Creating an error page

In this section, we will understand how error pages work in React Router before implementing one in our app.

Understanding error pages

Currently, a React Router built-in error page is shown when an error occurs. We can check this by entering an invalid path in the running app:

Figure 6.10 – Standard React Router error page

An error is raised because no matching routes are found in the router. The *404 Not Found* message on the error page confirms this.

This standard error page isn't ideal because the information is targeted at a developer rather than a real user. Also, the app header isn't shown, so users can't easily navigate to a page that does exist.

As the error message suggests, an `errorElement` prop can be used on a route to override the standard error page. The following is an example of a custom error page defined for a customer's route; if any error occurs on this route, the `CustomersErrorPage` component will be rendered:

```
const router = createBrowserRouter([
  ...,
  {
    path: 'customers',
    element: <CustomersPage />,
    errorElement: <CustomersErrorPage />
  },
  ...
]);
```

Now that we have started understanding error pages in React Router, we will implement one in our app.

Adding an error page

Carry out the steps below to create an error page in the app:

1. First, create a new page called `ErrorPage.tsx` in the `src/pages` folder with the following content:

```
import { Header } from '../Header';

export function ErrorPage() {
  return (
    <>
      <Header />
      <div className="text-center p-5 text-xl">
        <h1 className="text-xl text-slate-900">
          Sorry, an error has occurred
        </h1>
      </div>
    </>
  );
}
```

   The component simply returns the app header with a *Sorry, an error has occurred* message underneath.

2. Open `Routes.tsx` and add an `import` statement for the error page:

```
import { ErrorPage } from './pages/ErrorPage';
```

3. Specify the error page on the root route as follows:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <App />,
    errorElement: <ErrorPage />,
    children: ...
  },
]);
```

   Specifying the error page on the root route means that it will show if any routes have errors.

4. Switch back to the running app and change the browser URL to `http://localhost:3000/invalid`. The error page will be shown:

Figure 6.11 – Error page

5. This is a good start, but we can improve it by providing the user with more information, which we can get from React Router's `useRouteError` hook. Open `ErrorPage.tsx` again and add an `import` statement for `useRouteError`:

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

6. Assign the error to an `error` variable before the component's `return` statement using `useRouteError`:

```
export function ErrorPage() {
  const error = useRouteError();
  return ...
}
```

7. The `error` variable is of the `unknown` type – you can verify this by hovering over it. We can use a type predicate function to allow TypeScript to narrow it to something we can work with. Add the following type predicate function beneath the component:

```
function isError(error: any): error is { statusText:
string } {
  return "statusText" in error;
}
```

The function checks whether the error object has a `statusText` property and if so, gives it a type with this property.

8. We can now use this function to render the information in the `statusText` property:

```
return (
  <>
    <Header />
    <div className="text-center p-5 text-xl">
      <h1 className="text-xl text-slate-900">
        Sorry, an error has occurred
      </h1>
      {isError(error) && (
        <p className="text-base text-slate-700">
          {error.statusText}
        </p>
      )}
    </div>
  </>
);
```

9. In the running app, the information about the error appears on the error page as an invalid path:

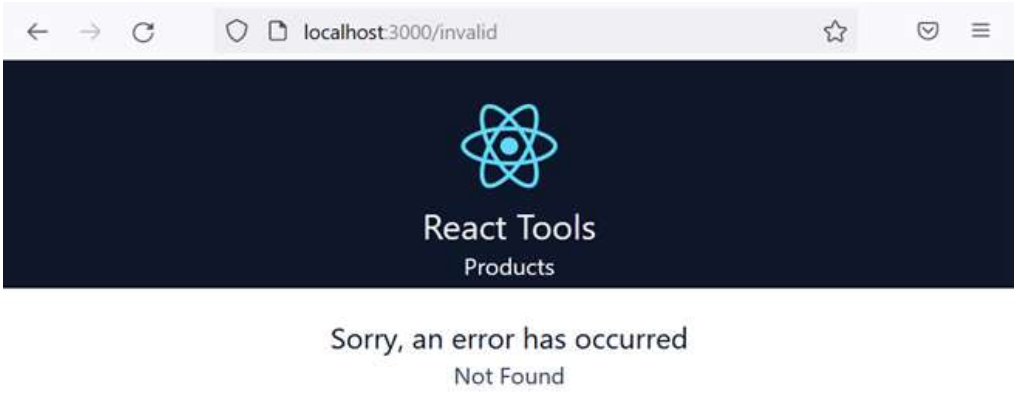

Figure 6.12 – Error page containing information about the error

That completes this section on error pages. The key point is to use an `errorElement` prop on a route to catch and display errors. Specific error information can be obtained using the `useRouteError` hook.

For more information on `errorElement`, see the following link: `https://reactrouter.com/en/main/route/error-element`. For more information on the `useRouteError` hook, see the following link: `https://reactrouter.com/en/main/hooks/use-route-error`.

Next, we will learn about index routes.

Using index routes

Currently, the app's root path displays nothing other than the header. In this section, we will learn about index routes to display a nice welcome message in the root path.

Understanding index routes

An **index route** can be thought of as a default child route. In React Router, if no children match a parent route, it will display an index route if one is defined. An index route has no path and instead has an `index` Boolean property, as in the following example:

```
{
  path: "/",
  element: <App />,
  children: [
    {
      index: true,
      element: <HomePage />,
    },
    ...,
  ]
}
```

Next, we will add a home page using an index route in our app.

Using an index route in the app

Carry out the following steps to add a home page using an index route in our app:

1. Create a new file in the `src/pages` folder called `HomePage.tsx` with the following content:

```
export function HomePage() {
  return (
    <div className="text-center p-5 text-xl">
      <h1 className="text-xl text-slate-900">Welcome to
        React Tools!</h1>
    </div>
  );
}
```

   The page displays a welcome message.

2. Open `Routes.tsx` and import the home page we just created:

```
import { HomePage } from './pages/HomePage';
```

3. Add the home page as an index page for the root path as follows:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <App />,
    errorElement: <ErrorPage />,
    children: [
      {
        index: true,
        element: <HomePage />,
      },
      ...
    ]
  }
]);
```

4. We will add links to the logo and app title in the header to go to the home page. Open `Header.tsx` and import the `Link` component from React Router:

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

5. Wrap links to the root page around the logo and title as follows:

```
<header ...>
  <Link to="">
    <img src={logo} ... />
  </Link>
  <Link to="">
    <h1 ...>React Tools</h1>
  </Link>
  <nav>
    ...
  </nav>
</header>
```

6. In the running app, click the app title to go to the root page, and you will see the welcome message displayed:

Figure 6.13 – Welcome page

That completes this section on index routes.

To recap, an index route is a default child route and is defined using an `index` Boolean property.

For more information on index routes, see the following link: `https://reactrouter.com/en/main/route/route#index`.

Next, we will learn about search parameters.

Using search parameters

In this section, we will learn about search parameters in React Router and use them to implement a search feature in the app.

Understanding search parameters

Search parameters are part of a URL that comes after the `?` character and separated by the `&` character. Search parameters are sometimes referred to as **query parameters**. In the following URL, `type` and when are search parameters: `https://somewhere.com/?type=sometype&when=recent`.

React Router has a hook that returns functions for getting and setting search parameters called `useSearchParams`:

```
const [searchParams, setSearchParams] = useSearchParams();
```

`searchParams` is a JavaScript `URLSearchParams` object. There is a `get` method on `URLSearchParams`, which can be used to get the value of a search parameter. The following example gets the value of a search parameter called `type`:

```
const type = searchParams.get('type');
```

`setSearchParams` is a function used to set search parameter values. The function parameter is an object as in the following example:

```
setSearchParams({ type: 'sometype', when: 'recent' });
```

Next, we will add search functionality to our app.

Adding search functionality to the app

We will add a search box to the header of the app. Submitting a search will take the user to the products list page and list a filtered set of products matching the search criteria. Carry out the following steps:

1. Open `Header.tsx` and add `useSearchParams` to the React Router import. Also, add an `import` statement for the `FormEvent` type from React:

```
import { FormEvent } from 'react';
import {
  NavLink,
  Link,
  useSearchParams
} from 'react-router-dom';
```

2. Use the `useSearchParams` hook to destructure functions to get and set search parameters before the `return` statement:

```
export function Header() {
  const [searchParams, setSearchParams] =
useSearchParams();
  return ...
}
```

3. Add the following search form above the logo:

```
<header ...>
  <form
    className="relative text-right"
    onSubmit={handleSearchSubmit}
  >
    <input
      type="search"
      name="search"
      placeholder="Search"
```

```
        defaultValue={searchParams.get('search') ?? ''}
        className="absolute right-0 top-0 rounded py-2 px-3
          text-gray-700"
      />
    </form>
    <Link to="">
      <img src={logo} ... />
    </Link>
    ...
  </header>
```

The form contains a search box with its default value as the value of a `search` parameter. `searchParams.get` returns `null` if the parameter doesn't exist, so a **nullish coalescing operator** (`??`) is used to set the search box's default value to an empty string in this case.

> **Note**
>
> The **nullish coalescing operator** (`??`) returns the right operand if the left operand is `null` or `undefined`; otherwise, it returns the left operand. For more information, see the following link: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Nullish_coalescing`.

The form submission calls a `handleSearchSubmit` function, which we'll implement in the next step.

4. Add a `handleSearchSubmit` function as follows, just above the `return` statement:

```
export function Header() {
  const [searchParams, setSearchParams] =
    useSearchParams();
  function handleSearchSubmit(e:
    FormEvent<HTMLFormElement>) {
    e.preventDefault();
    const formData = new FormData(e.currentTarget);
    const search = formData.get('search') as string;
    setSearchParams({ search });
  }
  return ...
}
```

The submit handler parameter is typed using `FormEvent`. `FormEvent` is a generic type that takes in the type of the element, which is `HTMLFormElement` for a form submit handler.

We use the `preventDefault` method on submit handler parameters to prevent the form from being submitted to the server because we handle all the logic in this function.

We use the JavaScript `FormData` interface to get the value of the search field. Then, we use a type assertion to set the type of the search field value to a string.

The last line of code in the submit handler sets the value of the search parameter. This will update the browser's URL to have this search parameter.

> **Note**
> We will learn a lot more about forms in React in *Chapter 7*, *Working with Forms*.

5. Now, we need to filter the products list with the value of the search parameter. Open `ProductsPage.tsx` and add `useSearchParams` to the `import` statement:

```
import { Link, useSearchParams } from 'react-router-dom';
```

6. At the top of the `ProductsPage` component, destructure `searchParams` from `useSearchParams` as follows:

```
export function ProductsPage() {
  const [searchParams] = useSearchParams();
  return ...
}
```

7. Add the following function just before the `return` statement to filter the products list by the search value:

```
const [searchParams] = useSearchParams();
function getFilteredProducts() {
  const search = searchParams.get('search');
  if (search === null || search === "") {
    return products;
  } else {
    return products.filter(
      (product) =>
        product.name
          .toLowerCase()
```

```
            .indexOf(search.toLowerCase()) > -1
      );
    }
  }
  return ...
```

The function starts by getting the value of the `search` parameter. The full product list is returned if there is no search parameter or if the value is an empty string. Otherwise, the product list is filtered using the array's `filter` function, checking that the search value is contained within the product name irrespective of the case.

8. Use the function we just created in the JSX to output the filtered products. Replace the reference to `products` with a call to `getFilteredProducts` as follows:

```
<ul className="list-none m-0 p-0">
  {getFilteredProducts().map((product) => (
    <li
      key={product.id}
      className="p-1 text-base text-slate-800"
    >
      <Link
        to={`${product.id}`}
        className="p-1 text-base text-slate-800
          hover:underline"
      >
        {product.name}
      </Link>
    </li>
  ))}
</ul>
```

9. In the running app, whilst on the home page, enter some search criteria in the search box and press *Enter* to submit the search.

The search parameter is added to the URL in the browser. However, it doesn't navigate to the products list page. Don't worry about this because we'll address this issue in the next section:

Figure 6.14 – The search parameter added to the URL

The key point in this section is that the `useSearchParams` hook from React Router allows you to set and get URL search parameters. The parameters are also structured in a JavaScript `URLSearchParams` object.

For more information on the `useSearchParams` hook, see the following link in the React Router documentation: `https://reactrouter.com/en/main/hooks/use-search-params`. More information on `URLSearchParams` is available at `https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams`.

Next, we will explore another React Router hook that enables programmatic navigation.

Navigating programmatically

React Router's `Link` and `NavLink` components allow declarative navigation. However, sometimes we must navigate imperatively – in fact, this would be useful for the search feature in our app to navigate to the products list page. In this section, we will learn how to programmatically navigate with React Router and use this to complete the app's search feature. Carry out the following steps:

1. Open `Header.tsx` and add import the `useNavigate` hook from React Router:

```
import {
  NavLink,
  Link,
  useSearchParams,
  useNavigate
} from 'react-router-dom';
```

The `useNavigate` hook returns a function we can use to perform programmatic navigation.

2. Invoke `useNavigate` after the call to the `useSearchParams` hook. Assign the result to a variable called `navigate`:

```
export function Header() {
  const [searchParams, setSearchParams] =
    useSearchParams();
  const navigate = useNavigate();
  ...
}
```

The `navigate` variable is a function that can be used to navigate. It takes in an argument for the path to navigate to.

3. In `handleSearchSubmit`, replace the `setSearchParams` call with a call to `navigate` to go to the products list page with the relevant search parameter:

```
function handleSearchSubmit(e: FormEvent<HTMLFormElement>)
{
  e.preventDefault();
  const formData = new FormData(e.currentTarget);
  const search = formData.get('search') as string;
  navigate(`/products/?search=${search}`);
}
```

4. We no longer need `setSearchParams` because the setting of the search parameter is included in the navigation path, so remove this from the `useSearchParams` call:

```
const [searchParams] = useSearchParams();
```

5. In the running app, enter some search criteria in the search box and press *Enter* to submit the search.

The search parameter is used to navigate to the products list page. When the products list page appears, the correctly filtered products are shown:

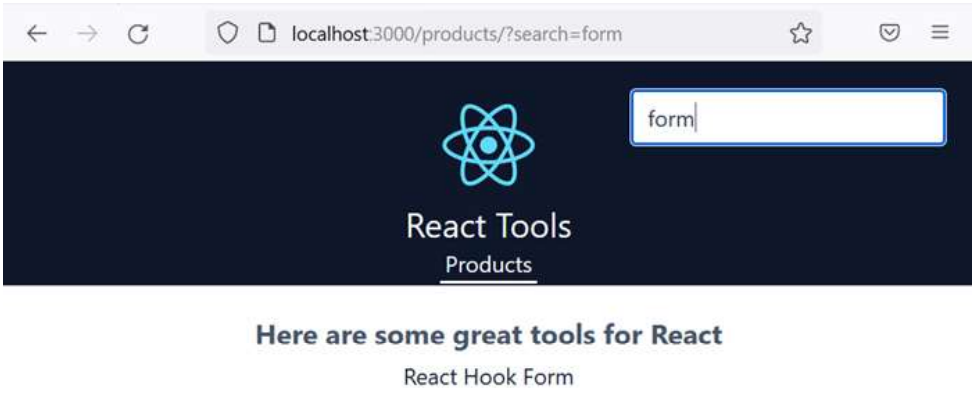

Figure 6.15 – Products list page with filter products

So, programmatic navigation is achieved using the `useNavigate` hook. This returns a function that can navigate to the path passed into it.

For more information on the `useNavigate` hook, see the following link in the React Router documentation: `https://reactrouter.com/en/main/hooks/use-navigate`.

Next, we will refactor the search form's navigation to use React Router's `Form` component.

Using form navigation

In this section, we will use React Router's `Form` component to navigate to the products list page when the search criteria are submitted. `Form` is a wrapper around the HTML `form` element that handles the form submission on the client side. This will replace the use of `useNavigate` and simplify the code. Carry out the following steps:

1. In `Header.tsx`, start by removing `useNavigate` from `import` for the React Router and replace it with the `Form` component:

```
import {
  NavLink,
  Link,
  useSearchParams,
  Form
} from 'react-router-dom';
```

2. In the JSX, replace the `form` element with React Router's `Form` component:

```
<Form
  className="relative text-right"
  onSubmit={handleSearchSubmit}
```

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

3. In the `Form` element in the JSX, remove the `onSubmit` handler. Replace this with the following `action` attribute so that the form is sent to the `products` route:

```
<Form
  className="relative text-right"
  action="/products"
>
  ...
</Form>
```

   React Router's form submission mimics how a native `form` element submits to a server path. However, React Router submits the form to a client-side route instead. In addition, `Form` mimics an HTTP `GET` request by default, so a `search` parameter will automatically be added to the URL.

4. The remaining tasks are to remove the following code:

   - Remove the React import statement because `FormEvent` is redundant now
   - Remove the call to `useNavigate` because this is no longer required
   - Remove the `handleSearchSubmit` function because this is no longer required

5. In the running app, enter some search criteria in the search box and press *Enter* to submit the search. This will work as it did before.

   That has simplified the code quite a bit!

We will learn more about React Router's `Form` component in *Chapter 7* and *Chapter 9*. The key takeaway from this section is that `Form` wraps the HTML `form` element, handling form submission on the client.

For more information on `Form`, see the following link in the React Router documentation: `https://reactrouter.com/en/main/components/form`.

Next, we will learn about a type of performance optimization that can be applied to large pages in the app.

Implementing lazy loading

Currently, all the JavaScript for our app is loaded when the app first loads. This can be problematic in large apps. In this section, we will learn how to only load the JavaScript for components when their route becomes active. This pattern is often referred to as **lazy loading**. In our app, we will create a lazily loaded admin page.

Understanding React lazy loading

By default, all React components are bundled together and loaded when the app first loads. This is inefficient for large apps – particularly when a user does not use many components. Lazily loading React components addresses this issue because lazy components aren't included in the initial bundle that is loaded; instead, their JavaScript is fetched and loaded when rendered.

There are two main steps to lazy loading React components. First, the component must be dynamically imported as follows:

```
const LazyPage = lazy(() => import('./LazyPage'));
```

In the code block, `lazy` is a function from React that enables the imported component to be lazily loaded. Note that the lazy page must be a default export – lazy loading doesn't work with named exports.

Webpack can then split the JavaScript for `LazyPage` into a separate bundle. Note that this separate bundle will include any child components of `LazyPage`.

The second step is to render the lazy component inside React's `Suspense` component as follows:

```
<Route
  path="lazy"
  element={
    <Suspense fallback={<div>Loading…</div>}>
      <LazyPage />
    </Suspense>
  }
/>
```

The `Suspense` component's `fallback` prop can be set to an element to render while the lazy page is being fetched.

Next, we will create a lazy admin page in our app.

Adding a lazy admin page to the app

Carry out the following steps to add a lazy admin page to our app:

1. Create a file called `AdminPage.tsx` in the `src/pages` folder with the following content:

```
export default function AdminPage() {
  return (
    <div classNa"e="text-center p-5 text"xl">
      <h1 classNa"e="text-xl text-slate-"00">Admin
        Panel</h1>
      <p classNa"e="text-base text-slate-"00">
        You shou'dn't come here often becaus' I'm lazy
      </p>
    </div>
  );
}
```

   The page is very small, so it is not a great use case for lazy loading. However, its simplicity will allow us to focus on how to implement lazy loading.

2. Open `Routes.tsx` and import `lazy` and `Suspense` from React:

```
import { lazy, Suspense } fr'm 're'ct';
```

3. Import the admin page as follows (it is important that this comes after all the other `import` statements):

```
const AdminPage = lazy(() => impo't('./pages/
AdminP'ge'));
```

4. Add the `admin` route as follows:

```
const router = createBrowserRouter([
  {
    pat':''/',
    element: <App />,
    errorElement: <ErrorPage />,
    children: [
      ...,
      {
        pat': 'ad'in',
```

```
          element: (
            <Suspense
              fallback={
                <div classNa"e="text-center p-5 text-xl
                  text-slate-"00">
                  Loading...
                </div>
              }
            >
              <AdminPage />
            </Suspense>
          )
        }
      ]
    }
]);
```

The path to the admin page is `/admin`. A loading indicator will render as the admin page's JavaScript is fetched.

5. Open `Header.tsx` and add a link to the admin page after the `Products` link as follows:

```
<nav>
  <NavLink ... >
    Products
  </NavLink>
  <NavLink
    "o="ad"in"
    className={({ isActive }) =>
      `text-white no-underline p-1 pb-0.5 border-solid
        border-b-2 ${
      isActive"? "border-wh"te"": "border-transpar"nt"
      }`
    }
  >
    Admin
  </NavLink>
</nav>
```

6. In the running app, open the browser DevTools and go to the **Network** tab and clear out any existing requests. Slow down the connection by selecting **Slow 3G** from the **No throttling** menu:

Figure 6.16 – Setting a slow connection

7. Now, click on the **Admin** link in the header. The loading indicator appears because the JavaScript for the admin page is being downloaded:

Figure 6.17 – The loading indicator

After the admin page has been downloaded, it will render in the browser. If you look at the **Network** tab in DevTools, you will see confirmation of the admin page bundle being lazily loaded:

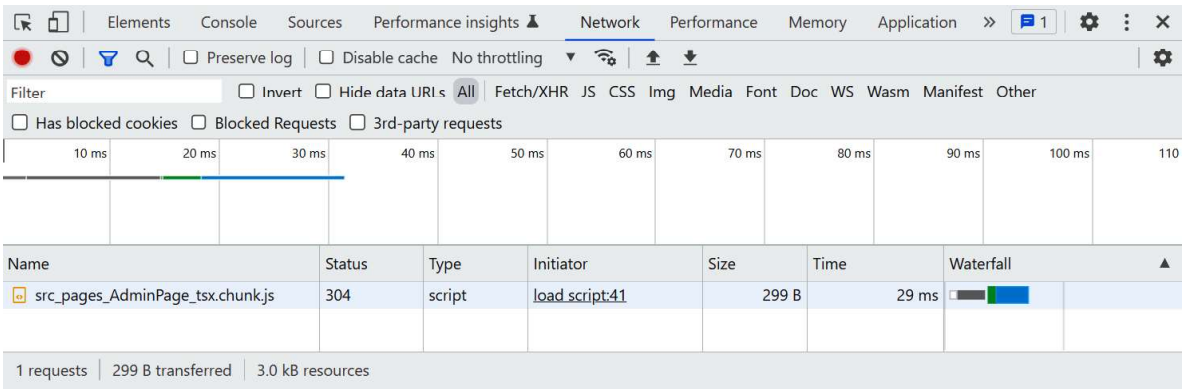

Figure 6.18 – Admin page download

That completes this section on lazily loading React components. In summary, lazily loading React components is achieved by dynamically importing the component file and rendering the component inside a `Suspense` component.

For more information on lazily loading React components, see the following link in the React documentation: `https://reactjs.org/docs/code-splitting.html`.

That also completes this chapter. Next, we will recap what we have learned about React Router.

Summary

React Router gives us a comprehensive set of components and hooks for managing the navigation between pages in our app. We used `createBrowserRouter` to define all our web app's routes. A route contains a path and a component to render when the path matches the browser URL. We used an `errorElement` prop for a route to render a custom error page in our app.

We used nested routes to allow the `App` component to render the app shell and page components within it. We used React Router's `Outlet` component inside the `App` component to render page content. We also used an index route on the root route to render a welcome message.

We used React Router's `NavLink` component to render navigation links that are highlighted when their route is active. The `Link` component is great for other links that have static styling requirements – we used this for product links on the product list. We used React Router's `Form` component to navigate to the products list page when the search form is submitted.

Route parameters and search parameters allow parameters to be passed into components so that they can render dynamic content. `useParams` gives access to route parameters, and `useSearchParams` provides access to search parameters.

React components can be lazily loaded to increase startup performance. This is achieved by dynamically importing the component file and rendering the component inside a `Suspense` component.

In the next chapter, we will learn all about forms in React.

Questions

Let's test our knowledge of React Router with the following questions:

1. We have declared the following routes in an app:

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <App />,
    errorElement: <ErrorPage />,
```

```
      children: [
        { path: "customers", element: <CustomersPage /> }
      ]
    }
]);
```

What component will render when the path is `/customers`?

What component will render when the path is `/products`?

2. What would the path be in a route that could handle a `/customers/37` path? `37` is a customer ID and could vary.

3. The routes for a `settings` page are defined as follows:

```
{
  path: "/settings",
  element: <SettingsPage />,
  children: [
    { path: "general", element: <GeneralSettingsTab /> },
    { path: "dangerous", element: <DangerousSettingsTab
      /> }
  ]
}
```

The settings page has **General** and **Dangerous** tabs, which show when the path is `/settings/general` and `/settings/dangerous`, respectively. However, when these paths are requested, no tab content is shown on the settings page – so, what could we have forgotten to add in the `SettingsPage` component?

4. We are implementing a navigation bar in an app. When clicking on a navigation item, the app should navigate to the relevant page. Which React Router component should we use to render the navigational items? `Link` or `NavLink`?

5. A route is defined as follows:

```
{ path: "/user/:userId", element: <UserPage /> }
```

Inside the `UserPage` component, the following code is used to get the user `id` information from the browser URL:

```
const params = useParams<{id: string}>();
const id = params.id;
```

However, `id` is always `undefined`. What is the problem?

6. The following URL contains an example of a search parameter on a `customers` page:

   `/customers/?search=cool company`

   However, an error occurs in the following implementation:

```
function getFilteredCustomers() {
  const criteria = useSearchParams.get('search');
  if (criteria === null || criteria === "") {
    return customers;
  } else {
    return customers.filter(
      (customer) =>
        customer.name.toLowerCase().indexOf(criteria.
          toLowerCase()) > -1
    );
  }
}
```

   What is the problem?

7. A React component is lazily loaded as follows:

```
const SpecialPage = lazy(() => import('./pages/
SpecialPage'));

const router = createBrowserRouter([
  ...,
  {
    path: '/special',
    element: <SpecialPage />,
  },
  ...
]);
```

   However, React throws an error. What is the problem?

Answers

1. `CustomersPage` will render when the path is `/customers`.

   `ErrorPage` will render when the path is `/products`.

2. The path could be `path="customers/:customerId"`.
3. It is likely that the `Outlet` component has not been added to `SettingsPage`.
4. Both will work, but `NavLink` is better because it enables items to be styled when active.
5. The route parameter referenced should be `userId`:

```
const params = useParams<{userId: string}>();
const id = params.userId;
```

6. Hooks must be called at the top level of function components. Also, the `useSearchParams` hook doesn't directly have a `get` method. Here's the corrected code:

```
const [searchParams] = useSearchParams();
function getFilteredCustomers() {
  const criteria = searchParams.get('search');
  ...
}
```

7. The lazy component must be nested inside a `Suspense` component as follows:

```
{
  path: '/special',
  element: (
    <Suspense fallback={<Loading />}>
      <SpecialPage />
    </Suspense>
  )
}
```

7
Working with Forms

Forms are extremely common in apps, so it's essential to be able to efficiently implement them. In some apps, forms can be large and complex, and getting them to perform well is challenging.

In this chapter, we'll learn how to build forms in React using different approaches. The example form we will make here is a contact form that you would often see on company websites. It will contain a handful of fields and some validation logic.

The first approach to building a form will be to store field values in the state. We will see how this approach can bloat code and hurt performance. The next approach embraces the browser's native form capabilities, reducing the amount of code required and improving performance. We will then use React Router's `Form` component, which we briefly covered in *Chapter 6*, *Routing with React Router*. The final approach will be to use a popular library called **React Hook Form**. We'll experience how React Hook Form helps us implement form validation and submission logic while maintaining excellent performance.

We'll cover the following topics:

- Using controlled fields
- Using uncontrolled fields
- Using React Router Form
- Using native validation
- Using React Hook Form

Technical requirements

We will use the following technologies in this chapter:

- **Browser**: A modern browser such as Google Chrome
- **Node.js** and **npm**: You can install them from `https://nodejs.org/en/download/`
- **Visual Studio Code**: You can install it from `https://code.visualstudio.com/`

All the code snippets in this chapter can be found online at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter7`.

Using controlled fields

In this section, we will build the first version of our contact form. It will contain fields for the user's name, email address, contact reason, and any additional notes the user may want to make.

This method will involve the use of **controlled fields**, which is where field values are stored in the state. We will use this approach to implement the form – however, in doing so, we will pay attention to the amount of code required and the negative impact on performance; this will help you see why other methods are better.

To get started, first, we need to create a React and TypeScript project, as in previous chapters.

Creating the project

We will develop the form using Visual Studio Code and a new Create React App based project setup. We've previously covered this several times, so we will not cover the steps in this chapter – instead, refer to *Chapter 3*, *Setting Up React and TypeScript*. Create the project for the contact form with the name of your choice.

We will style the form with Tailwind CSS. We have previously covered how to install and configure Tailwind in Create React App in *Chapter 5*, *Approaches to Styling Frontends*, so after you have created the React and TypeScript project, install and configure Tailwind.

We will use a Tailwind plugin to help us style the form – it provides nice styles for field elements out of the box. Carry out the steps below to install and configure this plugin:

1. Install this plugin by running the following command in a terminal:

```
npm i -D @tailwindcss/forms
```

2. Open `tailwind.config.js` to configure the plugin. Add the highlighted code to this file to tell Tailwind to use the forms plugin we just installed:

```
module.exports = {
  content: ['./src/**/*.{js,jsx,ts,tsx}'],
  theme: {
    extend: {},
  },
  plugins: [require('@tailwindcss/forms')],
};
```

That completes the project setup. Next, we will create the first version of the form.

Creating a contact form

Now, carry out the following steps to create the first version of the contact form:

1. Create a file called `ContactPage.tsx` in the `src` folder with the following `import` statement:

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

   We have imported the `useState` hook and the `FormEvent` type from React, which we will eventually use in the implementation.

2. Add the following `type` alias under the import statement. This type will represent all the field values:

```
type Contact = {
  name: string;
  email: string;
  reason: string;
  notes: string;
};
```

3. Add the following `function` component:

```
export function ContactPage() {
  return (
    <div className="flex flex-col py-10 max-w-md
      mx-auto">
      <h2 className="text-3xl font-bold underline
        mb-3">Contact Us</h2>
      <p className="mb-3">
        If you enter your details we'll get back to you
          as soon as we
        can.
      </p>
      <form></form>
    </div>
  );
}
```

   This displays a heading and some instructions, horizontally centered on the page.

4. Add the following fields inside the `form` element:

```
<form ...>
  <div>
    <label htmlFor="name">Your name</label>
    <input type="text" id="name" />
  </div>
  <div>
    <label htmlFor="email">Your email address</label>
    <input type="email" id="email" />
  </div>
  <div>
    <label htmlFor="reason">Reason you need to contact
      us</label>
    <select id="reason">
      <option value=""></option>
      <option value="Support">Support</option>
      <option value="Feedback">Feedback</option>
      <option value="Other">Other</option>
    </select>
  </div>
  <div>
    <label htmlFor="notes">Additional notes</label>
    <textarea id="notes" />
  </div>
</form>
```

We have added fields for the user's name, email address, contact reason, and additional notes. Each field label is associated with its editor by setting the `htmlFor` attribute to the editor's `id` value. This helps assistive technology such as screen readers to read out labels when fields gain focus.

5. Add a `submit` button to the bottom of the `form` element as follows:

```
<form ...>
  ...
  <div>
    <button
      type="submit"
```

```
        className="mt-2 h-10 px-6 font-semibold bg-black
          text-white"
      >
        Submit
      </button>
    </div>
  </form>
```

6. The field containers will all have the same style, so create a variable for the style and assign it to all the field containers as follows:

```
const fieldStyle = "flex flex-col mb-2";
return (
  <div ...>
    ...
    <form ...>
      <div className={fieldStyle}>...</div>
      <div className={fieldStyle}>...</div>
      <div className={fieldStyle}>...</div>
      <div className={fieldStyle}>...</div>
      <div>
        <button
          type="submit"
          className="mt-2 h-10 px-6 font-semibold
            bg-black text-white"
        >
          Submit
        </button>
      </div>
    </form>
  </div>
);
```

The fields are nicely styled now using a vertically flowing flexbox and a small margin under each one.

7. Add the state to hold the field values as follows:

```
export function ContactPage() {
  const [contact, setContact] = useState<Contact>({
    name: "",
    email: "",
```

```
    reason: "",
    notes: "",
  });
  const fieldStyle = ...;
  ...
}
```

We have given the state the `Contact` type we created earlier and initialized the field values to empty strings.

8. Bind the state to the `name` field editor as follows:

```
<div className={fieldStyle}>
  <label htmlFor="name">Your name</label>
  <input
    type="text"
    id="name"
    value={contact.name}
    onChange={(e) =>
      setContact({ ...contact, name: e.target.value })
    }
  />
</div>
```

`value` is set to the current value of the state. `onChange` is triggered when the user fills in the input element, which we use to update the state value. To construct the new state object, we clone the current state and override its `name` property with the new value from the `onChange` parameter.

9. Repeat the same approach for the binding state to the other field editors as follows:

```
<div className={fieldStyle}>
  ...
  <input
    type="email"
    id="email"
    value={contact.email}
    onChange={(e) =>
      setContact({ ...contact, email: e.target.value })
    }
  />
```

```
</div>
<div className={fieldStyle}>
  ...
  <select
    id="reason"
    value={contact.reason}
    onChange={(e) =>
      setContact({ ...contact, reason: e.target.value })
    }
  >
    ...
  </select>
</div>
<div className={fieldStyle}>
  ...
  <textarea
    id="notes"
    value={contact.notes}
    onChange={(e) =>
      setContact({ ...contact, notes: e.target.value })
    }
  />
</div>
```

10. Add a submit handler to the `form` element as follows:

```
function handleSubmit(e: FormEvent<HTMLFormElement>) {
  e.preventDefault();
  console.log('Submitted details:', contact);
}
const fieldStyle = ...;
return (
  <div>
    ...
    <form onSubmit={handleSubmit}>
      ...
    </form>
  </div>
);
```

The submit handler parameter is typed using React's `FormEvent` type. The submit handler function prevents the form from being sent to the server using the `preventDefault` method on the handler parameter. Instead of sending the form to a server, we output the `contact` state to the console.

11. The last step is to render `ContactPage` in the `App` component. Open `App.tsx` and replace its content with the following:

```
import { ContactPage } from './ContactPage';

function App() {
  return <ContactPage />;
}

export default App;
```

The `App` component simply renders the `ContactPage` page component we just created. The `App` component also remains a default export so that `index.tsx` isn't broken.

That completes the first iteration of the form. We will now use the form and discover a potential performance problem. Carry out the following steps:

1. Run the app in development mode by executing `npm start` in the terminal. The form appears as follows:

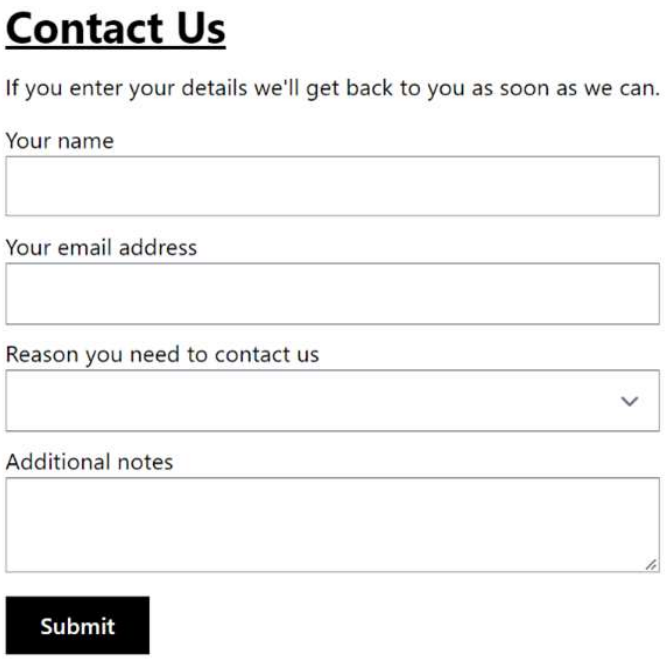

Figure 7.1 – Contact form

2. We are going to highlight component re-rendering using React DevTools, which will highlight a potential performance problem.

   Open the browser DevTools and select the **Components** panel. If there is no **Components** panel, ensure that the React DevTools are installed in the browser (see *Chapter 3* for how to install React DevTools).

   Click on the settings cog to view the React DevTools settings and tick the **Highlight updates when components render** option. This option will display a blue-green outline around components in the page that re-render.

3. Fill in the form and notice a bluey green outline appears around the form every time a character is entered into a field:

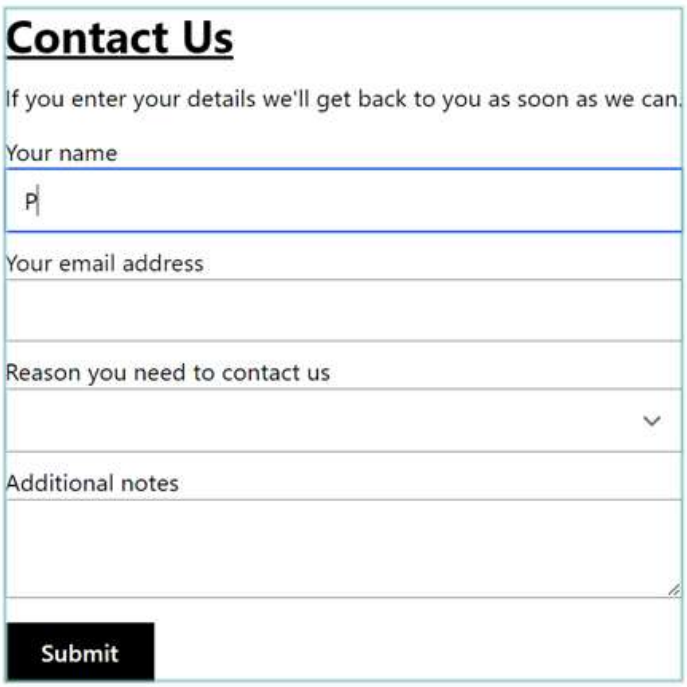

Figure 7.2 – Highlighted re-render for every keystroke

   So, every time a character is entered into a field, the whole form is re-rendered. This makes sense because a state change occurs when a field changes, and a state change causes a re-render. This isn't a huge problem in this small form but can be a significant performance problem in larger forms.

4. Complete all the fields in the form and click on the **Submit** button. The field values are output to the console.

That completes the first iteration of the form. Keep the app running as we reflect on the implementation and move to the next section.

The key takeaway from this section is that controlling field values with the state can lead to performance problems. Having to bind the state to each field also feels a bit repetitive.

Next, we will implement a more performant and succinct version of the form.

Using uncontrolled fields

Uncontrolled fields are the opposite of controlled fields – it's where field values *aren't* controlled by state. Instead, native browser features are used to obtain field values. In this section, we will refactor the contact form to use uncontrolled fields and see the benefits.

Carry out the following steps:

1. Open `ContactPage.tsx` and start by removing `useState` from the React import, because this is no longer required.
2. Then, at the top of the `component` function, remove the call to `useState` (this iteration of the form won't use any state).
3. Remove the `value` and `onChange` props from the field editors, because we are no longer controlling field values with the state.
4. Now, add a `name` attribute on all the field editors as follows:

```
<form onSubmit={handleSubmit}>
  <div className={fieldStyle}>
    <label htmlFor="name">Your name</label>
    <input type="text" id="name" name="name" />
  </div>
  <div className={fieldStyle}>
    <label htmlFor="email">Your email address</label>
    <input type="email" id="email" name="email" />
  </div>
  <div className={fieldStyle}>
    <label htmlFor="reason">
      Reason you need to contact us
    </label>
    <select id="reason" name="reason">
      ...
    </select>
  </div>
  <div className={fieldStyle}>
```

```
      <label htmlFor="notes">Additional notes</label>
      <textarea id="notes" name="notes" />
    </div>
    ...
  </form>;
```

The `name` attribute is important because it will allow us to easily extract field values in the form submit handler, which we will do next.

5. Add the following code in the submit handler to extract the field values before they are output to the console:

```
function handleSubmit(e: FormEvent<HTMLFormElement>) {
  e.preventDefault();
  const formData = new FormData(e.currentTarget);
  const contact = {
    name: formData.get('name'),
    email: formData.get('email'),
    reason: formData.get('reason'),
    notes: formData.get('notes'),
  } as Contact;
  console.log('Submitted details:', contact);
}
```

`FormData` is an interface that allows access to values in a form and takes in a form element in its constructor parameter. It contains a `get` method that returns the value of the field whose name is passed as an argument. For more information on `FormData`, see `https://developer.mozilla.org/en-US/docs/Web/API/FormData`.

That completes the refactoring of the form. To recap, uncontrolled fields don't have values stored in the state. Instead, field values are obtained using `FormData`, which relies on field editors having a `name` attribute.

Notice the reduced code in the implementation compared to the controlled fields implementation. We will now try the form and check whether the form is re-rendered on every keystroke. Carry out the following steps:

1. In the running app, make sure DevTools is still open with the **Highlight updates when the components render** option still ticked.
2. Fill in the form and you will notice that the re-render outline never appears. This makes sense because there is no longer any state, so a re-render can't occur because of a state change.

3. Complete all the fields in the form and click on the **Submit** button. The field values are output to the console just as they were before.

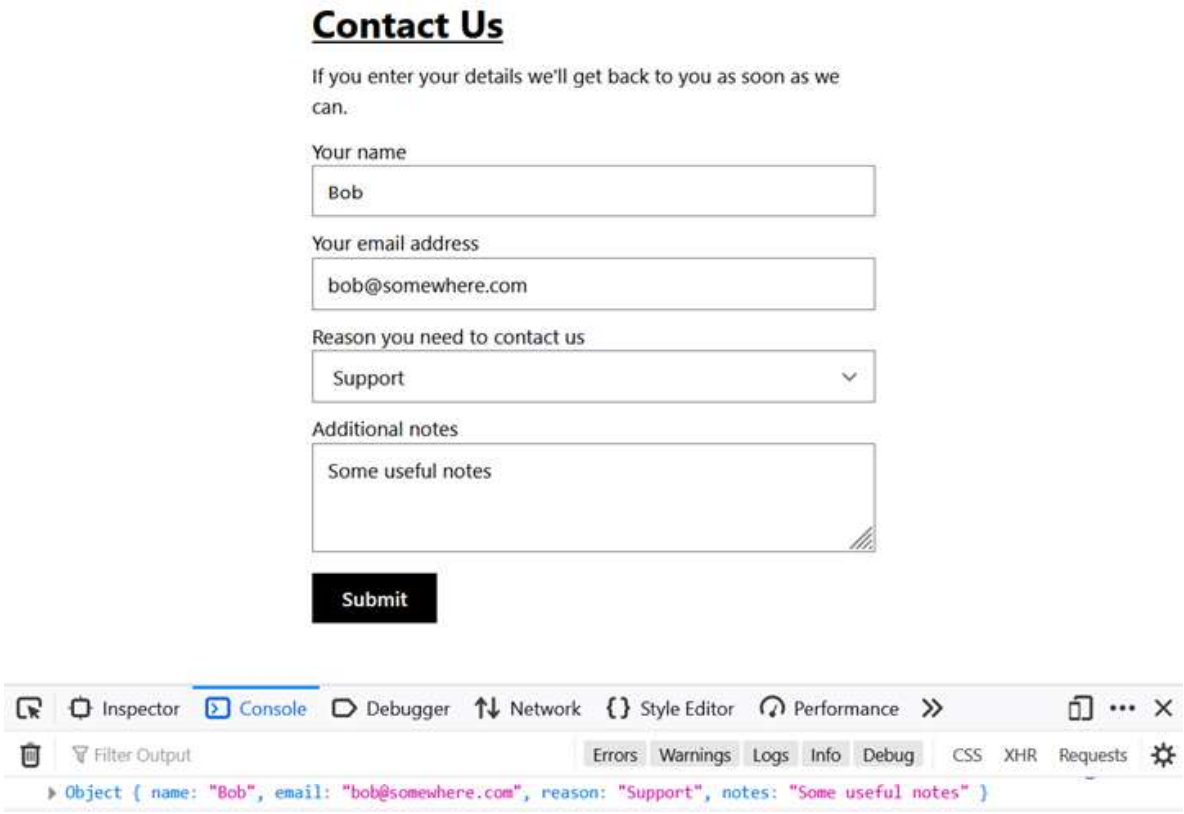

Figure 7.3 – Completed form with submitted data in console

4. Stop the app from running by pressing *Ctrl* + *C*.

So, this implementation is shorter, more performant, and an excellent approach for simple forms. The key points in the implementation are to include a `name` attribute for field editors and use the `FormData` interface to extract the form values.

The current implementation is very simple though – for example, there is no submission success message. In the next section, we will use React Router and add a submission message.

Using React Router Form

In *Chapter 6*, we started to learn about React Router's `Form` component. We learned that `Form` is a wrapper around the HTML `form` element that handles the form submission. We will now cover `Form` in more detail and use it to provide a nice submission success message on our contact form.

Carry out the following steps:

1. First, install React Router by executing the following command in the terminal:

```
npm i react-router-dom
```

2. Now, let's create a `ThankYouPage` component, which will inform the user that their submission has been successful. To do this, create a file called `ThankYouPage.tsx` in the `src` folder with the following content:

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

export function ThankYouPage() {
  const { name } = useParams<{ name: string }>();
  return (
    <div className="flex flex-col py-10 max-w-md
      mx-auto">
      <div
        role="alert"
        className="bg-green-100 py-5 px-6 text-base text-
          green-700 "
      >
        Thanks {name}, we will be in touch shortly
      </div>
    </div>
  );
}
```

The component uses a route parameter for the person's name that is included in the thank you message.

3. Next, open `App.tsx` and add the following imports from React Router:

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

We haven't come across React Router's `Navigate` component before – it is a component that performs navigation. We will use this in *step 5*, in the route definitions, to redirect from the root path to the contact page.

4. Add `contactPageAction` to the `import` statement for `ContactPage` and also import the `ThankYouPage` component:

```
import {
  ContactPage,
  contactPageAction
} from './ContactPage';
import { ThankYouPage } from './ThankYouPage';
```

Note that `contactPageAction` doesn't exist yet, so a compile error will occur. We will resolve this error in *step 9*.

5. Still in `App.tsx`, set up routes that render the contact and thank you pages:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Navigate to="contact" />,
  },
  {
    path: '/contact',
    element: <ContactPage />,
    action: contactPageAction,
  },
  {
    path: '/thank-you/:name',
    element: <ThankYouPage />,
  },
]);
```

There is an `action` property on the `contact` route that we haven't covered yet – this handles form submission. We have set this to `contactPageAction`, which we will create in *step 9*.

6. The last task in `App.tsx` is to change the `App` component to return `RouterProvider` with the route definitions:

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

7. Now, open `ContactPage.tsx` and add the following imports from React Router:

```
import {
  Form,
```

```
  ActionFunctionArgs,
  redirect,
} from 'react-router-dom';
```

8. In the JSX, change the `form` element to be a React Router `Form` component and remove the `onSubmit` attribute:

```
<Form method="post">
  ...
</Form>
```

We have set the form's method to `"post"` because the form will mutate data. The default form method is `"get"`.

9. Now, move the `handleSubmit` function outside the component, to the bottom of the file. Rename the function to `contactPageAction`, allow it to be exported, and make it asynchronous:

```
export async function contactPageAction(
  e: FormEvent<HTMLFormElement>
) {
  e.preventDefault();
  const formData = new FormData(e.currentTarget);
  const contact = {
    name: formData.get('name'),
    email: formData.get('email'),
    reason: formData.get('reason'),
    notes: formData.get('notes'),
  } as Contact;
  console.log('Submitted details:', contact);
}
```

This will now be a React Router action that handles part of the form submission.

10. Change the parameters on `contactPageAction` to the following:

```
export async function contactPageAction({
  request,
}: ActionFunctionArgs)
```

React Router will pass in a `request` object when it calls this function.

11. Remove the `e.preventDefault()` statement in `contactPageAction` because React Router does this for us.
12. Change the `formData` assignment to get the data from the React Router's `request` object:

```
const formData = await request.formData();
```

13. The last change to the `contactPageAction` function is to redirect to the thank you page at the end of the submission:

```
export async function contactPageAction({
  request,
}: ActionFunctionArgs) {
  ...
  return redirect(
    `/thank-you/${formData.get('name')}`
  );
}
```

14. Remove the `FormEvent` import because this is redundant now.
15. Run the app in development mode by executing `npm start` in the terminal.
16. The app will automatically redirect to the `Contact` page. Complete the form and submit it.

    The app will redirect to the thank you page:

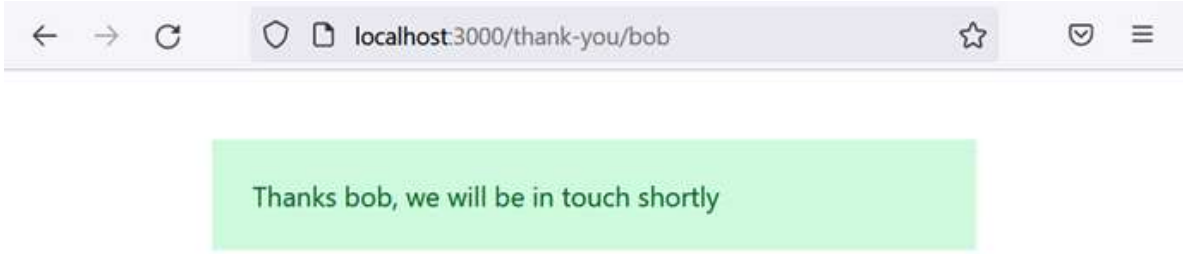

Figure 7.4 – Thank you page

That completes this section on React Router's form capability. Keep the app running as we recap and move to the next section.

The key points on React Router's `Form` component are as follows:

- React Router's `Form` component is a wrapper around the HTML `form` element
- The form is submitted to the current route by default, but can be submitted to a different path using the `path` attribute

- We can write logic inside the submission process using an action function defined on the route that is submitted to

For more information on React Router's Form component, see the following link: `https://reactrouter.com/en/components/form`.

Next, we will implement validation on the form.

Using native validation

In this section, we will add the required validation to the name, email, and reason fields and ensure that the email matches a particular pattern. We will use standard HTML form validation to implement these rules.

Carry out the following steps:

1. In `ContactPage.tsx`, add a `required` attribute to the name, email, and reason field editors to add HTML form required validation for these fields:

```
<Form method="post">
  <div className={fieldStyle}>
    ...
    <input type="text" id="name" name="name" required />
  </div>
  <div className={fieldStyle}>
    ...
    <input type="email" id="email" name="email" required
/>
  </div>
  <div className={fieldStyle}>
    ...
    <select id="reason" name="reason" required >...</
      select>
  </div>
  ...
</Form>
```

2. Add the following pattern-matching validation on the `email` field editor:

```
<input
  type="email"
  id="email"
```

```
    name="email"
    required
    pattern="\S+@\S+\.\S+"
  />
```

This pattern will ensure that the entry is in an email format.

3. In the running app, without populating any fields, submit the form. Validation kicks in and the form submission doesn't complete. Instead, the name field is focused and an error message appears beneath it:

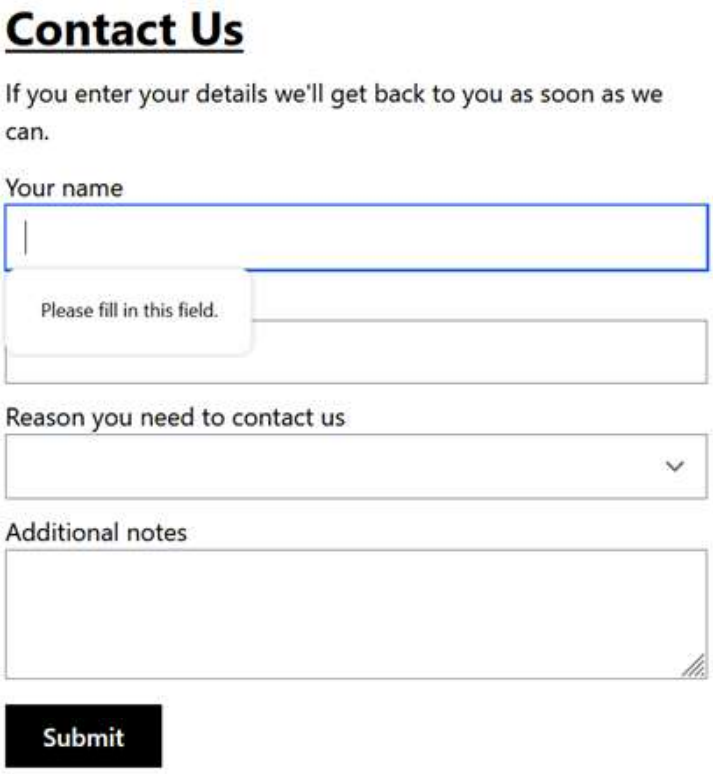

Figure 7.5 – HTML form validation message for the name field

Note that the error message is styled slightly differently in different browsers – the preceding screenshot is from Firefox.

4. Correctly fill in the name field so that it is valid. Then, move on to experiment with the email field validation. For example, try entering an email address without an @ character; you will find that the email field needs to be populated with a correctly formatted email address.

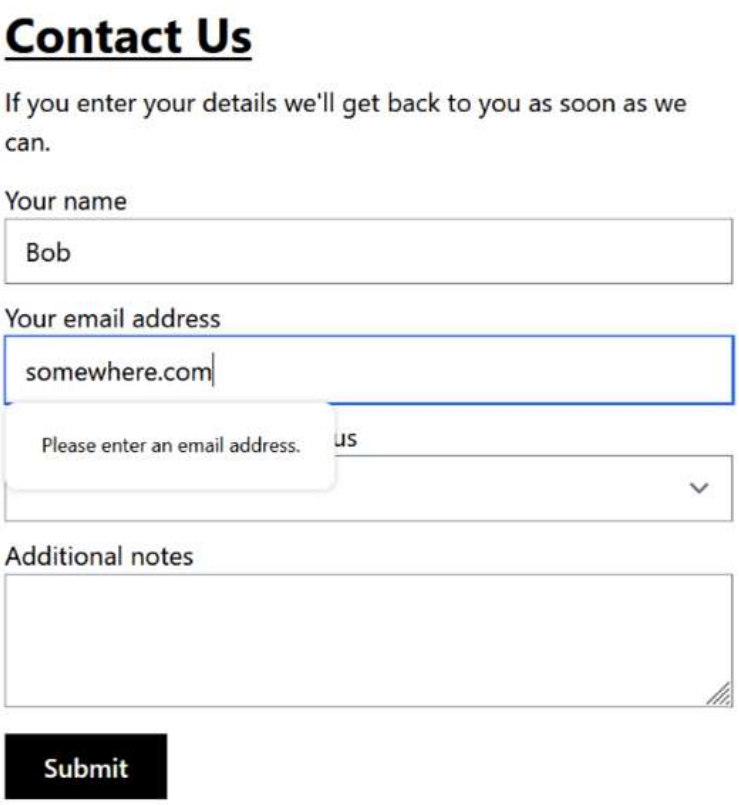

Figure 7.6 – HTML form validation message for the email field

5. Correctly fill in the email field so that it is valid. Then, move on to experiment with the reason field validation. Try to select the blank reason and you will find that a validation error occurs.
6. Correctly fill in all the fields and submit the form. You will find that the thank you message appears as it did previously.
7. Stop the app from running by pressing *CTRL + C*.

The simplicity of the implementation of standard HTML form validation is nice. However, if we want to customize the validation user experience, we'll need to write JavaScript to use the constraint validation API. For information on this API and more information on HTML form validation, see the following link: `https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation`.

In the next section, we'll use a popular form library to improve the validation user experience. This is a little easier to work with in React than the constraint validation API.

Using React Hook Form

In this section, we will learn about React Hook Form and use it to improve the validation user experience in our contact form.

Understanding React Hook Form

As the name suggests, React Hook Form is a React library for building forms. It is very flexible and can be used for simple forms such as our contact form, as well as large forms with complex validation and submission logic. It is also very performant and optimised not to cause unnecessary re-renders. It is also very popular with tens of thousands of GitHub stars and maturing nicely having been first released in 2019.

The key part of React Hooks Form is a `useForm` hook, which returns useful functions and the state. The following code snippet shows the `useForm` hook being called:

```
const {
  register,
  handleSubmit,
  formState: { errors, isSubmitting, isSubmitSuccessful }
} = useForm<FormData>();
```

`useForm` has a generic type parameter for the type of the field values. In the preceding example, the field values type is `FormData`.

Understanding the register function

A key function that `useForm` returns is a `register` function, which takes in a unique field name and returns the following in an object structure:

- An `onChange` handler, which happens when the field editor's value changes
- An `onBlur` handler, which happens when the field editor loses focus
- A reference to the field editor element
- The field name

These items returned from the `register` function are spread onto the field editor element to allow React Hook Form to efficiently track its value. The following code snippet allows a name field editor to be tracked by React Hook Form:

```
<input {...register('name')} />
```

After the result of `register` has been spread on to the `input` element, it will contain `ref`, `name`, `onChange`, and `onBlur` attributes.:

```
<input
  ref={someVariableInRHF}
  name="name"
  onChange={someHandlerInRHF}
  onBlur={anotherHandlerInRHF}
/>
```

The `ref`, `onChange`, and `onBlur` attributes will reference code in React Hook Form that tracks the value of the `input` element.

Specifying validation

Field validation is defined in the `register` field in an options parameter as follows:

```
<input {...register('name', {required: true})} />
```

In the preceding example, required validation is specified. The associated error message can be defined as an alternative to the `true` flag as follows:

```
<input
  {...register('name', { required: 'You must enter a name' })}
/>
```

There are a range of different validation rules that can be applied. See this page in the React Hook Form documentation for a list of all the rules that are available: `https://react-hook-form.com/get-started#applyvalidation`.

Obtaining validation errors

The `useForm` returns a state called `errors`, which contains the form validation errors. The `errors` state is an object containing invalid field error messages. For example, if a `name` field is invalid because a `required` rule has been violated, the `errors` object could be as follows:

```
{
  name: {
    message: 'You must enter your email address',
    type: 'required'
  }
}
```

Fields in a valid state don't exist in the `errors` object, so a field validation error message can be conditionally rendered as follows:

```
{errors.name && <div>{errors.name.message}</div>}
```

In the preceding code snippet, if the name field is valid, `errors.name` will be `undefined`, and so the error message won't render. If the name field is invalid, `errors.name` will contain the error and so the error message will render.

Handling submission

The `useForm` hook also returns a handler called `handleSubmit` that can be used for form submission. `handleSubmit` takes in a function that React Hook Form calls when it has successfully validated the form. Here's an example of `handleSubmit` being used:

```
function onSubmit(data: FormData) {
  console.log('Submitted data:', data);
}
return (
  <form onSubmit={handleSubmit(onSubmit)}>
  </form>
);
```

In the preceding example, `onSubmit` is only called in the submission when the form is successfully validated and not when the form is invalid.

The `isSubmitting` state can be used to disable elements whilst the form is being submitted. The following example disables the `submit` button while the form is being submitted:

```
<button type="submit" disabled={isSubmitting}>Submit</button>
```

`isSubmitSuccessful` can be used to conditionally render a successful submission message:

```
if (isSubmitSuccessful) {
  return <div>The form was successfully submitted</div>;
}
```

There are many more features in React Hook Form, but these are the essential functions and states that are commonly used. Refer to the React Hook Form documentation for more information at `https://react-hook-form.com/`.

Now that we understand React Hook Form's basics, we will refactor our contact form to use it.

Using React Hook Form

We will refactor the contact form that we have been working on to use React Hook Form. The form will contain the same features but the implementation will use React Hook Form. After making the code changes, we will check how often the form is re-rendered using React's DevTools.

We will remove the use of React Router's `Form` component– it currently handles form submission. React Hook Form is capable of handling submission as well, and we need it to fully control submission to ensure the form is valid as part of this process.

Carry out the following steps:

1. Let's start by installing React Hook Form. Run the following command in the terminal:

   ```
   npm i react-hook-form
   ```

   TypeScript types are included in this package, so there is no need for a separate installation.

2. Open `ContactPage.tsx` and add an `import` statement for `useForm` from React Hook Form:

   ```
   import { useForm } from 'react-hook-form';
   ```

3. Remove `Form`, `redirect`, and `ActionFunctionArgs` from the React Router `import` statement and replace them with `useNavigate`:

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

   We will use `useNavigate` to navigate to the thank you page at the end of the form submission.

4. Add the following call to `useForm` at the top of the `ContactPage` component and destructure the `register` and `handleSubmit` functions:

   ```
   export function ContactPage() {
     const { register, handleSubmit } = useForm<Contact>();
     ...
   }
   ```

5. Add the following call to `useNavigate` after the call to `useForm` to get a function that we can use to perform navigation:

   ```
   export function ContactPage() {
     const { register, handleSubmit } = useForm<Contact>();
     const navigate = useNavigate();
   }
   ```

6. In the JSX, replace the use of React Router's `Form` element with a native `form` element. Remove the `method` attribute on the `form` element, and replace it with the following `onSubmit` attribute:

```
<form onSubmit={handleSubmit(onSubmit)}>
  ...
</form>
```

We will implement the `onSubmit` function in the next step.

7. Add the following `onSubmit` function after the call to `useNavigate`. React Hook Form will call this function with the form data after it has ensured the form is valid:

```
const navigate = useNavigate();
function onSubmit(contact: Contact) {
  console.log('Submitted details:', contact);
  navigate(`/thank-you/${contact.name}`);
}
```

The function outputs the form data to the console and then navigates to the thank you page.

8. Remove the `contactPageAction` function at the bottom of the file because this is no longer required.

9. Replace the `name` attribute on the field editors with a call to `register`. Pass the field name into `register` and spread the result of `register` as follows:

```
<form onSubmit={handleSubmit(onSubmit)}>
  <div className={fieldStyle}>
    <label htmlFor="name">Your name</label>
    <input ... {...register('name')} />
  </div>
  <div className={fieldStyle}>
    <label htmlFor="email">Your email address</label>
    <input ... {...register('email')} />
  </div>
  <div className={fieldStyle}>
    <label htmlFor="reason">Reason you need to contact
      us</label>
    <select ... {...register('reason')}>
      ...
    </select>
  </div>
```

```
    <div className={fieldStyle}>
      <label htmlFor="notes">Additional notes</label>
      <textarea ... {...register('notes')} />
    </div>
    ...
  </Form>
```

React Hook Form will now be able to track these fields.

10. Open `App.tsx` and remove `contactPageAction` from the `import` statement for `ContactPage` and remove it from the `/contact` route:

```
import { ContactPage } from './ContactPage';
...
const router = createBrowserRouter([
  {
    path: '/',
    element: <Navigate to="contact" />,
  },
  {
    path: '/contact',
    element: <ContactPage />
  },
  {
    path: '/thank-you/:name',
    element: <ThankYouPage />,
  }
]);
```

11. Run the app in development mode by executing `npm start` in the terminal.
12. Open the React DevTools and ensure the **Highlight updates when components render** option is still ticked so that we can observe when the form is re-rendered.
13. Fill in the form with valid values. The re-render outline doesn't appear when the field values are entered because they aren't using the state to cause a re-render yet. This is confirmation that React Hook Form efficiently tracks field values.
14. Now, click on the **Submit** button. After the form has been successfully submitted, the field values are output to the console and the thank you page appears, just as before.

We've had to do a little more work to set up the form in React Hook Form so that it can track the fields. This enables React Hook Form to validate the fields, which we will implement in the next section.

Adding validation

We will now remove the use of standard HTML form validation and use React Hook Form's validation. Using React Hook Form's validation allows us to more easily provide a great validation user experience.

Carry out the following steps:

1. Open `ContactPage.tsx` and add the `FieldError` type to the React Hook Form `import` statement:

```
import { useForm, FieldError } from 'react-hook-form';
```

2. Destructure the `errors` state from `useForm` as follows:

```
const {
  register,
  handleSubmit,
  formState: { errors }
} = useForm<Contact>();
```

`errors` will contain validation errors if there are any.

3. Add a `noValidate` attribute to the form element to prevent any native HTML validation:

```
<form noValidate onSubmit={handleSubmit(onSubmit)}>
```

4. Remove the native HTML validation rules on all the field editors by removing the validation `required` and `pattern` attributes.

5. Add the required validation rules to the `register` function for the name, email, and reason fields, as follows:

```
<div className={fieldStyle}>
  <label htmlFor="name">Your name</label>
  <input
    type="text"
    id="name"
    {...register('name', {
      required: 'You must enter your name'
    })}
  />
</div>
<div className={fieldStyle}>
  <label htmlFor="email">Your email address</label>
```

```
    <input
      type="email"
      id="email"
      {...register('email', {
        required: 'You must enter your email address'
      })}
    />
  </div>
  <div className={fieldStyle}>
    <label htmlFor="reason">Reason you need to contact us</
  label>
    <select
      id="reason"
      {...register('reason', {
        required: 'You must enter the reason for contacting
          us'
      })}
    >
      ...
    </select>
  </div>
```

We have specified the validation error message with the validation rules.

6. Add an additional rule for the email field to ensure it matches a particular pattern:

```
<input
  type="email"
  id="email"
  {...register('email', {
    required: 'You must enter your email address',
    pattern: {
      value: /\S+@\S+\.\S+/,
      message: 'Entered value does not match email
        format',
    }
  })}
/>
```

7. We will now style fields if they are invalid. Each field is going to have the same styling logic, so define the style in a function as follows:

```
function getEditorStyle(fieldError: FieldError |
undefined) {
  return fieldError ? 'border-red-500' : '';
}
return (
  <div>
    ...
  </div>
);
```

The field error is passed into the `getEditorStyle` function. The function returns a Tailwind CSS class that styles the element with a red border if there is an error.

8. Use the `getEditorStyle` function to style the name, email, and reasons fields when invalid:

```
<div className={fieldStyle}>
  <label htmlFor="name">Your name</label>
  <input ... className={getEditorStyle(errors.name)} />
</div>
<div className={fieldStyle}>
  <label htmlFor="email">Your email address</label>
  <input ... className={getEditorStyle(errors.email)} />
</div>
<div className={fieldStyle}>
  <label htmlFor="reason">Reason you need to contact us</
label>
  <select ... className={getEditorStyle(errors.reason)} >
    ...
  </select>
</div>
```

React Hook Form's `errors` state contains a property for a field containing a validation error if the field has an invalid value. For example, if the name field value is invalid, `errors` will contain a property called `name`.

9. Now, let's display the validation error under each field when it's invalid. The structure and style of the error will be the same for each field with a varying message, so we will create a reusable validation error component. Create a file in the `src` folder called `ValidationError.tsx` with the following content:

```
import { FieldError } from 'react-hook-form';

type Props = {
  fieldError: FieldError | undefined;
};
export function ValidationError({ fieldError }: Props) {
  if (!fieldError) {
    return null;
  }
  return (
    <div role="alert" className="text-red-500 text-xs
      mt-1">
      {fieldError.message}
    </div>
  );
}
```

The component has a `fieldError` prop for the field error from React Hook Form. Nothing is rendered if there is no field error. If there is an error, it is rendered with red text inside a `div` element. The `role="alert"` attribute allows a screen reader to read the validation error.

10. Return to `ContactPage.tsx` and import the `ValidationError` component:

```
import { ValidationError } from './ValidationError';
```

11. Add instances of `ValidationError` beneath each field editor as follows:

```
<div className={fieldStyle}>
  <label htmlFor="name">Your name</label>
  <input ... />
  <ValidationError fieldError={errors.name} />
</div>
<div className={fieldStyle}>
  <label htmlFor="email">Your email address</label>
  <input ... />
  <ValidationError fieldError={errors.email} />
```

```
</div>
<div className={fieldStyle}>
  <label htmlFor="reason">Reason you need to contact us</
label>
  <select ... >...</select>
  <ValidationError fieldError={errors.reason} />
</div>
```

That completes the implementation of the form validation. We will now test our enhanced form in the following steps:

1. In the running app, ensure the **Highlight updates when components render** option is still ticked in DevTools so we can observe when the form is re-rendered.
2. Click on the **Submit** button without filling in the form.

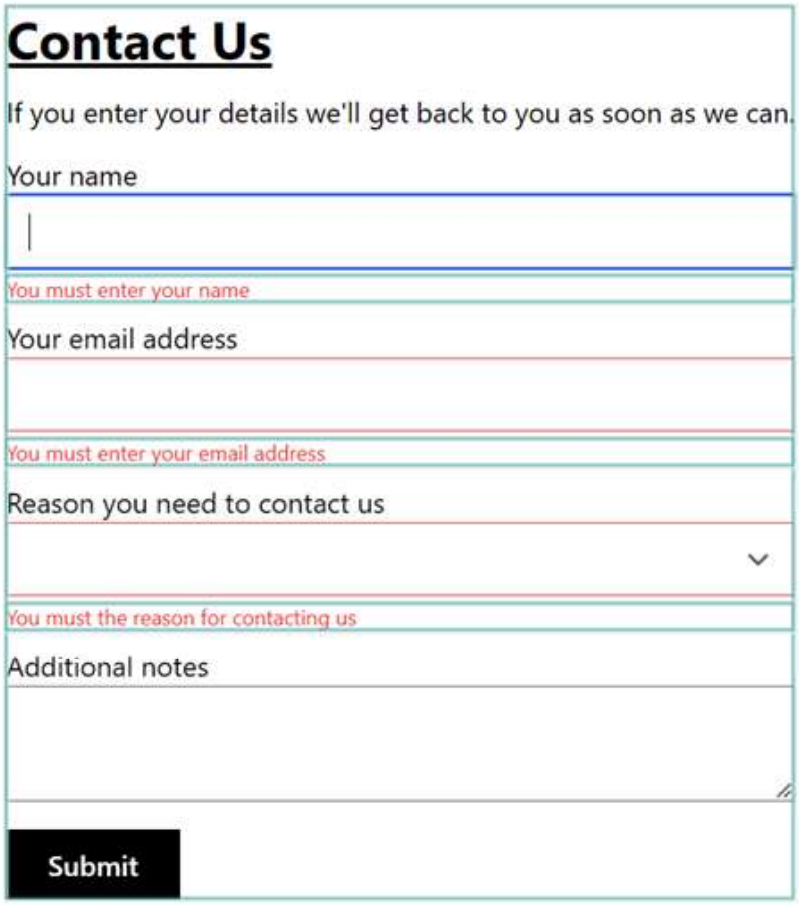

Figure 7.7 – Highlighted re-render and validation errors when the form is submitted

3. Fill in the form and click on the **Submit** button. The validation errors appear. You will also notice that the form now re-renders when submitted because of the `errors` state. This is a necessary re-render because we need the page to update with the validation error messages.

   Another thing that you may have noticed is that nothing is output to the console because our `onSubmit` function is not called until the form is valid.

4. Fill in the form correctly and submit it. The field values are now output to the console and the thank you page appears.

The form is working nicely now.

One thing you may have spotted is when validation actually happens – it happens when the form is submitted. We will change the validation to happen every time a field editor loses focus. Carry out the following steps:

1. On `ContactPage.tsx`, add the following argument to the `useForm` call:

```
const {
  register,
  handleSubmit,
  formState: { errors },
} = useForm<Contact>({
  mode: "onBlur",
  reValidateMode: "onBlur"
});
```

   The `mode` option now tells React Hook Form to initially validate when a field editor loses focus. The `reValidationMode` option now tells React Hook Form to validate subsequently when a field editor loses focus.

2. In the running app, visit the fields in the form without filling them in and see the validation happen.

That completes the form and indeed this section on React Hook Form. Here's a recap of the critical parts of React Hook Form:

- React Hook Form doesn't cause unnecessary re-renders when a form is filled in.
- React Hook Form's `register` function needs to be spread on field editors. This function allows field values to be efficiently tracked and allows validation rules to be specified.
- React Hook Form's submit handler automatically prevents a server post and ensures the form is valid before our submission logic is called.

Next, we will summarize this chapter.

Summary

At the start of this chapter, we learned that field values in a form can be controlled by the state. However, this leads to lots of unnecessary re-rendering of the form. We then realised that not controlling field values with the state and using the `FormData` interface to retrieve field values instead is more performant and requires less code.

We used React Router's `Form` component, which is a wrapper around the native `form` element. It submits data to a client-side route instead of a server. However, it doesn't cover validation – we tried using native HTML validation for that, which was simple to implement, but providing a great user experience with native HTML validation is tricky.

We introduced ourselves to a popular forms library called React Hook Form to provide a better validation user experience. It contains a `useForm` hook that returns useful functions and a state. The library doesn't cause unnecessary renders, so it is very performant.

We learned that React Hook Form's `register` function needs to be spread on every field editor. This is so that it can efficiently track field values without causing unnecessary renders. We learned that React Hook Form contains several common validation rules, including required fields and field values that match a particular pattern. Field validation rules are specified in the `register` function and can be specified with an appropriate validation message. `useForm` returns an `errors` state variable, which can be used to render validation error messages conditionally.

We explored the submit handler that React Hook Form provides. We learned that it prevents a server post and ensures that the form is valid. This submit handler has an argument for a function that is called with the valid form data.

In the next chapter, we will focus on state management in detail.

Questions

Answer the following questions to check what you have learned about forms in React:

1. How many times will the following form render after the initial render when `Bob` is entered into the name field?

```
function ControlledForm () {
  const [name, setName] = useState('');
  return (
    <form
      onSubmit={(e) => {
        e.preventDefault();
        console.log(name);
      }}
```

```
    >
      <input
        placeholder="Enter your name"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
    </form>
  );
}
```

2. How many times will the following form render after the initial render when `Bob` is entered into the name field?

```
function UnControlledForm() {
  return (
    <form
      onSubmit={(e) => {
        e.preventDefault();
        console.log(
          new FormData(e.currentTarget).get('name')
        );
      }}
    >
      <input placeholder="Enter your name" name="name" />
    </form>
  );
}
```

3. The following form contains an uncontrolled search field. When search criteria is entered into it and submitted, `null` appears in the console rather than the search criteria. Why is that so?

```
function SearchForm() {
  return (
    <form
      onSubmit={(e) => {
        e.preventDefault();
        console.log(
          new FormData(e.currentTarget).get('search')
        );
      }}
    >
```

```
        <input type="search" placeholder="Search ..." />
      </form>
    );
  }
```

4. The following component is a search form implemented using React Hook Form. When search criteria are entered into it and submitted, an empty object appears in the console, rather than an object containing the search criteria. Why is that so?

```
function SearchReactHookForm() {
  const { handleSubmit } = useForm();
  return (
    <form
      onSubmit={handleSubmit((search) => console.
        log(search))}
    >
      <input type="search" placeholder="Search ..." />
    </form>
  );
}
```

5. The following component is another search form implemented using React Hook Form. The form does function correctly but a type error is raised on the `onSubmit` parameter. How can the type error be resolved?

```
function SearchReactHookForm() {
  const { handleSubmit, register } = useForm();
  async function onSubmit(search) {
    console.log(search.criteria);
    // send to web server to perform the search
  }
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input
        type="search"
        placeholder="Search ..."
        {...register('criteria')}
      />
    </form>
  );
}
```

6. Continuing with the search form from the last question, how can we disable the `input` element while the search is being performed on the web server?
7. Continuing with the search form from the last question, how can we prevent a search from executing if the criteria is blank?

Answers

1. The form will render three times when `Bob` is entered into the name field. This is because each change of value causes a re-render because the value is bound to the state.
2. The form will never re-render when `Bob` is entered into the name field because its value isn't bound to the state.
3. The `FormData` interface requires the `name` attribute to be on the `input` element or it won't be able to find it and it will return `null`:

```
<input type="search" placeholder="Search ..."
name="search" />
```

4. For React Hook Form to track the field, the `register` function needs to be spread on the `input` element as follows:

```
function SearchReactHookForm() {
  const { handleSubmit, register } = useForm();
  return (
    <form
      onSubmit={handleSubmit((search) => console.
        log(search))}
    >
      <input
        type="search"
        placeholder="Search ..."
        {...register('criteria')}
      />
    </form>
  );
}
```

5. A type for field values can be defined and specified in the call to `useForm` and also the `onSubmit` search parameter:

```
type Search = {
  criteria: string;
```

```
};
function SearchReactHookForm() {
  const { handleSubmit, register } = useForm<Search>();
  async function onSubmit(search: Search) {
    ...
  }
  return ...
}
```

6. React Hook Form's `isSubmitting` state can be used to disable the `input` element while the search is being performed on the web server:

```
function SearchReactHookForm() {
  const {
    handleSubmit,
    register,
    formState: { isSubmitting },
  } = useForm<Search>();
  ...
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input
        type="search"
        placeholder="Search ..."
        {...register('criteria')}
        disabled={isSubmitting}
      />
    </form>
  );
}
```

7. The required validation can be added to the search form to prevent a search from executing if the criteria are blank:

```
<input
  type="search"
  placeholder="Search ..."
  {...register('criteria', { required: true })}
  disabled={isSubmitting}
/>
```