5

Approaches to Styling React Frontends

In this chapter, we will style the alert component we worked on in previous chapters using four different approaches. First, we will use plain CSS and understand the downsides of this approach. Then, we will move on to use **CSS modules**, which will resolve plain CSS's main problem. We will then use a **CSS-in-JS** library called Emotion and a library called Tailwind CSS and will understand the benefits of each of these libraries.

We will also learn how to use SVGs in React apps and use them in the alert component for the information and warning icons.

We'll cover the following topics:

- Using plain CSS
- Using CSS modules
- Using CSS-in-JS
- Using Tailwind CSS
- Using SVGs

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 used in this chapter can be found online at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter5`.

Using plain CSS

We will start this section by setting up a React and TypeScript project with the alert component from *Chapter 3, Setting Up React and TypeScript*. Next, we will add the alert component from *Chapter 3* and style it using plain CSS. Finally, we will look at one of the challenges with plain CSS and discover how we could mitigate it.

Creating the project

The project we will be using is the one we completed at the end of *Chapter 3*. This can be found at the following location: `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter3/Section2-Creating-a-project-with-Create-React-App/myapp`. To copy this locally, carry out the following steps:

1. Open Visual Studio Code in a folder of your choice.
2. Run the following command in the terminal to clone the GitHub repository for the book:

```
git clone https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition.git
```

3. Reopen Visual Studio Code in the `Learn-React-with-TypeScript-2nd-Edition\Chapter3\Section2-Creating-a-project-with-Create-React-App\myapp` subfolder. This contains the project in the form it was in at the end of *Chapter 3*.
4. Run the following command to install all the dependencies:

```
npm i
```

The project is now set up. Next, we will take some time to understand how to use plain CSS in React components.

Understanding how to reference CSS

Create React App has already enabled the use of plain CSS in the project. In fact, if you look in `App.tsx`, it already uses plain CSS:

```
...
import './App.css';
...
```

```
function App() {
  return (
    <div className="App">
      ...
    </div>
  );
}
...
```

CSS styles from the `App.css` file are imported, and the `App` CSS class is referenced on the outer `div` element.

React uses a `className` attribute rather than `class` because `class` is a reserved word in JavaScript. The `className` attribute is converted to a `class` attribute during transpilation.

The CSS import statement is a webpack feature. As webpack processes all the files, it will include all the imported CSS in the bundle.

Carry out the following steps to explore the CSS bundle that the project produces:

1. Start by opening and looking at `App.css`. As we have seen already, `App.css` is used within `App.tsx`. However, it contains CSS classes that are no longer used, such as `App-header` and `App-logo`. These classes were referenced in the `App` component before we removed them when we added the alert component. Leave the redundant CSS classes in place.
2. Open the `index.tsx` file and you'll notice that `index.css` is imported. However, no CSS classes are referenced within this file. If you open `index.css`, you will notice that it only contains CSS rules that target element names and no CSS classes.
3. Run the following command in the terminal to produce a production build:

   ```
   npm run build
   ```

   After a few seconds, the build artifacts will appear in a `build` folder at the project's root.
4. Open `index.html` in the `build` folder and notice all the whitespace has been removed because it is optimized for production. Next, find the `link` element that references the CSS file and note down the path – it will be something similar to `/static/css/main.073c9b0a.css`.

Figure 5.1 – The link element in index.html

5. Open up the referenced CSS file. All the whitespace has been removed because it is optimized for production. Notice that it contains all the CSS from `index.css` and `App.css`, including the redundant `App-header` and `App-logo` CSS classes.

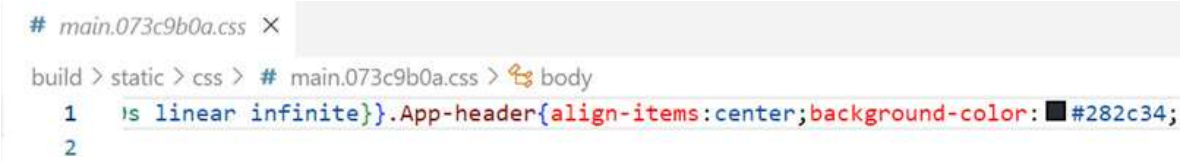

Figure 5.2 – The bundled CSS file, including redundant App-header CSS class

The key point here is that webpack doesn't remove any redundant CSS – it will include all the content from all the CSS files that have been imported.

Next, we will style the alert component with plain CSS.

Using plain CSS in the alert component

Now that we understand how to use plain CSS within React, let's style the alert component. Carry out the following steps:

1. Add a CSS file called `Alert.css` in the `src` folder. This is available in GitHub at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter5/Section1-Using-plain-CSS/app/src/Alert.css` to copy.
2. We will add the CSS classes step by step and understand the styles in each class. Start by adding a `container` class into `Alert.css`:

```
.container {
  display: inline-flex;
  flex-direction: column;
  text-align: left;
  padding: 10px 15px;
  border-radius: 4px;
  border: 1px solid transparent;
}
```

This will be used on the outer `div` element. The style uses an inline flexbox, with the items flowing vertically and left-aligned. We've also added a nice rounded border and a bit of padding between the border and child elements.

3. Add the following additional classes that can be used within `container`:

```
.container.warning {
  color: #e7650f;
  background-color: #f3e8da;
}
.container.information {
  color: #118da0;
  background-color: #dcf1f3;
}
```

We will use these classes for the different types of alerts to color them appropriately.

4. Add the following class for the header container element:

```
.header {
  display: flex;
  align-items: center;
  margin-bottom: 5px;
}
```

This will be applied to the element that contains the icon, heading, and close button. It uses a flexbox that flows horizontally with child elements vertically centered. It also adds a small gap at the bottom before the alert message.

5. Now add the following class for the icon to give it a width of 30 px:

```
.header-icon {
  width: 30px;
}
```

6. Next, add the following class to apply to the heading to make it bold:

```
.header-text {
  font-weight: bold;
}
```

7. Add the following class to apply to the close button:

```
.close-button {
  border: none;
  background: transparent;
  margin-left: auto;
```

```
    cursor: pointer;
  }
```

This removes the border and background. It also aligns the button to the right of the header and gives it a pointer mouse cursor.

8. Add the following class for the content element:

```
.content {
  margin-left: 30px;
  color: #000;
}
```

This adds a left margin so the message horizontally aligns with the heading and sets the text color to black.

That completes all the CSS class definitions.

9. Open `Alert.tsx` and add an import statement for the CSS file we just created:

```
import './Alert.css';
```

10. Now we are going to reference the CSS classes we just created in the elements of the alert component. Add the following highlighted CSS class name references in the alert JSX to do this:

```
<div className={`container ${type}`}>
  <div className="header">
    <span
      ...
      className="header-icon"
    >
      {type === "warning" ? "⚠" : "i"}
    </span>
    <span className="header-text">{heading}</span>
  </div>
  {closable && (
    <button
      ...
      className="close-button"
    >
      ...
    </button>
```

```
    )}
    <div className="content">{children}</div>
  </div>
```

The elements in the alert component are now being styled by the CSS classes in the imported CSS file.

11. Move the close button so that it is located inside the header container, under the `header` element:

```
<div className={`container ${type}`}>
  <div className="header">
    ...
    <span className="header-text">{heading}</span>
    {closable && (
      <button
        aria-label="Close"
        onClick={handleCloseClick}
        className="close-button"
      >
        <span role="img" aria-label="Close">
          ✕
        </span>
      </button>
    )}
  </div>
  <div className="content">{children}</div>
</div>;
```

12. Start the app in development mode by running `npm start` in the terminal.

    After a few seconds an improved alert component will appear in the browser:

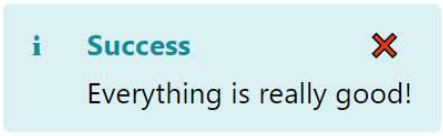

Figure 5.3 – A styled alert component with plain CSS

That completes the alert component's styling, but let's continue so that we can observe a downside of plain CSS.

Experiencing CSS clashes

We will now see an example of CSS with different components clashing. Keep the app running in development mode and then follow these steps:

1. Open `App.tsx` and change the referenced CSS class from `App` to `container` on the `div` element:

```
<div className="container">
  <Alert ...>
    ...
  </Alert>
</div>
```

2. Open `App.css` and rename the `App` CSS class to `container` and also add `20px` of padding to it:

```
.container {
  text-align: center;
  padding: 20px;
}
```

3. Now, look at the running app and notice that the alert is no longer centered horizontally on the page. Inspect the elements using the browser DevTools. If you inspect the `div` element from the `App` component, you will see that styles from the `container` CSS class in the alert component have been applied to it as well as the `container` CSS class we just added. So, the `text-align` CSS property is `left` rather than `center`.

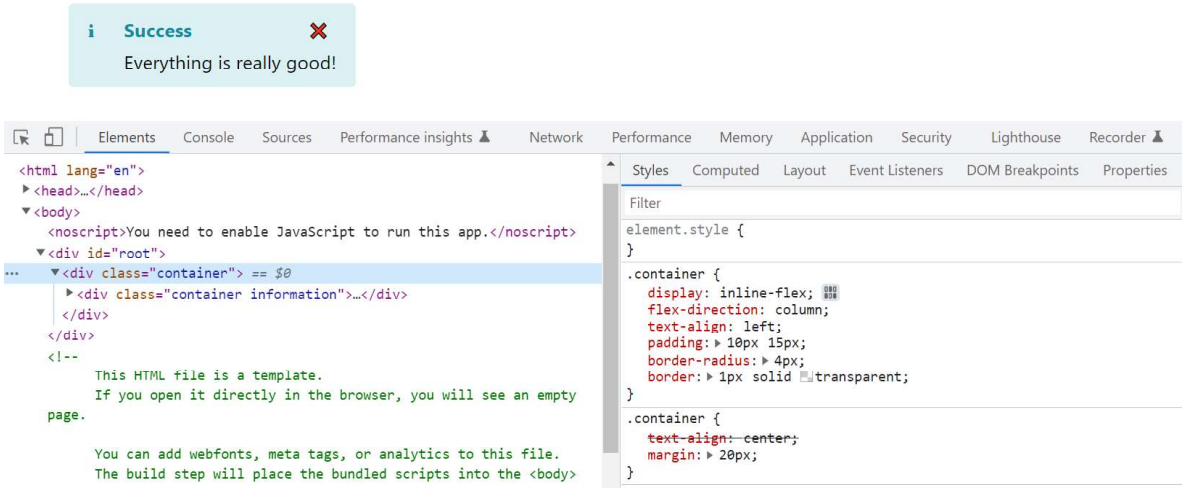

Figure 5.4 – Clashing CSS classes

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

The key point here is that plain CSS classes are scoped to the whole app and not just the file it is imported into. This means that CSS classes can clash if they have the same name, as we have just experienced

A solution to CSS clashes is to carefully name them using **BEM**. For example `container` in the `App` component could be called `App__container`, and `container` in the `Alert` component could be called `Alert__container`. However, this requires discipline from all members of a development team.

> **Note**
>
> **BEM** stands for **Block**, **Element**, **Modifier** and is a popular naming convention for CSS class names. More information can be found at the following link: `https://css-tricks.com/bem-101/`.

Here's a quick recap of this section:

- Create React App configures webpack to process CSS so that CSS files can be imported into React component files
- All the styles in an imported CSS file are applied to the app – there is no scoping or removing redundant styles

Next, we will learn about a styling approach that doesn't suffer from CSS clashes across components.

Using CSS modules

In this section, we will learn about an approach to styling React apps called **CSS modules**. We will start by understanding CSS modules and then we will use them within the alert component that we have been working on.

Understanding CSS modules

CSS modules is an open source library available on GitHub at `https://github.com/css-modules/css-modules`, which can be added to the webpack process to facilitate the automatic scoping of CSS class names.

A CSS module is a CSS file, just like in the previous section; however, the filename has an extension of `.module.css` rather than `.css`. This special extension allows webpack to differentiate a CSS module file from a plain CSS file so that it can be processed differently.

A CSS module file is imported into a React component file as follows:

```
import styles from './styles.module.css';
```

This is similar to the syntax of importing plain CSS, but a variable is defined to hold CSS class name mapping information. In the preceding code snippet, the CSS class name information is imported into a variable called `styles`, but the variable name can be anything we choose.

The CSS class name mapping information variable is an object containing property names corresponding to the CSS class names. Each class name property contains a value of a scoped class name to be used on a React component. Here is an example of the mapping object that has been imported into a component called `MyComponent`:

```
{
  container: "MyComponent_container__M7tzC",
  error: "MyComponent_error__vj8Oj"
}
```

The scope CSS class name starts with the component filename, then the original CSS class name, followed by a random string. This naming construct prevents class names from clashing.

Styles within a CSS module are referenced in a component's `className` attribute as follows:

```
<span className={styles.error}>A bad error</span>
```

The CSS class name on the element would then resolve to the scoped class name. In the preceding code snippets, `styles.error` would resolve to `MyComponent_error__ vj8Oj`. So, the styles in the running app will be the scoped style names and not the original class names.

Projects created using Create React App already have CSS modules installed and configured with webpack. This means we don't have to install CSS modules in order to start using them in our project.

Next, we will use CSS modules in the alert component we have worked on.

Using CSS modules in the alert component

Now that we understand CSS modules, let's use them in the alert component. Carry out the following steps:

1. Start by renaming `Alert.css` to `Alert.module.css`; this file can now be used as a CSS module.
2. Open `Alert.module.css` and change the CSS class names to camel case rather than kebab case. This will allow us to reference the scoped CSS class names more easily in the component – for example, `styles.headerText` rather than `styles["header-text"]`. The changes are as follows:

```
…
.headerIcon {
  ...
```

```
}
.headerText {
  ...
}
.closeButton {
  ...
}
```

3. Now, open `Alert.tsx` and change the CSS import statement to import the CSS module as follows:

```
import styles from './Alert.module.css';
```

4. In the JSX, change the class name references to use the scoped names from the CSS module:

```
<div className={`${styles.container} ${styles[type]}`}>
  <div className={styles.header}>
    <span
      ...
      className={styles.headerIcon}
    >
      {type === "warning" ? "⚠" : "i"}
    </span>
    {heading && (
      <span className={styles.headerText}>{heading}</
        span>
    )}
    {closable && (
      <button
        ...
        className={styles.closeButton}
      >
        ...
      </button>
    )}
  </div>
  <div className={styles.content}>{children}</div>
</div>
```

5. Start the app by running `npm start` in the terminal.

   After a few seconds, the styled alert will appear. This time the alert will be horizontally centered, which is a sign that styles are no longer clashing.

6. Inspect the elements in the DOM using the browser's DevTools. You will see that the alert component is now using scoped CSS class names. This means the alert container styles no longer clash with the app container styles.

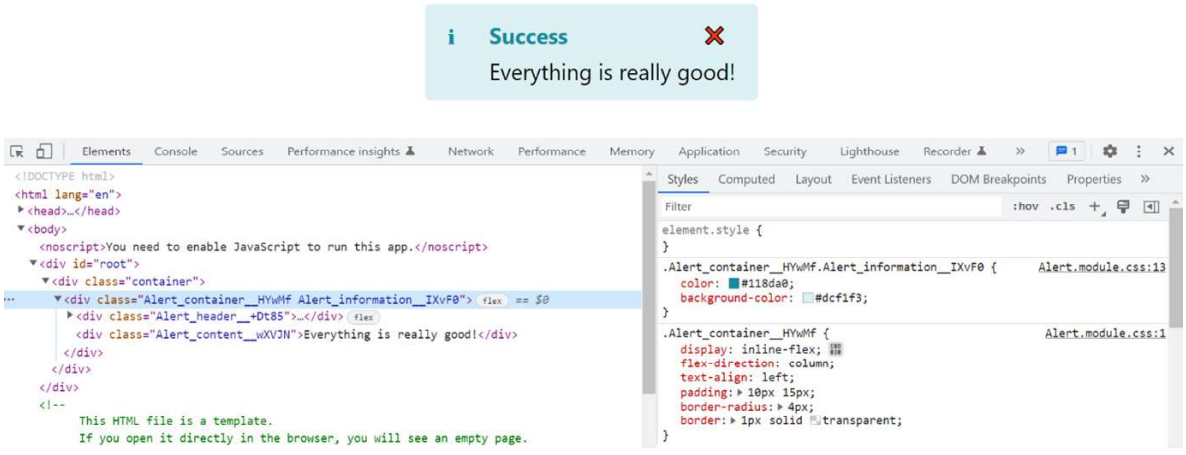

Figure 5.5 – The CSS module scoped class names

7. Stop the running app before continuing by pressing *Ctrl* + *C*.

8. To round off our understanding of CSS modules, let's see what happens to the CSS in a production build. However, before we do that, let's add a redundant CSS class at the bottom of `Alert.module.css`:

```
...
.content {
  margin-left: 30px;
  color: #000;
}
.redundant {
  color: red;
}
```

9. Now create a production build by executing `npm run build` in the terminal.

   After a few seconds, the build artifacts are created in the `build` folder.

10. Open the bundled CSS file and you will notice the following points:

    - It contains all the CSS from `index.css`, `App.css`, and the CSS module we just created.
    - The class names from the CSS module are scoped. This will ensure that the styles in production don't clash, as they did not in development mode.
    - It contains the redundant CSS class name from the CSS module.

Figure 5.6 – The redundant CSS class included in CSS bundle

That completes the refactoring of the alert component to use CSS modules.

> **Note**
> For more information on CSS modules, visit the GitHub repository at `https://github.com/css-modules/css-modules`.

Here's a recap of what we have learned about CSS modules:

- CSS modules allow CSS class names to be automatically scoped to a React component. This prevents styles for different React components from clashing.
- CSS modules isn't a standard browser feature; instead, it is an open source library that can be added to the webpack process.
- CSS modules are pre-installed and pre-configured in projects created with Create React App.
- Similar to plain CSS, redundant CSS classes are not pruned from the production CSS bundle.

Next, we will learn about another approach to styling React apps.

Using CSS-in-JS

In this section, we start by understanding CSS-in-JS and its benefits. We will then refactor the alert component we have used to implement CSS-in-JS and observe how it differs from CSS modules.

Understanding CSS-in-JS

CSS-in-JS isn't a browser feature, and it isn't even a specific library – instead, it is a type of library. Popular examples of CSS-in-JS libraries are **styled-components** and **Emotion**. There isn't a significant difference between styled-components and Emotion – they are both popular and have similar APIs. We will use Emotion in this chapter.

Emotion generates styles that are scoped, similar to CSS modules. However, you write the CSS in JavaScript rather than in a CSS file – hence the name *CSS-in-JS*. In fact, you can write the CSS directly on JSX elements as follows:

```
<span
  css={css`
    font-weight: 700;
    font-size: 14;
  `}
>
  {text}
</span>
```

Each CSS-in-JS library's syntax is slightly different – the preceding example is a code snippet from Emotion styling.

Having styles directly on the component allows a developer to fully understand the component without having to visit another file. This obviously increases the file size, which can make the code harder to read. However, child components can be identified and extracted out of the file to mitigate large file sizes. Alternatively, styles can be extracted from component files into a JavaScript function that is imported.

A massive benefit of CSS-in-JS is that you can mix logic into the style, which is really useful for highly interactive apps. The following example contains a conditional `font-weight` dependent on an `important` prop and a conditional `font-size` dependent on a `mobile` prop:

```
<span
  css={css`
    font-weight: ${important ? 700 : 400};
    font-size: ${mobile ? 15 : 14};
  `}
>
  {text}
</span>
```

JavaScript string interpolation is used to define the conditional statement.

The equivalent plain CSS would be similar to the following example, with separate CSS classes created for the different conditions:

```
<span
  className={`${important ? "text-important" : ""} ${
    mobile ? "text-important" : ""
  }`}
>
  {text}
</span>
```

If a style on an element is highly conditional, then CSS-in-JS is arguably easier to read and certainly easier to write.

Next, we will use Emotion in the alert component we have worked on.

Using Emotion in the alert component

Now that we understand CSS-in-JS, let's use Emotion in the alert component. To do so, carry out the following steps. All the code snippets used can be found at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter5/Section3-Using-CSS-in-JS/app/src/Alert.tsx`:

1. Create React App doesn't install and set up Emotion, so we first need to install Emotion. Run the following command in the terminal:

   ```
   npm i @emotion/react
   ```

   This will take a few seconds to install.

2. Open `Alert.tsx` and remove the CSS module import.
3. Add an import for the `css` prop from Emotion with a special comment at the top of the file:

   ```
   /** @jsxImportSource @emotion/react */
   import { css } from '@emotion/react';
   import { useState } from 'react';
   ```

   This special comment changes JSX elements to be transpiled using Emotion's `jsx` function instead of React's `createElement` function. Emotion's `jsx` function adds styles to elements containing Emotion's `css` prop.

4. In the JSX, we need to replace all the `className` props with the equivalent Emotion `css` attributes. The styles are largely the same as defined in the CSS file we created earlier, so the explanations won't be repeated.

   We will style one element at a time, starting with the outer `div` element:

```
<div
  css={css`
    display: inline-flex;
    flex-direction: column;
    text-align: left;
    padding: 10px 15px;
    border-radius: 4px;
    border: 1px solid transparent;
    color: ${type === "warning" ? "#e7650f" : "#118da0"};
    background-color: ${type === "warning"
      ? "#f3e8da"
      : "#dcf1f3"};
  `}
>
  ...
</div>
```

   There are a few important points to explain in this code snippet:

   - The `css` attribute isn't usually valid on JSX elements. The special comment at the top of the file (`/** @jsxImportSource @emotion/react */`) allows this.
   - The `css` attribute is set to a **tagged template literal**. This is a special string that gets processed by the function specified before it, which is a function called `css` in this case. For more information on tagged template literals, see `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals`.
   - The tagged template literal converts the style to a CSS class at runtime. We will verify this in *step 14*.
   - String interpolation is used to implement the conditional styles for the colors. Remember that we had to define three CSS classes using plain CSS or CSS modules. This CSS-in-JS version is arguably more readable and certainly more concise.

5. Next, style the header container:

```
<div
  css={css`
```

```
      display: flex;
      align-items: center;
      margin-bottom: 5px;
    `}
  >
    <span role="img" ... > ... </span>
    <span ...>{heading}</span>
    {closable && ...}
  </div>
```

6. Next, style the icon as follows:

```
<span
  role="img"
  aria-label={type === "warning" ? "Warning" :
    "Information"}
  css={css`
    width: 30px;
  `}
>
  {type === "warning" ? "⚠" : "i"}
</span>
```

7. Then, style the heading as follows:

```
<span
  css={css`
    font-weight: bold;
  `}
>
  {heading}
</span>
```

8. Now, style the close button as follows:

```
{closable && (
  <button
    ...
    css={css`
```

```
        border: none;
        background: transparent;
        margin-left: auto;
        cursor: pointer;
      `}
    >
      ...
    </button>
  )}
```

9. Finally, style the message container as follows:

```
<div
  css={css`
    margin-left: 30px;
    color: #000;
  `}
>
  {children}
</div>
```

10. Run the app by running `npm start` in the terminal. The alert component will appear like it was before.

11. Inspect the elements in the DOM using the browser's DevTools. The alert component uses scoped CSS class names, similar to CSS modules:

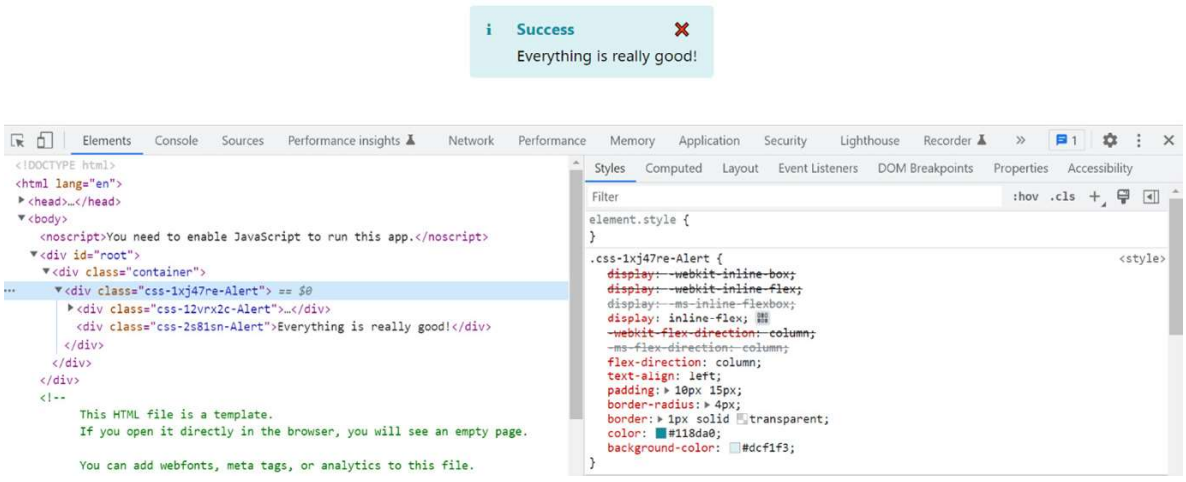

Figure 5.7 – Emotion's scoped class names

12. Stop the running app before continuing by pressing *Ctrl* + *C*.
13. To round off our understanding of Emotion, let's see what happens to the CSS in a production build. First, create a production build by executing `npm run build` in the terminal.

    After a few seconds, the build artifacts are created in the `build` folder.
14. Open the bundled CSS file from the `build/static/css` folder. Notice that the Emotion styles are not there. This is because Emotion generates the styles at runtime via JavaScript rather than at build time. If you think about it, the styles can't be generated at build time because they may depend on JavaScript variables whose values are only known at runtime.

This completes the refactoring of the alert component to use CSS-in-JS.

> **Note**
> For more information on emotion, visit their website at `https://emotion.sh/docs/introduction`.

Here's a recap of what we learned about Emotion and CSS-in-JS:

- Styles for a CSS-in-JS library are defined in JavaScript rather than a CSS file.
- Emotion's styles can be defined directly on a JSX element using a `css` attribute.
- A huge benefit is that conditional logic can be added directly to the styles, which helps us style interactive components more quickly.
- Emotion styles are applied at runtime rather than at build time because they depend on JavaScript variables. While this allows conditional styling logic to be elegantly defined, it does mean a small performance penalty because the styles are created and applied at runtime.

Next, we will learn about another different approach to styling React frontends.

Using Tailwind CSS

In this section, we will start by understanding Tailwind CSS and its benefits. We will then refactor the alert component we have been using to use Tailwind and observe how it differs from other approaches we have tried.

Understanding Tailwind CSS

Tailwind is a set of prebuilt CSS classes that can be used to style an app. It is referred to as a **utility-first CSS framework** because the prebuilt classes can be thought of as flexible utilities.

An example CSS class is `bg-white`, which styles the background of an element white – *bg* is short for *background*. Another example is `bg-orange-500`, which sets the background color to a 500 shade of orange. Tailwind contains a nice color palette that can be customized.

The utility classes can be used together to style an element. The following example styles a button element in JSX:

```
<button className="border-none rounded-md bg-emerald-700 text-
white cursor-pointer">
  ...
</button>
```

Here's an explanation of the classes used in the preceding example:

- `border-none` removes the border of an element.
- `rounded-md` rounds the corners of an element border. The *md* stands for *medium*. Alternatively, `lg` (large) could have been used or even `full`, for more rounded borders.
- `bg-emerald-700` sets the element background color to a 700 shade of emerald.
- `text-white` sets the element text color to white.
- `cursor-pointer` sets the element cursor to a pointer.

The utility classes are low-level and focused on styling a very specific thing. This makes the classes flexible, allowing them to be highly reusable.

Tailwind can specify that a class should be applied when the element is in a hover state by prefixing it with `hover:`. The following example sets the button background to a darker shade of emerald when hovered:

```
<button className="md border-none rounded-md bg-emerald-700
text-white cursor-pointer hover:bg-emerald-800">
  ...
</button>
```

So, a key point of Tailwind is that we don't write new CSS classes for each element we want to style – instead, we use a large range of well-thought-through existing classes. A benefit of this approach is that it helps an app look nice and consistent.

> **Note**
>
> For more information on Tailwind, refer to their website at the following link: `https://tailwindcss.com/`. The Tailwind website is a crucial resource for searching and understanding all the different utility classes that are available.

Next, we will install and configure Tailwind in the project containing the alert component we have been working on.

Installing and configuring Tailwind CSS

Now that we understand Tailwind, let's install and configure it in the alert component project. To do this, carry out the following steps:

1. In the Visual Studio project, start by installing Tailwind by running the following command in a terminal:

   ```
   npm i -D tailwindcss
   ```

   The Tailwind library is installed as a development dependency because it's not required at runtime.

2. Tailwind integrates into Create React App projects using a library called **PostCSS**. PostCSS is a tool that transforms CSS using JavaScript and Tailwind runs as a plugin in it. Install PostCSS by running the following command in the terminal:

   ```
   npm i -D postcss
   ```

3. Tailwind also recommends another PostCSS called **Autoprefixer**, which adds vendor prefixes to CSS. Install this by running the following command in the terminal:

   ```
   npm i -D autoprefixer
   ```

4. Next, run the following command in a terminal to generate configuration files for Tailwind and PostCSS:

   ```
   npx tailwindcss init -p
   ```

   After a few seconds, the two configuration files are created. The Tailwind configuration file is called `tailwind.config.js`, and the PostCSS configuration file is called `postcss.config.js`.

5. Open `tailwind.config.js` and specify the path to the React components as follows:

   ```
   module.exports = {
     content: [
       './src/**/*.{js,jsx,ts,tsx}'
   ```

```
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

6. Now, open `index.css` in the `src` folder and add the following three lines at the top of the file:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

These are called **directives** and will generate the CSS required by Tailwind during the build process.

Tailwind is now installed and ready to use.

Next, we will use Tailwind to style the alert component we have been working on.

Using Tailwind CSS

Now, let's use Tailwind to style the alert component. We will remove emotion's `css` JSX attribute and replace it with Tailwind utility class names in the JSX `className` attribute. To do this, carry out the following steps:

1. Open `Alert.tsx` and start by removing the special emotion comment and the `css` import statement from the top of the file.
2. Replace the `css` attribute with a `className` attribute on the outermost `div` element as follows:

```
<div
  className={`inline-flex flex-col text-left px-4 py-3
    rounded-md border-1 border-transparent`}
>
  ...
</div>
```

Here is an explanation of the utility classes that were just used:

- `inline-flex` and `flex-col` create an inline flexbox that flows vertically
- `text-left` aligns items to the left
- `px-4` adds 4 spacing units of left and right padding
- `py-3` adds 3 spacing units of top and bottom padding

- We have encountered `rounded-md` before – this rounds the corners of the `div` element
- `border-1` and `border-transparent` add a transparent 1 px border

> **Note**
> Spacing units are defined in Tailwind and are a proportional scale. One spacing unit is equal to `0.25rem`, which translates roughly to `4px`.

3. Still on the outermost `div` element, add the following conditional styles using string interpolation:

```
<div
  className={`inline-flex flex-col text-left px-4 py-3
rounded-md border-1 border-transparent ${
    type === 'warning' ? 'text-amber-900' : 'text-
      teal-900'
  } ${type === 'warning' ? 'bg-amber-50' : 'bg-teal-
    50'}`}
>
  ...
</div>
```

The text color is set to a 900 amber shade for warning alerts and a 900 teal shade for information alerts. The background color is set to a 50 amber shade for warning alerts and a 50 teal shade for information alerts.

4. Next, replace the `css` attribute with a `className` attribute on the header container as follows:

```
<div className="flex items-center mb-1">
  <span role="img" ... > ... </span>
  <span ... >{heading}</span>
  {closable && ...}
</div>
```

Here is an explanation of the utility classes that were just used:

- `flex` and `items-center` create a horizontal flowing flexbox where the items are centered vertically
- `mb-1` adds a 1 spacing unit margin at the bottom of the element

5. Replace the `css` attribute with a `className` attribute on the icon as follows:

```
<span role="img" ... className="w-7">
  {type === 'warning' ? '⚠' : 'i'}
```

```
  </span>
```

`w-7` sets the element to a width of 7 spacing units.

6. Replace the `css` attribute with a `className` attribute on the heading as follows:

```
<span className="font-bold">{heading}</span>
```

`font-bold` sets the font weight to be bold on the element.

7. Replace the `css` attribute with a `className` attribute on the close button as follows:

```
{closable && (
  <button
    ...
    className="border-none bg-transparent ml-auto cursor-
      pointer"
  >
    ...
  </button>
)}
```

Here, `border-none` removes the element border, and `bg-transparent` makes the element background transparent. `ml-auto` sets the left margin to automatic, which right aligns the element. `cursor-pointer` sets the mouse cursor to a pointer.

8. Finally, replace the `css` attribute with a `className` attribute on the message container as follows:

```
<div className="ml-7 text-black">
  {children}
</div>
```

`ml-7` sets the left margin on the element to 7 spacing units and `text-black` sets the text color to black.

9. Run the app by running `npm start` in the terminal. After a few seconds, the app will appear in the browser.

   Notice that the alert component looks a bit nicer because of Tailwind's default color palette and consistent spacing.

10. Inspect the elements in the DOM using the browser's DevTools. Notice the Tailwind utility classes being used and notice the spacing units use CSS rem units.

    A key point to notice is that no CSS class name scoping occurs. There is no need for any scoping because the classes are general and reusable and not specific to any element.

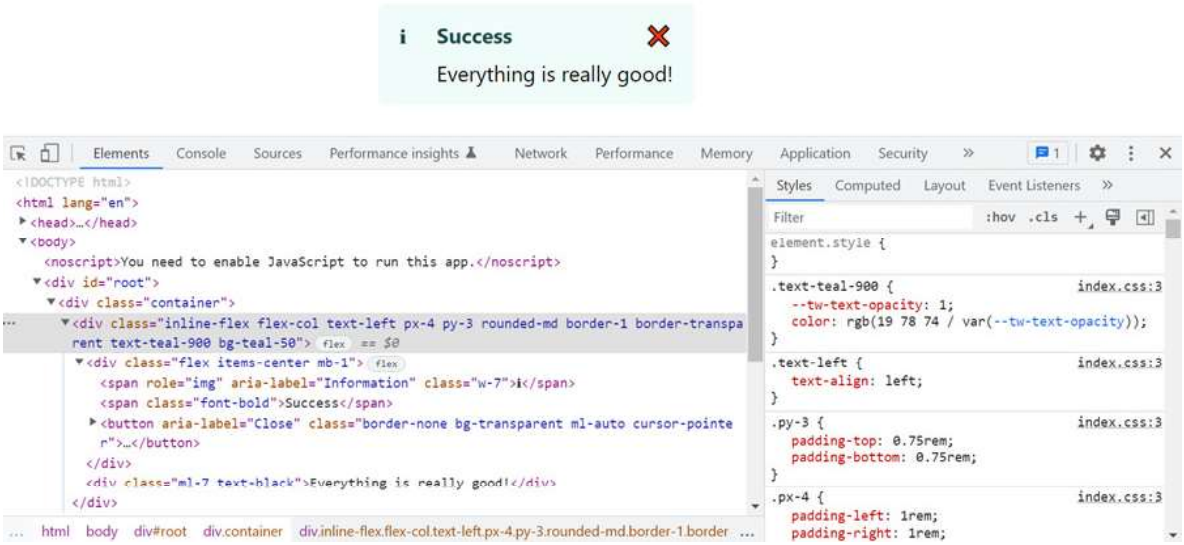

Figure 5.8 – A styled alert using Tailwind

11. Stop the running app before continuing by pressing *Ctrl* + *C*.
12. To round off our understanding of Tailwind, let's see what happens to the CSS in a production build. First, create a production build by executing `npm run build` in the terminal.

    After a few seconds, the build artifacts are created in the `build` folder.
13. Open the bundled CSS file from the `build/static/css` folder. Notice the base Tailwind styles at the start of the file. You will also see that all the Tailwind classes that we used are in this file.

Figure 5.9 – Tailwind CSS classes in a bundled CSS file

> **Note**
>
> An important point is that Tailwind doesn't add all its CSS classes – that would produce a massive CSS file! Instead, it only adds the CSS classes used in the app.

That completes the process of refactoring the alert component to use Tailwind.

Here's a recap of what we learned about Tailwind:

- Tailwind is a well-thought-through collection of reusable CSS classes that can be applied to React elements
- Tailwind has a nice default color palette and a 4 px spacing scale, both of which can be customized
- Tailwind is a plugin for PostCSS and executed at build time
- Tailwind does not incur a runtime performance penalty like Emotion, because the styles aren't created and applied at runtime
- Only classes used on React elements are included in the CSS build bundle

Next, we will make the icons in the alert component look a bit nicer.

Using SVGs

In this section, we will learn how to use SVG files in React and how to use them for the icons in the alert component.

Understanding how to use SVGs in React

SVG stands for **Scalable Vector Graphics** and it is made up of points, lines, curves, and shapes based on mathematical formulas rather than specific pixels. This allows them to scale when resized without distortion. The quality of icons is important to get right – if they are distorted, they make the whole app feel unprofessional. Using SVGs for icons is common in modern web development.

Create React App configures webpack to use SVG files when a project is created. In fact, `logo.svg` is referenced in the template `App` component as follows:

```
import logo from './logo.svg';
...
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        ...
      </header>
    </div>
  );
}
export default App;
```

In the preceding example, `logo` is imported as a path to the SVG file, which is then used on the `src` attribute on the `img` element to display the SVG.

An alternate way of referencing SVGs is to reference them as a component as follows:

```
import { ReactComponent as Logo } from './logo.svg';

function SomeComponent() {
  return (
    <div>
      <Logo />
    </div>
  );
}
```

SVG React components are available in a named import called `ReactComponent`. In the preceding example, the SVG component is aliased with the name `Logo`, which is then used in the JSX.

Next, we will learn how to use SVGs in the alert component.

Adding SVGs to the alert component

Carry out the following steps to replace the emoji icons in the alert component with SVGs:

1. First, create three files called `cross.svg`, `info.svg`, and `warning.svg` in the `src` folder. Then, copy and paste the content of these from the GitHub repository at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter5/Section5-Using-SVGs/app/src`.
2. Open `Alert.tsx` and add the following import statements to import the SVGs as React components:

   ```
   import { ReactComponent as CrossIcon } from './cross.
   svg';
   import { ReactComponent as InfoIcon } from './info.svg';
   import { ReactComponent as WarningIcon } from './warning.
   svg';
   ```

   We have given the SVG components appropriately named aliases.
3. Update the `span` element containing the emoji icons to use SVG icon components as follows:

   ```
   <span
     role="img"
   ```

```
    aria-label={type === 'warning' ? 'Warning' :
      'Information'}
    className="inline-block w-7"
  >
    {type === 'warning ' ? (
      <WarningIcon className="fill-amber-900 w-5 h-5" />
    ) : (
      <InfoIcon className="fill-teal-900 w-5 h-5" />
    )}
  </span>;
```

We have used Tailwind to size and color the icons appropriately.

4. Next, update the emoji close icon to the SVG close icon as follows:

```
<button
  aria-label="Close"
  onClick={handleCloseClick}
  className="border-none bg-transparent ml-auto cursor-
    pointer"
>
  <CrossIcon />
</button>
```

5. Run the app by running `npm start` in the terminal. After a few seconds, the app will appear in a browser containing the improved alert component:

Figure 5.10 – An alert with an SVG icon

That completes the alert component – it is looking much better now.

Here's a quick recap of what we learned about using SVGs in React apps:

- Webpack needs to be configured to bundle SVG files and Create React App does this configuration for us
- The default import for an SVG file is the path to the SVG, which can then be used in an `img` element

- A named import called `ReactComponent` can be used to reference the SVG as a React component in JSX

Next, we will summarize what we have learned in this chapter.

Summary

In this chapter, we learned about four methods of styling.

First, we learned that plain CSS could be used to style React apps, but all the styles in the imported CSS file are bundled regardless of whether a style is used. Also, the styles are not scoped to a specific component – we observed the `container` CSS class names clashing with the `App` and `Alert` components.

Next, we learned about CSS modules, which allows us to write plain CSS files imported in a way that scopes styles to the component. We learned that CSS modules is an open source library pre-installed and pre-configured in projects created with Create React App. We saw how this resolved the CSS clashing problem but didn't remove redundant styles.

Then, we discussed CSS-in-JS libraries, which allow styles to be defined directly on the React component. We used emotion's `css` prop to style the alert component without an external CSS file. The nice thing about this approach is that conditional-style logic can be implemented more quickly. We learned that emotion's styles are scoped like CSS modules, but the scoping happens at runtime rather than at build time. We also understood that the small performance cost of this approach is because of the styles being created at runtime.

The fourth styling approach we looked at was using Tailwind CSS. We learned that Tailwind provides a set of reusable CSS classes that can be applied to React elements, including a nice default color palette and a 4 px spacing scale, both of which can be customized. We learned that only the used Tailwind classes are included in the production build.

Finally, we learned that Create React App configures webpack to enable the use of SVG files. SVGs can be referenced as a path in an `img` element or as a React component using a `ReactComponent` named `import`.

In the next chapter, we will look at implementing multiple pages in React apps with a popular library called React Router.

Questions

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

1. Why could the following use of plain CSS be problematic?

```
<div className="wrapper"></div>
```

2. We have a component styled using CSS modules as follows:

```
import styles from './styles1.module.css';

function ComponentOne() {
  return <div className={styles.wrapper}></div>;
}
```

We have another component styled using CSS modules as follows:

```
import styles from './styles2.module.css';

function ComponentTwo() {
  return <div className={styles.wrapper}></div>;
}
```

Will the styles of these `div` elements clash, given that they are using the `wrapper` class name?

3. We have a component styled using CSS modules as follows:

```
import styles from './styles3.module.css';

function ComponentThree() {
  return <div className={styles.wrapper}>
</div>
}
```

The styles in `styles3.module.css` are as follows:

```
.wrap {
  display: flex;
  align-items: center;
  background: #e7650f;
}
```

The styles aren't being applied when the app is run. What is the problem?

4. We are defining a reusable button component with a `kind` prop that can be `"square"` or `"rounded"`. The rounded button should have a 04 px border radius, and the square button should have no border radius. How could we define this conditional style using Emotion's `css` prop?

5. We are styling a button element using Tailwind. It is currently styled as follows:

```
<button className="bg-blue-500 text-white font-bold py-2
px-4 rounded">
```

```
  Button
</button>
```

How can we enhance the style by making the button background a 700 shade of blue when the user hovers over it?

6. A logo SVG is referenced as follows:

```
import Logo from './logo.svg';

function LogoComponent() {
  return <Logo />;
}
```

However, the logo isn't rendered. What is the problem?

7. We are styling a button element using Tailwind that has a `color` prop to determine its color and is styled as follows:

```
<button className={`bg-${color}-500 text-white font-bold
py-2 px-4 rounded`}>
  Button
</button>
```

However, the button color doesn't work. What is the problem?

Answers

1. The wrapper CSS class could clash with other classes. To reduce this risk, the class name can be manually scoped to the component:

```
<div className="card-wrapper"></div>
```

2. The CSS won't clash because CSS modules will scope the class names to each component.
3. The wrong class name is referenced in the component – it should be `wrap` rather than `wrapper`:

```
import styles from './styles3.module.css';

function ComponentThree() {
  return <div className={styles.wrap}>
</div>
}
```

4. The `css` prop on the button could be as follows:

```
<button
  css={css`
    border-radius: ${kind === "rounded" ? "4px" : "0px"};
  `}
>
  ...
</button>
```

5. The style can be adjusted as follows to include the hover style:

```
<button className="bg-blue-500 hover:bg-blue-700 text-
white font-bold py-2 px-4 rounded">
  ...
</button
```

6. `Logo` will hold the path to the SVG rather than a component. The import statement can be adjusted as follows to import a logo component:

```
import { ReactComponent as Logo } from './logo.svg';

function LogoComponent() {
  return <Logo />;
}
```

7. The `bg-${color}-500` class name is problematic because this can only be resolved at runtime because of the `color` variable. The used Tailwind classes are determined at build time and added to the bundle, meaning the relevant background color classes won't be bundled. This means that the background color style won't be applied to the button.

6
Routing with React Router

In this chapter, we will build a simple app implementing the following pages:

- A home page that welcomes the user
- A products list page that lists all the products
- A product page that provides details about a particular product
- An admin page for privileged users

This will all be managed using a library called **React Router**.

Through this, we will learn how to implement static links from the products list to the product page and implement route parameters on the product page for the product ID. We will also learn about form navigation and query parameters when it comes to the search feature of our app.

Finally, the chapter will end with how to lazily load code for a page to improve performance.

So, in this chapter, we will cover the following topics:

- Introducing React Router
- Declaring routes
- Creating navigation
- Using nested routes
- Using route parameters
- Creating an error page
- Using index routes
- Using search parameters
- Navigating programmatically

- Using form navigation
- Implementing lazy loading

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/Chapter6`.

Introducing React Router

In this section, we start by creating a new React project for the app before understanding what React Router is and how to install it.

Creating the project

We will develop the app locally using Visual Studio Code, which requires a new Create React App-based project setup. We have covered this several times, so we will not cover the steps in this chapter – instead, see *Chapter 3*, *Setting Up React and TypeScript*. Create the project for the app with a name of your choice.

We will style the app with Tailwind CSS. We 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 for the app, install and configure Tailwind.

Understanding React Router

As the name suggests, React Router is a routing library for React apps. A router is responsible for selecting what to show in the app for a requested path. For example, React Router is responsible for determining what components to render when a path of `/products/6` is requested. For any app containing multiple pages, a router is essential, and React Router has been a popular router library for React for many years.

Installing React Router

React Router is in a package called `react-router-dom`. Install this in the project using the following command in the terminal:

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

TypeScript types are included in `react-router-dom`, so there is no need for a separate installation.

Next, we will create a page in the app and declare a route that shows it.

Declaring routes

We will start this section by creating a page component that lists the app's products. We will then learn how to create a router and declare routes using React Router's `createBrowserRouter` function.

Creating the products list page

The products list page component will contain the list of the React tools in the app. Carry out the following steps to create this:

1. We will start by creating the data source for the page. First, create a folder called `data` in the `src` folder and then a file called `products.ts` within `data`.
2. Add the following content into `products.ts` (you can copy and paste it from the GitHub repository at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter6/src/data/products.ts`):

```
export type Product = {
  id: number,
  name: string,
  description: string,
  price: number,
};

export const products: Product[] = [
  {
    description:
      'A collection of navigational components that
        compose declaratively with your app',
    id: 1,
    name: 'React Router',
    price: 8,
```

```
  },
  {
    description: 'A library that helps manage state
      across your app',
    id: 2,
    name: 'React Redux',
    price: 12,
  },
  {
    description: 'A library that helps you implement
      robust forms',
    id: 3,
    name: 'React Hook Form',
    price: 9,
  },
  {
    description: 'A library that helps you interact with
      a REST API',
    id: 4,
    name: 'React Apollo',
    price: 10,
  },
  {
    description: 'A library that provides utility CSS
      classes',
    id: 5,
    name: 'Tailwind CSS',
    price: 7,
  },
];
```

This is a list of all the React tools in the app held in a JavaScript array.

Note

Usually, this kind of data is on a server somewhere, but this adds complexity beyond the scope of this chapter. We cover how to interact with server data in detail in *Chapter 9*, *Interacting with RESTful APIs*, including how to do this efficiently with React Router.

3. We will create the products list page component now. First, create a folder for all the page components in the `src` folder called `pages`. Next, create a file called `ProductsPage.tsx` in the `pages` folder for the products list page component.
4. Add the following `import` statement into `ProductsPage.tsx` to import the products we just created:

```
import { products } from '../data/products';
```

5. Next, start to create the `ProductsPage` component by outputting a heading for the page:

```
export function ProductsPage() {
  return (
    <div className="text-center p-5">
      <h2 className="text-xl font-bold text-slate-600">
        Here are some great tools for React
      </h2>
    </div>
  );
}
```

This uses Tailwind classes to make the heading large, bold, gray, and horizontally centered.

6. Next, add the list of the products in the JSX:

```
<div className="text-center p-5 text-xl">
  <h2 className="text-base text-slate-600">
    Here are some great tools for React
  </h2>
  <ul className="list-none m-0 p-0">
    {products.map((product) => (
      <li key={product.id} className="p-1 text-base text-
        slate-800">
        {product.name}
      </li>
    ))}
  </ul>
</div>
```

The Tailwind classes remove the bullet points, margin, and padding from the unordered list element, and make the list items gray.

Notice that we use the products array `map` function to iterate over each product and return a `li` element. Using `Array.map` is common practice for JSX looping logic.

Notice the `key` prop on the list item elements. React requires this on elements in a loop to update the corresponding DOM elements efficiently. The value of the `key` prop must be unique and stable within the array, so we have used the product ID.

That completes the product page for now. This page won't show in the app yet because it isn't part of its component tree – we need to declare it as a page using React Router, which we'll do next.

Understanding React Router's router

A router in React Router is a component that tracks the browser's URL and performs navigation. Several routers are available in React Router, and the one recommended for web applications is called a **browser router**. As its name suggests, the `createBrowserRouter` function creates a browser router.

`createBrowserRouter` requires an argument containing all the **routes** in the application. A route contains a path and what component to render when the app's browser address matches that path. The following code snippet creates a router with two routes:

```
const router = createBrowserRouter([
  {
    path: 'some-page',
    element: <SomePage />,
  },
  {
    path: 'another-page',
    element: <AnotherPage />,
  }
]);
```

When the path is `/some-page`, the `SomePage` component will be rendered. When the path is `/another-page`, the `AnotherPage` component will be rendered.

The router returned by `createBrowserRouter` is passed to a `RouterProvider` component and placed high up in the React component tree, as shown here:

```
const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
```

```
    <RouterProvider router={router} />
  </React.StrictMode>
);
```

Now that we are starting to understand React Router's router, we will use it in our project.

Declaring the products route

We will declare the products list page in the app using `createBrowserRouter` and `RouterProvider`. Carry out the following steps:

1. We will create our own component to hold all the route definitions. Create a file called `Routes.tsx` in the `src` folder containing the following `import` statements:

   ```
   import {
     createBrowserRouter,
     RouterProvider,
   } from 'react-router-dom';
   import { ProductsPage } from './pages/ProductsPage';
   ```

   We have imported `createBrowserRouter` and `RouterProvider` from React Router. We have also imported `ProductsPage`, which we'll render in a `products` route next.

2. Add the following component underneath the `import` statements to define the router with a `products` route:

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

   So, the `ProductsPage` component will be rendered when the path is `/products`.

3. Still in `Routes.tsx`, create a component called `Routes` under the router as follows:

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

   This component wraps `RouterProvider` with the router passed into it.

4. Open the `index.tsx` file and add an `import` statement for the `Routes` component we just created beneath the other `import` statements:

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

5. Render `Routes` instead of `App` as the top-level component as follows:

```
root.render(
  <React.StrictMode>
    <Routes />
  </React.StrictMode>
);
```

This causes the `products` route we defined to be part of the component tree. This means the products list page will be rendered in the app when the path is `/products`.

6. Remove the `import` statement for the `App` component, as this is not needed at the moment.
7. Run the app using `npm start`.

An error screen appears explaining that the current route isn't found:

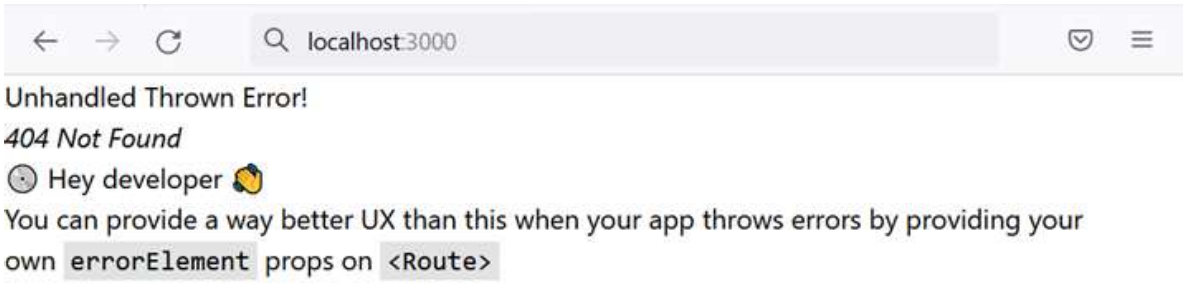

Figure 6.1 – React Router's standard error page

The error page is from React Router. As the error message suggests, we can provide our own error screen, which we will do later in this chapter.

8. Change the browser URL to `http://localhost:3000/products`.

You will see the products list page component rendered as follows:

Figure 6.2 – Products list page

This confirms that the `products` route is working nicely. Keep the app running as we recap and move to the next section.

Here's a recap of what we have learned in this section:

- In a web app, routes in React Router are defined using `createBrowserRouter`
- Each route has a path and a component to render when the browser's URL matches that path
- The router returned from `createBrowserRouter` is passed into a `RouterProvider` component, which should be placed high up in the component tree

For more information on `createBrowserRouter`, see the following link in the React Router documentation: `https://reactrouter.com/en/main/routers/create-browser-router`. For more information on `RouterProvider`, see the following link in the React Router documentation: `https://reactrouter.com/en/main/routers/router-provider`.

Next, we will learn about React Router components that can perform navigation.

Creating navigation

React Router comes with components called `Link` and `NavLink`, which provide navigation. In this section, we will create a navigation bar at the top of the app containing the `Link` component from React Router. We will then swap `Link` for the `NavLink` component and understand the difference between the two components.

Using the Link component

Carry out the following steps to create an app header containing React Router's `Link` component:

1. Start by creating a file for the app header called `Header.tsx` in the `src` folder containing the following `import` statements:

```
import { Link } from 'react-router-dom';
import logo from './logo.svg';
```

We have imported the `Link` component from React Router.

We have also imported the React logo because we will include this in the app header with the navigation options.

2. Create the `Header` component as follows:

```
export function Header() {
  return (
    <header className="text-center text-slate-50
      bg-slate-900 h-40 p-5">
      <img
        src={logo}
        alt="Logo"
        className="inline-block h-20"
      />
      <h1 className="text-2xl">React Tools</h1>
      <nav></nav>
    </header>
  );
}
```

The component contains a `header` element containing the React logo, the app title, and an empty `nav` element. We have used Tailwind classes to make the header gray with the logo and title horizontally centered.

3. Now, create a link inside the `nav` element:

```
<nav>
  <Link
    to="products"
    className="text-white no-underline p-1"
  >
```

```
      Products
    </Link>
  </nav>
```

The `Link` component has a `to` prop that defines the path to navigate to. The text to display can be specified in the `Link` content.

4. Open `Routes.tsx` and add an import statement for the `Header` component we just created:

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

5. In the `router` definition, add a root path that renders the `Header` component as follows:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Header />,
  },
  {
    path: 'products',
    element: <ProductsPage />,
  },
]);
```

What we have just done isn't ideal because the `Header` component needs to show on all routes and not just the root route. However, it will allow us to explore React Router's `Link` component. We will tidy this up in the *Using nested routes* section.

6. In the running app, change the browser address to the root of the app. The new app header appears, containing the **Products** link:

Figure 6.3 – App header

7. Now, inspect the app header elements using the browser's DevTools:

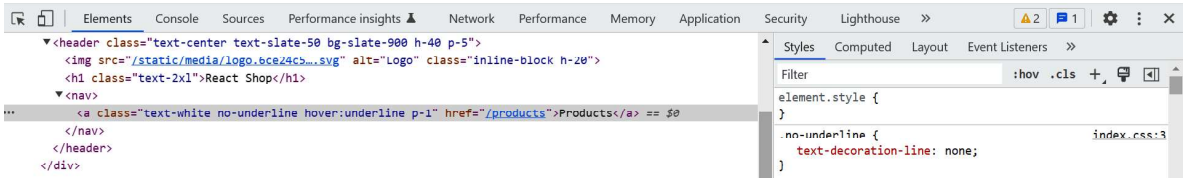

Figure 6.4 – Header component inspection

We can see that the `Link` component is rendered as an HTML anchor element.

8. Select the **Network** tab in DevTools and clear any existing requests that are shown. Click on the **Products** link in the app header. The browser will navigate to the products list page.

   Notice that a network request wasn't made for the products list page. This is because React Router overrides the anchor element's default behavior with client-side navigation:

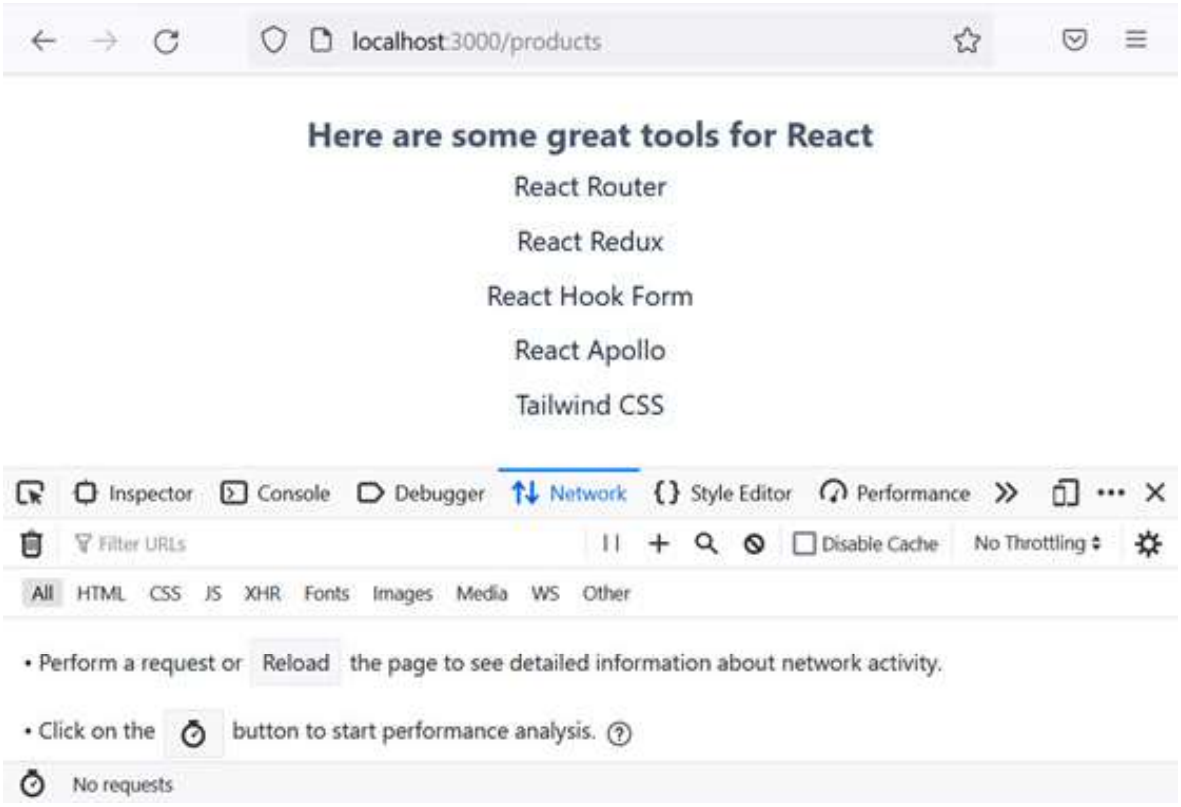

Figure 6.5 – Client-side navigation

Finally, notice that the app header disappears on the products list page, which is not what we want. We will resolve this in the *Using nested routes* section.

Keep the app running as we move to the next section.

The navigation is working well but it would be nice if the **Products** link had a different style when the products list page was active. We will make this improvement next.

Using the NavLink component

React Router's `NavLink` is like a `Link` element but allows it to be styled differently when active. This is really handy for a navigation bar.

Carry out the following steps to replace `Link` with `NavLink` in the app header:

1. Open `Header.tsx` and change the `Link` references to `NavLink`:

```
import { NavLink } from 'react-router-dom';
...
export function Header() {
  return (
    <header ...>
      ...
      <nav>
        <NavLink
          to="products"
          className="..."
        >
          Products
        </NavLink>
      </nav>
    </header>
  );
}
```

   The app header will look and behave exactly the same at the moment.

2. The `className` prop on the `NavLink` component accepts a function that can be used to conditionally style it, depending on whether its page is active. Update the `className` attribute to the following:

```
<NavLink
  to="products"
```

```
    className={({ isActive }) =>
      `text-white no-underline p-1 pb-0.5 border-solid
         border-b-2 ${
        isActive ? "border-white" : "border-transparent"
      }`
    }
  >
    Products
  </NavLink>
```

The function takes in a parameter, `isActive`, for defining whether the link's page is active. We've added a bottom border to the link if it is active.

We can't see the impact of this change just yet, because the **Products** link doesn't appear on the products list page yet. We will resolve this in the next section.

That completes the app header and our exploration of the `NavLink` component.

To recap, `NavLink` is great for main app navigation when we want to highlight an active link, and `Link` is great for all the other links in our app.

For more information on the `Link` component, see the following link: `https://reactrouter.com/en/main/components/link`. For more information on the `NavLink` component, see the following link: `https://reactrouter.com/en/main/components/nav-link`.

Next, we will learn about nested routes.

Using nested routes

In this section, we will cover **nested routes** and the situations in which they are useful, before using a nested route in our app. The nested route will also resolve the disappearing app header problem we experienced in the previous sections.

Understanding nested routes

A nested route allows a segment of a route to render a component. For example, the following mock-up is commonly implemented using nested routes:

Figure 6.6 – Use case for nested routes

The mock-up displays information about a customer. The path determines the active tab – in the mockup, **Profile** is the active tab because that is the last segment in the path. If the user selects the **History** tab, the path would change to `/customers/1234/history`.

A `Customer` component could render the shell of this screen, including the customer's name, picture, and tab headings. The component that renders the tab contents could be decoupled from the `Customer` component and coupled to the path instead.

This feature is referred to as *nested routes* because `Route` components are nested inside each other. Here's what the routes for the mock-up could be:

```
const router = createBrowserRouter([
  {
    path: 'customer/:id',
    element: <Customer />,
    children: [
      {
        path: 'profile',
        element: <CustomerProfile />,
      },
      {
        path: 'history',
        element: <CustomerHistory />,
      },
      {
```

```
        path: 'tasks',
        element: <CustomerTasks />,
      },
    ],
  },
]);
```

This nested approach to defining routes makes them easy to read and understand, as you can see in the preceding code snippet.

A critical part of nested routes is where child components are rendered in their parent. In the preceding code snippet, where would the `CustomerProfile` component be rendered in the `Customer` component? The solution is React Router's `Outlet` component. Here's an example of `Outlet` in the `Customer` component from the mock-up:

```
export function Customer() {
  ...
  return (
    <div>
      <Name ... />
      <Picture ... />
      <nav>
        <NavLink to="profile" ... >Profile</NavLink>
        <NavLink to="history" ... >History</NavLink>
        <NavLink to="tasks" ... >Tasks</NavLink>
      </nav>
      <Outlet />
    </div>
  );
}
```

So, in this example, the `CustomerProfile` component would be rendered after the navigation options in the `Customer` component. Notice that the `Customer` component is decoupled from the nested content. This means new tabs can be added to the customer page without changing the `Customer` component. This is another benefit of nested routes.

Next, we will use a nested route within our app.

Using nested routes in the app

In our app, we will use the `App` component for the app's shell, which renders the root path. We will then nest the products list page within this:

1. Open `App.tsx` and replace all the existing content with the following:

```
import { Outlet } from 'react-router-dom';
import { Header } from './Header';

export default function App() {
  return (
    <>
      <Header />
      <Outlet />
    </>
  );
}
```

   The component renders the app header with nested content underneath it.

> **Note**
>
> The empty JSX elements, `<>` and `</>`, are **React fragments**. React fragments are not added to the DOM and are used as a workaround to React components only being able to return a single element, so they are a way of returning multiple elements in a React component that keeps React happy.

2. Open `Routes.tsx`, import the `App` component we just modified, and remove the `import` component for `Header`:

```
import {
  createBrowserRouter,
  RouterProvider,
} from 'react-router-dom';
import { ProductsPage } from './pages/ProductsPage5';
import App from './App';
```

3. Update the `router` definition as follows:

```
const router = createBrowserRouter([
  {
```

```
      path: '/',
      element: <App />,
      children: [
        {
          path: 'products',
          element: <ProductsPage />,
        }
      ]
    }
]);
```

The products list page is now nested inside the `App` component.

4. If you return to the running app, you will see that the app header now appears on the products list page. You will also see the **Products** link underlined because it is an active link:

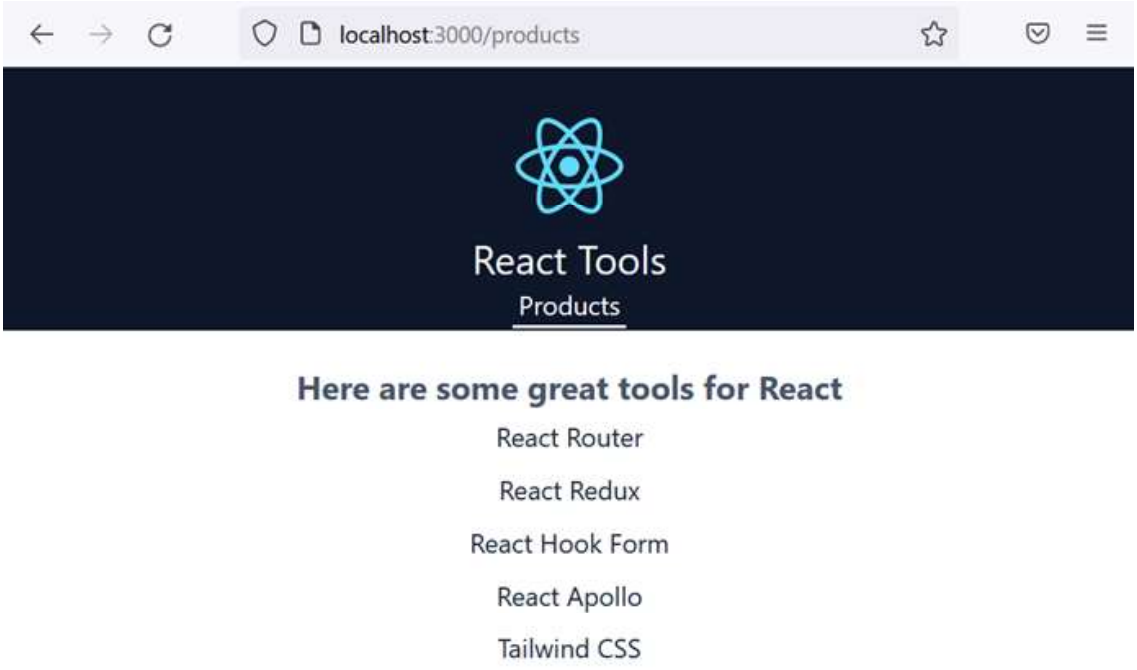

Figure 6.7 – App header on the products list page

To recap, nested routes allow components to be rendered for different path segments. An `Outlet` component is to render nested content within a parent component.

For more information on the `Outlet` component, see the following link: `https://reactrouter.com/en/main/components/outlet`.

Next, we will learn about route parameters.

Using route parameters

In this section, we will understand **route parameters** and how they are useful before using a route parameter in our app.

Understanding route parameters

A route parameter is a segment in the path that varies. The value of the variable segment is available to components so that they can render something conditionally.

In the following path, `1234` is the ID of a customer: `/customers/1234/`.

This can be defined as a route parameter in a route as follows:

```
{ path: '/customer/:id', element: <Customer /> }
```

A colon (`:`) followed by a name defines a route parameter. It is up to us to choose a parameter name that makes sense, so the `:id` segment in the path is the route parameter definition in the preceding route.

Multiple route parameters can be used in a path as follows:

```
{
  path: '/customer/:customerId/tasks/:taskId',
  element: <CustomerTask />,
}
```

Route parameter names obviously have to be unique within a path.

Route parameters are available to components using React Router's `useParams` hook. The following code snippet is an example of how the `customerId` and `taskId` route parameter values could be obtained:

```
const params = useParams<Params>();
console.log('Customer id', params.customerId);
console.log('Task id', params.taskId);
```

As we can see from the code snippet, `useParams` has a generic argument that defines the type for the parameters. The `type` definition for the preceding code snippet is as follows:

```
type Params = {
  customerId: string;
  taskId: string;
};
```

It is important to note that the route parameter values are always strings because they are extracted from paths, which are strings.

Now that we understand route parameters, we will use a route parameter in our app.

Using route parameters in the app

We will add a product page to our app to show the description and price of each product. The path to the page will have a route parameter for the product ID. Carry out the following steps to implement the product page:

1. We will start by creating the product page. In the `src/pages` folder, create a file called `ProductPage.tsx` with the following `import` statements:

   ```
   import { useParams } from 'react-router-dom';
   import { products } from '../data/products';
   ```

   We have imported the `useParams` hook from React Router, which will allow us to get the value of an `id` route parameter – the product's ID. We have also imported the `products` array.

2. Start creating the `ProductPage` component as follows:

   ```
   type Params = {
     id: string;
   };
   export function ProductPage() {
     const params = useParams<Params>();
     const id =
       params.id === undefined ? undefined :
         parseInt(params.id);
   }
   ```

   We use the `useParams` hook to obtain the `id` route parameter and turn it into an integer if it has a value.

3. Now, add a variable that is assigned to the product with the ID from the route parameter:

```
export function ProductPage() {
  const params = useParams<Params>();
  const id =
    params.id === undefined ? undefined :
      parseInt(params.id);
  const product = products.find(
    (product) => product.id === id
  );
}
```

4. Return the product information from the `product` variable in the JSX:

```
export function ProductPage() {
  ...
  return (
    <div className="text-center p-5 text-xl">
      {product === undefined ? (
        <h1 className="text-xl text-slate-900">
          Unknown product
        </h1>
      ) : (
        <>
          <h1 className="text-xl text-slate-900">
            {product.name}
          </h1>
          <p className="text-base text-slate-800">
            {product.description}
          </p>
          <p className="text-base text-slate-800">
            {new Intl.NumberFormat('en-US', {
              currency: 'USD',
              style: 'currency',
            }).format(product.price)}
          </p>
        </>
      )}
```

```
      </div>
    );
  }
```

`Unknown product` is returned if the product can't be found. Its name, description, and price are returned if the product is found. We use JavaScript's `Intl.NumberFormat` function to nicely format the price.

That completes the product page.

5. The next task is to add the route for the product page. Open `Routes.tsx` and add an `import` statement for the product page:

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

6. Add the following highlighted route for the product page:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <App />,
    children: [
      {
        path: 'products',
        element: <ProductsPage />,
      },
      {
        path: 'products/:id',
        element: <ProductPage />,
      }
    ]
  }
]);
```

So, the `/products/2` path should return a product page for React Redux.

7. In the running app, change the browser URL to `http://localhost:3000/products/2`. The React Redux product should show up:

Figure 6.8 – The product page

8. The last task in this section is to turn the products list on the products list page into links that open the relevant product page. Open `ProductsPage.tsx` and import the `Link` component from React Router:

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

9. Add a `Link` component around the product name in the JSX:

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

Link paths are relative to the component's path. Given that the component path is `/products`, we set the link path to the product ID, which should match the `product` route.

10. Return to the running app and go to the products list page. Hover over the products and you will now see that they are links:

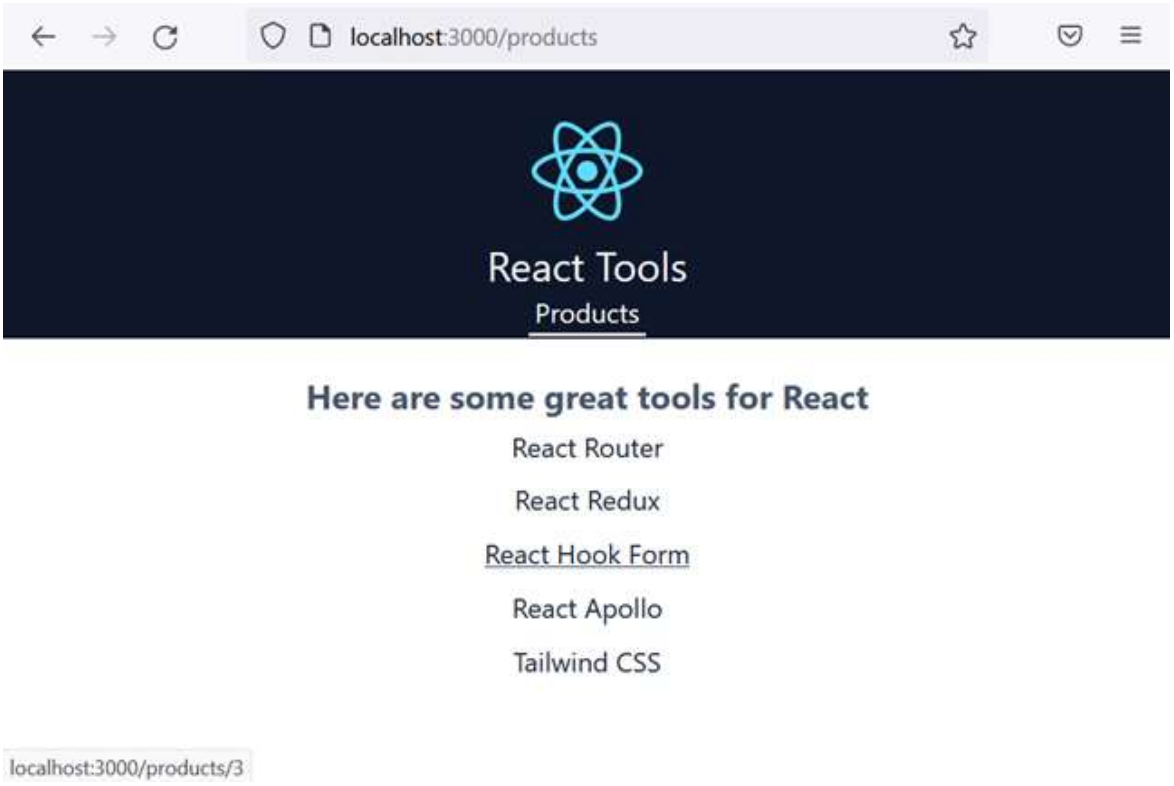

Figure 6.9 – Product list links

11. Click one of the products and the relevant product page will appear.

That completes this section on route parameters. Here's a quick recap:

- A route parameter is a varying segment in a path defined using a colon followed by the parameter name
- Route parameters can be accessed using React Router's `useParams` hook

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

Remember React Router's error page we experienced in the *Declaring routes* section? Next, we will learn how to customize that error page.