 'react';

import CartContext from '../../store/cart-context';
import classes from './EventItem.module.css';

function EventItem({ event }) {
  const cartCtx = useContext(CartContext);

  const isInCart = cartCtx.items.some((item) => item.id === event.
id);

  let buttonCaption = 'Add to Cart';
  let buttonAction = () => cartCtx.addItem(event);

  if (isInCart) {
    buttonCaption = 'Remove from Cart';
    buttonAction = () => cartCtx.removeItem(event.id);
  }

  return (
    <li className={classes.event}>
      <img src={event.image} alt={event.title} />
      <div className={classes.content}>
        <h2>{event.title}</h2>
        <p className={classes.price}>${event.price}</p>
        <p>{event.description}</p>
        <div className={classes.actions}>
          <button onClick={buttonAction}>{buttonCaption}</button>
```

```
          </div>
        </div>
      </li>
    );
  }

  export default EventItem;
```

10. And confirm that the **`Cart`** component contains this code:

```
  import { useContext } from 'react';
  import ReactDOM from 'react-dom';

  import CartContext from '../../store/cart-context';
  import classes from './Cart.module.css';

  function Cart({ onClose }) {
    const cartCtx = useContext(CartContext);

    const total = cartCtx.items.reduce(
      (prevVal, item) => prevVal + item.price,
      0
    );

    return ReactDOM.createPortal(
      <>
        <div className={classes.backdrop} onClick={onClose} />
        <aside className={classes.cart}>
          <h2>Your Cart</h2>
          <ul>
            {cartCtx.items.map((item) => (
              <li key={item.id}>
                {item.title} (${item.price})
              </li>
            ))}
          </ul>
          <p className={classes.total}>Total: ${total}</p>
          <div className={classes.actions}>
            <button onClick={onClose}>Close</button>
            <button onClick={onClose}>Buy</button>
          </div>
```

```
        </aside>
      </>,
      document.getElementById('modal')
    );
  }

  export default Cart;
```

> **NOTE**
>
> All code files used for this activity can be found at https://packt.link/vjkCr.

ACTIVITY 10.2: REPLACING USESTATE() WITH USEREDUCER()

Solution:

Perform the following solution steps to complete this activity:

1. Remove the existing logic in the **`Form`** component that uses the **`useState()`** Hook for state management.
2. Import and use the **`useReducer()`** Hook in the **`Form`** component. For the moment, pass an empty (newly added) reducer function as a first argument to **`useReducer()`**. Make sure to create the reducer function outside of the component function. Also, pass an initial state value as the second argument to **`useReducer()`**. That initial value should be an object containing values and validity information for both the **`email`** and **`password`** fields. Overall, for the moment, the code in **`Form.js`** looks like this:

```
import { useReducer } from 'react';

import classes from './Form.module.css';

const initialFormState = {
  email: {
    value: '',
    isValid: false,
  },
  password: {
    value: '',
    isValid: false,
```

```
  },
};

function formReducer(state, action) {
  // to be added
}

function Form() {
  useReducer(formReducer, initialFormState);

  const formIsValid = true; // will be changed!

  function changeEmailHandler(event) {
    const value = event.target.value;
  }

  function changePasswordHandler(event) {
    const value = event.target.value;
  }

  function submitFormHandler(event) {
    event.preventDefault();

    if (!formIsValid) {
      alert('Invalid form inputs!');
      return;
    }

    console.log('Good job!');
    console.log(formState.email.value, formState.password.value);
  }

  return (
    <form className={classes.form} onSubmit={submitFormHandler}>
      <div className={classes.control}>
        <label htmlFor="email">Email</label>
        <input id="email" type="email" onChange={changeEmailHandler}
/>
      </div>
      <div className={classes.control}>
        <label htmlFor="password">Password</label>
```

```
          <input id="password"
            type="password" onChange={changePasswordHandler} />
        </div>
        <button>Submit</button>
      </form>
    );
  }

  export default Form;
```

3. As the next step, fill the reducer function. It must handle two main types of actions: an email address change and a password change. In both cases, the state value must be updated appropriately (i.e., the newly entered value must be stored and the validity of the input must be derived and stored as well). You also must return a default value in case of unknown action types. The updated reducer function looks like this:

```
function formReducer(state, action) {
  if (action.type === 'EMAIL_CHANGE') {
    return {
      ...state,
      email: {
        value: action.payload,
        isValid: action.payload.includes('@'),
      },
    };
  }

  if (action.type === 'PASSWORD_CHANGE') {
    return {
      ...state,
      password: {
        value: action.payload,
        isValid: action.payload.trim().length > 7,
      },
    };
  }

  return initialFormState;
}
```

Please note that the **`type`** values you want to support (**`'EMAIL_CHANGE'`** and **`'PASSWORD_CHANGE'`** in the preceding snippet) are up to you. You can use any identifier values of your choice. You can also assign a different name to the **`type`** property (e.g., use **`action.identifier`** instead of **`action.type`**). Similarly, you can also use any other name than **`payload`** for the extra data required by the state updates. If you do choose different identifier values or property names, you must use the same values and names in the following steps.

4. Update the code where **`useReducer()`** is called. That Hook returns an array with exactly two elements: the current state value and a function that can be used for dispatching actions. Store both elements in different constants (or variables):

```
// ... more code ...
function Form() {
  const [formState, dispatch] = useReducer(formReducer,
initialFormState);
  // ... more code ...
}
```

5. Dispatch the actions in the appropriate places. **`'EMAIL_CHANGE'`** is dispatched in the **`changeEmailHandler`** function and **`'PASSWORD_CHANGE'`** is dispatched in the **`changePasswordHandler`** function. Both actions also need payload data (the value entered by the user). Pass that data along with the action via the **`payload`** property:

```
// ... more code ...
function changeEmailHandler(event) {
  const value = event.target.value;
  dispatch({ type: 'EMAIL_CHANGE', payload: value });
}

function changePasswordHandler(event) {
  const value = event.target.value;
  dispatch({ type: 'PASSWORD_CHANGE', payload: value });
}
// ... more code ...
```

6. Lastly, use the state managed via **`useReducer()`** in all the places where it's needed (in the **`Form`** component) to derive the **`formIsValid`** Boolean and log (via **`console.log()`**) the entered values inside the **`submitFormHandler`** function. The finished **`Form`** component code looks like this:

```
function Form() {
  const [formState, dispatch] = useReducer(formReducer,
initialFormState);

  const formIsValid = formState.email.isValid && formState.password.
isValid;

  function changeEmailHandler(event) {
    const value = event.target.value;
    dispatch({ type: 'EMAIL_CHANGE', payload: value });
  }

  function changePasswordHandler(event) {
    const value = event.target.value;
    dispatch({ type: 'PASSWORD_CHANGE', payload: value });
  }

  function submitFormHandler(event) {
    event.preventDefault();

    if (!formIsValid) {
      alert('Invalid form inputs!');
      return;
    }

    console.log('Good job!');
    console.log(formState.email.value, formState.password.value);
  }
```

```
  return (
    <form className={classes.form} onSubmit={submitFormHandler}>
      <div className={classes.control}>
        <label htmlFor="email">Email</label>
        <input id="email" type="email" onChange={changeEmailHandler}
/>
      </div>
      <div className={classes.control}>
        <label htmlFor="password">Password</label>
        <input id="password" type="password"
onChange={changePasswordHandler} />
      </div>
      <button>Submit</button>
    </form>
  );
}
```

> **NOTE**
>
> All code files used for this activity can be found at https://packt.link/Ua7VH.

CHAPTER 11: BUILDING CUSTOM REACT HOOKS

ACTIVITY 11.1: BUILD A CUSTOM KEYBOARD INPUT HOOK

Solution:

Perform the following solution steps to complete this activity:

1. As a first step, create a **`hooks/use-key-event.js`** file. This file will hold the custom Hook function.
2. Create the **`useKeyEvent`** Hook function in the newly added **`use-key-event.js`** file. Also, export the **`useKeyEvent`** function so that it can be used in other files (it will be used in the **`App`** component later):

```
function useKeyEvent() {
  // logic to be added...
}

export default useKeyEvent;
```

3. Move the **`useEffect()`** import and call (and all the logic inside of it) from the **`App`** component body to the **`useKeyEvent`** function body:

```
import { useEffect } from 'react';

function useKeyEvent() {
  useEffect(() => {
    function keyPressedHandler(event) {
      const pressedKey = event.key;

      if (!['s', 'c', 'p'].includes(pressedKey)) {
        alert('Invalid key!');
        return;
      }
      setPressedKey(pressedKey);
    }

    window.addEventListener('keydown', keyPressedHandler);

    return () => window.removeEventListener('keydown',
keyPressedHandler);
```

```
  }, []);
}

export default useKeyEvent;
```

4. Make sure to remove that **`useEffect()`** logic (and the **`useEffect`** import) from the **`App`** component file. At the moment, the **`useKeyEvent`** Hook won't work correctly as there are multiple problems:

   - Inside the effect function, it's calling **`setPressedKey(pressedKey)`** without that function existing in the Hook function.
   - The custom Hook does not communicate with the component in which it might be used. It should return the key that was pressed (after validating that it's an allowed key).
   - The allowed keys (**`'s'`**, **`'c'`**, and **`'p'`**) are hardcoded into the custom Hook.

   These issues will be fixed over the next steps.

5. Start by adding state to the **`useKeyEvent`** Hook. Import and use **`useState`** to manage the **`pressedKey`** state:

```
import { useEffect, useState } from 'react';

function useKeyEvent() {
  const [pressedKey, setPressedKey] = useState();

  // ... unchanged rest of the code
}
```

6. Add a **`return`** statement at the end of the custom Hook function and return the **`pressedKey`** state value. Since it's the only value that must be returned, you don't need to group it into an array or object:

```
function useKeyEvent() {
  const [pressedKey, setPressedKey] = useState();

  useEffect(() => {
    // unchanged logic ...
  }, []);

  return pressedKey;
}
```

7. Make the Hook more reusable by converting the hardcoded list of allowed keys (**`['s', 'c', 'p']`**) to a parameter (**`allowedKeys`**) that is received and used by the **`useKeyEvent`** Hook function. Don't forget to add the parameter variable as a dependency to the **`useEffect()`** dependencies array since the value is used inside of **`useEffect()`**:

```
import { useEffect, useState } from 'react';

function useKeyEvent(allowedKeys) {
  const [pressedKey, setPressedKey] = useState();

  useEffect(() => {
    function keyPressedHandler(event) {
      const pressedKey = event.key;

      if (!allowedKeys.includes(pressedKey)) {
        alert('Invalid key!');
        return;
      }
      setPressedKey(pressedKey);
    }

    window.addEventListener('keydown', keyPressedHandler);

    return () => window.removeEventListener('keydown',
keyPressedHandler);
  }, [allowedKeys]);

  return pressedKey;
}

export default useKeyEvent;
```

8. The custom Hook is now finished and hence can be used in other components. Import and use it in the **App** component like this:

```
import useKeyEvent from './hooks/use-key-event';

function App() {
  const pressedKey = useKeyEvent(['s', 'c', 'p']);

  let output = '';

  if (pressedKey === 's') {
    output = '😊';
  } else if (pressedKey === 'c') {
    output = '😭';
  } else if (pressedKey === 'p') {
    output = '🎉';
  }

  return (
    <main>
      <h1>Press a key!</h1>
      <p>
        Supported keys: <kbd>s</kbd>, <kbd>c</kbd>, <kbd>p</kbd>
      </p>
      <p id="output">{output}</p>
    </main>
  );
}

export default App;
```

> **NOTE**
>
> All code files used for this activity can be found at https://packt.link/LzxOO.

CHAPTER 12: MULTIPAGE APPS WITH REACT ROUTER

ACTIVITY 12.1: CREATING A BASIC THREE-PAGE WEBSITE

Solution:

Perform the following steps to complete this activity:

1. Create a new React project via **`npx create-react-app`** as explained in *Chapter 1*, *React: What & Why?*. Then, install the React Router library by running **`npm install react-router-dom`** inside the project folder.
2. For the three required pages, create three components: a **`Welcome`** component, a **`Products`** component, and a **`ProductDetail`** component. Store these components in files inside the **`src/routes`** folder since these components will only be used for routing.

   For the **`Welcome`** component, enter the following code:

```
// src/routes/Welcome.js

function Welcome() {
  return (
    <main>
      <h1>Welcome to our shop!</h1>
      <p>
        Please explore our products or share this
        site with others.
      </p>
    </main>
  );
}

export default Welcome;
```

   To create the **`Products`** component, run the following:

```
// src/routes/Products.js

import products from '../data/products';

function Products() {
  return (
    <main>
```

```
      <h1>Our Products</h1>
      <ul id="products-list">
        {products.map((product) => (
          <li key={product.id}>
            {product.title} (${product.price})
          </li>
        ))}
      </ul>
    </main>
  );
}

export default Products;
```

The code for the **`ProductDetail`** component will look as follows:

```
// src/routes/ProductDetail.js

function ProductDetail() {
  return (
    <main>
      <h1>PRODUCT TITLE</h1>
      <p id="product-price">$PRODUCT PRICE</p>
      <p>PRODUCT DESCRIPTION</p>
    </main>
  );
}

export default ProductDetail;
```

3. At the moment, no routing logic has been added yet. Therefore, dummy content such as "**`PRODUCT TITLE`**" is output in **`ProductDetail`**. This will change later.

4. With the components added, it's time to add route definitions. For this, you must first enable React Router by importing and using the **`BrowserRouter`** component (in the **`App`** component):

```
import { BrowserRouter } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
```

```
        // ...
      </BrowserRouter>
    );
  }

  export default App;
```

5. Between the **`BrowserRouter`** tags, add the route definitions for the three routes. This is done via the **`Route`** component. Keep in mind that the individual **`Route`** definitions must be wrapped in the **`Routes`** components. For each route definition, you must add a **`path`** and an **`element`** prop—the latter of which should render the respective component that belongs to the route:

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';

import ProductDetail from './routes/ProductDetail';
import Products from './routes/Products';
import Welcome from './routes/Welcome';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Welcome />} />
        <Route path="/products" element={<Products />} />
        <Route path="/products/:id" element={<ProductDetail />} />
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

6. The paths are up to you, but with the provided page descriptions, **`/`**, **`/products`**, and **`/products/:id`** are sensible choices. Though, instead of **`:id`**, you could, of course, use **`:productId`** or any other identifier.

7. The website should also have a main navigation bar. Therefore, as a next step, create a **`MainNavigation`** component and store it in an **`src/components/MainNavigation.js`** file. It's not a component that will be assigned directly to a route, and therefore it does not go in the **`src/routes`** folder. Inside the **`MainNavigation`** component, you should render a **`<header>`** element that contains a **`<nav>`** element, which then outputs a list (**`<ul>`**) of links. The actual links, however, will be added in a later step:

```
function MainNavigation() {
  return (
    <header id="main-nav">
      <nav>
        <ul>
          <li>
            Home
          </li>
          <li>
            Our Products
          </li>
        </ul>
      </nav>
    </header>
  );
}

export default MainNavigation;
```

8. Import the newly created **`MainNavigation`** component into the **`App`** component file and output it right above the routes:

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';

import MainNavigation from './components/MainNavigation';
import ProductDetail from './routes/ProductDetail';
import Products from './routes/Products';
import Welcome from './routes/Welcome';

function App() {
  return (
    <BrowserRouter>
      <MainNavigation />
```

```
        <Routes>
          <Route path="/" element={<Welcome />} />
          <Route path="/products" element={<Products />} />
          <Route path="/products/:id" element={<ProductDetail />} />
        </Routes>
      </BrowserRouter>
    );
  }

  export default App;
```

9. It's time to add some links. Place one link in the **`Welcome`** component. There, the text "our products" (in **`'Please explore "Our Products" …'`**) should be turned into a link. Since it's an internal link, use the **`<Link>`** element:

```
  import { Link } from 'react-router-dom';

  function Welcome() {
    return (
      <main>
        <h1>Welcome to our shop!</h1>
        <p>
          Please explore <Link to="/products">our products</Link>
          or share this site with others.
        </p>
      </main>
    );
  }

  export default Welcome;
```

10. In the **`MainNavigation`** component, use **`NavLink`** so that the navigation items reflect whether or not they are linked to the currently active route:

```
  import { NavLink } from 'react-router-dom';

  function MainNavigation() {
    return (
      <header id="main-nav">
        <nav>
          <ul>
            <li>
```

```
            <NavLink
              to="/"
              className={({ isActive }) => (isActive ? 'active' :
'')}
            >
              Home
            </NavLink>
          </li>
          <li>
            <NavLink
              to="/products"
              className={({ isActive }) => (isActive ? 'active' :
'')}
            >
              Our Products
            </NavLink>
          </li>
        </ul>
      </nav>
    </header>
  );
}

export default MainNavigation;
```

11. More links must be added to the **`Products`** component. In the list of products that's rendered there, ensure every list item links to the **`ProductDetail`** component (i.e., to the **`/products/:id`** route). The link, therefore, must be generated dynamically with the help of the product **`id`**:

```
import { Link } from 'react-router-dom';

import products from '../data/products';

function Products() {
  return (
    <main>
      <h1>Our Products</h1>
      <ul id="products-list">
        {products.map((product) => (
          <li key={product.id}>
            <Link to={`/products/${product.id}`}>
```

```
                {product.title} (${product.price})
              </Link>
            </li>
          ))}
        </ul>
      </main>
    );
  }

  export default Products;
```

12. To finish this project, dynamic product detail data must be output in the **`ProductDetail`** component. For this, use the **`useParams()`** Hook to get access to the product **`id`** that's encoded in the URL path. With the help of that ID, you can find the product that's needed and output its data:

```
  import { useParams } from 'react-router-dom';

  import products from '../data/products';

  function ProductDetail() {
    const params = useParams();
    const prodId = params.id;
    const product = products.find((product) => product.id === prodId);

    return (
      <main>
        <h1>{product.title}</h1>
        <p id="product-price">${product.price}</p>
        <p>{product.description}</p>
      </main>
    );
  }

  export default ProductDetail;
```

The finished pages should look like this:

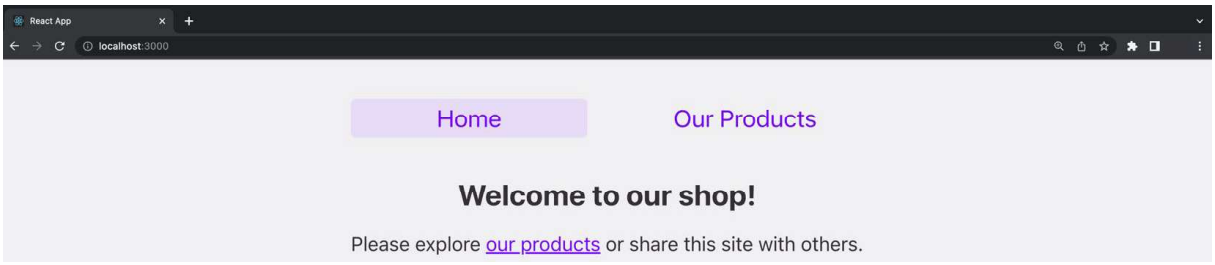

Figure 12.16: The final welcome page

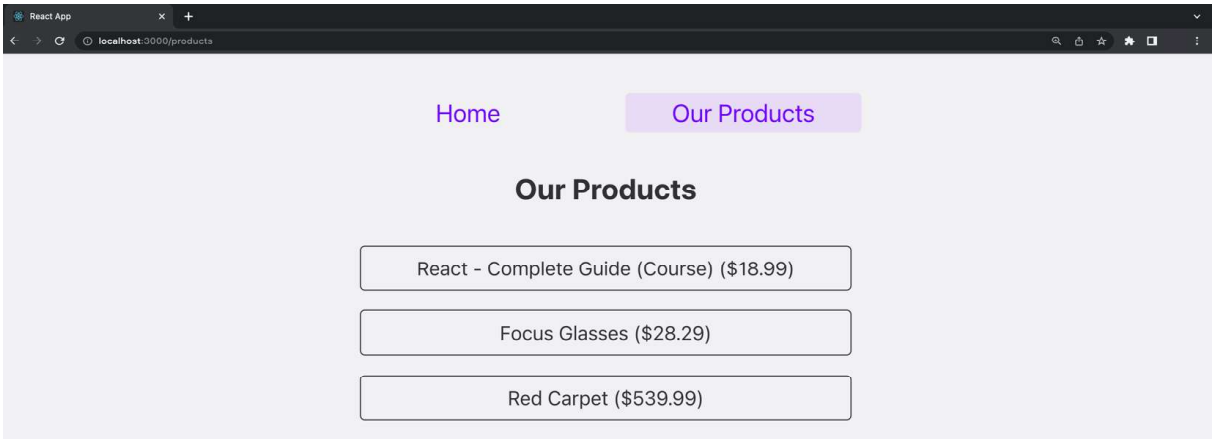

Figure 12.17: The final products page

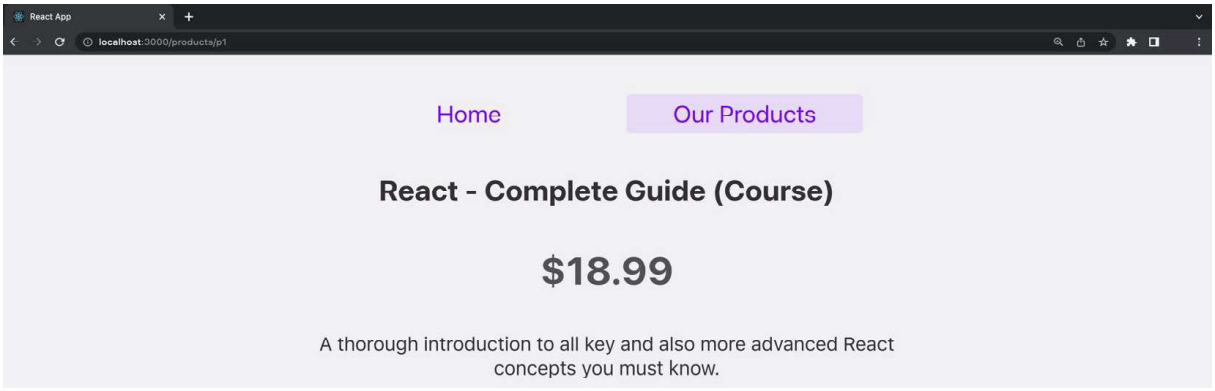

Figure 12.18: The final product details page

> **NOTE**
>
> All code files for this solution can be found at https://packt.link/wPcgN.

ACTIVITY 12.2: ENHANCING THE BASIC WEBSITE

Solution:

Perform the following steps to complete the activity:

1. Create a new **`Layout`** component in the **`src/components`** folder. In that component, include **`MainNavigation`** (remove that component from the **`App`** component), as well as a **`<main>`** element, which should be wrapped around the **`special children`** prop:

```
import MainNavigation from './MainNavigation';

function Layout({ children }) {
  return (
    <>
      <MainNavigation />
      <main>{children}</main>
    </>
  );
}

export default Layout;
```

2. Since you are using React Router's nested routes feature, you must also add a new route component—e.g., a component named **`Root`**. Add it in the **`src/routes`** folder and make sure it outputs the **`Layout`** component, wrapped around the special **`Outlet`** component:

```
import { Outlet } from 'react-router-dom';

import Layout from '../components/Layout';

function Root() {
  return (
    <Layout>
      <Outlet />
    </Layout>
  );
}

export default Root;
```

3. Next, add a new route definition to **`Routes`** in the **`App`** component. The definition for the **`Root`** route should wrap all other routes (i.e., the other routes are now nested into the **`Root`** route). Since the existing routes are converted into nested routes, you must change their paths to relative paths. You also should remove the path from the **`Welcome`** route and instead add the **`index`** prop to it:

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';

import Root from './routes/Root';
import ProductDetail from './routes/ProductDetail';
import Products from './routes/Products';
import Welcome from './routes/Welcome';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Root />}>
          <Route element={<Welcome />} index />
          <Route path="products" element={<Products />} />
          <Route path="products/:id" element={<ProductDetail />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

4. Add code splitting to the application by importing **`lazy`** and **`Suspense`** from React in the **`App`** component file. Thereafter, replace the existing imports for **`Products`** and **`ProductDetail`** with the code to load these components lazily. Don't forget to use the **`Suspense`** component as a wrapper around your route definitions:

```
import { Suspense, lazy } from 'react';
import { BrowserRouter, Routes, Route } from 'react-router-dom';

import Root from './routes/Root';
```

```
import Welcome from './routes/Welcome';

const ProductDetail = lazy(() => import('./routes/ProductDetail'));
const Products = lazy(() => import('./routes/Products'));

function App() {
  return (
    <BrowserRouter>
      <Suspense fallback={<p>Loading...</p>}>
        <Routes>
          <Route path="/" element={<Root />}>
            <Route path="" element={<Welcome />} index />
            <Route path="products" element={<Products />} />
            <Route path="products/:id" element={<ProductDetail />} />
          </Route>
        </Routes>
      </Suspense>
    </BrowserRouter>
  );
}

export default App;
```

Lazy loading makes a lot of sense for **`Products`** and **`ProductDetail`** since their routes are not necessarily visited by users (at least not immediately). It makes no sense for the **`Root`** component as this component wraps all other routes and is therefore always loaded. The **`Welcome`** component is the starting page and so does not significantly benefit from code splitting either.

5. For the *"Not Found"* page, first add a new **`NotFound`** component and store it in **`src/routes/NotFound.js`**. Add the content shown in *Figure 12.15* in the activity description:

```
function NotFound() {
  return (
    <>
      <h1>We're really sorry!</h1>
      <p>We could not find this page.</p>
    </>
  );
```

```
}

export default NotFound;
```

6. Thereafter, add a new (nested) route definition in your **App** component and use the special "catch-all" path (*****) to ensure that this route becomes active when no other route is matched. Since this page should rarely be shown, you can load it lazily:

```
import { Suspense, lazy } from 'react';
import { BrowserRouter, Routes, Route } from 'react-router-dom';

import Root from './routes/Root';
import Welcome from './routes/Welcome';

const ProductDetail = lazy(() => import('./routes/ProductDetail'));
const Products = lazy(() => import('./routes/Products'));
const NotFound = lazy(() => import('./routes/NotFound'));

function App() {
  return (
    <BrowserRouter>
      <Suspense fallback={<p>Loading...</p>}>
        <Routes>
          <Route path="/" element={<Root />}>
            <Route path="" element={<Welcome />} index />
            <Route path="products" element={<Products />} />
            <Route path="products/:id" element={<ProductDetail />} />
            <Route path="*" element={<NotFound />} />
          </Route>
        </Routes>
      </Suspense>
    </BrowserRouter>
  );
}

export default App;
```

The final *"Not Found"* page should look like this:

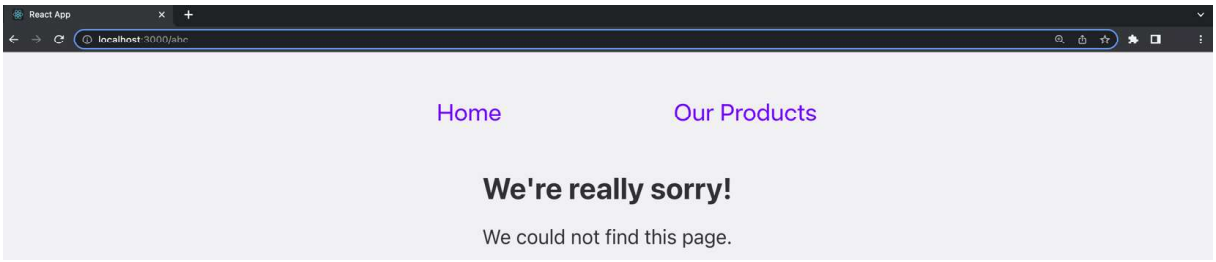

Figure 12.19: The final "Not Found" page

> **NOTE**
>
> All code files for this solution can be found at https://packt.link/HChZi.

CHAPTER 13: MANAGING DATA WITH REACT ROUTER

ACTIVITY 13.1: A TO-DOS APP

Solution:

Perform the following steps to complete this activity:

1. Create a new React project via **`npx create-react-app`** as explained in *Chapter 1*, *React—What and Why?*. Then, install the React Router library by running **`npm install react-router-dom`** inside the project folder. Check the **`package.json`** file to confirm that you have at least version 6.4 of the **`react-router-dom`** package installed.
2. For the three required pages, create three components: a **`Todos`** component, a **`NewTodo`** component, and a **`SelectedTodo`** component. Store these components in files inside the **`src/routes`** folder since these components will only be used for routing.
3. For the **`Todos`** component, enter the following code:

```
// src/routes/Todos.js

function Todos() {
  const todos = [];

  let content = (
    <main>
      <h1>No todos found</h1>
      <p>Start adding some!</p>
      <p>
        <Link className=”btn-cta” to=”/new”>
          Add Todo
        </Link>
      </p>
    </main>
  );

  if (todos && todos.length > 0) {
    content = (
      <main>
```

```
          <section>
            <Link className=”btn-cta” to=”/new”>
              Add Todo
            </Link>
          </section>
          <ul id=”todos”>

          </ul>
        </main>
      );
    }

    return (
      <>
        {content}
        <Outlet />
      </>
    );
  }

  export default Todos;
```

4. The code for the **`NewTodo`** component should look like this:

```
// src/routes/NewTodo.js

import Modal from '../components/Modal';

function NewTodo() {
  return (
    <Modal>
      <Form method=”post”>
        <p>
          <label htmlFor=”text”>Your todo</label>
          <input type=”text” id=”text” name=”text” />
        </p>
        <p className=”form-actions”>
          <button>Save Todo</button>
        </p>
```

```
      </Form>
    </Modal>
  );
}

export default NewTodo;
```

5. The code for the **`SelectedTodo`** component will look as follows:

```
// src/routes/SelectedTodo.js
import Modal from '../components/Modal';

function SelectedTodo() {
  return (
    <Modal>
      <Form method="patch">
        <p>
          <label htmlFor="text">Your todo</label>
          <input type="text" id="text" name="text" />
        </p>
        <p className="form-actions">
          <button>Update Todo</button>
        </p>
      </Form>
      <Form method="delete">
        <p className="form-actions">
          <button className='btn-alt'>Delete Todo</button>
        </p>
      </Form>
    </Modal>
  );
}

export default SelectedTodo;
```

6. At the moment, the route definitions are missing, and no data loading or submission logic has been added. Please note, however, that the components already use **`<Link>`** and **`<Form>`**.

7. With the components added, it's time to add route definitions. For this, you must first enable React Router by importing and using the **`RouterProvider`** component (in the **`App`** component):

```
import { RouterProvider } from 'react-router-dom';

function App() {
  return <RouterProvider />;
}

export default App;
```

`RouterProvider` requires a value for its **`router`** prop. That value must be an array of route definition objects. These objects can be created directly by you, or indirectly via the **`createRoutesFromElements()`** function. For this solution, use the direct approach (in **`App.js`**):

```
import { createBrowserRouter, RouterProvider } from 'react-router-
dom';

import Todos from './routes/Todos';
import NewTodo from './routes/NewTodo';
import SelectedTodo from './routes/SelectedTodo';

const router = createBrowserRouter([
  {
    path: '/',
    element: <Todos />,
    children: [
      { path: 'new', element: <NewTodo /> },
      { path: ':id', element: <SelectedTodo /> },
    ],
  },
]);

function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

8. Please note that the **`/new`** and **`/:id`** routes are child routes of the **`/`** route. The **`/`** route is thus a layout route, wrapping these child routes. That's why this layout route (**`Todos`** in **`Todos.js`**) renders an **`<Outlet />`** element.

9. To load and display to-dos, add a **`loader()`** function to the **`Todos`** route. As a first step, export such a function in the **`routes/Todos.js`** file:

```
// other imports ...
import { getTodos } from '../data/todos';

export function loader() {
  // getTodos() is a utility function that uses localStorage under
the hood
  return getTodos();
}
```

10. Thereafter, assign it as a value for the loader prop on the **`/`** route definition:

```
import Todos, { loader as todosLoader } from './routes/Todos';
// other imports ...

const router = createBrowserRouter([
  {
    path: '/',
    element: <Todos />,
    loader: todosLoader,
    children: [
      // child routes ...
    ],
  },
]);
```

11. **`getTodos()`** from the previous step is a utility function that reaches out to **`localStorage`** to retrieve and parse stored to-dos. Implement this function using the following code:

```
function getTodosFromStorage() {
  return JSON.parse(localStorage.getItem('todos'));
}



export function getTodos() {
  return getTodosFromStorage();
}
```

12. To use the loaded to-dos data, use the **`useLoaderData()`** Hook inside the **`Todos`** component. The loaded to-dos are then output via an unordered list (**`<ul>`**):

```
import { Link, Outlet, useLoaderData } from 'react-router-dom';

import { getTodos } from '../data/todos';

function Todos() {
  const todos = useLoaderData();

  let content = (
    <main>
      <h1>No todos found</h1>
      <p>Start adding some!</p>
      <p>
        <Link className="btn-cta" to="/new">
          Add Todo
        </Link>
      </p>
    </main>
  );

  if (todos && todos.length > 0) {
    content = (
      <main>
        <section>
          <Link className="btn-cta" to="/new">
            Add Todo
          </Link>
        </section>
        <ul id="todos">
          {todos.map((todo) => (
            <li key={todo.id}>
              <Link to={todo.id}>{todo.text}</Link>
            </li>
          ))}
        </ul>
      </main>
    );
```

```
  }

  return (
    <>
      {content}
      <Outlet />
    </>
  );
}
```

13. Another route that needs to-do data is the **`/:id`** route. There, a single to-do item must be loaded as the route is activated. You could reuse the to-dos data from the **`/`** route (via **`useRouteLoaderData()`**) but for practice purposes, use a separate **`loader()`** function for this activity. This **`loader()`** function, which is added to and exported from **`routes/SelectedTodo.js`** has this shape:

```
import { getTodo } from '../data/todos';
// ... other imports

export async function loader({ params }) {
  return getTodo(params.id);
}
```

14. **`getTodo()`** is yet another utility function. Implement it as follows:

```
export function getTodo(id) {
  const todos = getTodosFromStorage();
  const todo = todos.find((t) => t.id === id);

  if (!todo) {
    throw new Error('Could not find todo for id ' + id);
  }

  return todo;
}
```

15. Please note that this function throws an error if no to-do is found for the specified **`id`**. For that reason, error handling will be implemented in a later step.

16. Inside the **`SelectedTodo`** component, access the selected to-do item data via **`useLoaderData()`**. Then, use that data to set a default value on the form input:

```
function SelectedTodo() {
  const todo = useLoaderData();

  return (
    <Modal>
      <Form method="post">
        <p>
          <label htmlFor="text">Your todo</label>
          <input
            type="text"
            id="text"
            name="text"
            defaultValue={todo.text} />
        </p>
        <p className="form-actions">
          <button>Update Todo</button>
        </p>
      </Form>
      <Form method="delete">
        <p className="form-actions">
          <button className='btn-alt'>Delete Todo</button>
        </p>
      </Form>
    </Modal>
  );
}
```

17. To ensure users are able to submit new to-dos, export an **`action()`** function in the **`NewTodo`** component file, as shown here:

```
import { addTodo } from '../data/todos';
// ... other imports

export async function action({ request }) {
  const formData = await request.formData();
  const enteredText = formData.get('text');
  addTodo(enteredText);
```

```
  return redirect('/');
}
```

18. This function extracts the submitted form data, retrieves the entered text value, calls the **`addTodo()`** utility function, and then redirects the user back to the main page (**`/`**). Since the **`NewTodo`** component uses **`<Form>`** instead of **`<form>`**, React Router will automatically prevent the browser default and call the **`action()`** function assigned to the route that contains the form (**`/new`** route, in this case).

19. Implement the **`addTodo()`** function (defined in **`data/todos.js`**) by running the following code:

```
function saveTodosToStorage(todos) {
  const serializedTodos = JSON.stringify(todos);
  localStorage.setItem('todos', serializedTodos);
}

export function addTodo(text) {
  let todos = getTodosFromStorage();
  const newTodo = {
    id: new Date().toISOString(),
    text,
  };
  if (todos) {
    todos.unshift(newTodo);
  } else {
    todos = [newTodo];
  }
  saveTodosToStorage(todos);
}
```

20. To allow React Router to execute the preceding **`action()`** function when the **`<Form>`** in **`NewTodo`** is submitted, register the action in the route definition:

```
import NewTodo, { action as newTodoAction } from './routes/NewTodo';
// ... other imports

const router = createBrowserRouter([
  {
    path: '/',
    element: <Todos />,
    loader: todosLoader,
```

```
      children: [
        { path: 'new', element: <NewTodo />, action: newTodoAction }
      ],
    },
]);
```

21. To allow users to update or delete to-do items, add an **`action()`** function to the **`SelectedTodo`** component file. In that function, use **`request.method`** to differentiate between **`PATCH`** and **`DELETE`** requests. This is required because inside the **`SelectedTodo`** component, two forms are created via **`<Form>`** (see *step 3*): one **`<Form>`** creates a **`PATCH`** request (for updating to-do data), and the other **`<Form>`** creates a **`DELETE`** request (for deleting a to-do). The **`action()`** function is implemented like this:

```
import { deleteTodo, getTodo, updateTodo } from '../data/todos';

// ... other imports

export async function action({ request, params }) {
  const todoId = params.id;

  if (request.method === 'PATCH') {
    const formData = await request.formData();
    const enteredText = formData.get('text');
    updateTodo(todoId, enteredText);
  }

  if (request.method === 'DELETE') {
    deleteTodo(todoId);
  }
  return redirect('/');
}
```

22. Once again, define **`updateTodo()`** and **`deleteTodo()`** in the **`data/todos.js`** file:

```
export function updateTodo(id, newText) {
  const todos = getTodos();
  const updatedTodo = todos.find((t) => t.id === id);
  updatedTodo.text = newText;
  saveTodosToStorage(todos);
}
```

```
export function deleteTodo(id) {
  const todos = getTodos();
  const updatedTodos = todos.filter((t) => t.id !== id);
  saveTodosToStorage(updatedTodos);
}
```

23. Finally, to make React Router aware of this **`action()`** function and ensure that it gets executed as the respective forms are submitted, register the action created in *step 16*, as follows:

```
import SelectedTodo, {
  action as changeTodoAction,
  loader as todoLoader,
} from './routes/SelectedTodo';
// ... other imports

const router = createBrowserRouter([
  {
    path: '/',
    element: <Todos />,
    loader: todosLoader,
    children: [
      { path: 'new', element: <NewTodo />, action: newTodoAction },
      {
        path: ':id',
        element: <SelectedTodo />,
        action: changeTodoAction,
        loader: todoLoader,
      },
    ],
  },
]);
}
```

24. To handle any errors that have occurred, add a new **`Error`** component that is displayed when things go wrong. This component is stored in **`routes/Error.js`**:

```
import { useRouteError } from 'react-router-dom';

import Modal from '../components/Modal';

function Error() {
  const error = useRouteError();

  return (
    <Modal>
      <h1>An error occurred!</h1>
      <p>{error.message}</p>
    </Modal>
  );
}

export default Error;
```

25. This component uses React Router's **`useRouteError()`** Hook to access the error that was thrown. The error is then used to output the error message.

26. To use this **`Error`** component as a fallback, add it as a value for the **`errorElement`** property in your route definitions:

```
import Error from './routes/Error';
// ... other imports

const router = createBrowserRouter([
  {
    path: '/',
    element: <Todos />,
    errorElement: <Error />,
    loader: todosLoader,
    children: [
      { path: 'new', element: <NewTodo />, action: newTodoAction },
      {
        path: ':id',
        element: <SelectedTodo />,
        action: changeTodoAction,
```

```
            loader: todoLoader,
          },
        ],
      },
    ]);
```

27. Here, **`Error`** is set as an **`errorElement`** on the main route and is used by React Router for all errors occurring anywhere in the entire app.

> **NOTE**
>
> All code for this solution is available at https://packt.link/oaFoZ.