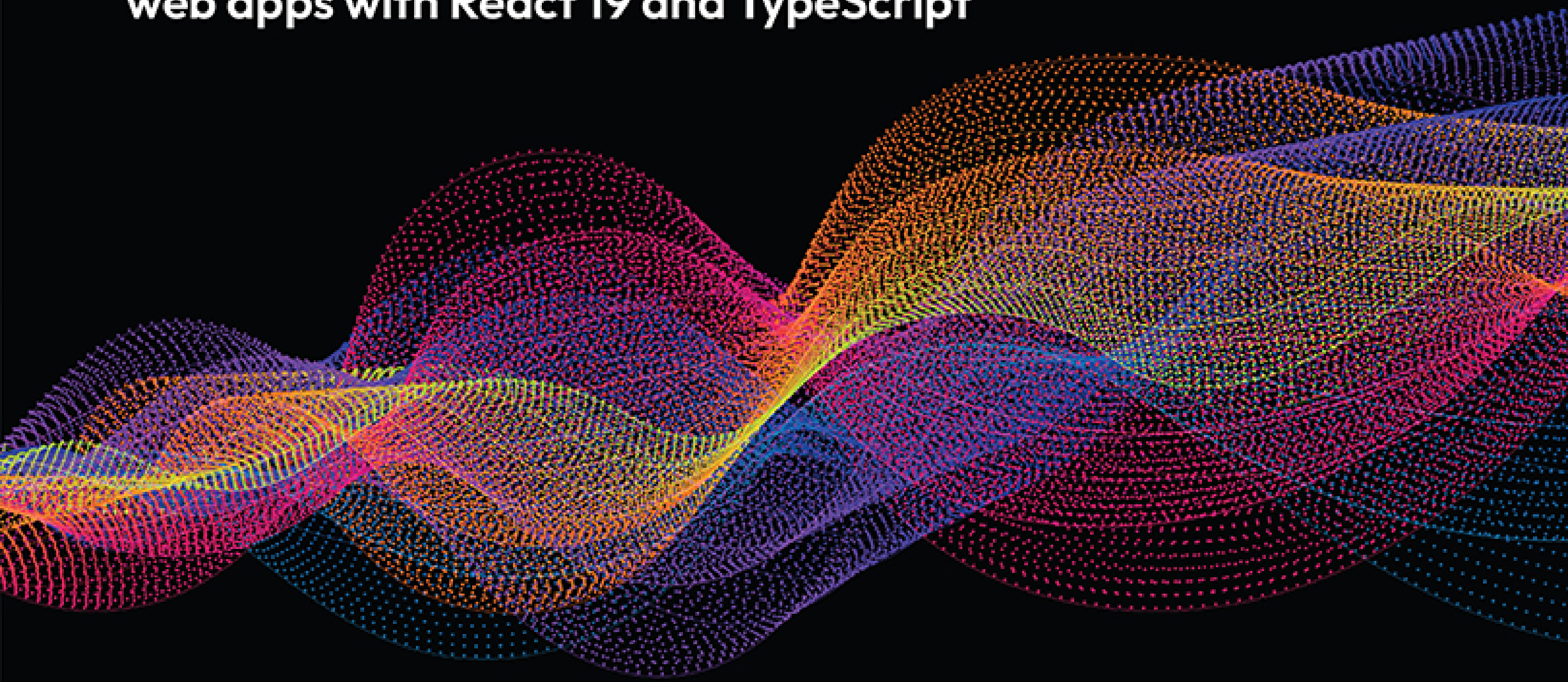
EXPERT INSIGHT

Learn React with TypeScript

A beginner's guide to building real-world, production-ready web apps with React 19 and TypeScript



Third Edition

Carl Rippon

<packt>

Learn React with TypeScript

Third Edition

Portfolio Director: Ashwin Nair

Relationship Lead: Bhavya Rao

Project Manager: Aparna Nair

Content Engineer: Adrija Mitra

Technical Editor: Sweety Pagaria

Copy Editor: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Shantanu Zagade

Growth Lead: Priyadarshini Sharma

First published: November 2018

Second edition: March 2023

Third edition: June 2025

Production reference: 1050625

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83664-317-3

www.packt.com

Contributors

About the author

Carl Rippon is a seasoned software developer with over 25 years of experience in building complex business applications across a range of industries. For the past 15 years, he has specialized in modern JavaScript technologies – particularly React, TypeScript, and Next.js. A passionate educator and writer, Carl has authored more than 100 blog posts, sharing practical insights and solutions with the developer community.

I'd like to thank Sarah, Ellie-Jayne, Lily-Rose, Fudge, and Arlo for all the support and encouragement they've given me while writing this book. A special thanks to everyone on the Packt editorial team for their hard work and great feedback, especially Adrija Mitra.

About the reviewers

Gurjit Singh is a Berlin-based senior frontend engineer at Storyblok, with over six years of experience in building modern web applications using React, TypeScript, and Node.js. Formerly employed at zendesk.com, he contributed at an organizational level across AI-powered and customer-facing initiatives. Gurjit is also an active open source contributor. His work has led to collaborations with engineers at Apple, Wix, and more, and he was invited to a hackathon in San Francisco, US, for his contributions to the Khalis Foundation. He enjoys speaking at conferences and sharing practical engineering insights with the developer community. In his free time, he's passionate about Indian classical music, reading psychology books, and traveling the globe.

Andrew Baisden is an experienced software developer skilled in the JavaScript and Python ecosystems. He builds cross-platform applications using frontend technologies such as React, TypeScript, and modern frameworks. Experienced with backend and mobile development, Andrew is also passionate about sharing knowledge and writes technical articles for various publications. He also engages with his social media audience by offering valuable resources and content. Andrew combines education with constant self-improvement to stay current with technology.

The author acknowledges the use of cutting-edge AI, in this case ChatGPT, with the sole aim of enhancing the language and clarity within the book, thereby ensuring a smooth reading experience for readers. It's important to note that the content itself has been crafted by the author and edited by a professional publishing team.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://packt.link/GxSkC

Table of Contents

Preface

React was built by Meta to provide more structure to its code base and allow it to scale much better. React worked so well for Facebook that they eventually made it open source. Today, React is the dominant technology for building frontends; it allows us to build small, isolated, and highly reusable components that can be composed together to create complex frontends. With advancements such as React Server Components, React has further expanded its capabilities, enabling developers to seamlessly combine server-side rendering and client-side interactivity for highly optimized and dynamic applications.

TypeScript was built by Microsoft to help developers more easily develop large JavaScript-based programs. It is a superset of JavaScript that brings a rich type system to it. This type system helps developers catch bugs early and allows tools to be created to navigate and refactor code robustly.

This book will teach you how to use both of these technologies to create large, sophisticated frontends that are easy to maintain, while also exploring modern features such as React Server Components to enhance performance and productivity.

Who this book is for

If you are a developer who wants to create large and complex frontends with React and TypeScript, this book is for you. The book doesn't assume you have any previous knowledge of React or TypeScript – however, basic knowledge of JavaScript, HTML, and CSS will help you get to grips with the concepts covered.

What this book covers

Chapter 1, *Getting Started with React*, covers creating React projects and the fundamentals of building React components. This includes making a component configurable using props and interactive using state.

Chapter 2, *Getting Started with TypeScript*, starts with the fundamentals of TypeScript and its type system. This includes using inbuilt types as well as creating new types. The chapter then covers creating a React component with TypeScript types.

Chapter 3, *Using React Hooks*, details the common React Hooks and their typical use cases. The chapter also covers how to use the Hooks with TypeScript to make them type-safe.

Chapter 4, *Approaches to Styling React Frontends*, walks through how to style React components using several different approaches. The benefits of each approach are also explored.

Chapter 5, *Using React Server and Client Components*, covers how and when to use React Server Components and Client Components and also how to compose them together.

Chapter 6, *Creating a Multi-Page App with Next.js*, covers the fundamentals of building multi-page apps in a popular React framework called Next.js. This includes implementing different pages, links between them, and page parameters.

Chapter 7, *Server Component Data Fetching and Server Function Mutations*, demonstrates how React Server Components can fetch data from a database. The chapter also includes mutating database data using a React Server Function.

Chapter 8, *Client Component Data Fetching and Mutations with TanStack Query*, covers how React Client Components can fetch and mutate data from a database using a popular library called TanStack Query.

Chapter 9, *Working with Forms*, explores how forms can be implemented using several different approaches, including the latest React Hooks and a popular forms library.

Chapter 10, *State Management*, walks through how React state can be shared between different components. Several approaches are explored along with their benefits.

Chapter 11, *Reusable Components*, brings in several patterns for making React components highly reusable but still type-safe.

Chapter 12, *Unit Testing with Vitest and the React Testing Library*, first delves into how functions can be tested with Vitest. The chapter then moves on to how React components can be tested with the help of the React Testing Library.

To get the most out of this book

To follow along with this book, you'll need to have the following technologies installed on your Windows or macOS computer:

- A modern browser, such as Google Chrome, which you can download from https://www.google.com/chrome
- Node.js and npm, available at https://nodejs.org/en/download
- Visual Studio Code, downloadable from https://code.visualstudio.com

Software/hardware covered in the book
React 19 or later
Next.js 15 or later
TypeScript 5 or later

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "We used the **`Form`** component from Next.js to optimize the form submission performance."

A block of code is set as follows:

```
export default function Home() {
  return (
    <main>
    </main>
  );
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
import Form from ‘next/form’;
export function ContactForm() {
  return (
    <Form ... >
      ...
    </Form>
  );
}
```

Any command-line input or output is written as follows:

```
npm run dev
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “In the running app, try clicking the **Done** button to mark an item as done.”

> ***TIPS OR IMPORTANT NOTES***
>
> *Appear like this.*

Get in touch

Feedback from our readers is always welcome.

Share Your Thoughts

Once you've read *Learn React with TypeScript, Third Edition*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review pag for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a Free PDF Copy of This Book

Part 1: Introduction

This part will get you started with both React and TypeScript, learning how to create a new project and implement interactive type-safe components. We will also learn about React's common Hooks in detail and the cases in which they are used in applications.

This part has the following chapters:

1
Getting Started with React

Facebook has become an incredibly popular app. As its popularity has grown, so has the demand for new features. **React** is Meta's answer to helping more people work on the Facebook code base and deliver features more quickly. React has worked so well for Facebook that Meta eventually made it open source. Today, React is a mature library for building component-based frontends that is extremely popular and has a massive community and ecosystem.

TypeScript is also a popular, mature library maintained by another big company, Microsoft. It allows users to add a rich type system to their JavaScript code, helping them be more productive, particularly in large code bases.

This book will teach you how to use these awesome libraries to build robust frontends that are easy to maintain. The first two chapters in the book will introduce React and TypeScript separately. You'll then learn how to use React and TypeScript together to compose robust components with strong typing. There is a whole chapter on the recently released **React Server Components** (**RSCs**), which offer significant performance and productivity gains. The book covers all the key topics you'll need to build a web frontend, such as styling, forms, data fetching, and data mutation.

In this chapter, we will introduce React and understand its benefits. We will then build a simple React component, learning about the component syntax and how to make it configurable. After that, we will learn how to make a component interactive using component state and events. At the end of the chapter, we will learn how to use React's development tools.

By the end of this first chapter, you'll be able to create simple React components and will be ready to learn how to strongly type them with TypeScript.

In this chapter, we'll cover the following topics:

- Understanding the benefits of React
- Setting up a React project
- Understanding the structure of a React app
- Creating a component
- Using props
- Using state
- Using events
- Using React developer tools

Technical requirements

We use the following tools in this chapter:

- **Browser**: A modern browser such as Google Chrome.
- **Terminal**: We will use a terminal to execute commands to create a React project. The default terminal available in your operating system will work fine.
- **Visual Studio Code**: We need a code editor to create our first React component. Visual Studio Code is a popular editor that we'll use throughout this book. This can be downloaded and installed from https://code.visualstudio.com.
- **Node.js and npm**: Node.js will be required to build our React app and run it on a development server. npm is a package manager that allows us to easily install libraries into our app. These tools come together and can be downloaded and installed from https://nodejs.org/en/download.

All the code snippets in this chapter can be found online at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter01.

Understanding the benefits of React

Before we start creating our first React component, in this section, we will understand what React is and explore some of its benefits.

React is an incredibly popular frontend library. We have already mentioned that Meta uses React for Facebook, but many other famous companies use it too, such as Netflix, Uber, and Airbnb. React's popularity has resulted in a huge ecosystem surrounding it that includes great tools, popular libraries, and many experienced developers.

One reason for React's popularity is that it is simple. This is because it focuses on doing one thing very well – providing a powerful mechanism for building UI components. Components are pieces of the UI that can be composed together to create a frontend. Furthermore, components can be reusable so that they can be used on different screens or even in other apps.

React's narrow focus means it can be incorporated into an existing app, even if it uses a different framework. This is because it doesn't need to take over the whole app to run; it is happy to run as part of an app's frontend.

React components are displayed performantly using a virtual **Document Object Model** (**DOM**). You may be familiar with the real DOM – it provides the structure for a web page. However, changes to the real DOM can be costly, leading to performance problems in an interactive app. React solves this performance problem by using an in-memory representation of the real DOM called a **virtual DOM**. Before React changes the real DOM, it produces a new virtual DOM and compares it against the current virtual DOM to calculate the minimum amount of changes required to the real DOM. The real DOM is then updated with those minimum changes.

React's recent addition of Server Components enables very performant data fetching and reduces the amount of JavaScript sent from the server to the browser. Couple this with React server functions and you also have an extremely productive development experience.

React Native, a framework based on React, allows us to build cross-platform apps for iOS and Android, similar to how we use React to build web applications. The core React development skills are the same across both technologies, and code can be shared and reused as well.

The fact that Meta uses React for Facebook is a major benefit because it ensures that it is of the highest quality – React breaking Facebook is not good for Meta! It also means a lot of thought and care goes into ensuring new versions of React are cheap to adopt, which helps reduce the maintenance costs of an app.

React's simplicity means it is easy and quick to learn. There are many great learning resources, such as this book. There is also a range of tools that make it very easy to scaffold a React app – one such tool is called **Vite**, which we will learn about later in this chapter.

Now that we are starting to understand React, we will create our first React project in the next section.

Creating a React project

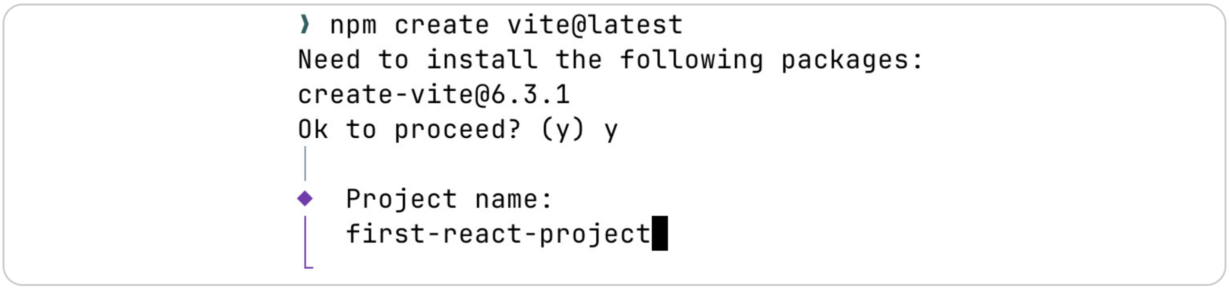
In this section, we will create a React project and configure Visual Studio Code to work optimally with it. We will also cover how to run a React app in development mode and also how to produce a production build.

We will create a React project using Vite, a popular build tool and development server for React apps. Carry out the following steps:

1. In a terminal, in a folder of your choice, execute the following command to instruct Vite to create a project:

   ```
   npm create vite@latest
   ```

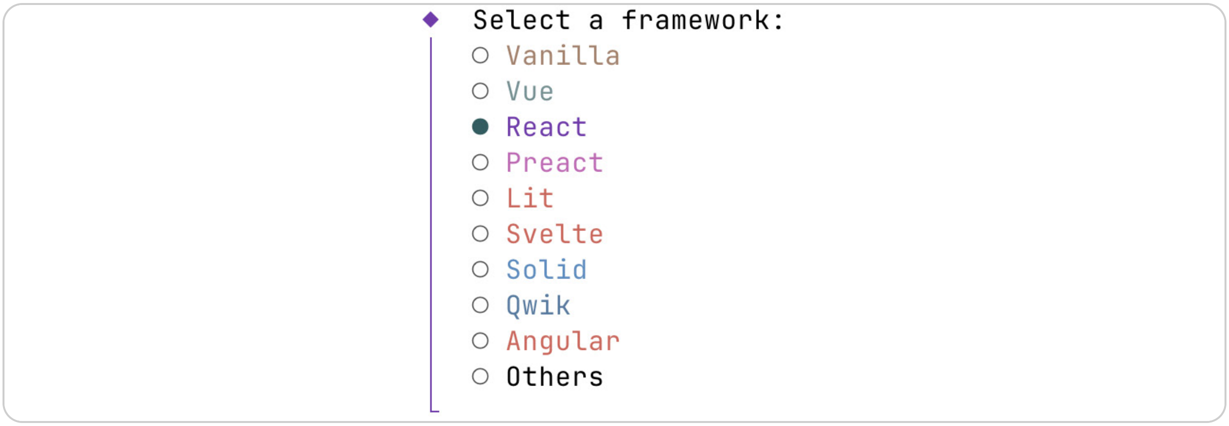
2. A prompt for the project name appears. The project name will be the folder name containing the project code. So, enter a name of your choice and press *Enter*.



```
❯ npm create vite@latest
Need to install the following packages:
create-vite@6.3.1
Ok to proceed? (y) y
│
◆  Project name:
│  first-react-project
└
```

Figure 1.1 – Creating a Vite project

3. A prompt now appears for the framework for the project. Select **React** by using the *down arrow* key to move to **React** and press *Enter*.



```
◆  Select a framework:
│  ○ Vanilla
│  ○ Vue
│  ● React
│  ○ Preact
│  ○ Lit
│  ○ Svelte
│  ○ Solid
│  ○ Qwik
│  ○ Angular
│  ○ Others
└
```

Figure 1.2 – Selecting the React framework

4. Lastly, the variant is prompted for. Select **JavaScript**, using the *down arrow* key, and press *Enter*. Note that we will explore the **TypeScript** option in the next chapter.
5. The project is created in the folder name you chose. The terminal lists the following three commands that it suggests should be run:
    - **`cd <your-project-name>`**: This will change the working directory to the one just created

    - **`npm install`**: This will install npm packages that the initial project depends on
    - **`npm run dev`**: This will run the app in development mode

6. Run the first two suggested commands in the terminal so that the project is the working directory and the project dependencies are installed.

Next, we are going to inspect the project before running the last suggested command to run the app in development mode.

Understanding the project

Now that the project is created, we are going to take some time to understand the folders and files that Vite has created. Carry out the following steps:

1. Open Visual Studio Code in the project directory. This can be done using the following command in the terminal:

   ```
   code .
   ```
2. Look at the folders and files in the project in the **Explorer** panel on the left. The following are brief descriptions of these:

    - **`node_modules`**: This contains all dependent npm packages and was created when they were installed in the project using the **`npm install`** command.

- **`Public`**: This stores static assets such as images to be served at the root path (`/`).
- **`Src`**: This holds our source code files, including the following:
    - **`main.jsx`**: Contains logic for loading the React app into the root web page, **`index.html`**
    - **`index.css`**: Global styles for the app
    - **`App.jsx`**: The top-level React component called **`App`**
    - **`App.css`**: Styles for the **`App`** component
- **`.gitignore`**: This specifies which folders and files are to be ignored by Git.
- **`eslint.config.js`**: This is a configuration file for ESLint, which is a tool that checks code for potential errors and deviations from coding standards.
- **`index.html`**: This is the root web page. The React app is loaded into this at runtime.
- **`package.json`**: This defines the project name, version, dependencies, scripts, and other project metadata.
- **`package-lock.json`**: This holds exact versions for dependencies, ensuring consistency when the project is run in different environments.
- **`README.md`**: This contains information about the Vite template used to create the project. It is typically overwritten

with information about the app being developed, such as an overview and steps to set up the development environment.

- **`vite.config.js`**: This contains the configuration for Vite. For this project, a Vite React plugin has been specified.

Now that we are starting to understand the folders and files in the React project, we'll fully set up linting.

Adding linting to Visual Studio Code

Linting is the process of checking code for potential problems. It is common practice to use linting tools to catch problems early in the development process as code is written. **ESLint** is a popular tool that can lint React and TypeScript code. Fortunately, Vite has already installed and configured ESLint in our project.

Editors such as Visual Studio Code can be integrated with ESLint to highlight potential problems. Carry out the following steps to install an ESLint extension into Visual Studio Code:

1. Open up the **EXTENSIONS** area in Visual Studio Code. The **Extensions** option is in the **Preferences** menu in the **File** menu on Windows or the **Settings…** menu in the **Code** menu on a Mac.
2. A list of extensions will appear on the left-hand side and the search box above the extensions list can be used to find a particular extension. Enter **`eslint`** into the extensions list search box.

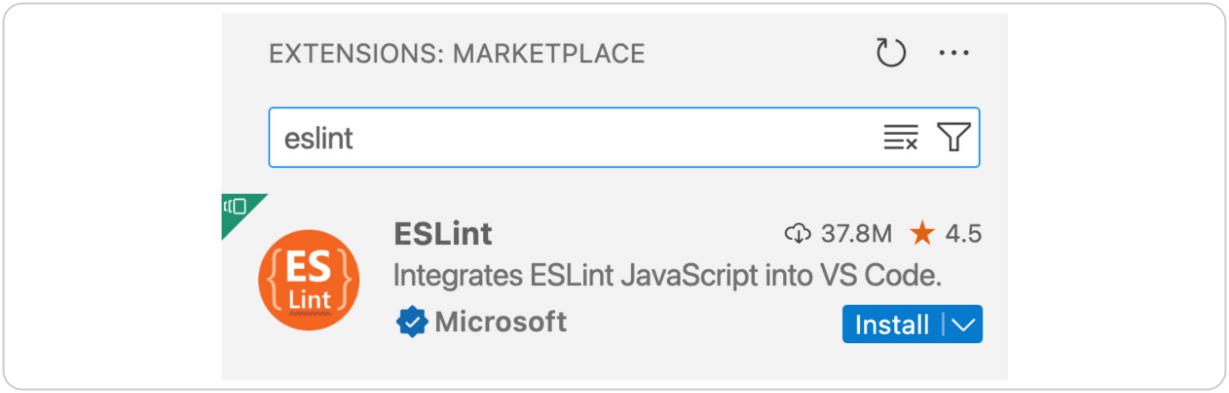
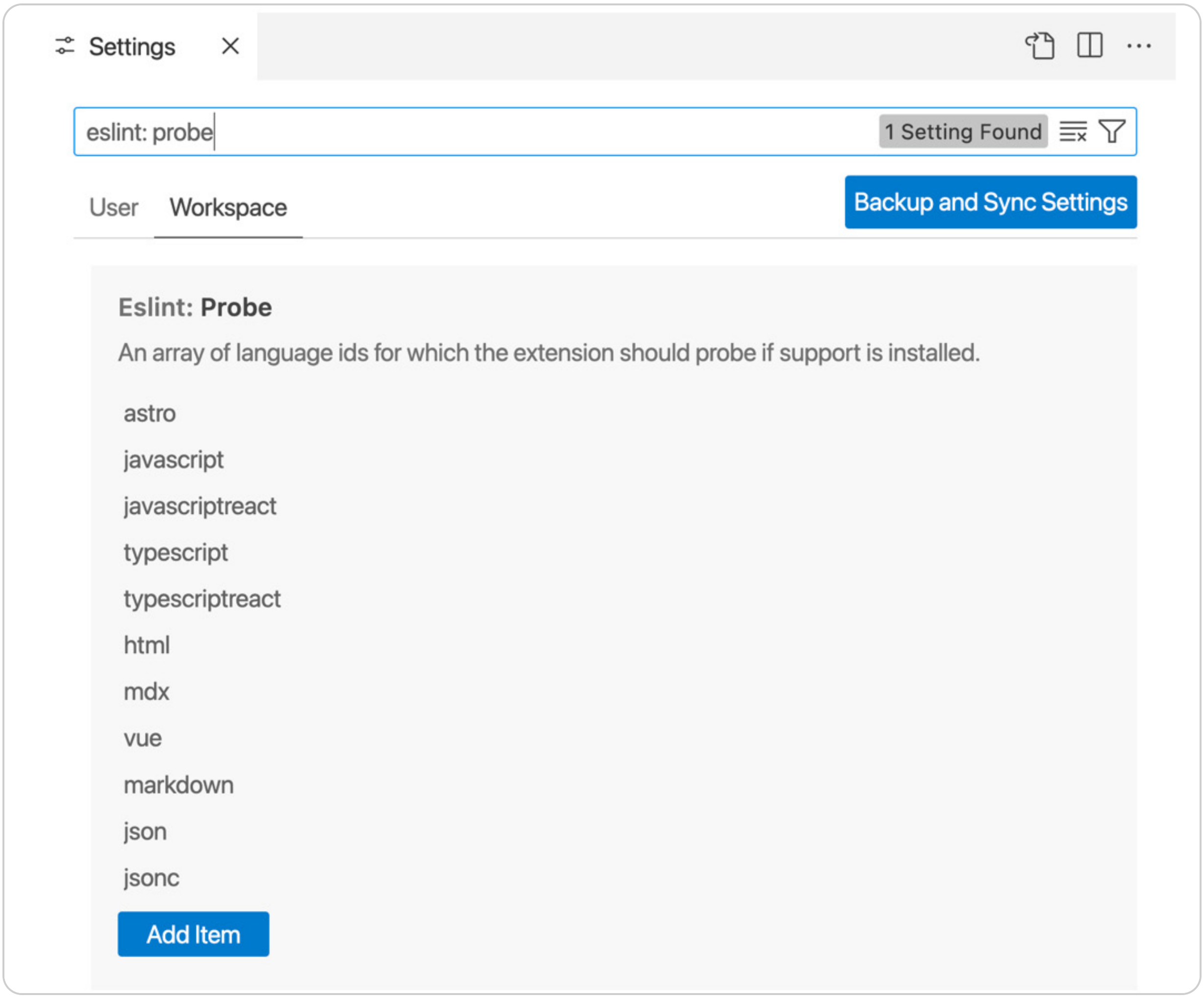


Figure 1.3 – Visual Studio Code ESLint extension

An extension by Microsoft called ESLint should appear at the top of the list.

3. Click the **Install** button to install the extension.
4. Now, we need to make sure the ESLint extension is configured to check React and TypeScript. So, open the **Settings** area in Visual Studio Code. The **Settings** option is in the **Preferences** menu in the **File** menu on Windows or the **Settings…** menu in the **Code** menu on a Mac.
5. In the **Settings** search box, enter **`eslint: probe`** and select the **Workspace** tab:



Eslint: Probe

An array of language ids for which the extension should probe if support is installed.

astro
javascript
javascriptreact
typescript
typescriptreact
html
mdx
vue
markdown
json
jsonc

Figure 1.4 – Visual Studio Code ESLint Probe settings

This setting defines the languages to use when ESLint checks code.

6. Make sure that **typescript** and **typescriptreact** are on the list. If not, add them using the **Add Item** button.

   The ESLint extension for Visual Studio Code is now installed and configured in the project.

7. Before we move on, there is one ESLint rule we are going to switch off, which is the checking of React component prop types. We won't be using this React feature because we will eventually be strongly typing

React components using TypeScript. Open **`eslint.config.js`** and add the highlighted line to the **`rules`** field to switch this rule off:

```
rules: {
  ...,
  'react/prop-types': 'off',
}
```

For more information about ESLint, see the following link: https://eslint.org/.

Next, we will add automatic code formatting to the project.

Adding code formatting

The next tool we will set up automatically formats code. Automatic code formatting ensures code is consistently formatted, which helps its readability. Having consistently formatted code also helps developers see the important changes in a code review – rather than differences in formatting.

Prettier is a popular tool capable of formatting React and TypeScript code. Unfortunately, Vite doesn't install and configure this for us. Carry out the following steps to install and configure Prettier in the project:

1. Install Prettier using the following command in the terminal in Visual Studio Code:

```
npm install --save-dev prettier
```

The `--save-dev` option specifies that `prettier` should be installed as a **development-only** dependency. This is because Prettier is only required during development and not at runtime.

A shortened version of this command is as follows:

```
npm i -D prettier
```

Here, `i` is short for `install`, and `-D` is short for `--save-dev`.

2. Prettier has overlapping style rules with ESLint, so install the following library to allow Prettier to take responsibility for the styling rules from ESLint:

```
npm i -D eslint-config-prettier
```

3. The ESLint configuration needs to be updated to allow Prettier to manage the styling rules. Open the `eslint.config.js` file, which is at the root of the project, and add the following highlighted lines:

```
...
import prettier from "eslint-config-prettier";
export default [
  ...,
  prettier
];
```

4. Prettier can be configured in a file called `.prettierrc.json`. Create this file with the following content in the root folder:

```
{
  “printWidth”: 100,
  “singleQuote”: true,
  “semi”: true,
  “tabWidth”: 2,
  “trailingComma”: “all”,
  “endOfLine”: “auto”
}
```

We have specified the following:

- Lines wrap at 100 characters
- String qualifiers are single quotes
- Semicolons are placed at the end of statements
- The indentation level is two spaces
- A trailing comma is added to multi-line arrays and objects
- Existing line endings are maintained

> **NOTE**
>
> *More information on the configuration options can be found at the following link: https://prettier.io/docs/en/options.html.*

Prettier is now installed and configured in the project.

Visual Studio Code can integrate with Prettier to automatically format code when source files are saved. So, let’s install a Prettier extension in Visual Studio Code:

1. Open the **EXTENSIONS** area in Visual Studio Code and enter `prettier` into the extensions list search box. An extension called **Prettier - Code formatter** should appear at the top of the list:

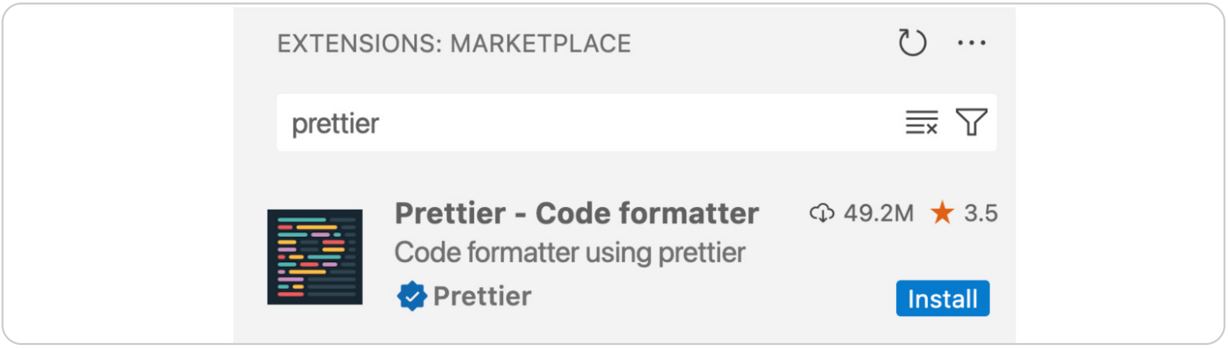
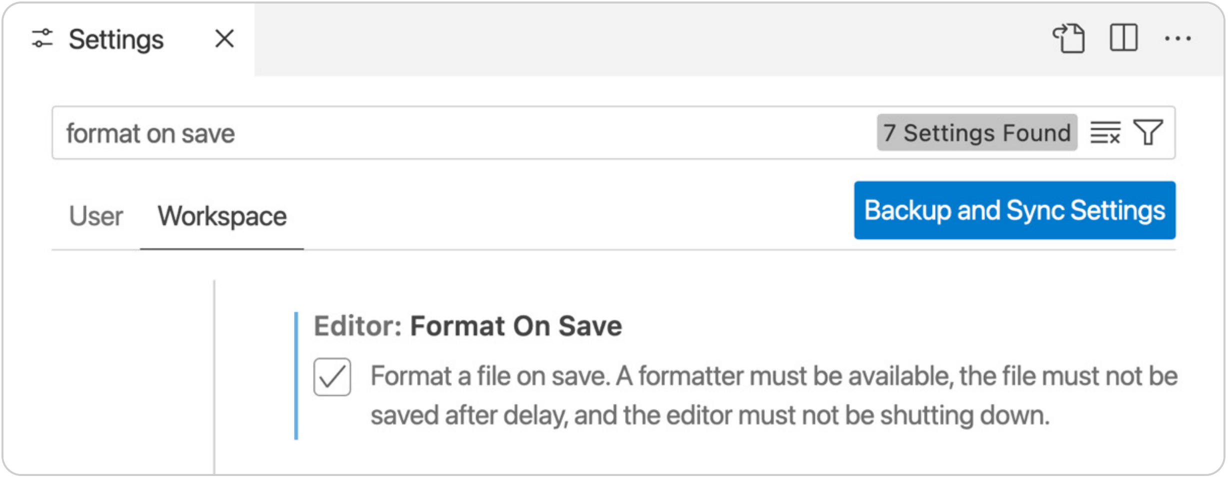


Figure 1.5 – Visual Studio Code Prettier extension

2. Click the **Install** button to install the extension.

3. Next, open the **Settings** area in Visual Studio Code. Select the **Workspace** tab and make sure the **Format On Save** option is ticked:



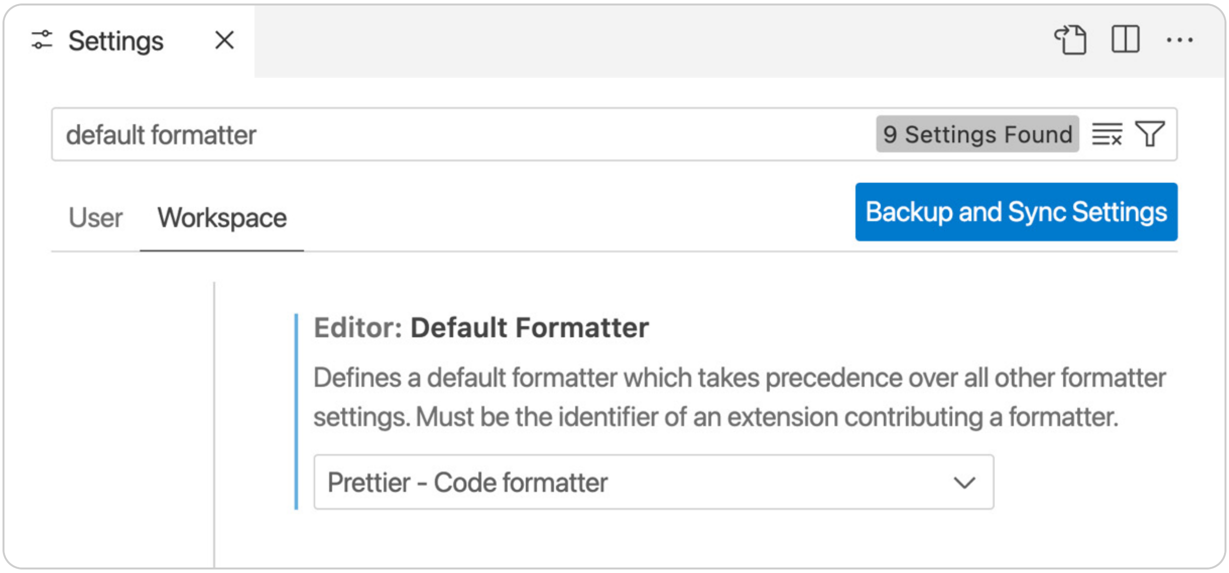
Editor: Format On Save

✓ Format a file on save. A formatter must be available, the file must not be saved after delay, and the editor must not be shutting down.

Figure 1.6 – Visual Studio Code Format On Save setting

This setting tells Visual Studio Code to automatically format code in files that are saved.

4. There is one more setting to set. This is the default formatter that Visual Studio Code should use to format the code. Click the **Workspace** tab and make sure **Default Formatter** is set to **Prettier - Code formatter**:



Editor: Default Formatter

Defines a default formatter which takes precedence over all other formatter settings. Must be the identifier of an extension contributing a formatter.

Prettier - Code formatter

Figure 1.7 – Visual Studio Code Default Formatter setting

The Prettier extension for Visual Studio Code is now installed and configured in the project. Next, we will run the app in development mode.

Starting the app in development mode

Vite has a development server that the project's app can run on. Carry out the following steps to run the app in development mode:

1. Vite has already created an npm script called **`dev`**, which runs the app in development mode. Run this script in the terminal as follows:

```
npm run dev
```

2. The app will start running on the Vite development server on localhost on port **5173** by default (the port can be changed in Vite's configuration). The browser URL for the app will appear in the terminal, which is http://localhost:5173/ by default. Go to this URL in a browser and you'll see the app running:

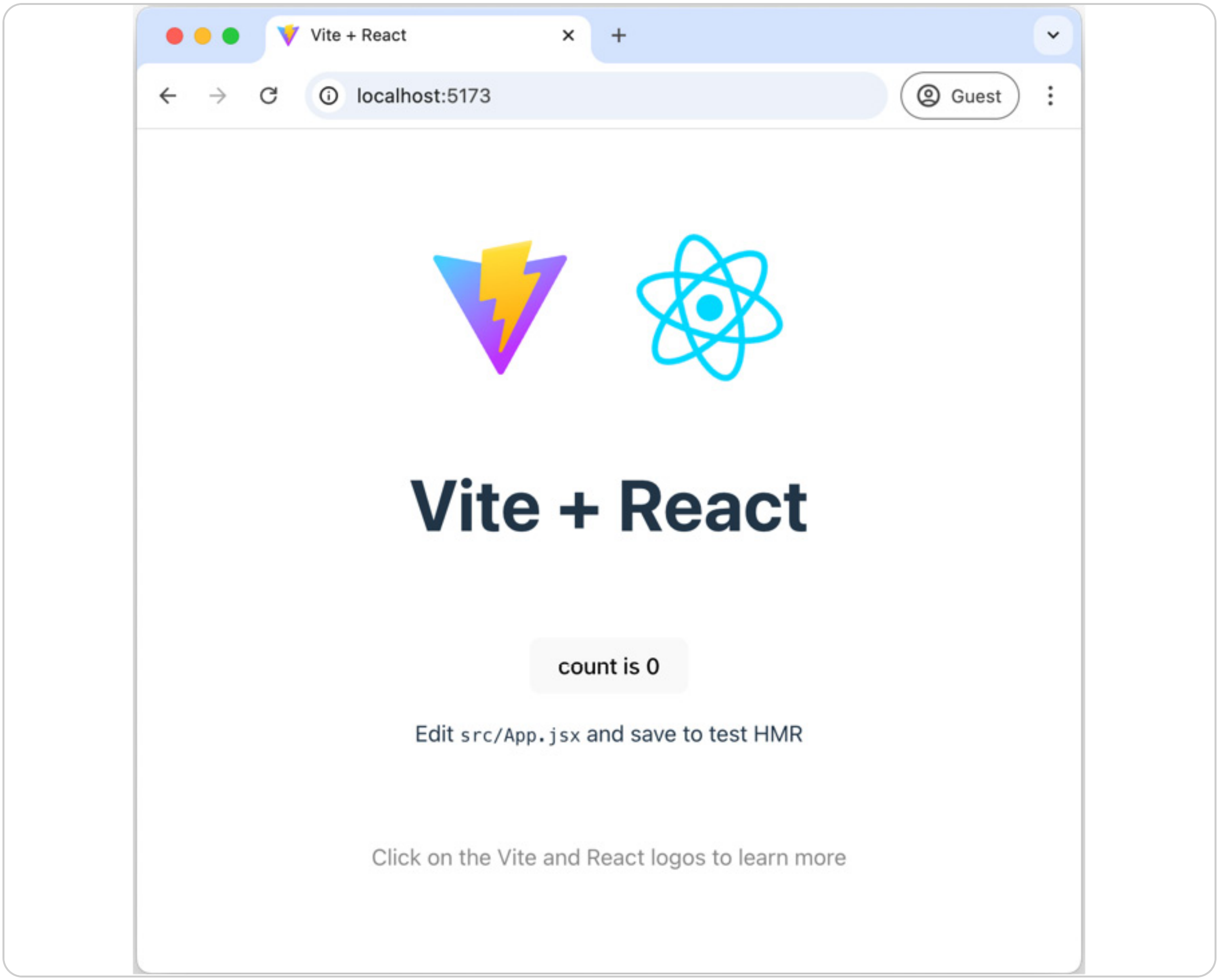

Vite + React
localhost:5173
Guest
Vite + React
count is 0
Edit src/App.jsx and save to test HMR
Click on the Vite and React logos to learn more

Figure 1.8 – The React app running in development mode

Vite not only serves the app on its development server but it also transpiles React components into JavaScript code that can run in the browser. It does all this incredibly fast!

3. We will make a simple change to the code now while the app is still running. In the code editor, open the **`index.html`** file at the project's root. Find the HTML **`title`** element, which specifies the title that appears on the browser tab.

4. Make a change to the contents of the **`title`** element by putting an exclamation mark at the end of it:

   ```
   <title>Vite + React!</title>
   ```

   Notice that the browser tab title updates immediately after you save the changes to the **`index.html`** file:

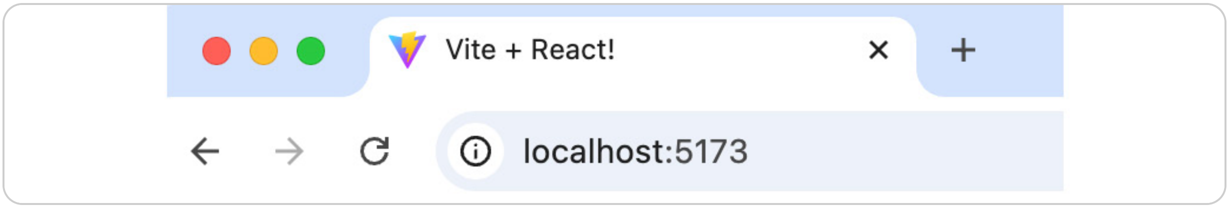


Figure 1.9 – Updated app title

Vite automatically does any required transpilation and reloads the app in the browser in a very efficient manner.

5. Stop the app from running before continuing. The shortcut key for stopping the app is *Ctrl* + *C*.

We have now seen how Vite provides a productive development experience. Next, we will produce a production build.

Producing a production build

A production build transpiles React components into JavaScript code similar to when running the app in development mode. However, it carries out several other processes on top of this so that the app runs in a performant manner in production.

One of the processes is **minification**. Minification is the process of removing all unnecessary characters from source code without impacting its functionality, which includes removing whitespace and comments and shortening variable names. This results in a smaller file size, leading to faster load times.

Another process also involves merging files so that the code is downloaded and executed in a performant manner in production. This process is often referred to as **bundling**, and the output file is often referred to as a **bundle**. Bundles are often separated into smaller chunks to decrease the app's load time (e.g., a bundle per page in the app). Bundlers also **tree-shake** redundant code out, to keep the size of the bundles as small as possible for better performance.

Carry out the following steps to produce a production build of our app:

1. Vite has already created an npm script called `build` that produces all the artifacts for deployment to production. Run this script in the terminal as follows:

```
npm run build
```

After a few seconds, the deployment artifacts are placed in a **`dist`** folder.

2. Open the **`dist`** folder – it contains many files. The root file is **`index.html`**, which references the other JavaScript, CSS, and image files. Open some of the files and view their content; you'll see that they are optimized for production with whitespace removed and the JavaScript minified.

This completes the production build and the React project setup with Vite. Here's a recap of the key points for creating a React project with Vite:

- Vite can quickly set up a React project using the **`npm create vite@latest`** command.
- Vite sets up many useful project features, such as linting. Using the ESLint Visual Code extension improves the linting experience when writing code.
- One feature that Vite doesn't set up is automatic code formatting. However, Prettier can be installed and configured to provide this capability.
- The app can be run in development code using the **`npm run dev`** command, and a production build can be created with **`npm run build`**.

Keep this project safe because we will continue to use it in the next section when we understand the structure of a React app.

Understanding the structure of a React app

In this section, we will explore the entry point of the React app created in the last section and how it is loaded into the HTML page. We will then learn about the React component tree and how a component is defined.

Understanding the React entry point

The entry point of this React app is in the **main.jsx** file in the **src** folder. Open this file and inspect its contents. It contains a call to React's **createRoot** function as follows:

```
createRoot(document.getElementById('root')).rende
r(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

Here's an explanation of this code:

- As the name suggests, **createRoot** creates a root in the HTML document for the React components. **createRoot** takes in a DOM element for where to place the React components, which is the element that has the ID of **'root'** in this case.

- **createRoot** returns an object containing a **render** function. The **render** function takes in the React components to display in the root DOM element. This displaying process is often referred to as **rendering**. In this case, the React components to display are an **App** component inside a **StrictMode** component. The syntax for the React components to display is **JSX**.

> **NOTE**
>
> *JSX stands for JavaScript XML, which is a syntax extension for JavaScript that allows developers to write HTML-like code within JavaScript/TypeScript. It enables the creation of React components in a readable and declarative way, making it easy to visualize the UI structure. JSX needs to be transpiled into regular JavaScript function calls using a tool such as* ***Babel****. For more information on JSX, see the following link: https://react.dev/learn/writing-markup-with-jsx.*

- The **StrictMode** component is a special React component that helps identify potential problems. It activates additional checks and outputs warnings to the browser console in development mode.

Next, we will take some time to understand the React component tree.

Understanding the React component tree

A React app is structured in a tree of components. The root component is the component at the top of the tree. In our project, the root component is the **`StrictMode`** component.

React components can be nested inside another React component. The **`App`** component is nested inside the **`StrictMode`** component in our project. This is powerful because any component can be placed inside **`StrictMode`** – it doesn't necessarily need to be **`App`**.

React components can output one or more other React components. The following is an example of a React component tree:

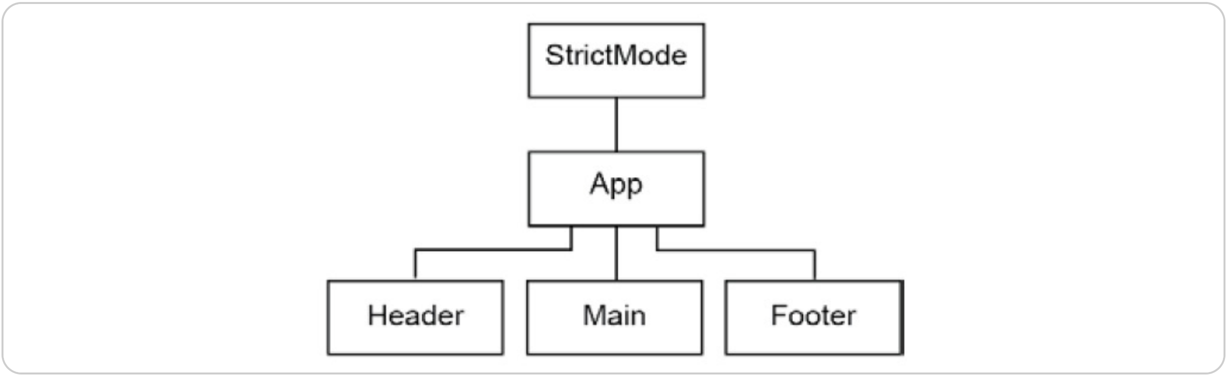


Figure 1.10 – A React component tree

If our **`App`** component rendered other React components (**`Header`**, **`Main`**, and **`Footer`**), the component tree would be as in the preceding figure.

Next, we'll start to understand how a React component is defined.

Understanding a React component

We will now understand the implementation of a basic React component.

Open **`App.jsx`**, which contains the definition for the **`App`** component. We won't fully understand the component at this stage, but notice it's just a regular JavaScript function.

Let's focus on what the function returns – it returns JSX representing the UI. Notice that the JSX references HTML elements such as **`div`**, **`a`**, **`h1`**, **`button`**, and **`p`**. So, JSX can output HTML elements as well as other React components. The **`App`** component currently only outputs HTML elements and not any other React components.

Notice the top-level JSX element in the return statement, **`<>`**, that doesn't have a name. This is a React **fragment**, which provides a way to group elements without creating a DOM element.

Still focusing on the JSX, notice the JavaScript code in curly brackets. For example, look at the JSX for the **`button`** element:

```
<button onClick={() => setCount((count) => count
+ 1)}>
  count is {count}
</button>
```

The **`onClick`** attribute is set to an anonymous JavaScript function that calls another function called **`setCount`**. We will understand what the **`onClick`** attribute does later in this chapter – the key point for now is that JSX can include JavaScript. Notice also that the `button` content also contains a reference to a JavaScript variable called `count`. Referencing JavaScript functions and variables in JSX allows component output to be dynamic.

That brings us to the end of this section. Let's recap:

- The entry point of a Vite React app is located in the `main.jsx` file, where the `createRoot` function is used to render React components
- A React app is structured into a tree of components
- A React component is a regular JavaScript function that returns JSX representing the dynamic UI

Next, it is time to create a React component.

Creating a component

In this section, we will create a React component and reference this within the `App` component.

Creating a basic Alert component

We are going to create a component that displays an alert, which we will simply call `Alert`. It will consist of an icon, a heading, and a message.

NOTE

> *A React component name must start with a capital letter. If a component name starts with a lowercase letter, it is treated as a DOM element and won't render properly.*

Carry out the following steps to create the component in the project:

1. Create a new file in the **`src`** folder called **`Alert.jsx`**.

> **NOTE**
>
> *The filename for component files isn't important to React or the React transpiler. It is common practice to use the same name as the component, either in Pascal or snake case. However, the file extension must be **`.js`** or **`.jsx`** for React transpilers to recognize these as React components.*

2. Open the **`Alert.jsx`** file and enter the following code in it:

```
function Alert() {
  return (
    <div>
      <div>
        <span role="img" aria-label="Warning">
          ⚠</span>
        <span>Oh no!</span>
      </div>
      <div>Something went wrong</div>
    </div>
  );
}
```

Remember that the code snippets are available online to copy. The link to the preceding snippet can be found at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter01/creating-a-component.

The component renders the following items:

- A warning icon (note that this is a warning emoji)
- A title: **`Oh no!`**
- A message: **`Something went wrong`**

> **NOTE**
>
> *The* ***`role`*** *and* ***`aria-label`*** *attributes have been added to the* ***`span`*** *element containing the warning icon to help screen readers understand that this is an image with a title of warning.*
>
> *For more information on the* ***`img`*** *role, see https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles/img_role.*
>
> *For more information on the* ***`aria-label`*** *attribute, see https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Attributes/aria-label.*

3. Alternatively, a React component can be implemented using arrow function syntax. The following code snippet is an arrow syntax version of the **`Alert`** component:

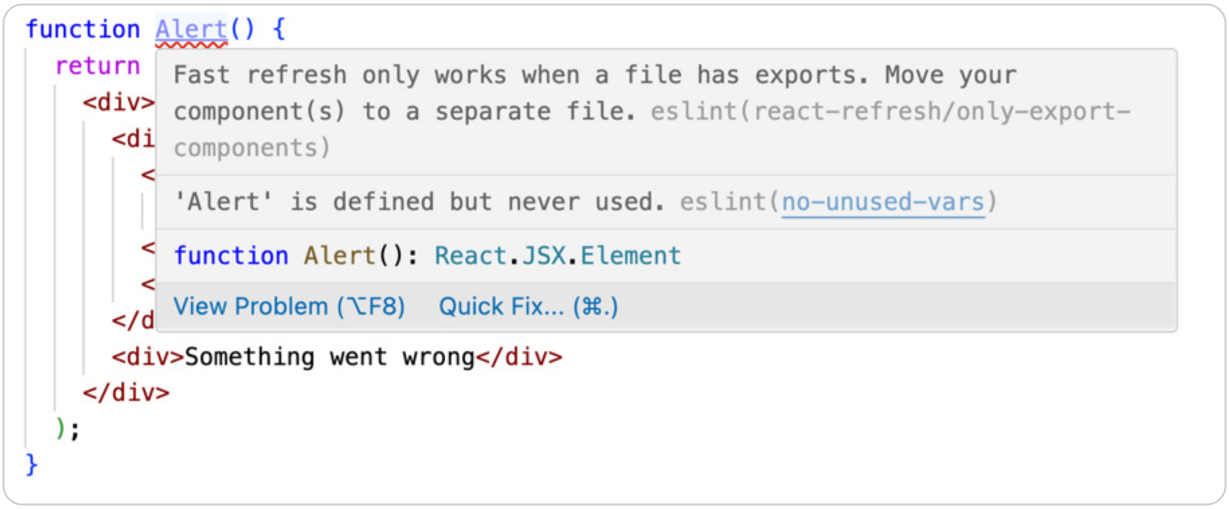
```
const Alert = () => {
  return (
    <div>
      <div>
        <span role="img" aria-label="Warning">
        ⚠
        </span>
        <span>Oh no!</span>
      </div>
      <div>Something went wrong</div>
    </div>
  );
};
```

NOTE

There aren't any significant differences between arrow functions and normal functions in the context of React function components. So, it is down to personal preference which one you choose. This book generally uses regular function syntax because it has fewer characters to type; however, if you wish, you can find more information on JavaScript arrow functions here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions.

Congratulations, you have created your first React component!

There is a linting error in the file we have just created, highlighted by ESLint. The error is that the **`Alert`** component is unused. Ignore the error for now – we'll resolve it later in this chapter.


```
function Alert() {
  return
    <div>
      <di
        <
        
        <
        <
      </d
      <div>Something went wrong</div>
    </div>
  );
}
```

```
Fast refresh only works when a file has exports. Move your
component(s) to a separate file. eslint(react-refresh/only-export-
components)

'Alert' is defined but never used. eslint(no-unused-vars)

function Alert(): React.JSX.Element
```

Figure 1.11 – ESLint error

If the app was running, the **Alert** component wouldn't appear in the browser yet. This is because it hasn't been added to the React component tree yet. We'll do this in the next section.

Adding Alert to the App component

Going back to the **Alert** component in our project, we will reference **Alert** in the **App** component. We will also remove the existing content in the **App** component so that it only renders the alert. To do this, carry out the following steps:

1. First, we need to export the **Alert** component so that it is available in other files. Open **Alert.jsx** and add the **export** keyword before the **Alert** function:

```
export function Alert() {
  ...
}
```

NOTE

It is common practice to have each React component in a separate file. This helps prevent files from becoming too large and helps the readability of the code base.

Notice that the ESLint error is now resolved because **`Alert`** can now potentially be used by other files.

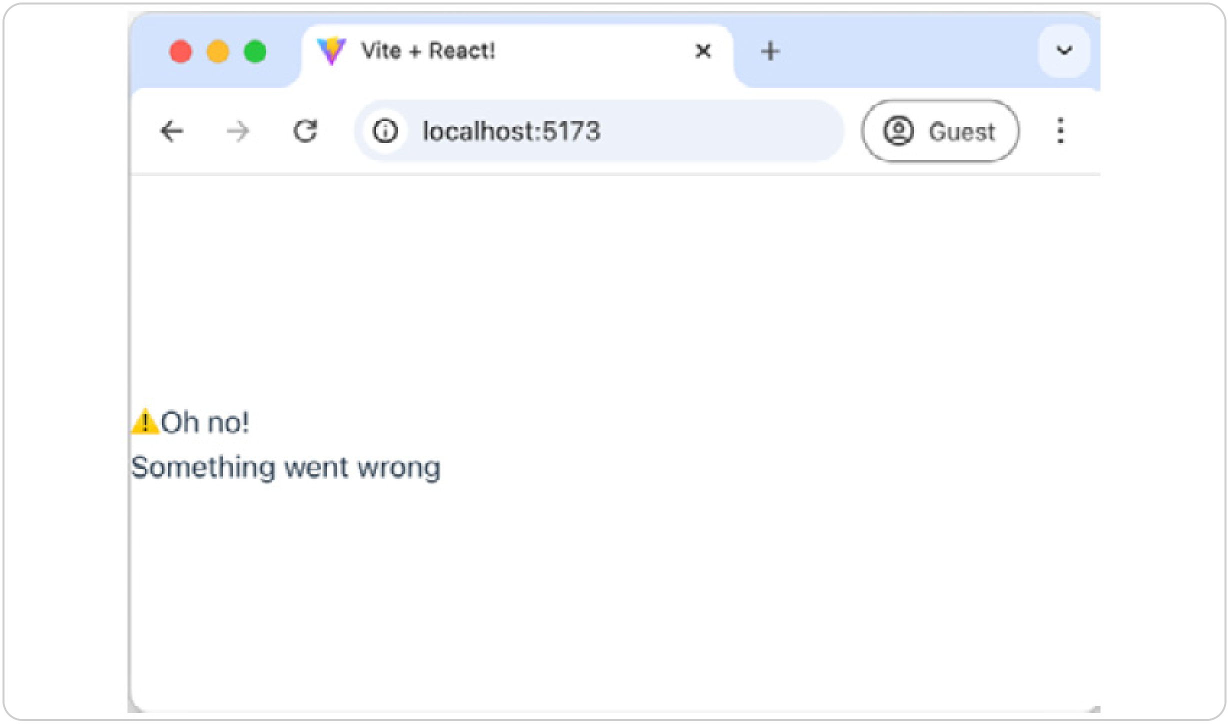
2. Now, we can import **`Alert`** into the **`App.jsx`** file. Open **`App.jsx`** and add the following **`import`** statement at the top of the file:

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

3. Remove the other **`import`** statements so that the alert is in the only import.
4. We can now reference **`Alert`** in the **`App`** component's JSX. Replace the **`App`** component definition with the following so that it only renders the alert:

   ```
   function App() {
     return <Alert />;
   }
   ```

5. Run the app in development mode by executing the **`npm run dev`** command in the terminal and opening the app's URL in a browser. The component will now display in the browser on the page:



⚠️Oh no!
Something went wrong

Figure 1.12 – Alert component in the app

Nice! If you have noticed that the **`Alert`** component isn't styled nicely, don't worry – we will learn how to style it in *Chapter 4*, *Approaches to Styling React Frontends*.

Here's a recap of this section:

- React component names start with an uppercase letter, and the filename should have a **`.js`** or **`.jsx`** extension.
- We created an **`Alert`** component that displays a warning icon, a title, and a message.
- Generally, a React component is structured in its own file and so needs to be exported before being referenced in another React component. We exported the **`Alert`** component and imported and used it within the **`App`** component.

Next, we will learn how to make the **`Alert`** component a little more flexible.

Using props

Currently, the **`Alert`** component is pretty inflexible. For example, the alert consumer can't change the heading or the message. At the moment, the heading or the message needs to be changed within **`Alert`** itself. **Props** solve this problem, and we will learn about them in this section.

> ***NOTE***
>
> *Props is short for properties. The React community often refers to these as props, so we will do so in this book.*

Understanding props

The **`props`** parameter is an optional parameter that is passed into a React component. This parameter is an object containing the properties of our choice, allowing a parent component to pass data. The following code snippet shows a **`props`** parameter in a **`ContactDetails`** component:

```
function ContactDetails(props) {
  console.log(props.name);
  console.log(props.email);
```

```
    ...
}
```

The **`props`** parameter contains the **`name`** and **`email`** properties in the preceding code snippet.

> **NOTE**
>
> *The parameter doesn't have to be named props, but it is common practice.*

Props are passed into a component in JSX as attributes. The prop names must match what is defined in the component. Here is an example of passing props into the preceding **`ContactDetails`** component:

```
<ContactDetails name="Fred"
email="fred@somewhere.com" />
```

So, props make the component output flexible. Consumers of the component can pass appropriate props into the component to get the desired output.

Next, we will add some props to the **`Alert`** component we have been working on.

Adding props to the Alert component

In the project, carry out the following steps to add props to the **`Alert`** component to make it more flexible:

1. Start by running the app in development mode if it's not already running. Do this by running the **`npm run dev`** command in the terminal.
2. Open **`Alert.jsx`** and add a **`props`** parameter to the function:

```
export function Alert(props) {
  ...
}
```

3. We will define the following props for the alert:

    - **`type`**: This will either be **`"information"`** or **`"warning"`** and will determine the icon in the alert.
    - **`heading`**: This will determine the heading of the alert.
    - **`children`**: This will determine the content of the alert. The **`children`** prop is actually a special prop used for the main content of a component.

Update the **`Alert`** component's JSX to use the props as follows:

```
export function Alert(props) {
  return (
    <div>
      <div>
        <span
          role="img"
          aria-label={
            props.type === 'warning'
              ? 'Warning'
              : 'Information'
          }
        >
        {props.type === 'warning' ? '⚠' : 'ℹ'}
      </span>
      <span>{props.heading}</span>
    </div>
    <div>{props.children}</div>
    </div>
  );
}
```

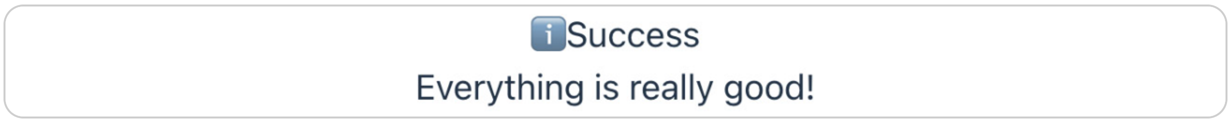
You may notice that the **Alert** component in the browser now displays nothing other than an information icon (this is an information emoji); this is because the **App** component isn't passing any props to **Alert** yet.

4. Open **App.jsx** and update the **Alert** component in the JSX to pass in props as follows:

```
export default function App() {
  return (
    <div className="App">
      <Alert type="information"
heading="Success">
        Everything is really good!
      </Alert>
    </div>
  );
}
```

Notice that the **Alert** component is no longer self-closing so **Everything is really good!** can be passed into its content. The content is passed to the **children** prop.

The app now displays the configured **`Alert`** component:


ℹ️Success

Everything is really good!

Figure 1.13 – Configured Alert component in the app

5. We can clean up the **`Alert`** component code a little by destructuring the **`props`** parameter.

> ## NOTE
>
> ***Destructuring*** *is a JavaScript feature that allows properties to be unpacked from an object. For more information, see the following link: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment.*

6. Open **`Alert.jsx`** again, destructure the **`function`** parameter, and use the unpacked props as follows:

```
export function Alert({ type, heading,
children }) {
  return (
    <div>
      <div>
        <span
          role="img"
          aria-label={
            type === 'warning' ? 'Warning' :
'Information'
          }
        >
```

```
          {type === 'warning' ? '⚠' : 'i'}
        </span>
        <span>{heading}</span>
      </div>
      <div>{children}</div>
      </div>
    );
  }
```

This is a little cleaner because we use the unpacked props directly rather than having to reference them through the **props** parameter.

7. We want the **type** prop to default to **"information"**. Define this default as follows:

```
  export function Alert({
    type = 'information',
    heading,
    children
  }) {
    ...
  }
```

That completes the implementation of the props in the **Alert** component for now. Here's a quick recap on props:

- Props allow a component to be configured by the consuming JSX and are passed as JSX attributes
- Props are received in the component definition in an object parameter and can then be used in its JSX

Next, we will continue to make the `Alert` component more sophisticated by allowing it to be closed by the user.

Using state

React component `state` is a special variable that may change over the lifecycle of a component. In this section, we'll learn about the `state` variable and use it within our `Alert` component. We will use `state` to allow the alert to be closed by the user.

Understanding state

There isn't a predefined list of states; we define what's appropriate for a given component. Some components won't even need any state – the `Alert` component hasn't required a state for the requirements so far.

The state is a key part of making a component interactive. When a user interacts with a component, the component's output may need to change. A change to the state causes the component to refresh, more often referred to as **re-render**.

The state is defined using a `useState` function from React. The `useState` function is one of React's **Hooks**. There is a whole chapter on React Hooks in *Chapter 3*, *Using React Hooks*.

The syntax for `useState` is as follows:

```
const [state, setState] = useState(initialState);
```

Here are the key points:

- The initial state value is passed into **`useState`**. If no value is passed, it will initially be **`undefined`**.
- **`useState`** returns a tuple containing the current state value and a function to update the state value. The tuple is destructured in the preceding code snippet.
- The state variable name is **`state`** in the preceding code snippet, but we can choose any meaningful name.
- We can also choose the state setter function name, but it is common practice to use the same name as the state variable preceded by **`set`**.
- Multiple states can be defined by defining multiple instances of **`useState`**. For example, here are definitions for loading and error states:

  ```
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState();
  ```

Next, we will implement **`state`** in the **`Alert`** component to determine whether it is visible or not.

Implementing a visible state in the Alert component

We will begin by implementing a feature in the **`Alert`** component that allows the user to close it. A key part of that feature is controlling the alert's visibility, which we will do with a **`visible`** state. This state will either be **`true`** or **`false`** and it will initially be set to **`true`**.

Follow these steps to implement a **`visible`** state in **`Alert`**:

1. If the app isn't already running, do so by running the **`npm run dev`** command in the terminal.
2. Open **`Alert.jsx`** in the project.
3. Add the following **`import`** statement at the top of the file to import the **`useState`** Hook from React:

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

4. Define the **`visible`** state as follows in the component definition:

```
export function Alert(...) {
  const [visible, setVisible] =
useState(true);
  return (
    ...
  );
}
```

5. After the state declaration, add a condition that returns **`null`** if the **`visible`** state is **`false`**. This means nothing will be rendered:

```
export function Alert(...) {
    const [visible, setVisible] =
  useState(true);
    if (!visible) {
      return null;
    }
    return (
      ...
    );
  }
```

The component will render in the app the same as before because the **`visible`** state is **`true`**. Try changing the initial state value to **`false`**, and you will see it disappear.

Currently, the **`Alert`** component is making use of the **`visible`** state's value by not rendering anything if it is **`false`**. However, the component isn't updating the **`visible`** state yet – that is, **`setVisible`** is unused at the moment. We will update the **`visible`** state after implementing a **`close`** button, which we will do next.

Adding a close button to Alert

We will add a **`close`** button to the **`Alert`** component to allow the user to close it. We will make this configurable so that the alert consumer can choose whether the **`close`** button is rendered.

Carry out the following steps:

1. Start by opening **`Alert.jsx`** and adding a **`closable`** prop:

```
export function Alert({
  type = "information",
  heading,
  children,
  closable
}) {
  ...
}
```

The consumer of the **`Alert`** component will use the **`closable`** prop to specify whether the **`close`** button appears.

2. Add a **`close`** button between the heading and content as follows:

```
export function Alert(...) {
  ...
  return (
    <div>
      <div>
      ...
        <span>{heading}</span>
      </div>
      <button aria-label="Close">
        <span role="img" aria-
label="Close">❌</span>
      </button>
      <div>{children}</div>
    </div>
  );
}
```

Notice that the **`span`** element that contains the **`close`** icon is given an **`"img"`** role and a **`"Close"`** label to help screen readers.

Likewise, the button is also given a **"Close"** label to help screen readers.

The **close** button appears in the **Alert** component as follows:

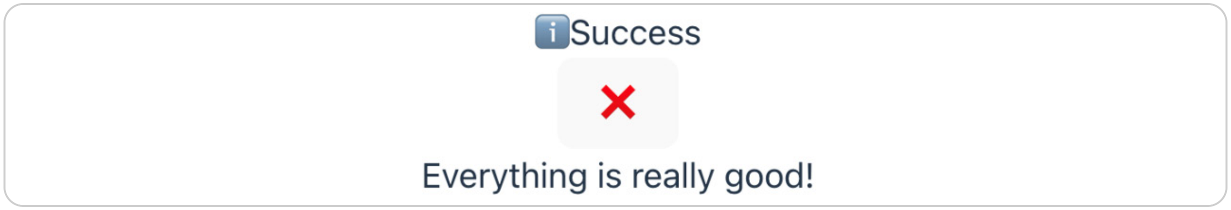


Figure 1.14 – The close button in the Alert component

3. At the moment, the **close** button will always render rather than just when the **closable** prop is **true**. We can use a JavaScript logical **AND** short circuit expression (represented by the **&&** characters) to render the **close** button conditionally. To do this, make the following highlighted changes:

```
import { useState } from 'react';
export function Alert(...) {
  ...
  return (
    <div>
      <div>
        ...
        <span>{heading}</span>
      </div>
      {closable && (
        <button aria-label="Close">
        <span role="img" aria-label="Close">
          ❌
        </span>
      </button>
      )}
      <div>{children}</div>
      </div>
    );
  }
```

If **`closable`** is a **falsy** value, the expression will **short-circuit** and, consequently, not render the button. However, if **`closable`** is **truthy**, the button will be rendered.

> **NOTE**
>
> *See the following link for more information about logical **AND** short-circuit expressions: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND.*
> *See the following link for JavaScript's falsy values: https://developer.mozilla.org/en-US/docs/Glossary/Falsy and https://developer.mozilla.org/en-US/docs/Glossary/Truthy for truthy values.*

4. Open **`App.jsx`** and pass the **`closable`** prop into **`Alert`**:

```
export default function App() {
  return (
    <div className="App">
      <Alert
        type="information"
        heading="Success"
        closable
      >
      Everything is really good!
      </Alert>
      </div>
    );
  }
```

Notice that a value hasn't been explicitly defined on the **`closable`** attribute. We could have passed the value as follows:

```
closable={true}
```

However, there is no need to pass the value on a Boolean attribute. If the Boolean attribute is present on an element, its value is automatically **`true`**.

When the **`closable`** attribute is specified, the **`close`** button appears in the **`Alert`** component as it did before, in *Figure 1.13*. When the **`closable`** attribute isn't specified, the **`close`** button doesn't appear:

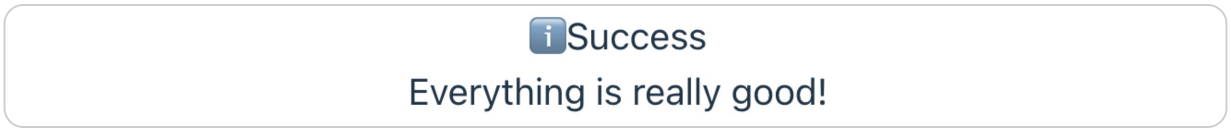

ℹ️Success
Everything is really good!

Figure 1.15 – The close button is not in the Alert component when closable is not specified

Excellent!

Here is a quick recap of what we have learned so far about React state:

- State is defined using React's **`useState`** Hook
- The initial value of the state can be passed into the **`useState`** Hook

- `useState` returns a state variable that can be used to render elements conditionally
- `useState` also returns a function that can be used to update the value of the state

You may have noticed that the `close` button doesn't actually close the alert. In the next section, we will rectify this as we learn about events in React.

Using events

Events are another key part of allowing a component to be interactive. In this section, we will understand what React events are and how to use events on DOM elements. We will also learn how to create our own React events.

We will continue to expand the `Alert` component's functionality as we learn about events. We will start by finishing the `close` button implementation before creating an event for when the alert has been closed.

Understanding events

Browser events happen as the user interacts with DOM elements. For example, clicking a button raises a `click` event from that button.

Logic can be executed when an event is raised. For example, an alert can be closed when its `close` button is clicked. A function called an **event handler** (sometimes referred to as an **event listener**) can be registered on an element for an event that contains the logic to execute when that particular event happens.

> ***NOTE***
>
> *See the following link for more information on browser events: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events.*

Events in React events are very similar to browser-native events. In fact, React events are a wrapper on top of the browser's native events.

Event handlers in React are generally registered to an element in JSX using an attribute. The following code snippet registers a `click` event handler called `handleClick` on a `button` element:

```
<button onClick={handleClick}>...</button>
```

Next, we will return to our `Alert` component and implement a `click` handler on the `close` button that closes the alert.

Implementing a close button click handler in the alert

At the moment, our **`Alert`** component contains a **`close`** button, but nothing happens when it is clicked. The alert also contains a **`visible`** state that dictates whether the alert is shown. So, to finish the **`close`** button implementation, we need to add an event handler when it is clicked that sets the **`visible`** state to **`false`**. Carry out the following steps to do this:

1. If the app isn't already running, do so by running the **`npm run dev`** command in the terminal.
2. Open **`Alert.jsx`** and register a **`click`** handler on the **`close`** button as follows:

   ```
   <button aria-label="Close" onClick=
   {handleCloseClick}>
   ```

   We have registered a **`click`** handler called **`handleCloseClick`** on the **`close`** button.
3. We then need to implement the **`handleCloseClick`** function in the component. Create an empty function to start with, just above the **`return`** statement:

   ```
   export function Alert(...) {
     const [visible, setVisible] =
   useState(true);
     if (!visible) {
       return null;
     }
     function handleCloseClick() {}
     return (
   ```

```
      ...
    );
  }
```

This may seem a little strange because we have put the **handleCloseClick** function inside another function, **Alert**. The handler needs to be inside the **Alert** function so that it has access to props and state.

Arrow function syntax can be used for event handlers if preferred. An arrow function version of the handler is as follows:

```
export function Alert(...) {
  const [visible, setVisible] =
useState(true);
  if (!visible) {
    return null;
  }
  const handleCloseClick = () => {}
  return (
    ...
  );
}
```

Event handlers can also be added directly to the element in JSX as follows:

```
<button aria-label="Close" onClick={() => {}}>
```

In the **Alert** component, we will stick to the named **handleCloseClick** event handler function.

4. Now, we can use the **visible** state setter function to make the **visible** state **false** in the event handler:

```
function handleCloseClick() {
  setVisible(false);
}
```

If you click the **`close`** button in the app, the alert disappears. Nice!

Note that the browser's *reload* option can be used to reload the app and make the **`Alert`** component reappear.

Next, we will extend the **`close`** button to raise an event when the alert closes.

Implementing an alert close event

We will now create a custom event in the **`Alert`** component. The event will be raised when the alert is closed so that consumers can execute logic when this happens.

A custom event in a component is implemented by using a prop. The prop is a function that is called to raise the event.

To implement an alert close event, follow these steps:

1. Start by opening **`Alert.jsx`** and add a prop for the event:

```
export function Alert({
  type = "information",
  heading,
  children,
  closable,
    onClose
  }) {}
```

We have called the prop **onClose**.

> ***NOTE***
>
> *It is common practice to start an event prop name with on.*

2. In the **handleCloseClick** event handler, raise the close event after the **visible** state is set to **false**:

   ```
   function handleCloseClick() {
     setVisible(false);
     if (onClose) {
       onClose();
     }
   }
   ```

   Notice that we only invoke **onClose** if it is defined and passed as a prop by the consumer. This means that we aren't forcing the consumer to handle this event.

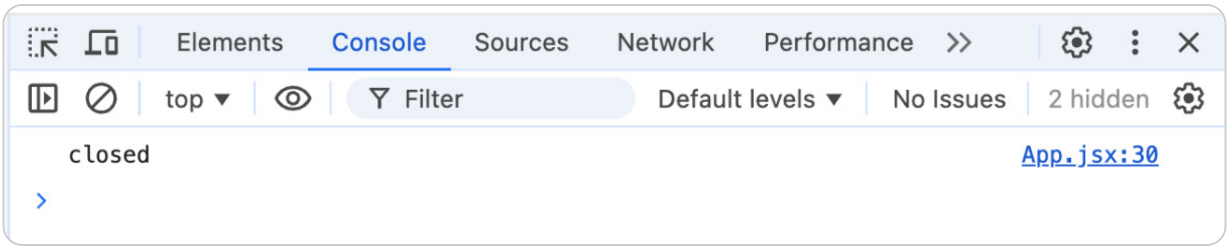
3. We can now handle when an alert is closed in the **App** component. Open **App.jsx** and add the following event handler to **Alert** in the JSX:

   ```
   <Alert
     type="information"
     heading="Success"
     closable
     onClose={() => console.log("closed")}
   ```

```
  >
    Everything is really good!
  </Alert>;
```

We have used an inline event handler this time.

4. In the app, if you click the *close* button and look at the console, you will see that **closed** has been output:



```
closed
```

Figure 1.16 – Console output after the alert is closed

That completes the close event and the implementation of the alert for this chapter.

Here's what we have learned about React events:

- Events, along with state, allow a component to be interactive
- Event handlers are functions that are registered on elements in JSX
- A custom event can be created by implementing a function prop and invoking it to raise the event

The component we created in this chapter is a function component. You can also create components using classes. For example, a class component version of the `Alert` component is at https://github.com/PacktPublishing/Learn-React-with-TypeScript-

Third-Edition/tree/main/Chapter01/class-component. However, function components are dominant in the React community. Here are a few of the reasons why:

- Generally, they require less code to implement
- Logic inside the component can be more easily reused
- React Hooks can't be used in class components

For these reasons, we will focus solely on function components in this book.

Next, we will learn how to use the React browser development tools.

Using React developer tools

React developer tools is a browser extension available for Chrome, Firefox, and Edge. The tools allow React apps to be inspected and debugged. In this section, we are going to install and use these tools on the `Alert` component we have implemented in this chapter.

The links to the extensions are as follows:

- Chrome: https://chromewebstore.google.com/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi
- Firefox: https://addons.mozilla.org/en-GB/firefox/addon/react-devtools/

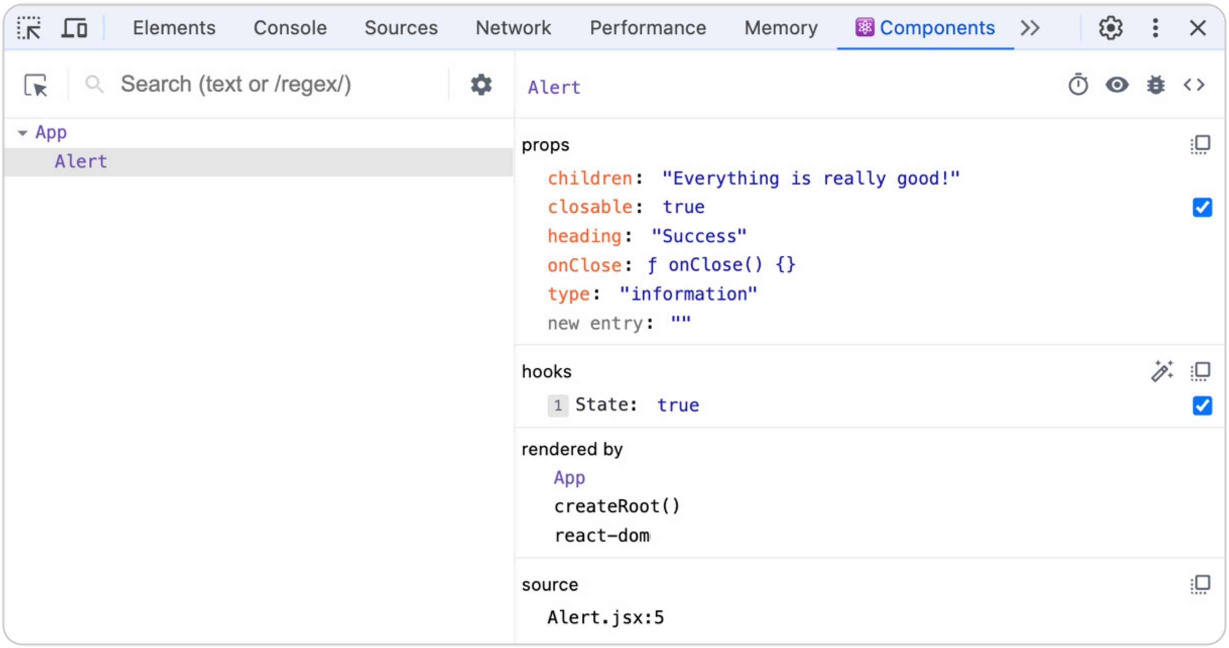
- Edge: https://microsoftedge.microsoft.com/addons/detail/react-developer-tools/gpphkfbcpidddadnkolkpfckpihlkkil

Follow the instructions in the relevant link to install the extension in your browser. You may need to reopen the browser for the tools to be available.

Using the Components tool

The first tool we are going to explore is the **Components** tool. It allows you to inspect the current props and state of a component. Carry out the following steps to try this tool on our `Alert` component:

1. If the app isn't running, start it by executing `npm run dev` in a terminal.
2. Open the browser's development tools by pressing *F12* on Windows or *Cmd* + *Option* + *I* on a Mac. React's developer tools can be found in two panels called **Components** and **Profiler**.
3. Select the **Components** panel. The React component tree appears. Selecting the `Alert` component in the component tree reveals current information about it:

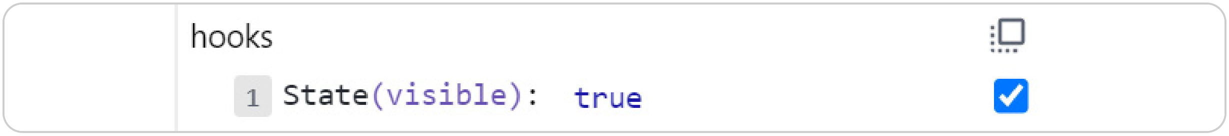


```
props
  children: "Everything is really good!"
  closable: true
  heading: "Success"
  onClose: ƒ onClose() {}
  type: "information"
  new entry: ""
hooks
  1 State: true
rendered by
  App
  createRoot()
  react-dom
source
  Alert.jsx:5
```

Figure 1.17 – React development tools Components panel

4. One useful set of component information is the current prop values. You can change these values and the component output in the browser will be updated. Try this by changing the `children` prop. This is a great way of manually testing that a prop is working as expected.

5. There is also a section for current Hook values, which includes state values. Notice, however, that the state isn't named – it has a generic name, `State`. However, Hooks appear in this section in the order they appear in the component code, so we can work out what these items are. In addition, there is a **Parse hook names** option (with a wizard icon) that can show the Hook variable names. Click this option to reveal the state variable name in brackets:



```
hooks
  1 State(visible): true
```

Figure 1.18 – State variable name after the wizard icon is clicked

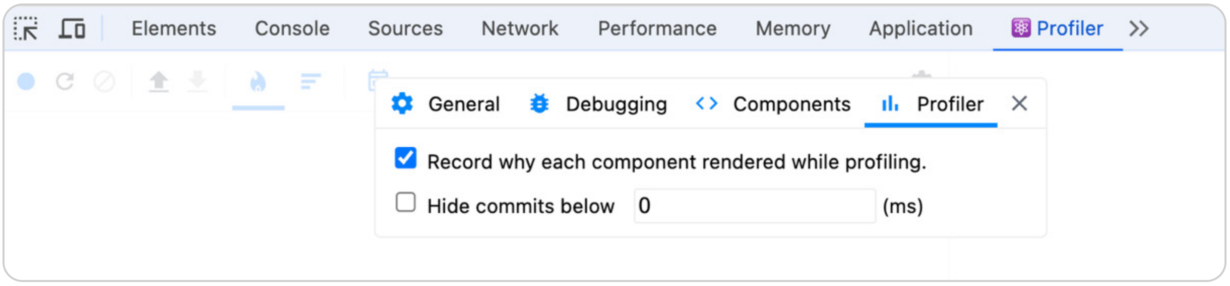
6. We're going to observe the `visible` state value change. Make sure `closable` is set on the component (you can set this prop to `true` in the `props` section if not). Like props, you can change state values using the developer tools. Click the checkbox to the right of `State(visible)`. This toggles the value of the `visible` state between `true` and `false` and updates the component in the browser accordingly.

This completes our exploration of the **Components** tool. Press *F5* to refresh the browser so that the `Alert` component reappears before continuing. Next, we will explore the **Profiler** tool.

Using the Profiler tool

Now, we will explore the **Profiler** panel. This tool allows interactions to be profiled, which is useful for tracking performance problems. Carry out the following steps to profile the closing of the alert:

1. Before we start profiling, we are going to make sure the **Profiler** tool captures why components render. Select the **Profiler** panel and click the **View settings** option (the cog icon). After the React developer tools settings open, click the **Profiler** tab to view the **Profiler** settings. Make sure the **Record why each component rendered while profiling.** setting is ticked.

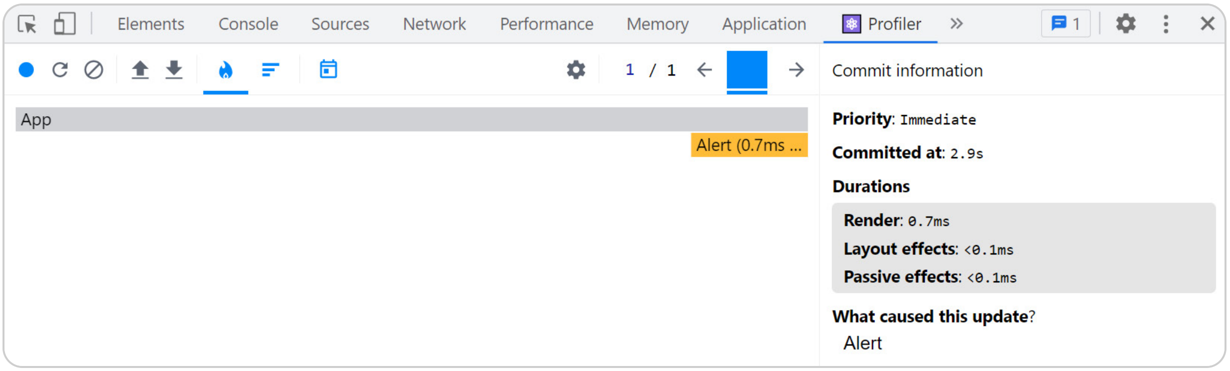


☑ Record why each component rendered while profiling.

☐ Hide commits below 0 (ms)

Figure 1.19 – Profiler settings

2. Close the settings and click the **Start profiling** option, which is the blue circle icon.
3. Click the *close* button in the alert in the app.
4. Click the **Stop profiling** option, which is the red circle icon. A timeline appears of all the component re-renders:



Commit information

Priority: `Immediate`
Committed at: `2.9s`
Durations
Render: `0.7ms`
Layout effects: `<0.1ms`
Passive effects: `<0.1ms`
What caused this update?
Alert

Figure 1.20 – Profile of the alert being closed

This shows that `Alert` was re-rendered when the *close* button was clicked, taking 0.7 milliseconds.

This tool is helpful in quickly spotting the slow components of a user interaction.

This completes our exploration of the React developer tools. Here's a recap:

- The React **Components** developer tool allows component props and state to be inspected and tested
- The React **Profiler** developer tool allows poor-performing user interactions to be profiled to help pinpoint the root problem

That brings us to the end of the chapter. Next is a chapter summary.

Summary

We now understand that React is a popular library for creating component-based frontends. In this chapter, we created an **`Alert`** component using React.

Component output is declared using a mix of HTML and JavaScript called JSX. JSX needs to be transpiled into JavaScript before it can be executed in a browser.

Props can be passed into a component as JSX attributes. This allows consumers of the component to control its output and behavior. A component receives props as an object parameter. The JSX attribute names form the object parameter property names. We implemented a range of props in this chapter in the **`Alert`** component.

Events can be handled to execute logic when the user interacts with the component. We created an event handler for the **`close`** button **`click`** event in the **`Alert`** component.

State can be used to re-render a component and update its output. The state is defined using the **`useState`** Hook and is often updated in event handlers. We created a state for whether the alert is visible.

Custom events can be implemented as a function prop. This allows consumers of the component to execute logic as the user interacts with it. We implemented a close event on the **`Alert`** component.

The **`Alert`** component is an example of a reusable component that can be used in many places across a large app and even across different apps.

In the next chapter, we will introduce ourselves to TypeScript. We will then use TypeScript to strongly type the **`Alert`** component we started in this chapter.

Questions

Answer the following questions to reinforce what you have learned in this chapter:

1. What is wrong with the following component definition?

```
export function important() {
    return <div>This is really important!</div>;
  }
```

2. Component props are passed into a component as follows:

```
<ContactDetails name="Fred"
email="fred@somewhere.com" />
```

The component is then defined as follows:

```
export function ContactDetails({ firstName,
email }) {
  return (
    <div>
      <div>{firstName}</div>
      <div>{email}</div>
    </div>
  );
}
```

The name **`Fred`** isn't output though. What is the problem?

3. What is the initial value of the **`loading`** state defined here?

```
const [loading, setLoading] = useState(true);
```

4. What is wrong with how the state is set in the following component?

```
export function Agree() {
  const [agree, setAgree] = useState();
  return (
    <button onClick={() => agree = true}>
      Click to agree
    </button>
  );
}
```

5. The following component implements an optional **`Agree`** event. What is wrong with this implementation?

   ```
   export function Agree({ onAgree }) {
     function handleClick() {
       onAgree();
     }
     return (
       <button onClick={handleClick}>
         Click to agree
       </button>
     );
   }
   ```

Answers

Here are the answers to the preceding questions:

1. The problem with the component definition is that its name is in lowercase. React functions must be named with an uppercase first character:

   ```
   export function Important() {
     ...
   }
   ```

2. The problem is that a **`name`** prop is passed rather than **`firstName`**. Here's the corrected JSX:

   ```
   <ContactDetails firstName="Fred"
   email="fred@somewhere.com" />
   ```

3. The initial value of the **`loading`** state is **`true`**.
4. The state isn't updated using the state setter function. Here's the corrected version of the state being set:

```
export function Agree() {
  const [agree, setAgree] = useState();
  return (
    <button onClick={() => setAgree(true)}>
      Click to agree
    </button>
  );
}
```

5. The problem is that clicking the button will cause an error if **`onAgree`** isn't passed because it will be **`undefined`**. Here's the corrected version of the component:

```
export function Agree({ onAgree }) {
  function handleClick() {
    if (onAgree) {
      onAgree();
    }
  }
  return (
    <button onClick={handleClick}>
      Click to agree
    </button>
  );
}
```

Learn more on Discord

2
Getting Started with TypeScript

In this chapter, we will start by understanding what TypeScript is and how it provides a much richer type system on top of JavaScript. We will learn about the basic types in TypeScript, such as numbers and strings, and then how to create our own types to represent objects and arrays using different TypeScript features. We will understand the TypeScript compiler and its key options in a React app. Finally, we will revise the alert component we built in the last chapter to use TypeScript.

In this chapter, we'll cover the following topics:

- Understanding the benefits of TypeScript
- Understanding JavaScript types
- Using basic TypeScript types
- Creating TypeScript types
- Using the TypeScript compiler
- Creating a React and TypeScript component

By the end of this chapter, you'll be able to create simple type-safe React components.

Technical requirements

We will use the following software in this chapter:

- **Browser**: A modern browser such as Google Chrome.
- **TypeScript Playground**: This is a website at https://www.typescriptlang.org/play/ that allows you to play around with and understand the features of TypeScript without installing it.
- **Terminal**: We will use a terminal to execute commands to create a TypeScript project. The default terminal available in your operating system will work fine.
- **Visual Studio Code**: We'll need a code editor to explore TypeScript. If you didn't install it in the last chapter, it can be installed from https://code.visualstudio.com/.
- **Node.js and npm**: TypeScript is dependent on these pieces of software. You can install them from https://nodejs.org/en/download/.

All the code snippets in this chapter can be found online at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter02.

Understanding the benefits of TypeScript

In this section, we will start by understanding what TypeScript is, how it relates to JavaScript, and how TypeScript enables teams to be more productive.

Understanding TypeScript

TypeScript was first released in 2012 and is still being developed, with new releases coming out every few months. But what is TypeScript, and what are its benefits?

TypeScript is often referred to as a superset or extension of JavaScript because any feature in JavaScript is available in TypeScript. Unlike JavaScript, TypeScript can't be executed directly in a browser – it must be transpiled into JavaScript first.

> ***NOTE***
>
> *It is worth being aware that a proposal is being considered that would allow TypeScript to be executed directly in a browser without transpilation. See the following link for more information: https://github.com/tc39/proposal-type-annotations.*

TypeScript adds a rich type system to JavaScript. It is often used with frontend frameworks such as Angular, Vue, and React. TypeScript can also be used to build a backend with Node.js, or even newer server runtimes such as Bun or Deno. This demonstrates how flexible TypeScript's type system is.

When a JavaScript code base grows, it can become hard to read and maintain. TypeScript's type system solves this problem. TypeScript uses the type system to allow code editors to catch type errors as developers write problematic code. Code editors also use the type system to provide productivity features, such as robust code navigation and code refactoring.

Next, we will step through an example of how TypeScript catches a type of error that JavaScript can't.

Catching type errors early

The type information helps the TypeScript compiler catch type errors. In code editors such as Visual Studio Code, a type error is underlined in red immediately after the developer has made a type mistake. Carry out the following steps to experience an example of TypeScript catching a type error:

1. Open Visual Studio Code in a folder of your choice.
2. Create a new file called `calculateTotalPrice.js` by choosing the **New File** option in the **Explorer** panel.

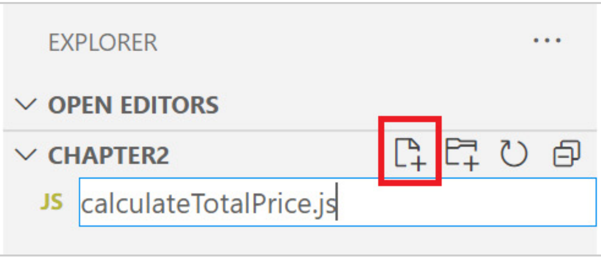


Figure 2.1 – Creating a new file in Visual Studio Code

3. Enter the following code into the file:

   ```
   function calculateTotalPriceJS(
     product,
     quantity,
     discount,
   ) {
     const priceWithoutDiscount =
       product.price * quantity;
     const discountAmount =
       priceWithoutDiscount * discount;
     return (
       priceWithoutDiscount -
       discountAmount
     );
   }
   ```

   The function calculates the total price for a product, as well as the quantity and discount passed into it.

   Remember that the code snippets are available online to copy. The link to the previous snippet is https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter02/understanding-typescript/calculateTotalPrice.js.

   There is a bug in the code that might be difficult to spot, and the error won't be highlighted by Visual Studio Code. The bug will become clear in step 5 after the code has been converted to TypeScript.

4. Now, create a copy of the file but with a **`.ts`** extension instead of **`.js`**. A file can be copied by right-clicking on it in the **Explorer** panel and selecting the **Copy** option. Then, right-click the **Explorer** panel again and select the **Paste** option to create the copied file.

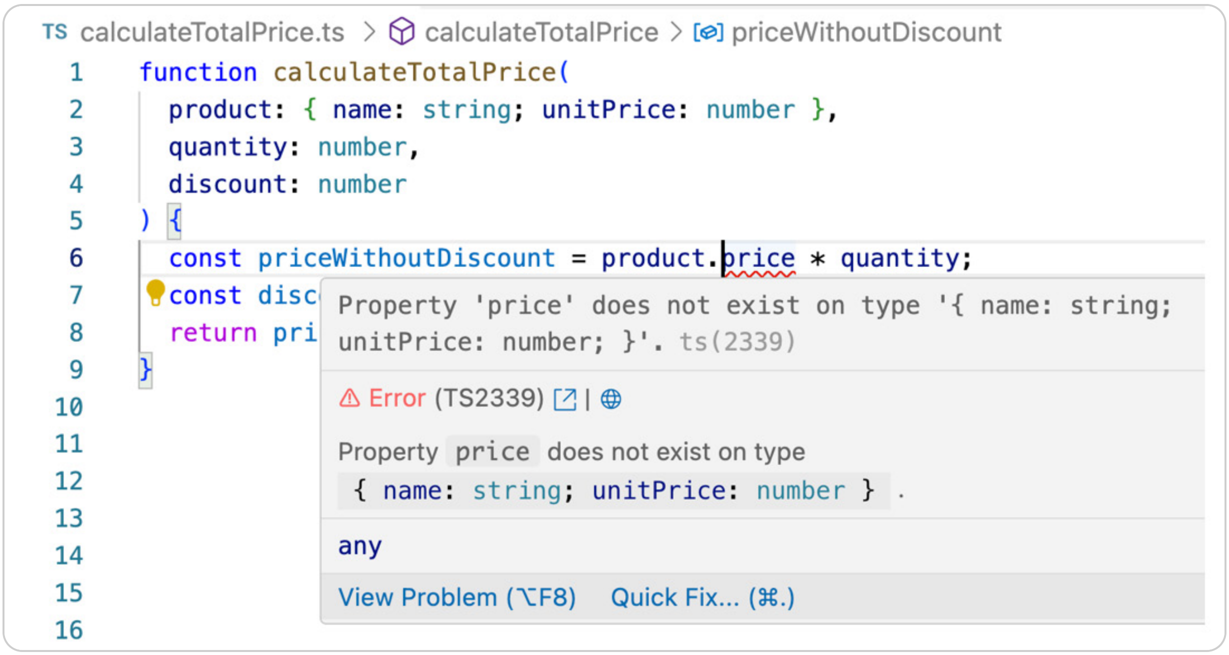
> **NOTE**
>
> *A **`.ts`** file extension denotes a TypeScript file. This means a TypeScript compiler will perform type checking on this file.*

5. In the **`calculateTotalPrice.ts`** file, remove the **`JS`** at the end of the function name and make the following highlighted updates to the code:

```
function calculateTotalPrice(
  product: { name: string; unitPrice: number
},
  quantity: number,
  discount: number
) {
  const priceWithoutDiscount = product.price *
quantity;
  const discountAmount = priceWithoutDiscount
* discount;
  return priceWithoutDiscount -
discountAmount;
}
```

Here, we have added TypeScript **type annotations** to the **`function`** parameters. We will learn about type annotations in detail in the next section.

The key point is that the type error is now highlighted by a red squiggly underline:



TS calculateTotalPrice.ts > calculateTotalPrice > priceWithoutDiscount

```
function calculateTotalPrice(
  product: { name: string; unitPrice: number },
  quantity: number,
  discount: number
) {
  const priceWithoutDiscount = product.price * quantity;
  const disc
  return pri
}
```

```
Property 'price' does not exist on type '{ name: string;
unitPrice: number; }'. ts(2339)
```

⚠ Error (TS2339)

Property `price` does not exist on type `{ name: string; unitPrice: number }` .

```
any
```

Figure 2.2 – Highlighted type error

The bug is that the function references a **`price`** property in the product object that doesn't exist. The property that should be referenced is **`unitPrice`**.

Catching these problems early in the development process increases the team's throughput and is one less thing for quality assurance to catch. It could be worse – the bug could have gotten into the live app and given users a bad experience.

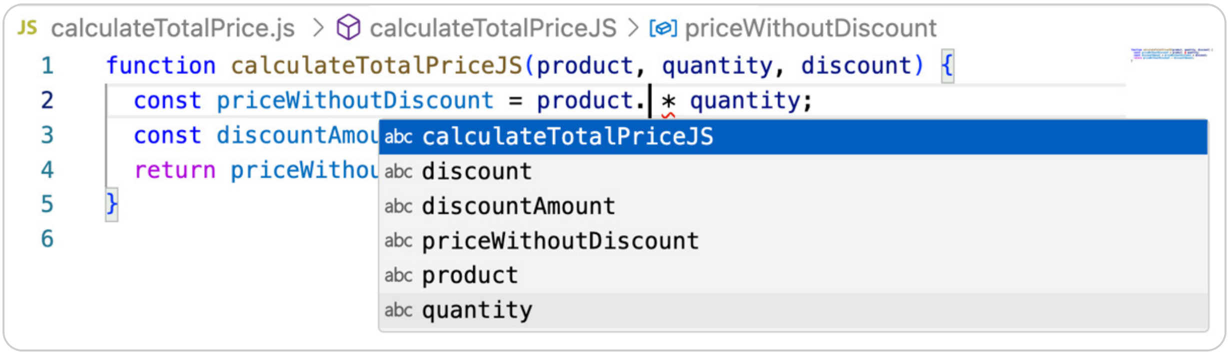
Keep these files open in Visual Studio Code because we will run through an example of TypeScript improving the developer experience next.

Improving developer experience and productivity with IntelliSense

IntelliSense is a feature in code editors that gives useful information about elements of code and allows code to be quickly completed. For example, IntelliSense can provide the list of properties available in an object.

Carry out the following steps to experience how TypeScript works better with IntelliSense than JavaScript and how this positively impacts productivity. As part of this exercise, we will fix the price bug from the previous section:

1. Open **`calculateTotalPrice.js`**, and on line 2, where **`product.price`** is referenced, remove **`price`**. Then, with the cursor after the dot (**`.`**), click *Ctrl* + spacebar. This opens Visual Studio Code's IntelliSense:

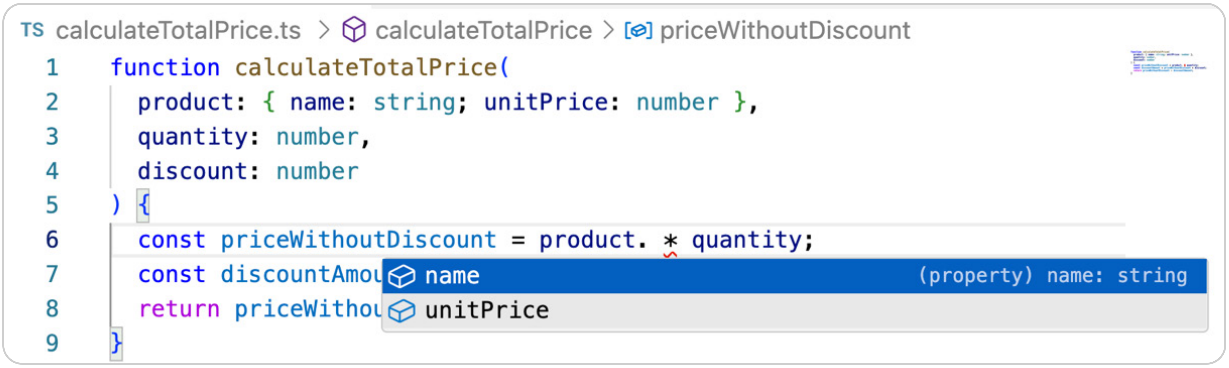


```
JS calculateTotalPrice.js > calculateTotalPriceJS > priceWithoutDiscount
function calculateTotalPriceJS(product, quantity, discount) {
  const priceWithoutDiscount = product. * quantity;
  const discountAmou
  return priceWithou
}
```

Figure 2.3 – IntelliSense in a JavaScript file

Visual Studio Code can only guess the potential property name, so it lists variable names and function names it has seen in the file. Unfortunately, IntelliSense doesn't help in this case because the correct property name, **`unitPrice`**, is not listed.

2. Now, open **`calculateTotalPrice.ts`**, remove **`price`** from **`product.price`**, and press *Ctrl* + spacebar to open IntelliSense again:



```
TS calculateTotalPrice.ts > calculateTotalPrice > priceWithoutDiscount
function calculateTotalPrice(
  product: { name: string; unitPrice: number },
  quantity: number,
  discount: number
) {
  const priceWithoutDiscount = product. * quantity;
  const discountAmou
  return priceWithou
}
```

Figure 2.4 – IntelliSense in a TypeScript file

This time, Visual Studio Code lists the correct properties.

3. Select **unitPrice** from IntelliSense to resolve the type error.

IntelliSense is just one tool that TypeScript provides. It can also provide robust refactoring features, such as renaming React components, and helps with accurate code navigation, such as going to a function definition.

To recap, we learned the following in this section:

- TypeScript's type-checking feature helps catch problems earlier in the development process
- TypeScript enables code editors to offer productivity features such as IntelliSense
- These advantages provide significant benefits when working in larger code bases

Next, we will learn about the type system in JavaScript. This will further underline the need for TypeScript in a large code base.

Understanding JavaScript types

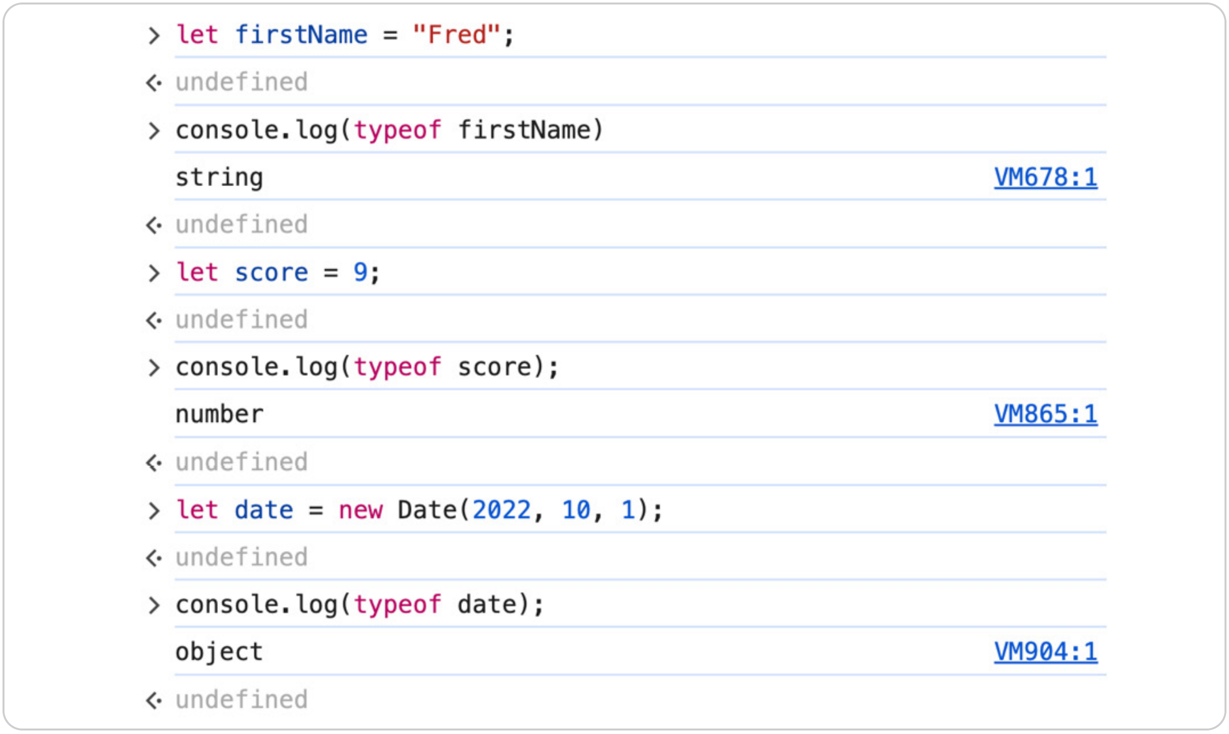
Before understanding the type system in TypeScript, let's briefly explore the type system in JavaScript. To do this, open a browser and carry out the following steps:

1. Open the browser development tools (*F12* on Windows or *Cmd + Option + I* on Mac) and go to the **Console** panel.
2. Enter the following lines into the console:

```
let firstName = “Fred”;
console.log(typeof firstName);
let score = 9;
console.log(typeof score);
let date = new Date(2022, 10, 1);
console.log(typeof date);
```

The code assigns three variables to various values. The code also outputs the variable values to the console, along with their JavaScript type.

Here’s the console output:



```
> let firstName = "Fred";
<· undefined
> console.log(typeof firstName)
  string                                         VM678:1
<· undefined
> let score = 9;
<· undefined
> console.log(typeof score);
  number                                         VM865:1
<· undefined
> let date = new Date(2022, 10, 1);
<· undefined
> console.log(typeof date);
  object                                         VM904:1
<· undefined
```

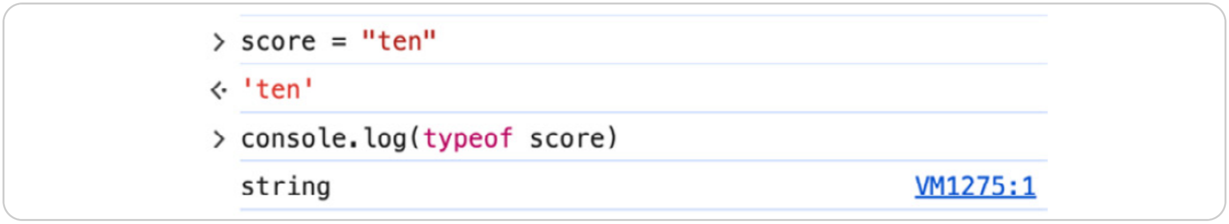
Figure 2.5 – Some JavaScript types

It isn’t surprising that **`firstName`** is a string and **`score`** is a number. However, it is a little surprising that **`date`** is an object rather than something more specific, such as a date.

3. Let's add another couple of lines of code to the console:

```
score = “ten”;
console.log(typeof score);
```

Again, the console output is a little surprising:



```
> score = "ten"
<· 'ten'
> console.log(typeof score)
  string                                      VM1275:1
```

Figure 2.6 – Variable changing type

The **`score`** variable has changed from a **`number`** type to a **`string`** type! This is because JavaScript is loosely typed.

A key point is that JavaScript only has a minimal set of types, such as **`string`**, **`number`**, and **`boolean`**. It is worth noting that all of the JavaScript types are available in TypeScript because Typescript is a superset of JavaScript.

Also, JavaScript allows a variable to change its type – meaning that the JavaScript engine won't throw an error if a variable is changed to a completely different type. This loose typing makes it impossible for code editors to catch type errors.

NOTE

For more information on JavaScript types, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures.

Now that we understand the limitations of the type system in JavaScript, we will learn about TypeScript's type system, starting with basic types.

Using basic TypeScript types

In this section, we'll start by understanding how TypeScript types can be declared and how they are inferred from assigned values. We will then learn the basic types commonly used in TypeScript that aren't available in JavaScript and understand helpful use cases.

Using type annotations

TypeScript type annotations enable variables to be declared with specific types. These allow the TypeScript compiler to check that the code adheres to these types. In short, type annotations allow TypeScript to catch bugs where our code uses the wrong type much earlier than we would if we were writing our code in JavaScript.

Open TypeScript Playground at https://www.typescriptlang.org/play and carry out the following steps to explore type annotations:

1. Remove any existing code in the left-hand pane and enter the following variable declaration:

   ```
   let unitPrice: number;
   ```

   The type annotation comes after the variable declaration. It starts with a colon followed by the type we want to assign to the variable. In this case, **`unitPrice`** is going to be a **`number`** type. Remember that **`number`** is a type in JavaScript, which means that it is available for us to use in TypeScript too.

   The transpiled JavaScript appears on the right-hand pane as follows:

   ```
   let unitPrice;
   ```

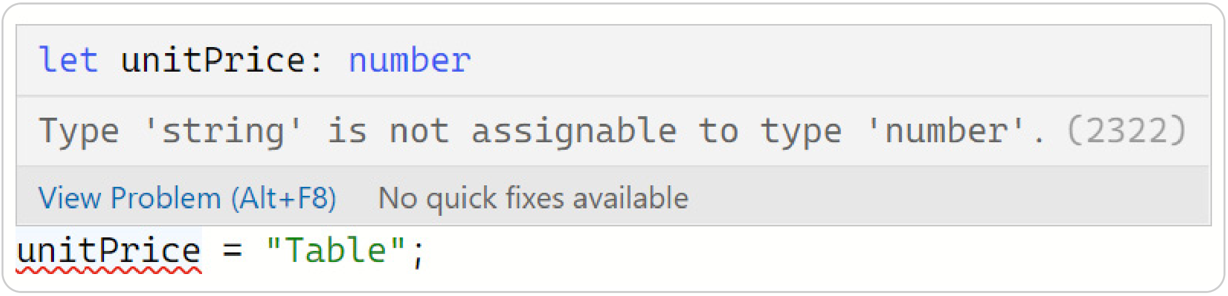
   However, notice that the type annotation has disappeared. This is because type annotations don't exist in JavaScript.

> **NOTE**
>
> *You may also see* **`"use strict";`** *at the top of the transpiled JavaScript. This means that the JavaScript will be executed in JavaScript strict mode, which will pick up more coding mistakes. For more information on JavaScript strict mode, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode.*

2. Add a second line to the program:

   ```
   unitPrice = "Table";
   ```

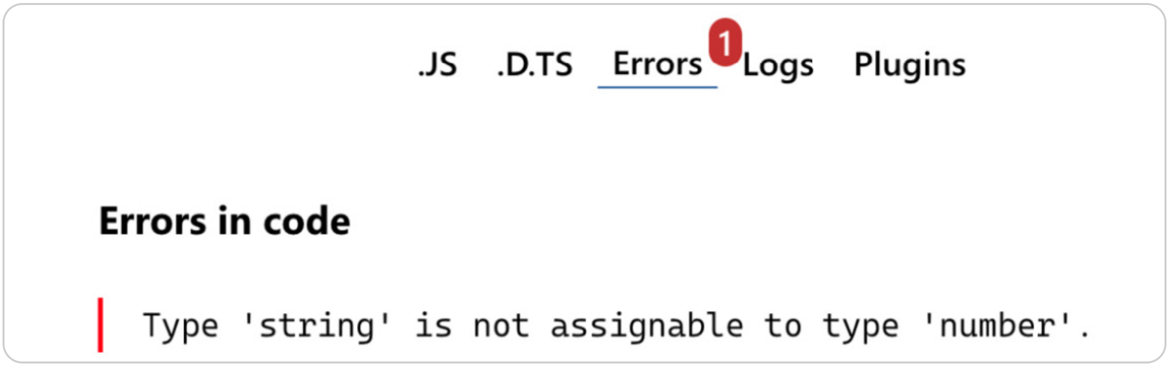
Notice that a red line appears under **`unitPrice`** on this line. If you hover over the underlined **`unitPrice`**, a type error is described:



```
let unitPrice: number
Type 'string' is not assignable to type 'number'. (2322)
View Problem (Alt+F8)    No quick fixes available
unitPrice = "Table";
```

Figure 2.7 – A type error being caught

3. The type error also appears in the **Errors** tab in the right-hand pane. There is a red badge containing **1** notifying us that there is one error. Click the **Errors** tab to discover the same error message that was reported in the previous step.



Errors in code

```
Type 'string' is not assignable to type 'number'.
```

Figure 2.8 – Type error reported on the Errors tab

4. You can also add type annotations to function parameters and a function's return value using the same syntax as annotating a variable. As an example, enter the following function into TypeScript Playground:

```
function getTotal(
  unitPrice: number,
  quantity: number,
  discount: number
): number {
  const priceWithoutDiscount = unitPrice *
quantity;
  const discountAmount = priceWithoutDiscount
* discount;
  return priceWithoutDiscount -
discountAmount;
}
```

We've declared the **`unitPrice`**, **`quantity`**, and **`discount`** parameters, all with a **`number`** type. The return type annotation comes after the function's parentheses, which is also a **`number`** type in the preceding example.

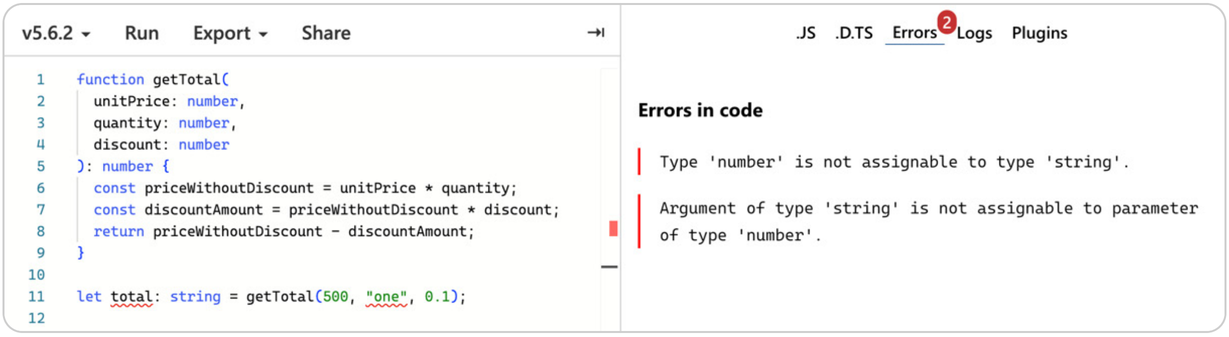
> **NOTE**
>
> *We have used both **`const`** and **`let`** to declare variables in different examples. **`let`** allows the variable to change the value after the declaration, whereas **`const`** variables can't change. In the preceding function, **`priceWithoutDiscount`** and **`discountAmount`** never change the value after the initial assignment, so we have used **`const`**.*

5. Add another line of code to call **`getTotal`** with an incorrect type for **`quantity`**. Assign the result of the call to **`getTotal`** to a variable

with an incorrect type:

```
let total: string = getTotal(500, "one", 0.1);
```

Both errors are immediately detected and highlighted:



```
function getTotal(
  unitPrice: number,
  quantity: number,
  discount: number
): number {
  const priceWithoutDiscount = unitPrice * quantity;
  const discountAmount = priceWithoutDiscount * discount;
  return priceWithoutDiscount - discountAmount;
}

let total: string = getTotal(500, "one", 0.1);
```

Errors in code

```
Type 'number' is not assignable to type 'string'.

Argument of type 'string' is not assignable to parameter
of type 'number'.
```

Figure 2.9 – Both type errors being caught

This strong type checking is something that we don't get in JavaScript, and it is very useful in large code bases because it helps us immediately detect type errors.

Next, we will learn how TypeScript doesn't always need type annotations in order to type check code.

Using type inference

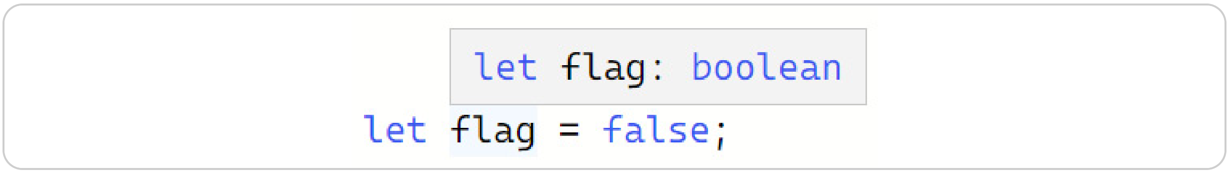
Type annotations are really valuable, but they require additional code to be written. This extra code takes time to write. Luckily, TypeScript's powerful **type inference** system means type annotations don't need to be specified all the time. TypeScript infers the type of a variable from its assigned value.

Explore type inference by carrying out the following steps in TypeScript Playground:

1. First, remove any previous code and then add the following line:

   ```
   let flag = false;
   ```

2. Hover over the **`flag`** variable. A tooltip will appear showing the type that **`flag`** has been inferred to:



```
let flag: boolean
let flag = false;
```

Figure 2.10 – Hovering over a variable reveals its type

3. Add another line beneath this to incorrectly set **`flag`** to an invalid value:

   ```
   flag = "table";
   ```

   A type error is immediately caught, just like when we used a type annotation to assign a type to a variable.

Type inference is an excellent feature of TypeScript and prevents the code bloat that lots of type annotations would bring. Therefore, it is common practice to use type inference and only revert to using type annotations where inference isn't possible.

Next, we will look at the **`Date`** type in TypeScript.

Using the Date type

We are already aware that a **`Date`** type doesn't exist in JavaScript, but luckily, a **`Date`** type does exist in TypeScript. The TypeScript **`Date`** type is a representation of the JavaScript **`Date`** object.

> ### *NOTE*
>
> *See the following link for more information on the JavaScript* ***`Date`*** *object: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date.*

To explore the TypeScript **`Date`** type, carry out the following steps in TypeScript Playground:

1. First, remove any previous code and then add the following lines:

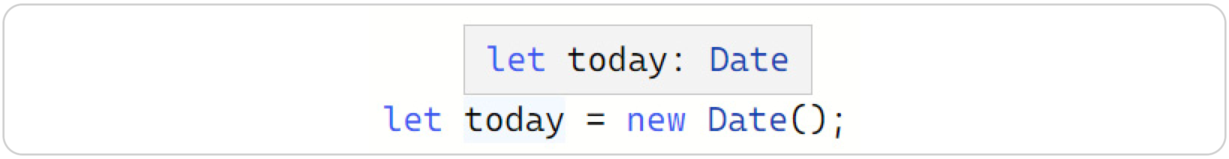
   ```
   let today: Date;
   today = new Date();
   ```

   A variable named **`today`** is declared with a **`Date`** type assigned to it. The value of the variable is set to today's date.

2. Refactor these two lines into the following single line that uses type inference rather than a type annotation:

   ```
   let today = new Date();
   ```

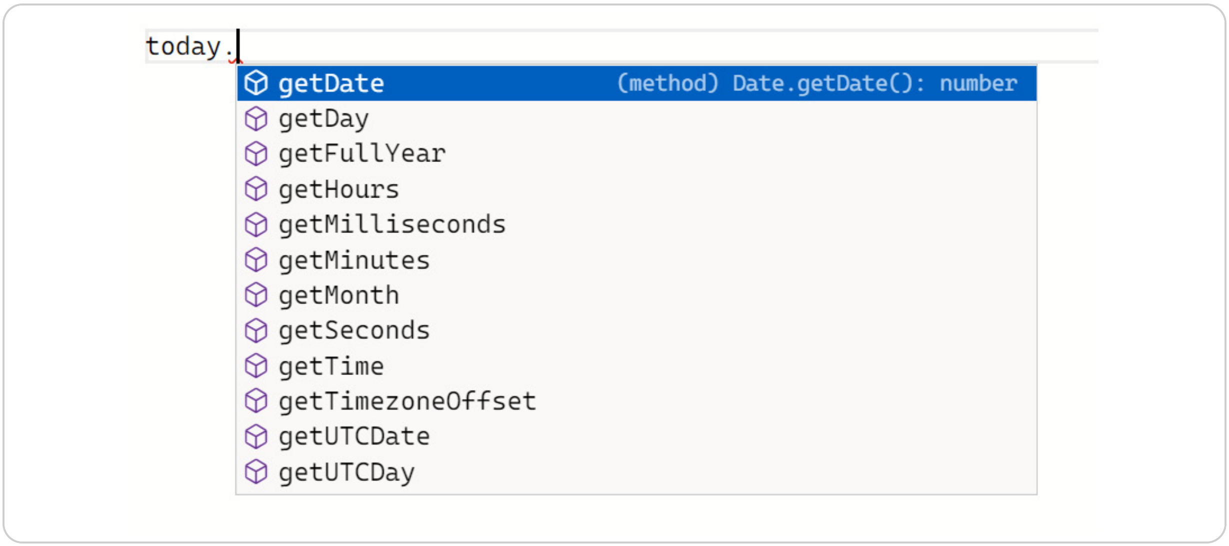
3. Check that **`today`** has been assigned the **`Date`** type by hovering over it and checking the tooltip:



```
let today: Date
let today = new Date();
```

Figure 2.11 – Confirmation that today has inferred the Date type

4. Now, check IntelliSense is working by adding **`today.`** on a new line:



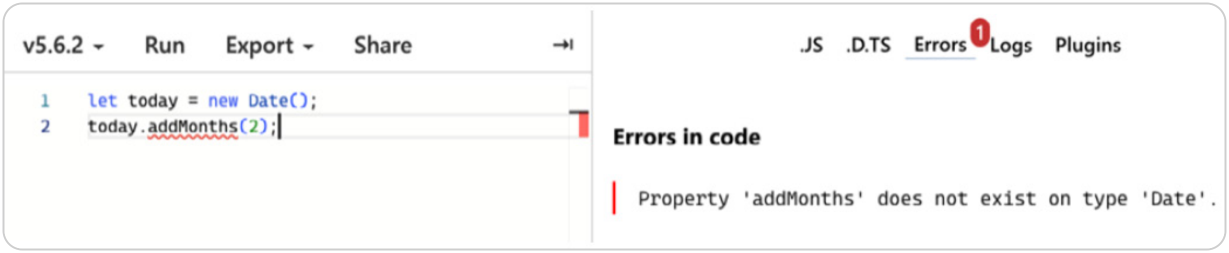
```
today.
getDate            (method) Date.getDate(): number
getDay
getFullYear
getHours
getMilliseconds
getMinutes
getMonth
getSeconds
getTime
getTimezoneOffset
getUTCDate
getUTCDay
```

Figure 2.12 – IntelliSense working nicely on a date

5. Remove this line and add a slightly different line of code:

```
today.addMonths(2);
```

An **`addMonths`** function doesn't exist in the **`Date`** object, so a type error is raised:



```
let today = new Date();
today.addMonths(2);
```

Errors in code

```
Property 'addMonths' does not exist on type 'Date'.
```

Figure 2.13 – Type error caught on a date

In summary, the **`Date`** type has all the features we expect – inference, IntelliSense, and type checking – which are really useful when working with dates.

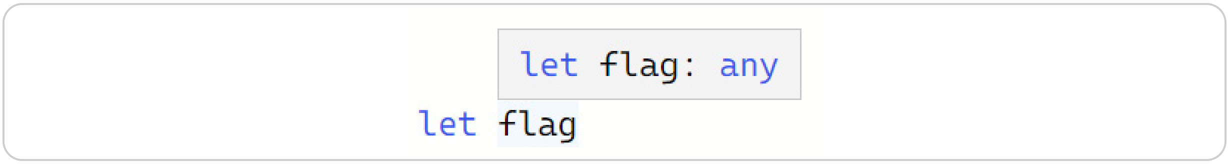
Next, we will learn about an escape hatch to TypeScript's type system.

Using the any type

What if we declare a variable with no type annotation and no value? What will TypeScript infer as the type? Let's find out by entering the following code in TypeScript Playground:

```
let flag;
```

Now, hover the mouse over **`flag`**:



```
let flag: any
let flag
```

Figure 2.14 – Variable given the any type

So, TypeScript gives a variable with no type annotation and no immediately assigned value the **`any`** type. It is a way of opting out of performing type checking on a particular variable and is commonly used for dynamic content or values from third-party libraries.

However, TypeScript's increasingly powerful type system means that we need to use **`any`** less often these days.

Instead, there is a better alternative: the **`unknown`** type.

Using the unknown type

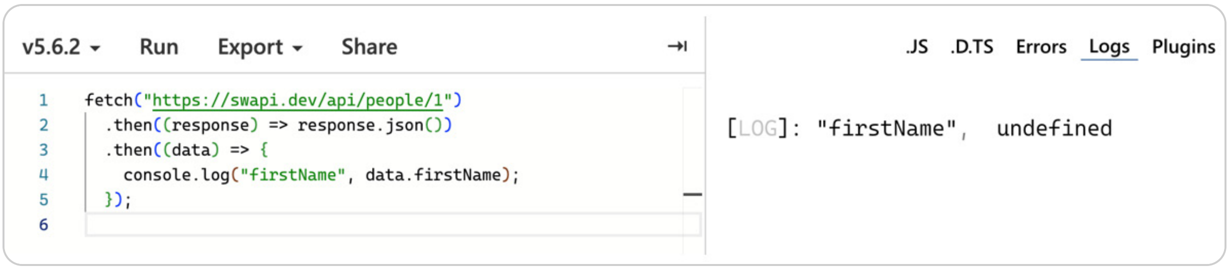
`unknown` is a type we can use when we are unsure of the type but want to interact with it in a strongly typed manner. Carry out the following steps to explore how this is a better alternative to the **`any`** type:

1. In TypeScript Playground, remove any previous code and enter the following:

   ```
   fetch("https://swapi.dev/api/people/1")
     .then((response) => response.json())
     .then((data) => {
       console.log("firstName", data.firstName);
     });
   ```

   The code fetches a Star Wars character from a web API. No type errors are raised, so the code appears okay.

2. Now, click on the **Run** option to execute the code:



```
fetch("https://swapi.dev/api/people/1")
  .then((response) => response.json())
  .then((data) => {
    console.log("firstName", data.firstName);
  });
```

```
[LOG]: "firstName",  undefined
```

Figure 2.15 – firstName property has an undefined value

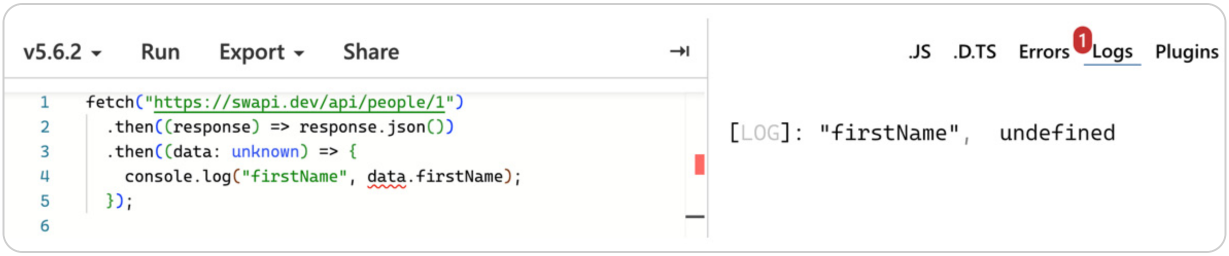
The **firstName** property doesn't appear to be in the fetched data because it is **undefined** when it is output to the console.

Why wasn't a type error raised on line 4 where **firstName** was referenced? Well, **data** is of type **any**, which means no type checking will occur on it. You can hover over **data** to confirm that it has been given the **any** type.

3. Give **data** the **unknown** type annotation:

```
fetch("https://swapi.dev/api/people/1")
 .then((response) => response.json())
 .then((data: unknown) => {
   console.log("firstName", data.firstName);
 });
```

A type error is now raised where **firstName** is referenced:



```
fetch("https://swapi.dev/api/people/1")
  .then((response) => response.json())
  .then((data: unknown) => {
    console.log("firstName", data.firstName);
  });
```

```
[LOG]: "firstName",  undefined
```

Figure 2.16 – Type error on unknown data parameter

The **unknown** type is the opposite of the **any** type, as it contains nothing within its type. A type that doesn't contain anything may seem useless. However, a variable's type can be widened if checks are made to allow TypeScript to widen it.

4. Before we give TypeScript information to widen **data**, change the property referenced within it from **firstName** to **name**:

```
fetch("https://swapi.dev/api/people/1")
  .then((response) => response.json())
  .then((data: unknown) => {
    console.log("name", data.name);
  });
```

   name is a valid property, but a type error is still occurring. This is because **data** is still **unknown**.

5. Now, make the highlighted changes to the code to widen the **data** type:

```
fetch("https://swapi.dev/api/people/1")
  .then((response) => response.json())
  .then((data: unknown) => {
    if (isCharacter(data)) {
      console.log("name", data.name);
    }
  });
function isCharacter(
  character: any
): character is { name: string } {
  return "name" in character;
}
```

   The code snippet can be copied from https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter02/using-basic-typescript-types/using-the-unknown-type/code.ts.

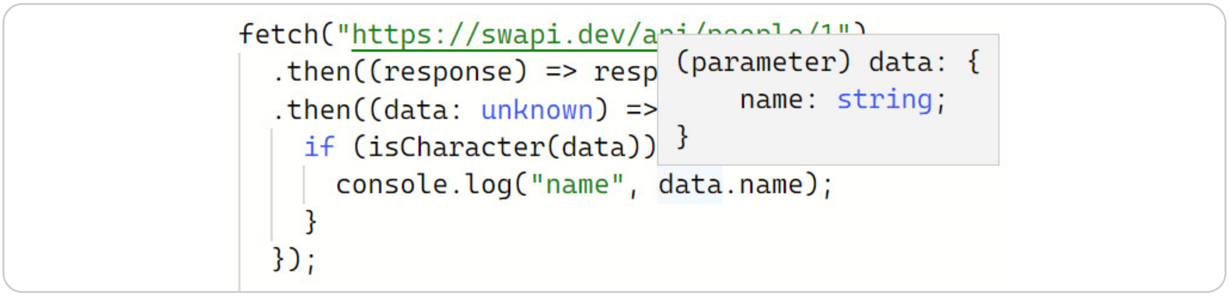
The **`if`** statement uses a function called **`isCharacter`** to verify that a **`name`** property is contained within the object. The result of this call is **`true`** in this example, so the logic will flow into the **`if`** branch.

Notice the return type of **`isCharacter`**, which is as follows:

```
character is { name: string }
```

This is a **type predicate**. TypeScript will narrow or widen the type of **`character`** to **`{ name: string }`** if the function returns **`true`**. The type predicate is **`true`** in this example, so **`character`** is widened to an object with a **`name`** string property.

6. Hover over the **`data`** variable on each line where it is referenced. **`data`** starts off with the **`unknown`** type where it is assigned with a type annotation. Then, it is widened to **`{name: string}`** inside the **`if`** branch:



```
fetch("https://swapi.dev/api/people/1")
  .then((response) => resp
  .then((data: unknown) =>
    if (isCharacter(data))
      console.log("name", data.name);
    }
  });
```

```
(parameter) data: {
    name: string;
}
```

Figure 2.17 – Widened type given to data

Notice that the type error has also disappeared. Nice!

7. Next, run the code. You will see **Luke Skywalker** output to the console.

In summary, the **`unknown`** type is an excellent choice for data whose type you are unsure about. However, you can't interact with **`unknown`** variables – the variable must be widened to a different type before any interaction.

Next up, let's cover arrays.

Using arrays

Arrays are structures that TypeScript inherits from JavaScript. We add type annotations to arrays as usual, but with square brackets, **`[]`**, at the end to denote that this is an array type.

Let's explore an example in TypeScript Playground:

1. Remove any existing code and enter the following:

   ```
   const numbers: number[] = [];
   ```

   Alternatively, the **`Array`** generic type syntax can be used:

   ```
   const numbers: Array<number> = [];
   ```

   We will learn about generics in TypeScript in *Chapter 11*, *Reusable Components*.

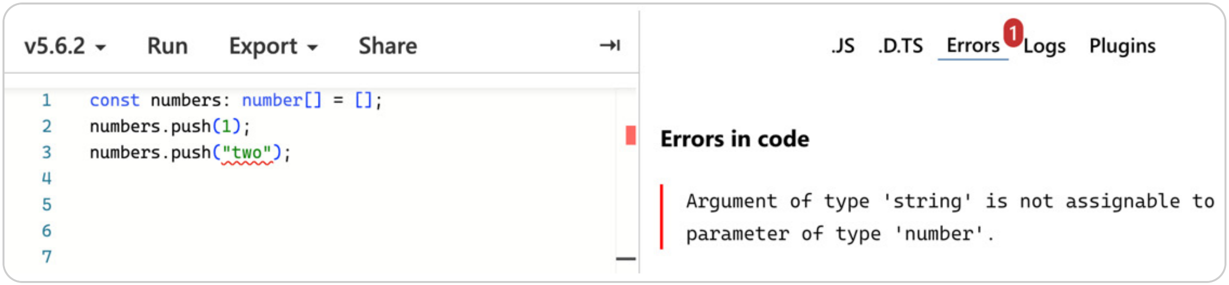
2. Add **`1`** to the array by using the array's **`push`** function:

   ```
   numbers.push(1);
   ```

3. Now add a string to the array:

```
numbers.push("two");
```

A type error is raised as we would expect:



```
const numbers: number[] = [];
numbers.push(1);
numbers.push("two");
```

Errors in code

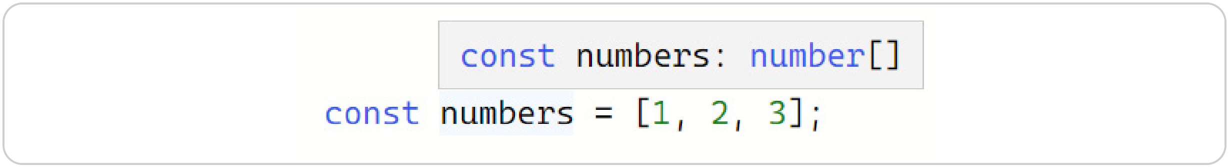
```
Argument of type 'string' is not assignable to
parameter of type 'number'.
```

Figure 2.18 – Type error when adding a string type to a number array

4. Now replace all the code with the following:

```
const numbers = [1, 2, 3];
```

5. Hover over **`numbers`** to verify that TypeScript has inferred its type to be **`number[]`**.



```
const numbers: number[]
const numbers = [1, 2, 3];
```

Figure 2.19 – Array type inference

Excellent – we can see that TypeScript's type inference works with arrays!

Arrays are one of the most common types used to structure data. In the preceding examples, we've only used an array with elements that have a `number` type, but any type can be used for elements, including objects, which have their own properties.

Here's a recap of all the basic types we have learned about in this section:

- TypeScript adds many useful types to JavaScript types, such as `Date`, and is capable of representing arrays.
- TypeScript can infer a variable's type from its assigned value. A type annotation can be used where type inference doesn't give the desired type.
- No type checking occurs on variables with the `any` type, so this type should be avoided.
- The `unknown` type is a strongly typed alternative to `any`, but `unknown` variables must be widened to be interacted with.
- Array types can be defined using square brackets after the array item type.

In the next section, we will learn how to create our own types.

Creating TypeScript types

The last section showed that TypeScript has a great set of standard types. In this section, we will learn how to create our own types. We will start by learning how to create types for objects before learning how to create types for variables that hold a range of values.

Using object types

Objects are very common in JavaScript programs, so learning how to represent them in TypeScript is really important. In fact, we already used an object type earlier in this chapter for the **`product`** parameter in the **`calculateTotalPrice`** function. Here is a reminder of the **`product`** parameter's type annotation:

```
function calculateTotalPrice(
  product: { name: string; unitPrice: number },
  ...
) {
  ...
}
```

An object type in TypeScript is represented a bit like a JavaScript object literal. However, instead of property values, property types are specified instead. Properties in the object definitions can be separated by semicolons or commas, but using a semicolon is common practice.

Clear any existing code in TypeScript Playground and follow this example to explore object types:

1. Enter the following variable assignment to an object:

   ```
   let table = {name: "Table", unitPrice: 450};
   ```

   If you hover over the **`table`** variable, you'll see it is inferred to be the following type:

```
{
  name: string;
  unitPrice: number;
}
```

So, type inference works nicely for objects.

2. Now, on the next line, try to set a **`discount`** property to **`10`**:

```
table.discount = 10;
```

A **`discount`** property doesn't exist in the type, though – only the **`name`** and **`unitPrice`** properties exist. So, a type error occurs.

3. Let's say we want to represent a **`product`** object containing the **`name`** and **`unitPrice`** properties, but we want **`unitPrice`** to be optional. Remove the existing code and replace it with the following:

```
const table: { name: string; unitPrice: number
} = {
  name: "Table",
};
```

4. This raises a type error because **`unitPrice`** is a required property in the type annotation. We can use a **`?`** symbol as follows to make this optional rather than required:

```
const table: { name: string; unitPrice?:
number } = {
  name: "Table",
};
```

The type error disappears.

> **NOTE**
>
> *The* ***`?`*** *symbol can be used in functions for optional parameters, for example,* ***`myFunction(requiredParam: string, optionalParam?: string)`****.*

Now, let's learn a way to streamline object type definitions.

Creating type aliases

The type annotation we used in the last example was quite lengthy and would be longer for more complex object structures. Also, having to write the same object structure to assign to different variables is a little frustrating:

```
const table: { name: string; unitPrice?: number }
= ...;
const chair: { name: string; unitPrice?: number }
= ...;
```

Type aliases solve these problems. As the name suggests, a type alias refers to another type, and the syntax is as follows:

```
type YourTypeAliasName = AnExistingType;
```

Open TypeScript Playground and follow along to explore type aliases:

1. Start by creating a type alias for the product object structure we used in the last example:

   ```
   type Product = { name: string; unitPrice?:
   number };
   ```

2. Now assign two variables to this **`Product`** type:

   ```
   let table: Product = { name: "Table" };
   let chair: Product = { name: "Chair",
   unitPrice: 40 };
   ```

   That's much cleaner!

3. A type alias can extend another object using the **`&`** symbol. Create a second type for a discounted product by adding the following type alias:

   ```
   type DiscountedProduct = Product & { discount:
   number };
   ```

   `DiscountedProduct` represents an object containing **`name`**, **`unitPrice`** (optional), and **`discount`** properties.

NOTE

A type that extends another using the **`&`** *symbol is referred to as an* ***intersection type****.*

4. Add the following variable with the **`DiscountedProduct`** type, as follows:

   ```
   let chairOnSale: DiscountedProduct = {
   ```

```
    name: "Chair on Sale",
    unitPrice: 30,
    discount: 5,
  };
```

5. A type alias can also be used to represent a function. Add the following type alias to represent a function:

```
type Purchase = (quantity: number) => void;
```

   The preceding type represents a function containing a **`number`** parameter and doesn't return anything.

> **NOTE**
>
> *The **`void`** type is used to indicate that a function doesn't return a value.*

6. Use the **`Purchase`** type to create a **`purchase`** function property in the **`Product`** type, as follows:

```
type Purchase = (quantity: number) => void;
type Product = {
  name: string;
  unitPrice?: number;
  purchase: Purchase;
};
```

   Type errors will be raised on the **`table`**, **`chair`**, and **`chairOnSale`** variable declarations because the **`purchase`** function property is required.

7. Add a **`purchase`** function property to the **`table`** variable declarations, as follows:

   ```
   let table: Product = {
     name: "Table",
     purchase: (quantity) =>
       console.log(`Purchased ${quantity}
   tables`),
   };
   table.purchase(4);
   ```

   The type error is resolved on the **`table`** variable declaration.

8. A **`purchase`** property could be added in a similar way to the **`chair`** and **`chairOnSale`** variable declarations to resolve their type errors. However, ignore these type errors for this exploration and move on to the next step.

9. Click the **Run** option to run the code that purchases four tables. "**Purchased 4 tables**" is output to the console.

In summary, type aliases allow existing types to be composed together and improve the readability and reusability of types. We will use type aliases extensively in this book.

Next, we will learn how to create a type to represent a range of values.

Creating union types

A **union type** is the mathematical union of multiple other types to create a new type and can represent a range of values. Type aliases can be used to create union types.

An example of a union type is as follows:

```
type Level = "H" | "M" | "L";
```

A variable of type **`Level`** can contain the values **`"H"`**, **`"M"`**, or **`"L"`**.

Clear any existing code in TypeScript Playground, and let's play around with union types:

1. Start by creating a type to represent **`"red"`**, **`"green"`**, or **`"blue"`**:

   ```
   type RGB = "red" | "green" | "blue";
   ```

   Note that this type is a union of strings, but a union type can consist of any type – even mixed types!

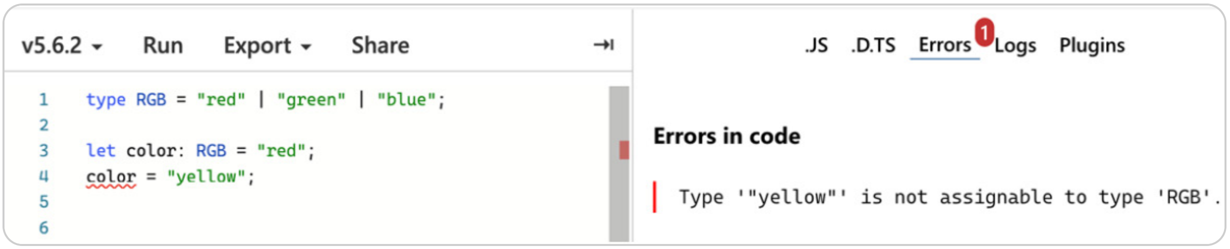
2. Create a variable with the **`RGB`** type and assign a valid value:

   ```
   let color: RGB = "red";
   ```

3. Now try assigning a value outside the type:

   ```
   color = "yellow";
   ```

   A type error occurs, as expected:



```
type RGB = "red" | "green" | "blue";

let color: RGB = "red";
color = "yellow";
```

Errors in code

```
Type '"yellow"' is not assignable to type 'RGB'.
```

Figure 2.20 – Type error on the union type

4. Now try to set **`color`** to **`null`**:

   ```
   color = null;
   ```

   As we would expect, this still creates an error.

5. Union types can reference multiple types. Let's add **`null`** to our **`RGB`** type, as follows:

   ```
   type RGB = "red" | "green" | "blue" | null;
   ```

   The **`color`** assignment to **`null`** now no longer raises an error because **`null`** is allowed within the **`RGB`** type.

That completes our exploration of union types. We'll use them extensively throughout this book.

Here's a recap of what we have learned about creating types:

- Union types are a great way of representing a specific set of strings. They also allow a variable to hold values of multiple types.
- Type aliases allow new, reusable types to be created and can be used for objects, functions, and union types.
- An existing type alias can be extended using the **`&`** symbol.

- The **?** symbol can specify that an object property or function parameter is optional.

Now that we have covered types, next, we will learn about the TypeScript compiler.

Using the TypeScript compiler

In this section, we will learn how to use the TypeScript compiler to type check code and transpile it into JavaScript. First, we will use Visual Studio Code to create a simple TypeScript project containing a simple function. We will then use the terminal within Visual Studio Code to interact with the TypeScript compiler.

Open Visual Studio Code in a blank folder of your choice, and carry out the following steps:

1. In the **Explorer** panel in Visual Studio Code, create a file called **package.json** containing the following content:

```
{
  “name”: “tsc-play”,
  “dependencies”: {
    “typescript”: “*”
  },
  “scripts”: {
    “build”: “tsc src/welcome.ts”
  }
}
```

The file defines a project name of **`tsc-play`** and sets the latest version of TypeScript as the only dependency. The file also defines an npm script called **`build`**, which will invoke the TypeScript compiler (**`tsc`**), passing it a **`welcome.ts`** file in the **`src`** folder. Don't worry that **`welcome.ts`** doesn't exist – we will create it in step 3.

2. In a terminal, navigate to the project folder and install the dependencies using the following command:

```
npm i
```

This will install all the libraries listed in the **`dependencies`** section of **`package.json`**. So, this will install TypeScript.

3. Create a folder called **`src`** and then create a file called **`welcome.ts`** within it.

4. Open **`welcome.ts`** and add the following content:

```
function welcome(name: string | null) {
    if (name === null) {
      return `Welcome!`;
    }
    return `Welcome, ${name}!`;
};
```

The function takes in a name and constructs a string that welcomes that name. If no name exists, a generic welcome is returned.

5. Enter the following command in the terminal:

   ```
   npm run build
   ```

   This will run the npm **`build`** script we defined in the first step.

   After the command finishes, notice that a **`welcome.js`** file appears next to **`welcome.ts`** in the **`src`** folder.

6. Open the transpiled **`welcome.js`** file and read the content. It will look as follows:

   ```
   function welcome(name) {
     if (name === null) {
       return "Welcome!";
     }
     return "Welcome, ".concat(name, "!");
   };
   ```

   Notice that the type annotations have been removed because they aren't valid JavaScript. Notice also that it has been transpiled to JavaScript, capable of running in very old browsers.

   The default configuration that the TypeScript compiler uses isn't ideal. For example, we probably want the transpiled JavaScript in a completely separate folder and are likely to want to target newer browsers, resulting in less JavaScript code.

7. The TypeScript compiler can be configured using a file called **`tsconfig.json`**. Add a **`tsconfig.json`** file at the root of the project, containing the following code:

```
{
  "compilerOptions": {
    "outDir": "dist",
    "target": "esnext",
    "module": "esnext",
    "lib": ["DOM", "esnext"],
    "strict": true,
    "jsx": "react",
    "moduleResolution": "node",
    "noEmitOnError": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "build"]
}
```

This code can be copied from https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter02/using-the-typescript-compiler/tsconfig.json.

Here's an explanation of each setting in the **`compilerOptions`** field:

- **`outDir`**: This is the folder that the transpiled JavaScript is placed in.
- **`target`**: This is the version of JavaScript we want to transpile to. The **`esnext`** target means the next version. This means transpilation uses modern JavaScript features instead of polyfilling them, reducing the size of the outputted code.

- **`Module`**: This is a module system within the code. The **`esnext`** module means standard JavaScript modules.
- **`Lib`**: Gives the standard library types included in the type-checking process. **`DOM`** gives the browser DOM API types, and **`esnext`** gives types for APIs in the next version of JavaScript.
- **`Strict`**: When set to **`true`**, it means the strictest level of type checking. This includes the **`strictNullChecks`** option, which means **`null`** is required in types when needed. For example, the following statement would error when **`Strict`** (and **`strictNullChecks`**) is **`true`** but not when **`false`**:

  ```
  let firstName: string = null;
  ```
- **`Jsx`**: When set to **`React`**, it allows the compiler to transpile React's JSX.
- **`moduleResolution`**: This is how dependencies are found. We want TypeScript to look in the **`node_modules`** folder, so we have chosen **`node`**.
- **`noEmitOnError`**: When set to **`true`**, it means the transpilation won't happen if a type error is found.

The **`include`** field specifies the TypeScript files to compile, and the **`exclude`** field specifies the files to exclude.

> ***NOTE***

> *For more information on the TypeScript compiler options, see the following link: https://www.typescriptlang.org/tsconfig.*

8. The TypeScript compiler configuration now specifies all files in the **src** folder to be compiled. So, remove the file path on the **build** script in **package.json**:

   ```
   {
     ...,
     "scripts": {
       "build": "tsc"
     }
   }
   ```

9. Delete the previous transpiled **welcome.js** in the **src** folder.

10. Rerun the **build** command in the terminal:

    ```
    npm run build
    ```

    This time, the transpiled file is placed in a **dist** folder – a copy of the transpiled function is provided here:

    ```
    function welcome(name) {
      if (name === null) {
        return `Welcome!`;
      }
      return `Welcome, ${name}!`;
    }
    ```

    You will notice that the transpiled JavaScript now uses backticks for the welcome string, which is supported in modern browsers.

11. The final thing we are going to try is a type error. Open **`welcome.ts`** and add a **`number`** return type to the function:

    ```
    function welcome(name: string | null): number
    {
    ```

    Two type errors are immediately raised in the editor.

12. Delete the **`dist`** folder to remove the previously transpiled JavaScript file.

13. Rerun the **`build`** command in the terminal:

    ```
    npm run build
    ```

    The type errors are reported in the terminal. Notice that the transpiled JavaScript file is not created.

In summary, TypeScript has a compiler, called **`tsc`**, that we can use to carry out type checking and transpilation as part of a continuous integration process. The compiler is very flexible and can be configured using a file called **`tsconfig.json`**. It is worth noting that Babel is often used to transpile TypeScript (as well as React), leaving TypeScript to focus on type checking.

Next, we will create a React component that is strongly typed with TypeScript.

Creating a React and TypeScript component

In *Chapter 1*, *Getting Started with React*, we built an alert component using React. In this section, we will use TypeScript to make the component strongly typed and experience the benefits. We start by adding a type to the alert component's props and then experiment with defining a type for its state.

Creating a project

We will use Vite to create a project as we did when we first built the alert component. However, this time, we will choose the React and TypeScript template. Carry out these steps:

1. In a terminal, in a folder of your choice, execute the following command to instruct Vite to create a React and TypeScript project:

   ```
   npm create vite@latest alert -- --template
   react-ts
   ```

   In the preceding code snippet, we have specified the project name and template in the command and so won't be prompted for this information. The **`react-ts`** template has been chosen to create a React and TypeScript project.

2. The project is created. Execute the following commands in the terminal to move the working directory to the **`alert`** folder, install the project

dependencies, open the project in Visual Studio Code, and run the app in development mode:

```
cd alert
npm i
code .
npm run dev
```

3. Feel free to add automatic code formatting. We covered this topic with Prettier in *Chapter 1*, *Getting Started with React*.
4. Create a new file for the alert component in the **`src`** folder called **`Alert.tsx`**.

> **NOTE**
>
> *TypeScript React components have a* **`.tsx`** *file extension.*

5. Paste into **`Alert.tsx`** the JavaScript version of the alert component, which can be found on GitHub at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter01/using-events/src/Alert.jsx.

   Notice that type errors are reported on some of the props because they can only be inferred as having the **`any`** type.

Next, we'll add a TypeScript type annotation on the component props, which will resolve the type error.

Adding a props type

Earlier in the chapter, we learned how to add type annotations to functions. We'll use this knowledge to strongly type the alert component props:

1. Add the following type just above the component. This will be the type for the component props:

   ```
   type Props = {
     type?: string;
     heading: string;
     children: ReactNode;
     closable?: boolean;
     onClose?: () => void;
   };
   ```

   The **`heading`** and **`children`** props are required but the rest of the props are optional.

   The **`children`** prop is given a special type from React called **`ReactNode`**. This allows it to accept JSX elements as well as strings. At the moment, a type error is occurring on this – we'll resolve this in the next step.

   The name of the type can be anything, but it is common practice to call it **`Props`**.

2. To resolve the type error on the **`ReactNode`** reference, update the React **`import`** statement as follows:

   ```
   import { useState, type ReactNode } from
   'react';
   ```

> **NOTE**
>
> *Including the* **`type`** *keyword before types in* **`import`** *statements isn't required. However, it is best practice because it helps the TypeScript-to-JavaScript compilation process remove all traces of type information.*

3. Now, assign the **`Props`** type to the alert component after the destructed parameters:

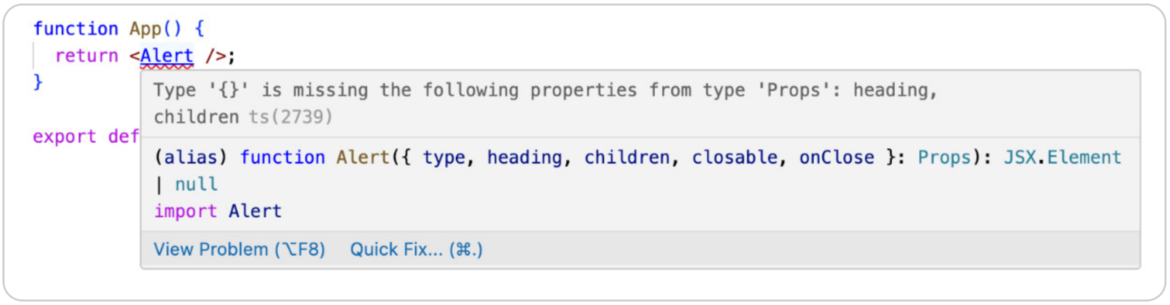
   ```
   export function Alert({
     type = "information",
     heading,
     children,
     closable,
     onClose,
   }: Props) {
     ...
   }
   ```

   The alert props are now strongly typed.

4. Open **`App.tsx`** and replace the contents with the following:

   ```
   import { Alert } from './Alert';
   import './App.css';
   function App() {
     return <Alert />;
   }
   export default App;
   ```

   We have imported the **`Alert`** component and output it in the **`App`** component. A type error occurs on **`Alert`** in the JSX because props that are required haven't been passed.



```
function App() {
  return <Alert />;
}

export def
```

```
Type '{}' is missing the following properties from type 'Props': heading,
children ts(2739)
(alias) function Alert({ type, heading, children, closable, onClose }: Props): JSX.Element
| null
import Alert
```

Figure 2.21 – Type error on the Alert component

5. Pass in a **header** prop to **Alert**, and give it some content:

```
<Alert heading="Success">
  Everything is really good!
</Alert>
```

The type errors will disappear now that the required props have been passed.

6. Start the app by executing **npm run dev** in a terminal.
7. If you visit the running app in a browser, the React app containing the alert appears on the page as expected.

Next, we will learn how to explicitly give React state a type.

Adding a state type

Follow these steps to experiment with the **visible** state type in the alert component:

1. Open **`Alert.tsx`** and hover over the **`visible`** state variable to determine its inferred type. It has been inferred to be **`boolean`** because it has been initialized with the **`true`** value. The **`boolean`** type is precisely what we want.

2. As an experiment, remove the initial value of **`true`** passed into **`useState`**. Then, hover over the **`visible`** state variable again. It has been inferred to be **`undefined`** because no default value has been passed into **`useState`**. This obviously isn't the type we want.

3. Sometimes, the **`useState`** type isn't inferred to be the type we want, like in the previous step. In these cases, the type of the state can be explicitly defined using a **generic argument** on **`useState`**. Explicitly give the **`visible`** state a **`boolean`** type by adding the following generic argument:

   ```
   const [visible, setVisible] =
   useState<boolean>();
   ```

NOTE

A generic argument is like a regular function argument but defines a type for the function. A generic argument is specified using angled brackets after the function name.

4. Restore the **`useState`** statement to what it originally was, with it initialized as **`true`** and no explicit type:

   ```
   const [visible, setVisible] = useState(true);
   ```

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

In summary, always check the inferred state type from **`useState`** and use its generic argument to explicitly define the type if the inferred type is not what is required.

That brings us to the end of the chapter. Next, we will recap what we have learned in this chapter.

Summary

TypeScript complements JavaScript with a rich type system, and in this chapter, we experienced catching errors early using TypeScript's type checking.

We also learned that JavaScript types, such as **`number`** and **`string`**, can be used in TypeScript, as well as types that only exist in TypeScript, such as **`Date`** and **`unknown`**.

New types can be created using type aliases. We learned that type aliases could be based on objects, functions, or even mixed types using a union type. We used a type alias to strongly type the props on an alert React component.

We now know that the **`?`** symbol in a type annotation makes an object property or function parameter optional. Also, an existing type can be extended using the **`&`** symbol.

We learned that the TypeScript compiler can be invoked via a CLI, allowing it to be integrated into a continuous integration pipeline. The compiler can carry out transpilation to JavaScript, as well as type checking, and can be configured with a **`tsconfig.json`** file.

So far in this book, we have used only one React Hook, **`useState`**. In the next chapter, we'll learn about many of React's other Hooks.

Questions

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

1. What would the inferred type be for the **`flag`** variable in the following code?

   ```
   let flag = false;
   ```

2. What is the type annotation for an array of dates?
3. Will a type error occur in the following code?

   ```
   type Point = {x: number; y: number; z?:
   number};
   const point: Point = { x: 24, y: 65 };
   ```

4. Use a type alias to create a number that can only hold integer values between and including 1 and 3.
5. The following code raises a type error because **`lastSale`** can't accept **`null`** values:

```
type Product = {
  name: string;
  lastSale: Date;
}
const table: Product = {name: "Table",
lastSale: null}
```

How can the **`Product`** type be changed to allow **`lastSale`** to accept **`null`** values?

Answers

1. The **`flag`** variable would be inferred to be a **`boolean`** type.
2. An array of dates can be represented as **`Date[]`** or **`Array<Date>`**.
3. A type error will not be raised on the **`point`** variable. It doesn't need to include the **`z`** property because it is optional.
4. A type for numbers 1-3 can be created as follows:

   ```
   type OneToThree = 1 | 2 | 3;
   ```

5. A union type can be used for the **`lastSale`** property to allow it to accept **`null`** values:

   ```
   type Product = {
     name: string;
     lastSale: Date | null;
   }
   const table: Product = {name: "Table",
   lastSale: null}
   ```

3
Using React Hooks

React Hooks are special functions that let you use React features, such as state, inside components. In this chapter, we will learn about React's common Hooks and how to use them with TypeScript. We will implement the knowledge of all these Hooks in a React component that allows a user to adjust a score for a person. We will start by exploring the effect Hook and begin to understand use cases where it is useful. We will then delve into two state Hooks, **`useState`** and **`useReducer`**, understanding when it is best to use each one. After that, we will cover the ref Hook and how it differs from the state Hooks, and then the memo and callback Hooks, looking at how they can help performance. In the last section, we will touch briefly on other React Hooks that are either less common or covered in depth later in this book.

By the end of this chapter, you'll have a working knowledge of the common React hooks.

So, we'll cover the following topics:

- Using the effect Hook
- Using state Hooks
- Using the ref Hook

- Using the memo Hook
- Using the callback Hook
- Other React Hooks

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-Third-Edition/tree/main/Chapter03.

Using the effect Hook

In this section, we will learn about the effect Hook and where it is useful. We will then create a new React project and a component that makes use of the effect Hook.

Understanding the effect Hook parameters

The effect Hook is used for handling side effects in a component. A component side effect is a set of instructions executed outside the scope of the component such as a web service request. The effect Hook is defined using the **`useEffect`** function from React. **`useEffect`** contains two parameters:

- A function that executes the effect; at a minimum, this function runs each time the component is rendered
- An optional array of dependencies that cause the effect function to rerun when changed

Here's an example of the effect Hook in a component:

```
function SomeComponent() {
    function someEffect() {
        console.log("Some effect");
    }
    useEffect(someEffect);
    return ...
}
```

The preceding effect Hook is passed an effect function called **`someEffect`**. No effect dependencies have been passed, so the effect function is executed each time the component renders.

Often, an anonymous arrow function is used for the effect function. Here's the same example but with an anonymous effect function instead:

```
function SomeComponent() {
    useEffect(() => {
         console.log("Some effect");
    });
    return ...
}
```

As you can see, this version of the code is a little shorter and arguably easier to read.

Here's another example of an effect:

```
function SomeOtherComponent({ search }:{search:
string}) {
    useEffect(() => {
         console.log("An effect dependent on a
search prop", search);
    }, [search]);
    Return ...;
}
```

This time, the effect has a dependency on a **`search`** prop. So, the **`search`** prop is defined in an array in the effect Hook's second parameter. The effect function will run every time the value of **`search`** changes.

The rules of Hooks

There are some rules that most React hooks, including **`useEffect`**, must obey:

- A Hook can only be called at the top level of a function component. So, a Hook can't be called in a loop or in a nested function such as an event handler.
- A Hook can't be called conditionally.
- A Hook can only be used in function components and not class components.

The following example is a violation of the rules:

```
export function AnotherComponent() {
    function handleClick() {
         useEffect(() => {
              console.log("Some effect");
         });
    }
    return (
         <button onClick={handleClick}>Cause
effect</button>
    );
}
```

This is a violation because **`useEffect`** is called in a handler function rather than at the top level. A corrected version is as follows:

```
export function AnotherComponent() {
    const [clicked, setClicked] =
useState(false);
    useEffect(() => {
         if (clicked) {
              console.log("Some effect");
```

```
            }
       }, [clicked]);
       function handleClick() {
            setClicked(true);
       }
       return (
            <button onClick={handleClick}>Cause
effect</button>
       );
}
```

`useEffect` has been lifted to the top level and now depends on the **`clicked`** state that is set in the handler function.

Effect cleanup

An effect can return a function that performs cleanup logic when the component is unmounted. Cleanup logic ensures nothing is left that could cause a memory leak. Let's consider the following example:

```
function ExampleComponent(
  { onClickAnywhere }: { onClickAnywhere: () =>
void }
) {
    useEffect(() => {
         function handleClick() {
              onClickAnywhere();
         }
         document.addEventListener("click",
handleClick);
    });
    return ...
}
```

The preceding effect function attaches an event handler to the **`document`** element. The event handler is never detached, though, so multiple event handlers will become attached to the **`document`** element as the effect is rerun. This problem is resolved by returning a **`cleanup`** function that detaches the event handler, as follows:

```
function ExampleComponent( ... ) {
    useEffect(() => {
         function handleClick() {
              onClickAnywhere();
         }
         document.addEventListener("click",
handleClick);
         return function cleanup() {
              document.removeEventListener("click
", handleClick);
         };
    });
    return ...;
}
```

Often, an anonymous arrow function is used for the cleanup function:

```
function ExampleComponent( ... ) {
    useEffect(() => {
         function handleClick() {
              onClickAnywhere();
         }
         document.addEventListener("click",
handleClick);
         return () => {
              document.removeEventListener("click
", handleClick);
         };
    });
    return ...;
```

An anonymous arrow function is a little shorter than the named function in the previous example.

Next, we will explore a common use case for the effect Hook.

Creating the project

Let's start by creating a new project in Visual Studio Code using Vite. We learned how to do this in *Chapter 2*, *Getting Started with TypeScript* – the steps are as follows:

1. In a terminal, in a folder of your choice, execute the following command to instruct Vite to create a React and TypeScript project:

   ```
   npm create vite@latest hooks -- --template
   react-ts
   ```

2. The project is created. Execute the following commands in the terminal to move the working directory to the **hooks** folder, install the project dependencies, and open the project in Visual Studio Code:

   ```
   cd hooks
   npm i
   code .
   ```

3. Feel free to add automatic code formatting. We covered this topic with Prettier in *Chapter 1*, *Getting Started with React*.
4. Run the app in development mode by executing the following command in a terminal:

   ```
   npm run dev
   ```

5. Open **App.tsx** and replace the content with the following:

   ```
   import './App.css';
   function App() {
       return <div></div>;
   }
   export default App;
   ```

   The app contains a blank page at the moment. Keep the app running as we explore the different React Hooks in the subsequent sections of this chapter.

That's the project created. Next, we will use the effect Hook.

Fetching data using the effect Hook

A common use of the effect Hook is fetching data. Carry out the following steps to implement an effect that fetches a person's name:

1. Create a function that will simulate a data request. To do this, create a file called **getPerson.ts** in the **src** folder and then add the following content to this file:

```
type Person = {
    name: string,
};
export function getPerson(): Promise<Person> {
    return new Promise((resolve) =>
         setTimeout(() => resolve({ name:
"Bob" }), 1000)
    );
}
```

The function asynchronously returns an object, **`{ name: "Bob" }`**, after a second has elapsed.

Notice the type annotation for the return type, **`Promise<Person>`**. The **`Promise`** type represents a JavaScript promise, which is something that will eventually be completed. The **`Promise`** type has a generic argument for the item type that is resolved in the promise, which is **`Person`** in this example. For more information on JavaScript promises, see the following link: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

2. Next, we will create a React component that will eventually display a person and a score. Create a file called **`PersonScore.tsx`** in the **`src`** folder and then add the following contents to the file:

```
import { useEffect } from 'react';
import { getPerson } from './getPerson';
export function PersonScore() {
    return null;
}
```

The **useEffect** Hook has been imported from React and the **getPerson** function we have just created has also been imported. At the moment, the component simply returns **null**.

3. Add the following effect above the **return** statement:

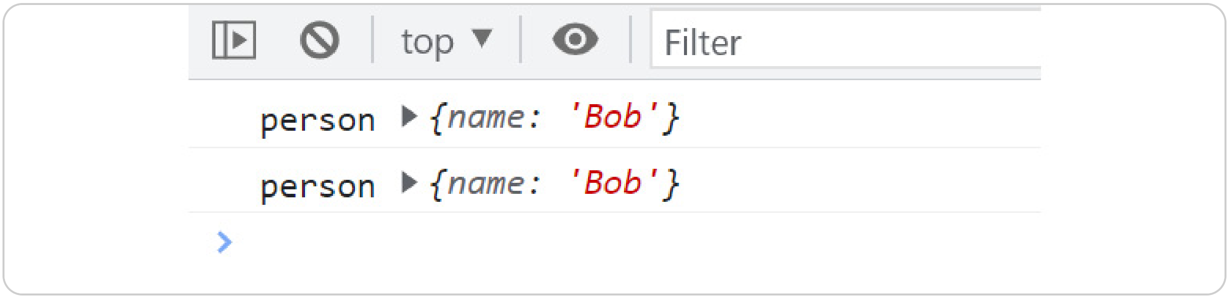
```
export function PersonScore() {
    useEffect(() => {
         getPerson().then((person) =>
console.log(person));
    }, []);
    return null;
}
```

The effect calls the **getPerson** function and outputs the returned person to the console. The effect is only executed after the component is initially rendered because an empty array has been specified as the effect dependencies in its second argument.

4. Open **App.tsx** and render the **PersonScore** component:

```
import './App.css';
import { PersonScore } from './PersonScore';
function App() {
    return <PersonScore />
}
export default App;
```

5. Go to the running app in the browser and go to the **Console** panel in the browser's DevTools. Notice that the `person` object appears in the console, which verifies that the effect that fetches the `person` data ran properly:



```
person ▶{name: 'Bob'}
person ▶{name: 'Bob'}
```

Figure 3.1 – The effect output

You may also notice that the effect function has been executed twice rather than once. This behavior is intentional and only happens in development mode with React Strict Mode.

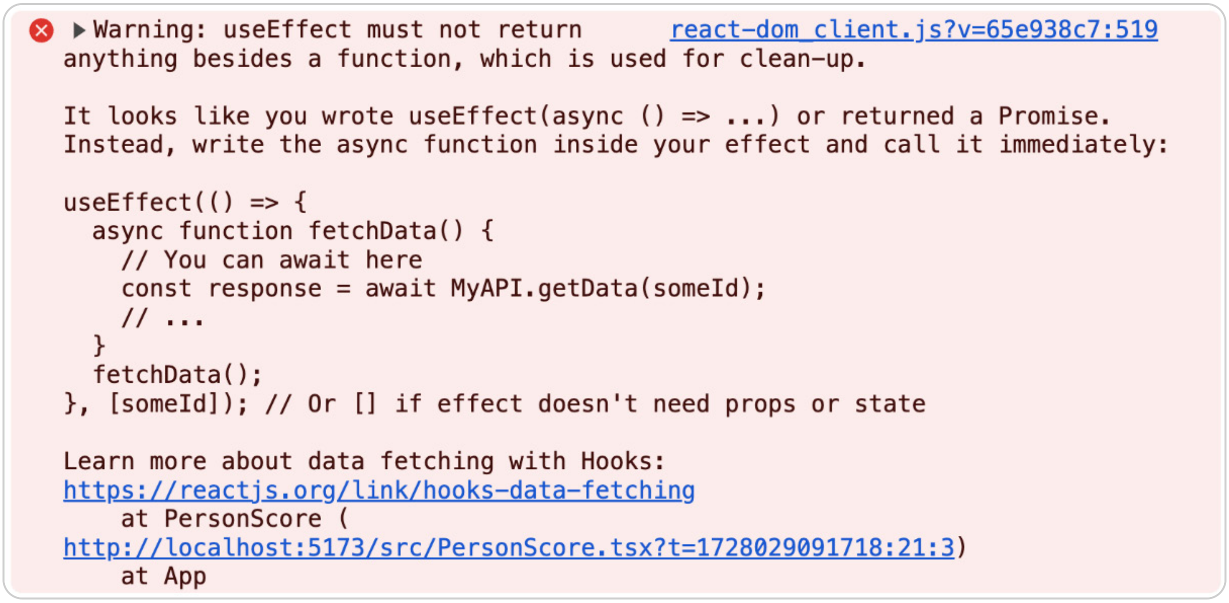
6. Next, we will refactor how the effect function is called to expose an interesting problem. Open `PersonScore.tsx` and change the `useEffect` call to use the `async`/`await` syntax:

```
useEffect(async () => {
    const person = await getPerson();
    console.log(person);
}, []);
```

> **NOTE**
>
> *The `async`/`await` syntax is an alternative way to write asynchronous code. Many developers prefer it because it reads like synchronous code. For*

*more information on **`async`**/**`await`**, see the following link: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Promises#async_and_await.*

The preceding code is arguably more readable, but React raises an error. Look in the browser's console and you'll see the following error:



```
▶Warning: useEffect must not return      react-dom_client.js?v=65e938c7:519
anything besides a function, which is used for clean-up.

It looks like you wrote useEffect(async () => ...) or returned a Promise.
Instead, write the async function inside your effect and call it immediately:

useEffect(() => {
  async function fetchData() {
    // You can await here
    const response = await MyAPI.getData(someId);
    // ...
  }
  fetchData();
}, [someId]); // Or [] if effect doesn't need props or state

Learn more about data fetching with Hooks:
https://reactjs.org/link/hooks-data-fetching
    at PersonScore (
http://localhost:5173/src/PersonScore.tsx?t=1728029091718:21:3)
    at App
```

Figure 3.2 – Effect async error

The error is very informative – the **`useEffect`** Hook doesn't allow a function marked with **`async`** to be passed into it.

7. Next, update the code and use the approach suggested in the error message:

```
useEffect(() => {
    async function getThePerson() {
        const person = await getPerson();
```

```
            console.log(person);
        }
        getThePerson();
  }, []);
```

A nested asynchronous function has been defined and immediately called in the effect function; this works nicely.

8. This implementation of the effect is arguably less readable than the initial version. So, switch back to that version before continuing to the next section. The code is available to copy from the following link: https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter03/use-effect/src/PersonScore.tsx.

> **NOTE**
>
> *Although* **`useEffect`** *is commonly used for data fetching, it can be problematic. The problems and alternative approaches are covered in Chapter 7, Server Component Data Fetching and Server Function Mutations.*

That completes our exploration of the effect Hook – here's a recap:

- The effect Hook is used to execute component side effects when a component is rendered or when certain props or states change
- Any required effect cleanup can be done in a function returned by the effect function

> **NOTE**

> *For more information on **`useEffect`**, see the React documentation at https://react.dev/reference/react/useEffect.*

Next, we will learn about the two state Hooks in React. Keep the app running as we move to the next section.

Using state Hooks

We have already learned about the **`useState`** Hook in previous chapters, but we will look at it again here and compare it against another state Hook we haven't covered yet, **`useReducer`**. We will expand the **`PersonScore`** component we created in the last section to explore these state Hooks.

Using useState

As a reminder, the **`useState`** Hook allows state to be defined in a variable. The syntax for **`useState`** is as follows:

```
const [state, setState] = useState(initialState);
```

We will enhance the **`PersonScore`** component we created in the last section to store the person's name in **`state`**. We will also have **`state`** for a score that is incremented, decremented, and reset using some buttons in the component. We will also add a **`loading`** state to the component, showing a loading indicator when **`true`**.

Carry out the following steps:

1. Open **`PersonScore.tsx`** and add **`useState`** to the React **`import`** statement:

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

2. Add the following state definitions for **`name`**, **`score`**, and **`loading`** at the top of the component function, above the **`useEffect`** call:

```
export function PersonScore() {
    const [name, setName] = useState<
         string | undefined
    >();
    const [score, setScore] = useState(0);
    const [loading, setLoading] =
useState(true);
    useEffect( ... );
    return null;
}
```

   The **`score`** state is initialized to **`0`** and **`loading`** is initialized to **`true`**.

3. Change the effect function to set the **`loading`** and **`name`** state values after the person data has been fetched. This should replace the existing **`console.log`** statement:

```
useEffect(() => {
    getPerson().then((person) => {
         setLoading(false);
         setName(person.name);
```

```
        });
    }, []);
```

After the **`person`** object has been fetched, **`loading`** is set to **`false`**, and **`name`** is set to the person's name.

4. Next, add the following **`if`** statement between the **`useEffect`** call and the **`return`** statement:

```
    useEffect( ... );
    if (loading) {
        return <div>Loading ...</div>;
    }
    return ...
```

This displays a loading indicator when the **`loading`** state is **`true`**.

5. Change the component's **`return`** statement to output the following:

```
    if (loading) {
        return <div>Loading ...</div>;
    }
    return (
        <div>
            <h3>
                {name}, {score}
            </h3>
            <button>Add</button>
            <button>Subtract</button>
            <button>Reset</button>
        </div>
    );
```

The person's name and score are displayed in a header with **Add**, **Subtract**, and **Reset** buttons underneath (don't worry that the output is unstyled – we will learn how to style components in the next chapter):

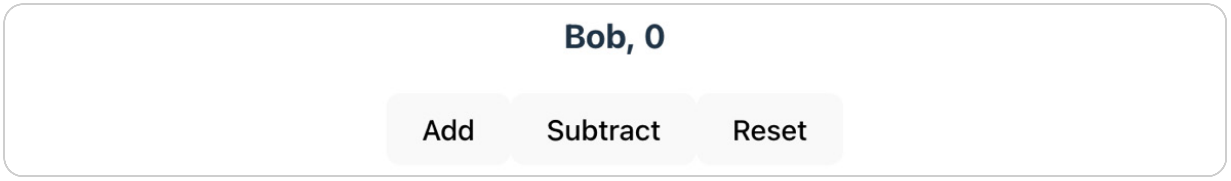


Figure 3.3 – The PersonScore component after data has been fetched

6. Update the **Add** button so that it increments the score when clicked:

```
<button onClick={() => setScore(score + 1)}>
    Add
</button>
```

The button click event calls the score state setter to increment the state.

There is an alternative method of updating the state values based on their previous value. The alternative method uses a parameter in the state setter that gives the previous state value, so our example could look as follows:

```
setScore(previousScore => previousScore + 1)
```

This is arguably a little harder to read, so we'll stick to our initial method.

7. Add score state setters to the other buttons, as follows:

```
<button onClick={() => setScore(score - 1)}>
    Subtract
</button>
<button onClick={() =>
setScore(0)}>Reset</button>
```

8. In the running app, click the different buttons. They should change the score as you would expect.

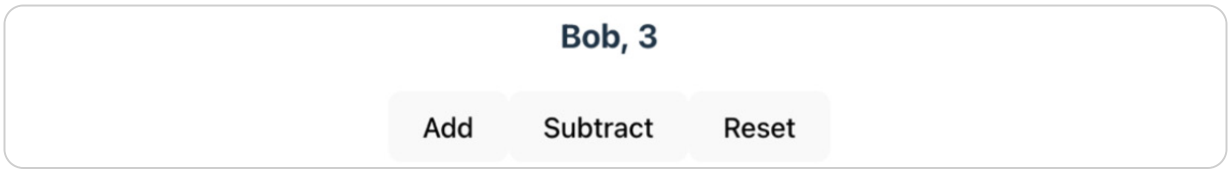


Figure 3.4 – The PersonScore component after the button is clicked

9. Before we finish this exercise, let's take some time to understand when the state values are actually set. Update the effect function to output the state values after they are set:

```
useEffect(() => {
    getPerson().then((person) => {
         setLoading(false);
         setName(person.name);
         console.log("State values", loading,
name);
    });
}, []);
```

You may notice that ESLint highlights missing dependencies, **loading** and **name**, on the **useEffect** call – ignore this warning because we'll remove this **`console.log`** statement at the end of this step.

Perhaps we would expect **`false`** and **`"Bob"`** as the output to the console? However, **`true`** and **`undefined`** are the output to the console. This is because updating state values is not immediate – instead, they are batched and updated before the next render. So, it isn't until the next render that **`loading`** will be **`false`** and **`name`** will be **`"Bob"`**.

We no longer need the **`console.log`** statement we added in this step, so remove it before continuing.

Next, we will learn about an alternative React Hook for using state.

Understanding useReducer

`useReducer` is an alternative, more complex, method of managing state. It uses a **reducer** function for state changes, which takes in the current state value and returns the new state value.

Here is an example of a **`useReducer`** call:

```
const [state, dispatch] = useReducer(
    reducer,
    initialState,
);
```

So, **`useReducer`** takes in a reducer function and the initial state value as parameters. It then returns a tuple containing the current state value and a function to **dispatch** state changes.

The dispatch function takes in an argument that describes the change. This object is often referred to as an **action**. An example **`dispatch`** call is as follows:

```
dispatch({ type: 'add', amount: 2 });
```

There is no defined structure for an action, but it is common practice for it to contain a property, such as **`type`**, to specify the type of change. Other properties in the action can vary depending on the type of change. Here's another example of a **`dispatch`** call:

```
dispatch({ type: 'loaded' });
```

This time, the action only needs the type to change the necessary state.

Turning our attention to the reducer function, it has parameters for the current state value and the action. Here's an example code snippet of a reducer:

```
function reducer(state: State, action: Action):
State {
    switch (action.type) {
         case 'add':
              return {
                   ...state,
                        total: state.total +
action.amount
                  };
          case ...
               ...
          default:
               return state;
     }
}
```

The reducer function usually contains a **`switch`** statement based on the action type. Each **`switch`** branch makes the required changes to the state and returns the updated state. A new state object is created during the state change – the current state is never mutated. Mutating the state would result in the component not re-rendering.

> **NOTE**
>
> *In the preceding code snippet, inside the* ***`'add'`*** *branch, the* ***spread syntax*** *is used on the* ***`state`*** *variable (****`...state`****). The spread syntax copies all the properties from the object after the three dots. In the preceding code snippet, all the properties are copied from the* ***`state`*** *variable into the new state object returned. The* ***`total`*** *property value will then be overwritten by* ***`state.total + action.amount`*** *because this is defined after the spread operation in the new object creation. For more information on the spread syntax, see the following link: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax.*

The types for **`useReducer`** can be explicitly defined in its generic parameter as follows:

```
const [state, dispatch] = useReducer<
    Reducer<State, Action>
>(reducer, initialState);
```

`Reducer` is a standard React type that has generic parameters for the type of state and the type of action.

So, **`useReducer`** is more complex than **`useState`** because state changes go through a reducer function that we must implement. However, the additional complexity may be worth it in the following cases:

- An action changes multiple state values – for example, a web API response updating a **`loading`** state value as well as a **`data`** state value.
- The next state value depends on the previous state value. The **`'add'`** action in the preceding example is an example of this.

Next, we will implement state using **`useReducer`**.

Using useReducer

We will refactor the **`PersonScore`** component we have been working on to use **`useReducer`** instead of **`useState`**. To do this, carry out the following steps. The code snippets used are available to copy from https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter03/use-reducer/src/PersonScore.tsx:

1. Open **PersonScore.tsx** and import **useReducer** instead of **useState** from React:

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

2. We will have the state in a single object, so define a type for the state beneath the **import** statements:

```
type State = {
    name: string | undefined;
    score: number;
    loading: boolean;
};
```

3. Next, let's define types for all the action objects:

```
type Action =
    | {
          type: 'initialize';
          name: string;
      }
    | {
          type: 'increment';
      }
    | {
          type: 'decrement';
      }
    | {
            type: 'reset';
        };
```

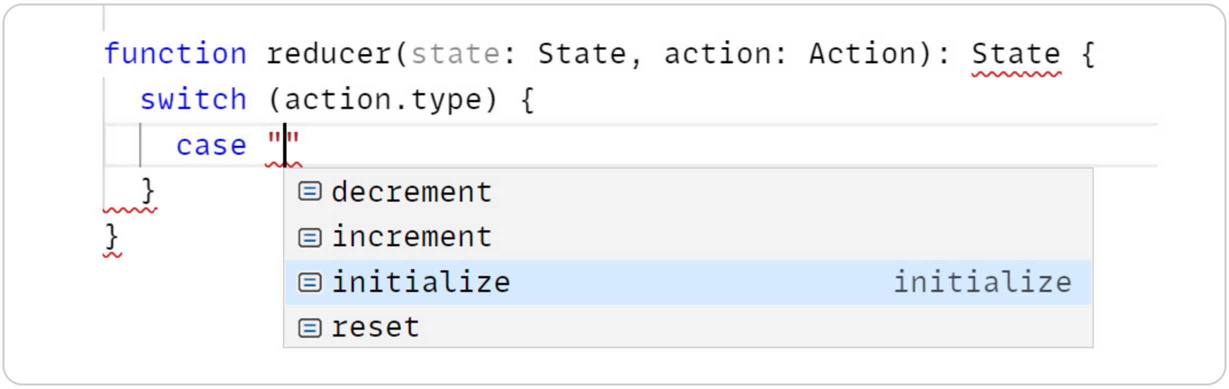
These action objects represent all the ways in which the state can change. The action object types are combined using a union type, allowing an action to be any of these. This kind of union type is a **discriminated union** because each item in the union contains a property to distinguish the items (the **`type`** property).

4. Now, define the following reducer function underneath the type definitions:

```
function reducer(state: State, action:
Action): State {
    switch (action.type) {
        case 'initialize':
            return {
                name: action.name,
                score: 0,
                loading: false
            };
        case 'increment':
            return { ...state, score:
state.score + 1 };
        case 'decrement':
            return { ...state, score:
state.score - 1 };
        case 'reset':
             return { ...state, score: 0 };
        default:
             return state;
        }
    }
```

The reducer function contains a **`switch`** statement that makes appropriate state changes for each type of action.

Notice the nice IntelliSense when referencing the **`state`** and **`action`** parameters:



```
function reducer(state: State, action: Action): State {
  switch (action.type) {
    case ""
  }
}
```

Figure 3.5 – IntelliSense inside the reducer function

5. Inside the **`PersonScore`** component, replace the **`useState`** calls with the following **`useReducer`** call:

```
const [{ name, score, loading }, dispatch] =
    useReducer(reducer, {
        name: undefined,
        score: 0,
        loading: true,
    });
```

The state has been initialized with an **`undefined`** name, a score of **`0`**, and **`loading`** set to **`true`**.

The current state value has been destructured into **`name`**, **`score`**, and **`loading`** variables. If you hover over these destructured state variables, you will see that their types have been inferred correctly.

6. We now need to amend the places in the component that update the state. Start with the effect function and dispatch an **`initialize`** action after the person has been returned:

```
useEffect(() => {
    getPerson().then(({ name }) =>
        dispatch({ type: 'initialize', name })
    );
}, []);
```

7. Lastly, dispatch the relevant actions in the button click handlers:

```
<button
    onClick={() => dispatch({ type:
'increment' })}
>
    Add
</button>
<button
    onClick={() => dispatch({ type:
'decrement' })}
>
    Subtract
</button>
<button onClick={() => dispatch({ type:
'reset' })}>
    Reset
</button>
```

8. If you try clicking the buttons in the running app, they will correctly update the displayed score.

That completes our exploration of the `useReducer` Hook. It is more useful for complex state management situations than `useState`, for example, when the state is a complex object with related properties and state changes depend on previous state values. The `useState` Hook is more appropriate when the state is based on primitive values independent of any other state.

> **NOTE**
>
> *For more information on* `useState` *and* `useReducer`*, see the React documentation at https://react.dev/reference/react/useState and https://react.dev/reference/react/useReducer.*

We will continue to expand the `PersonScore` component in the following sections. Next, we will learn how to move the focus to the **Add** button using the ref Hook.

Using the ref Hook

In this section, we will learn about the ref Hook and where it is useful. We will then walk through a common use case of the ref Hook by enhancing the `PersonScore` component we have been working on.

Understanding the ref Hook

The ref Hook is called **`useRef`** and it returns a variable whose value is persisted for the lifetime of a component. This means that the variable doesn't lose its value when a component re-renders.

The value returned from the ref Hook is often referred to as a **ref**. The ref can be changed without causing a re-render.

Here's the syntax for **`useRef`**:

```
const ref = useRef(initialValue);
```

An initial value can optionally be passed into **`useRef`**. The type of the ref can be explicitly defined in a generic argument for **`useRef`**:

```
const ref = useRef<Ref>(initialValue);
```

The generic argument is useful when no initial value is passed or is **`null`**. This is because TypeScript won't be able to infer the type correctly.

The value of the ref is accessed via its **`current`** property:

```
console.log("Current ref value", ref.current);
```

The value of the ref can be updated via its **`current`** property as well:

```
ref.current = newValue;
```

A common use of the ref Hook is to access HTML elements imperatively. HTML elements have a **`ref`** attribute in JSX that can be assigned to a ref. The following is an example of this:

```
function MyComponent() {
    const inputRef = useRef<HTMLInputElement>
(null);
    function doSomething() {
        console.log(
            "All the properties and methods of
the input",
            inputRef.current
        );
    }
    return <input ref={inputRef} type="text" />;
}
```

The ref used here is called **`inputRef`** and is initially **`null`**. So, it is explicitly given a type of **`HTMLInputElement`**, which is a standard type for input elements. The ref is then assigned to the **`ref`** attribute on an **`input`** element in JSX. All the input's properties and methods are then accessible via the ref's **`current`** property.

Next, we will use the ref Hook in the **`PersonScore`** component.

Using the ref Hook

We will enhance the **`PersonScore`** component we have been working on to use **`useRef`** to move the focus to the **Add** button. To do this, carry out the following steps. All the code snippets used are available at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter03/use-ref/src/PersonScore.tsx:

1. Open **`PersonScore.tsx`** and import **`useRef`** from React:

   ```
   import { useEffect, useReducer, useRef } from
   'react';
   ```

2. Create a ref for the **Add** button just below the **`useReducer`** statement:

   ```
   const [ ... ] = useReducer( ... );
   const addButtonRef = useRef<HTMLButtonElement>
   (null);
   useEffect( ... )
   ```

   The ref is named **`addButtonRef`** and is initially **`null`**. It is given the standard **`HTMLButtonElement`** type.

> **NOTE**
>
> *All the standard HTML elements have corresponding TypeScript types for React. Right-click on the* ***`HTMLButtonElement`*** *type and choose* ***Go to Definition*** *to discover all these types. The React TypeScript types will open containing all the HTML element types.*

3. Assign the ref to the **`ref`** attribute on the **`Add`** button JSX element:

   ```
   <button
       ref={addButtonRef}
   ```

```
    onClick={() => dispatch({ type:
'increment' })}
>
    Add
</button>
```

4. Now that we have a reference to the **Add** button, we can invoke its **`focus`** method to move the focus to it when the person's information has been fetched. Add the following highlighted line in the existing effect:

```
useEffect(() => {
    getPerson().then(({ name }) => {
        dispatch({ type: 'initialize', name
});
        addButtonRef.current?.focus();
    });
}, []);
```

   Notice the **`?`** symbol after the **`current`** property on the ref. This is the **optional chaining** operator, and it allows the **`focus`** method to be invoked without having to check that **`current`** is not **`null`**. Visit the following link for more information about optional chaining: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining.

5. There is a cleaner approach than adding that line of code to the existing effect. At the moment, that effect is mixing the concerns of fetching data, setting state, and setting focus to a button. Mixing concerns like this can make components hard to understand and change. So, remove

that line from the existing effect, and let's add another effect to do this below the existing effect:

```
useEffect(() => {
    getPerson().then(({ name }) =>
        dispatch({ type: 'initialize', name })
    );
}, []);
useEffect(() => {
    if (!loading) {
        addButtonRef.current?.focus();
    }
}, [loading]);
if (loading) {
    return <div>Loading ...</div>;
}
```

The effect is executed when the `loading` state is `true`, which will be after the person has been fetched.

6. If you refresh the browser containing the running app, you will see a focus indicator on the **Add** button:

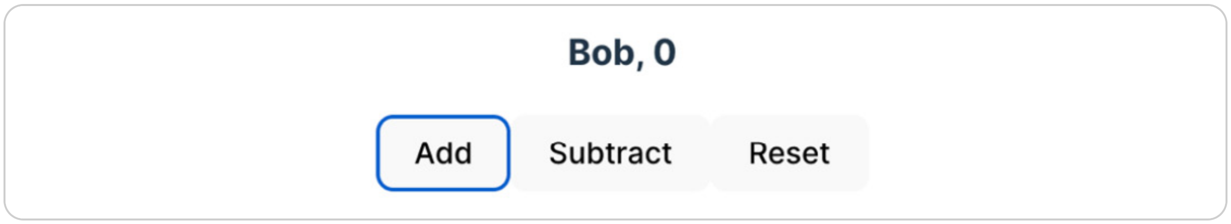


Figure 3.6 – The focused Add button

If you press the *Enter* key, you will see that the **Add** button is clicked and the score incremented. This proves that the **Add** button is focused.

That completes the enhancement and our exploration of the ref Hook.

To recap, the ref Hook creates a mutable value and doesn't cause a re-render when changed. It is commonly used to access HTML elements in React, imperatively.

> ***NOTE***
>
> *For more information on **`useRef`**, see the React documentation at https://react.dev/reference/react/useRef.*

Next, we will learn about the memo Hook.

Using the memo Hook

In this section, we will learn about the memo Hook and where it is useful. We will then walk through an example in the **`PersonScore`** component we have been working on.

Understanding the memo Hook

The memo Hook creates a memoized value and is beneficial for values that have computationally expensive calculations. The Hook is called **`useMemo`** and the syntax is as follows:

```
const memoizedValue = useMemo(
    () => expensiveCalculation(),
    []
);
```

A function that returns the value to memoize is passed into **`useMemo`** as the first argument. The function in this first argument should perform the expensive calculation.

The second argument passed to **`useMemo`** is an array of dependencies. So, if the **`expensiveCalculation`** function has dependencies **`a`** and **`b`**, the call will be as follows:

```
const memoizedValue = useMemo(
    () => expensiveCalculation(a, b),
    [a, b]
);
```

When any dependencies change, the function in the first argument is executed again to return a new value to memoize. In the previous example, a new version of **`memoizedValue`** is created every time **`a`** or **`b`** changes.

The type of the memoized value is inferred but can be explicitly defined in a generic parameter on **`useMemo`**. The following is an example of explicitly defining that the memoized value should have a **`number`** type:

```
const memoizedValue = useMemo<number>(
    () => expensiveCalculation(),
      []
  );
```

Next, we will experiment with **`useMemo`**.

Using the memo Hook

We will use the **`PersonScore`** component we have been working on to play with the **`useMemo`** Hook. To do so, carry out the following steps. The code snippets used are available at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter03/use-memo/src/PersonScore.tsx:

1. Open **`PersonScore.tsx`** and import **`useMemo`** from React:

   ```
   import {
       useEffect,
       useReducer,
       useRef,
       useMemo
   } from 'react';
   ```

2. Add the following expensive function below the **`import`** statements:

   ```
   function sillyExpensiveFunction() {
       console.log("Executing silly function");
       let sum = 0;
       for (let i = 0; i < 10000; i++) {
           sum += i;
       }
   ```

```
        return sum;
    }
```

The function adds all the numbers between `0` and `10000` and will take a while to execute.

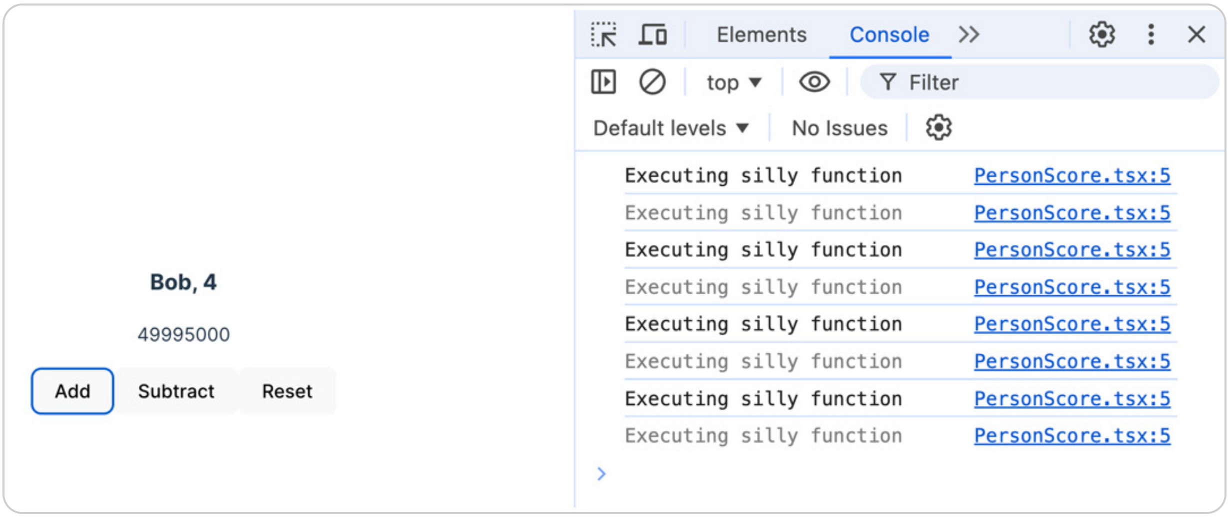
3. Add a call to the function in the `PersonScore` component beneath the effects:

```
    useEffect( ... );
    const expensiveCalculation =
    sillyExpensiveFunction();
    if (loading) {
        return <div>Loading ...</div>;
    }
```

4. Add the result of the function call to the JSX underneath `name` and `score`:

```
    <h3>
        {name}, {score}
    </h3>
    <p>{expensiveCalculation}</p>
    <button ... >
        Add
    </button>
```

5. Refresh the browser containing the app and click the buttons. If you look in the console, you will see that the expensive function is executed every time the component is re-rendered after a button click.



```
Executing silly function        PersonScore.tsx:5
Executing silly function        PersonScore.tsx:5
Executing silly function        PersonScore.tsx:5
Executing silly function        PersonScore.tsx:5
Executing silly function        PersonScore.tsx:5
Executing silly function        PersonScore.tsx:5
Executing silly function        PersonScore.tsx:5
Executing silly function        PersonScore.tsx:5
```

Figure 3.7 – The expensive function executed multiple times

Remember that a double render occurs in development mode and React's Strict Mode. So, once a button is clicked, you will see **Executing silly function** in the console twice.

An expensive function executing each time a component is re-rendered can lead to performance problems.

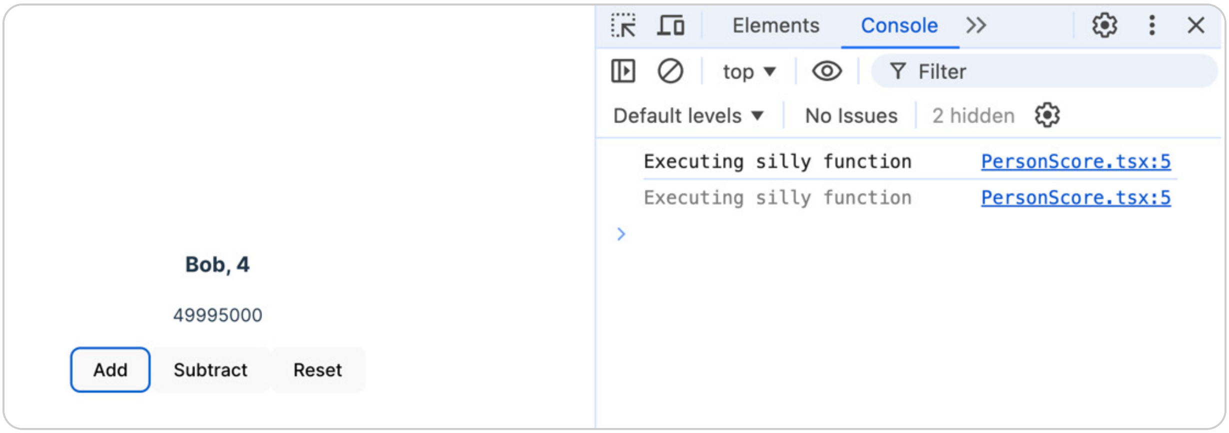
6. Rework the call to `sillyExpensiveFunction` as follows:

```
const expensiveCalculation = useMemo(
    () => sillyExpensiveFunction(),
    []
);
```

   The `useMemo` Hook is used to memoize the value from the function call.

7. Refresh the browser containing the running app and click the buttons. If you look in the console, you will see that the expensive function is

only executed on the initial render and not when the buttons are clicked because the memoized value is used instead.



Bob, 4

49995000

```
Executing silly function    PersonScore.tsx:5
Executing silly function    PersonScore.tsx:5
```

Figure 3.8 – The expensive function call memoized

That completes our exploration of the **`useMemo`** Hook. The takeaway from this section is that the **`useMemo`** Hook helps improve the performance of function calls by memoizing their results and using the memoized value when the function is re-executed.

NOTE

*For more information on **`useMemo`**, see the React documentation at https://react.dev/reference/react/useMemo.*

Next, we will look at another Hook that can help performance.

Using the callback Hook

In this section, we will learn about the callback Hook and where it is useful. We will then use the Hook in the **`PersonScore`** component we have been working on.

Understanding the callback Hook

The callback Hook memoizes a function so that it isn't recreated on each render. The Hook is called **`useCallback`** and the syntax is as follows:

```
const memoizedCallback = useCallback(() =>
someFunction(), []);
```

A function that executes the function to memoize is passed into **`useCallback`** as the first argument. The second argument passed to **`useCallback`** is an array of dependencies. So, if the **`someFunction`** function has dependencies **`a`** and **`b`**, the call will be as follows:

```
const memoizedCallback = useCallback(
     () => someFunction(a, b),
     [a, b]
);
```

When any dependencies change, the function in the first argument is executed again to return a new function to memoize. In the previous example, a new version of **`memoizedCallback`** is created every time **`a`** or **`b`** changes.

The type of the memoized function is inferred but can be explicitly defined in a generic parameter on **`useCallback`**. Here is an example of explicitly defining that the memoized function has no parameters and returns **`void`**:

```
   const memoizedValue = useCallback<() => void>(
      () => someFunction (),
      []
  );
```

A common use case for **`useCallback`** is to prevent unnecessary re-renders of child components. Before trying **`useCallback`**, we will take the time to understand when a component is re-rendered.

Understanding when a component is re-rendered

We already understand that a component re-renders when its state changes. Consider the following component:

```
  export function SomeComponent() {
      const [someState, setSomeState] =
  useState('something');
      return (
          <div>
               <ChildComponent />
               <AnotherChildComponent something=
  {someState} />
               <button
                   onClick={() =>
```

```
setSomeState('Something else')}
            ></button>
        </div>
    );
}
```

When **`someState`** changes, **`SomeComponent`** will re-render – for example, when the button is clicked. In addition, **`ChildComponent`** and **`AnotherChildComponent`** will re-render when **`someState`** changes. This is because a component is re-rendered when its parent is re-rendered.

It may seem like this re-rendering behavior will cause performance problems – particularly when a component is rendered near the top of a large component tree. However, it rarely does cause performance issues. This is because the DOM will only be updated after a re-render if the virtual DOM changes, and updating the DOM is the slow part of the process. In the preceding example, the DOM for **`ChildComponent`** won't be updated when **`SomeComponent`** is re-rendered if the definition of **`ChildComponent`** is as follows:

```
export function ChildComponent() {
    return <span>A child component</span>;
}
```

The DOM for **`ChildComponent`** won't be updated during a re-render because the virtual DOM will be unchanged.

While this re-rendering behavior generally doesn't cause performance problems, it can cause performance issues if a computationally expensive component is frequently re-rendered or a component with a slow side effect is frequently re-rendered. For example, we would want to avoid unnecessary re-renders in components with a side effect that fetches data.

There is a function called **`memo`** in React that can be used to prevent unnecessary re-renders. The **`memo`** function can be applied as follows to **`ChildComponent`** to prevent unnecessary re-renders:

```
export const ChildComponent = memo(() => {
    return <span>A child component</span>;
});
```

The **`memo`** function wraps the component and memoizes the result for a given set of props. The memoized function is then used during a re-render if the props are the same. Note that the preceding code snippet uses arrow function syntax so that the component can be a named export.

In summary, React's **`memo`** function can prevent the unnecessary re-rendering of slow components.

Next, we will use the **`memo`** function and the **`useCallback`** Hook to prevent unnecessary re-renders.

Using the callback Hook

We will now refactor the **`PersonScore`** component by extracting the Reset button into a separate component called **`Reset`**. This will lead to unnecessary re-rendering of the **`Reset`** component, which we will resolve using React's **`memo`** function and the **`useCallback`** Hook.

To do so, carry out the following steps. The code snippets used are available at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter03/use-callback/src/PersonScore.tsx:

1. Start by creating a new file in the **`src`** folder for the **`Reset`** button component called **`Reset.tsx`** with the following content:

   ```
   type Props = {
       onClick: () => void,
   };
   export function Reset({ onClick }: Props) {
       console.log("render Reset");
       return <button onClick=
   {onClick}>Reset</button>;
   }
   ```

   The component takes in a click handler and displays the Reset button.

   The component also outputs **render Reset** to the console so that we can clearly see when the component is re-rendered.

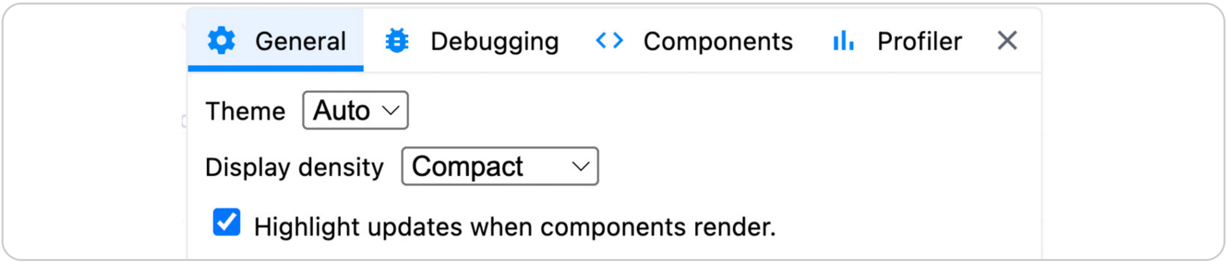
2. Open **`PersonScore.tsx`** and import the **`Reset`** component:

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

3. Replace the existing Reset button with the new **Reset** component as follows:

```
<div>
    ...
    <button onClick={() => dispatch({ type:
'decrement' })}>
        Subtract
    </button>
    <Reset
        onClick={() =>
            dispatch({ type: 'reset' })
        }
    />
</div>;
```

4. Go to the app running in the browser and open React's developer tools. Make sure the **Highlight updates when components render.** option is ticked in the **Components** panel's settings:



Theme Auto

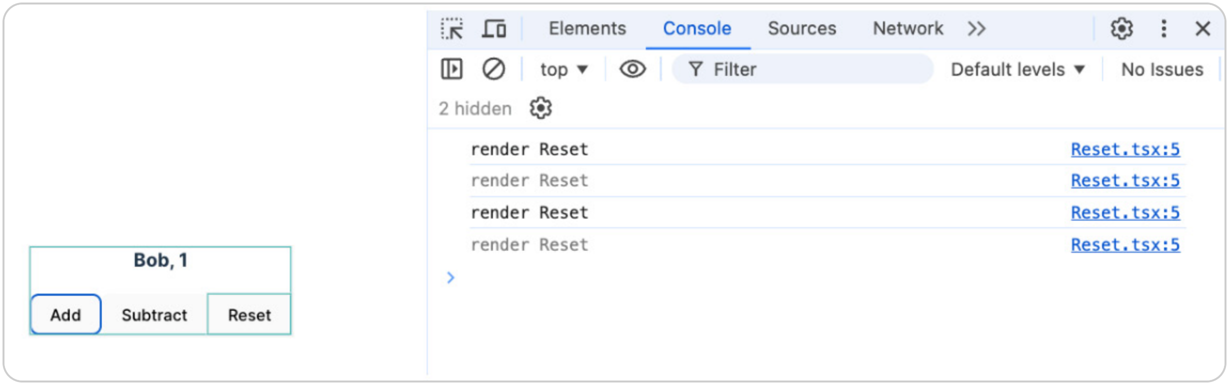
Display density Compact

✓ Highlight updates when components render.

Figure 3.9 – The re-render highlight option

5. In the browser, the **Reset** button will work as it did before. Click this button as well as the **Add** and **Subtract** buttons. If you look at the

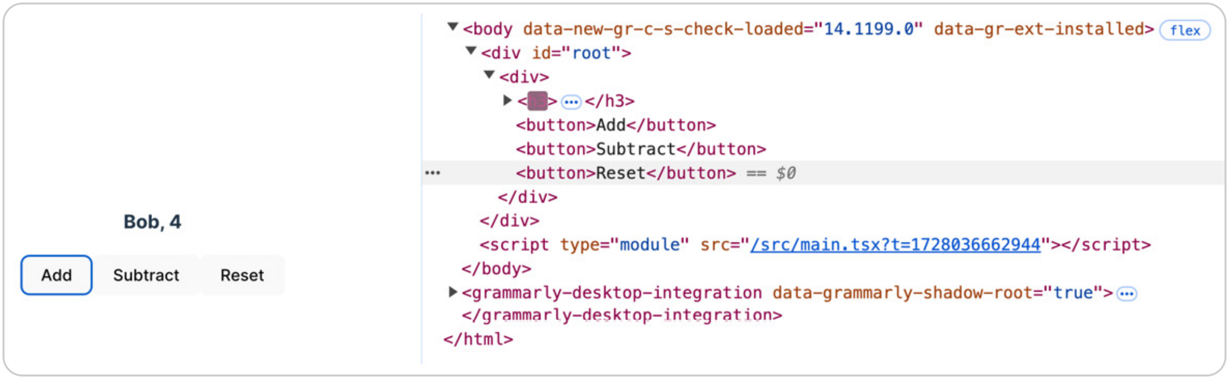
console, you'll notice that `Reset` is unnecessarily re-rendered. You will also see the re-render highlight around the **Reset** button.



```
render Reset        Reset.tsx:5
render Reset        Reset.tsx:5
render Reset        Reset.tsx:5
render Reset        Reset.tsx:5
```

Figure 3.10 – The unnecessary re-renders of the Reset component

6. Use the browser's **Inspect** tool to inspect the DOM. To do this, right-click on the **Reset** button and choose **Inspect**. Click the buttons and observe the DOM elements. The developer tools in Chrome highlight elements when they are updated. You will see that only the `h3` element content was updated – none of the other elements are highlighted due to an update occurring.



```
▼<body data-new-gr-c-s-check-loaded="14.1199.0" data-gr-ext-installed> flex
  ▼<div id="root">
    ▼<div>
      ▶<h3>…</h3>
       <button>Add</button>
       <button>Subtract</button>
...    <button>Reset</button> == $0
     </div>
   </div>
   <script type="module" src="/src/main.tsx?t=1728036662944"></script>
  </body>
 ▶<grammarly-desktop-integration data-grammarly-shadow-root="true">…
  </grammarly-desktop-integration>
</html>
```

Figure 3.11 – The h3 element was updated after a re-render

Even though `Reset` is unnecessarily re-rendered, it doesn't result in a DOM update. In addition, `Reset` isn't computationally expensive and doesn't contain any side effects. So, the unnecessary render isn't really a performance problem. However, we will use this example to learn how to use React's **`memo`** function, and the **`useCallback`** Hook can prevent the unnecessary render.

7. We will now add React's **`memo`** function to try to prevent unnecessary re-renders. Open **`Reset.tsx`** and add a React **`import`** statement to the **`memo`** import at the top of the file:

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

8. Now, wrap **`memo`** around the **`Reset`** component as follows:

```
export const Reset = memo(({ onClick }: Props)
=> {
    console.log("render Reset");
    return <button onClick=
{onClick}>Reset</button>;
});
```

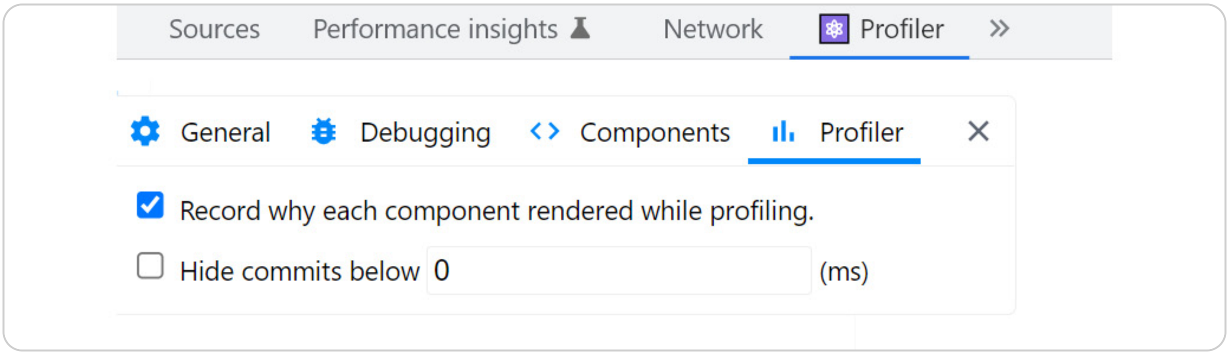
9. In addition, add the following line beneath the **`Reset`** component definition so that it has a meaningful name in React's development tools:

```
Reset.displayName = 'Reset';
```

10. In the browser, click the **Add**, **Subtract**, and **Reset** buttons. Then, look at the console and notice that **`Reset`** is *still* unnecessarily re-rendered.

11. We will use React's developer tools to start to understand why **`Reset`** is still unnecessarily re-rendered when its result is memoized. Open the

Profiler panel and click the cog icon to open the settings. Go to the **Profiler** settings section and make sure **Record why each component rendered while profiling.** is ticked:

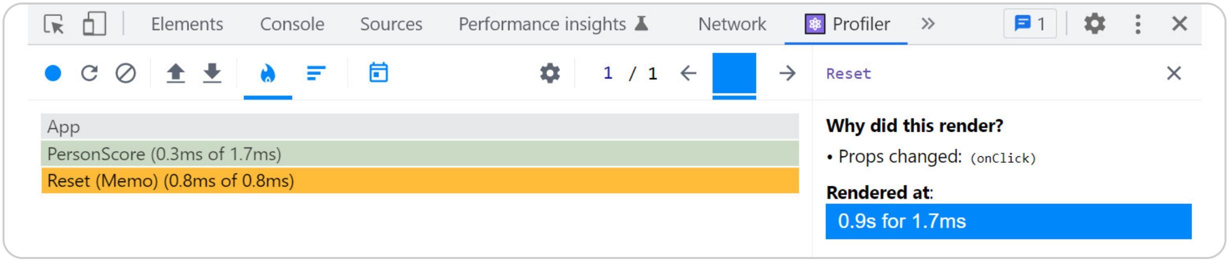


[✓] Record why each component rendered while profiling.

[ ] Hide commits below 0 (ms)

Figure 3.12 – Ensuring the Record why each component rendered while profiling. option is ticked

12. Click the blue circle icon to start profiling and then click the **Add** button in our app. Click the red circle icon to stop profiling.
13. In the flamegraph that appears, click the **Reset** bar. This gives useful information about the `Reset` component re-render:



App
PersonScore (0.3ms of 1.7ms)
Reset (Memo) (0.8ms of 0.8ms)

Why did this render?
• Props changed: (onClick)

Rendered at:
0.9s for 1.7ms

Figure 3.13 – Information about the Reset re-render

So, the unnecessary `Reset` render is happening because the `onClick` prop changes. The `onClick` handler contains the same code, but a new instance of the function is created on every render.

This means **onClick** will have a different reference on each render. The changing **onClick** prop reference means that the memoized result from **Reset** isn't used and a re-render occurs instead.

14. We can use the **useCallback** Hook to memoize the **onClick** handler and prevent the re-render. Open **PersonScore.tsx** and start by refactoring the handler into a named function:

```
const handleReset = () => dispatch({ type:
'reset' });
if (loading) {
    return <div>Loading ...</div>;
}
return (
    <div>
        ...
        <Reset onClick={handleReset} />
    </div>
);
```

15. Now, add **useCallback** to the React **import** statement:

```
import {
    useEffect,
    useReducer,
    useRef,
    useMemo,
    useCallback
} from 'react';
```

16. Add a memoized reset click handler, as follows:

```
const handleReset = () => dispatch({ type:
    "reset" });
    const handleResetMemoized = useCallback(
        handleReset,
        [],
    );
```

17. Lastly, change the **`onClick`** prop on **`Reset`** to reference the new memorized handler:

```
    <Reset onClick={handleResetMemoized} />;
```

18. Now, if you click the **Add**, **Subtract**, and **Reset** buttons, you will notice that **`Reset`** is no longer unnecessarily re-rendered.

    That completes our exploration of the **`useCallback`** Hook.

Here's a quick recap of everything we learned in this section:

- A component is re-rendered when its parent is re-rendered.
- React's **`memo`** function can be used to prevent unnecessary re-renders to child components.
- **`useCallback`** can be used to memoize functions. This can be used to create a stable reference for function props passed to child components to prevent unnecessary re-renders.
- React's **`memo`** function and **`useCallback`** should be used wisely – make sure they help performance before using them because they increase the complexity of the code.

NOTE

*For more information on **useCallback**, see the React documentation at https://react.dev/reference/react/useCallback.*

Next, we will touch on some other React Hooks.

Other React Hooks

In this section, we will touch on some other React Hooks. We will also mention some React Hooks that are covered in depth in subsequent chapters of this book.

useId

The **useId** Hook generates unique IDs and is typically used for accessibility attributes in reusable components. The following is a reusable **Field** component where **useId** is used to associate the label and input for screen readers:

```
export function Field({ label, name }: ... ) {
    const id = useId();
    return (
        <div>
            <label htmlFor={id}>{label}</label>
            <input id={id} name={name}
type="text" />
        </div>
    );
}
```

The full code for this example is at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter03/use-id.

For more information on **`useId`**, see the React documentation at https://react.dev/reference/react/useId.

useTransition

The **`useTransition`** Hook enables state transitions to not block the UI, making it ideal for state transitions that may take some time to complete.

The following component allows a user to filter a large list of names. When the user types a character in the search input, the **`list`** state is updated with the filtered names. The React transition marks the **`list`** state update as low priority, preventing the search input from lagging when the user types into it.

The **`useTransition`** Hook returns a tuple containing the following:

- A flag for whether the transition is in progress. The flag is called **`isPending`** in the following example.
- A function to start the transition. The function is called **`startTransition`** in the following example.

Here's the example:

```
function App() {
    const [query, setQuery] = useState("");
    const [list, setList] = useState(names);
    const [isPending, startTransition] =
useTransition();
    return (
        <div>
            <input ...
                 value={query}
                onChange={(e) => {
                     setQuery(e.target.value);
                     startTransition(() => {
                          setList(
                             names.filter((name)
=>
                                 name
                                   .toLowerCase()
                                   .includes(e.ta
rget.value.
toLowerCase()),
                            ),
                        );
                   });
                }}
            />
            {isPending && <p>Loading...</p>}
            <ul>
                {list.map((name, index) => (
                     <li key={index}>{name}</li>
                ))}
            </ul>
      </div>
    );
}
```

The full code for this example is at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter03/use-transition.

For more information on **useTransition**, see the React documentation at https://react.dev/reference/react/useTransition.

useDeferredValue

Like **useTransition**, the **useDeferredValue** Hook helps optimize the UI. The **useDeferredValue** Hook is used to defer the updating of a primitive value.

The following example is similar to the **useTransition** example – it allows a user to filter a large list of names. However, **useDeferredValue** is used to optimize the UI.

In this example, the filtered list isn't stored in state – instead, it is calculated on every render in the **List** component. So, the **List** component is slow to render on large lists.

useDeferredValue defers the updating of **deferredQuery**, which, in turn, defers the rendering of **List**. This allows the search input to feel less laggy when typed into:

```
function App() {
    const [query, setQuery] = useState("");
    const deferredQuery =
useDeferredValue(query);
```

```
    return (
        <div>
            <input ...
                value={query}
                 onChange={(e) => {
                     setQuery(e.target.value);
                 }}
            />
            {query !== deferredQuery &&
<p>Loading...</p>}
            <List query={deferredQuery} />
        </div>
    );
}
```

The **List** component is wrapped in React's **memo** function so that it isn't re-rendered when **query** changes and is only re-rendered when **deferredQuery** changes:

```
const List = memo(function List({ query }: ... )
{
    const list = names.filter((name) =>
         name.toLowerCase().includes(query.toLowe
rCase()),
    );
    return (
        <ul>
            {list.map((name, index) => (
                <li key={index}>{name}</li>
           ))}
        </ul>
    );
});
```

The full code for this example is at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter03/use-deferred-value.

For more information on **`useDeferredValue`**, see the React documentation at https://react.dev/reference/react/useDeferredValue.

Hooks covered in other chapters

We will cover the following hooks in detail in some of the later chapters:

- **`useContext`**: This is used to access a React context, which we cover in *Chapter 10*, *State Management*.
- **`useOptimistic`**: This is used for optimistic UI updates before confirmation from the server. It is covered in *Chapter 9*, *Working with Forms*.
- **`useActionState`**: This is commonly used for form state. It is covered in *Chapter 9*, *Working with Forms*.
- **`useFormStatus`**: This tracks the status of form submission and is also covered in *Chapter 9*, *Working with Forms*.

Next, we will summarize the chapter.

Summary

In this chapter, we learned that all React Hooks must be called at the top level of a function component and can't be called conditionally.

The `useEffect` Hook can be used to execute component side effects when it is rendered. We learned how to use `useEffect` to fetch data, which is a common use case.

`useReducer` is an alternative to `useState` for using state, and we experienced using both approaches in our `PersonScore` example component. `useState` is excellent for primitive state values. `useReducer` is great for complex object state values, particularly when state changes depend on previous state values.

The ref Hook creates a mutable value and doesn't cause a re-render when changed. We used `useRef` to set `focus` to an HTML element after it was rendered, which is a common use case.

The `useMemo` and `useCallback` Hooks can be used to memoize values and functions, respectively, and can be used for performance optimization. The examples we used for these Hooks were a little contrived and using `useCallback` didn't improve performance, so remember to check that the use of these Hooks does improve performance.

So far in this book, the components we have created are unstyled. In the next chapter, we will learn several approaches for styling

React components.

Questions

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

1. The following component renders some text for 5 seconds. This is problematic, though. What is the problem?

```
export function TextVanish({ text }: Props) {
    if (!text) {
        return null;
    }
    const [textToRender, setTextToRender] =
useState(text);
    useEffect(() => {
        setTimeout(() => setTextToRender(“”),
5000);
    }, []);
    return <span>{textToRender}</span>;
}
```

2. The following code is a snippet from a React component that fetches some data and stores it in state. There are several problems with this code, though. Can you spot any of the problems?

```
const [data, setData] = useState([]);
useEffect(async () => {
    const data = await getData();
    setData(data);
});
```

3. Consider the following **`reducer`** function:

```
type State = { steps: number };
type Action =
    | { type: 'forward'; steps: number }
    | { type: 'backwards'; steps: number };
function reducer(state: State, action:
Action): State {
    switch (action.type) {
        case 'forward':
            return { ...state, steps:
state.steps + action.steps };
        case 'backwards':
            return { ...state, steps:
state.steps - action.steps };
        default:
            return state;
    }
}
```

What will the type of the **`action`** parameter be narrowed down to in the **`'backwards'`** switch branch?

4. Consider the following **`Counter`** component:

```
export function Counter() {
    const [count, setCount] = useState(0);
    const memoCount = useMemo(() => count,
[]);
    return (
        <div>
             <button onClick={() =>
setCount(count + 1)}>
                  {memoCount}
```

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

What will the button content be after it is clicked once?

5. Consider the following **`Counter`** component:

```
    export function Counter() {
        const [count, setCount] = useState(0);
        const handleClick = useCallback(() => {
            setCount(count + 1);
        }, []);
        return (
            <div>
                <button onClick={handleClick}>
    {count}</button>
            </div>
        );
    }
```

What will the button content be after it is clicked twice?

Answers

Here are the answers to the preceding questions:

1. The problem with the component is that both **`useState`** and **`useEffect`** are called conditionally (when the **`text`** prop is defined), and React doesn't allow its Hooks to be called conditionally. Placing the Hooks before the **`if`** statement resolves the problem:

```
export function TextVanish({ text }: Props) {
    const [textToRender, setTextToRender] =
useState(text);
    useEffect(() => {
        setTimeout(() => setTextToRender(""),
5000);
    }, []);
    if (!text) {
        return null;
    }
    return <span>{textToRender}</span>;
}
```

2. The main problem with the code is that the effect function can't be marked as asynchronous with the **`async`** keyword. A solution is to revert to the older promise syntax:

```
const [data, setData] = useState([]);
useEffect(() => {
    getData().then((theData) =>
setData(theData));
});
```

The other major problem is that no dependencies are defined in the call to **`useEffect`**. This means the effect function will be executed on every render. The effect function sets some state, which causes a re-render. So, the component will keep re-rendering, and the effect function will keep executing indefinitely. An empty array passed into the second argument of **`useEffect`** will resolve the problem:

```
useEffect(() => {
    getData().then((theData) =>
setData(theData));
}, []);
```

Another problem is that the **`data`** state will have the **`any[]`** type, which isn't ideal. In this case, it is probably better to explicitly define the type of the state as follows:

```
const [data, setData] = useState<Data[]>([]);
```

The last problem is that the data state could be set after the component has been unmounted, which can lead to memory leaks. A solution is to set a flag when the component is unmounted and not set the state when the flag is set:

```
useEffect(() => {
    let cancel = false;
    getData().then((theData) => {
        if (!cancel) {
            setData(theData);
        }
    });
    return () => {
        cancel = true;
    };
}, []);
```

3. TypeScript will narrow the type of the **`action`** parameter to **`{ type: 'backwards'; steps: number }`** in the **`'backwards'`** switch branch.

4. The button content will always be **`0`** because the initial count of **`0`** is memoized and never updated.

5. The button content will be `1` after one click and will stay `1` after subsequent clicks. So, after two clicks, it will be `1`.

   The key here is that the `handleClick` function is only created when the component is initially rendered because `useCallback` memoizes it. So, the `count` state will always be `0` within the memoized function. This means the `count` state will always be updated to `1`, which will appear in the button content.

Part 2: App Fundamentals

This part covers key fundamental topics for building apps, starting with different styling approaches and their benefits. The use of React Server Components is then explored, and the use cases where they shine. The part concludes with building an app with multiple pages using the popular Next.js React framework.

This part has the following chapters:

4
Approaches to Styling React Frontends

React doesn't offer a standard styling mechanism, and the community has developed many different approaches. In this chapter, we will learn a few of the most popular styling approaches.

We will use different approaches to style the alert component we worked on in previous chapters. First, we will use plain CSS and understand the pros and cons of this approach. Then, we will move on to use **CSS modules**, which will resolve plain CSS's main problem. Next, we'll use a library called **Tailwind CSS**, again understanding its pros and cons. Lastly, we'll touch on a few other noteworthy styling approaches.

Additionally, we are going to learn how to use SVGs in React apps and use them in the alert component for the information and warning icons.

By the end of this chapter, you will have the knowledge to pick an appropriate styling approach for future React projects you work on.

We'll cover the following topics:

- Using plain CSS

- Using CSS modules
- Using Tailwind CSS
- Using SVGs
- Other styling approaches

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-Third-Edition/tree/main/Chapter04.

Using plain CSS

We will style the alert component we created in *Chapter 2*, *Getting Started with TypeScript*, using plain CSS. We will look at one of the challenges with plain CSS and discover how we could mitigate it.

Creating the project

We'll use the alert component we completed at the end of *Chapter 2*, *Getting Started with TypeScript*. We will create a new Vite React and TypeScript project and copy the alert component into it. Let's carry out the following steps:

1. In a terminal, in a folder of your choice, execute the following command to instruct Vite to create a React and TypeScript project:

   ```
   npm create vite@latest alert -- --template
   react-ts
   ```

2. The project is created. Execute the following commands in the terminal to move the working directory to the **`alert`** folder, install the project dependencies, and open the project in Visual Studio Code:

   ```
   cd alert
   npm i
   code .
   ```

3. Feel free to add automatic code formatting. We covered this topic with Prettier in *Chapter 1*, *Getting Started with React*.

4. Run the app in development mode by executing the following command in a terminal:

   ```
   npm run dev
   ```

5. Create a new file in the **`src`** folder called **`Alert.tsx`** and copy-paste the contents from https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter04/start/Alert.tsx.

   This is the last version of the alert component we worked on.

6. Open **`App.tsx`** and replace the content with the following:

```
import { Alert } from './Alert';
import './App.css';
export default function App() {
  return (
    <Alert heading="Success" closable>
      Everything is really good!
    </Alert>
  );
}
```

The project is now created and running in development mode. Next, we will take some time to understand how to use plain CSS in React components.

Understanding how to reference CSS

Vite has already enabled the use of plain CSS, and it does already use some of it in the project. Carry out the following steps to explore how plain CSS is referenced in a React app:

1. Open the **`main.tsx`** file and notice the **`index.css`** import statement:

   ```
   import './index.css'
   ```

   The import statement is a little different from importing a JavaScript or TypeScript module – it imports *all* the CSS from the file rather than part of the code in the file.

2. Open **`index.css`** and notice that it contains CSS rules that target elements outside of the React app. For example, there are CSS rules targeting the **`body`** element.

   Another CSS file used is **`App.css`**, which is imported into **`App.tsx`**. Most styles in **`App.css`** are redundant because they target elements we removed. For example, the logo CSS class is now redundant.

   We will keep the redundant CSS in place for now. Eventually, we'll check whether the build process strips it out.

3. Open **`App.css`** and add the following highlighted properties to the **`card`** class to add a rounded border:

   ```
   .card {
     padding: 2em;
     border-radius: 8px;
     border: 1px solid #c1c1c1;
   }
   ```

4. We will use the **`card`** CSS class in the **`App`** component. Open **`App.tsx`** and wrap a **`div`** element with a **`card`** CSS class around **`Alert`** as follows:

   ```
   function App() {
     return (
       <div className="card">
         <Alert ... >...</Alert>
       </div>
   ```

```
    );
  }
```

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.

In the running app, a border now appears around the alert:

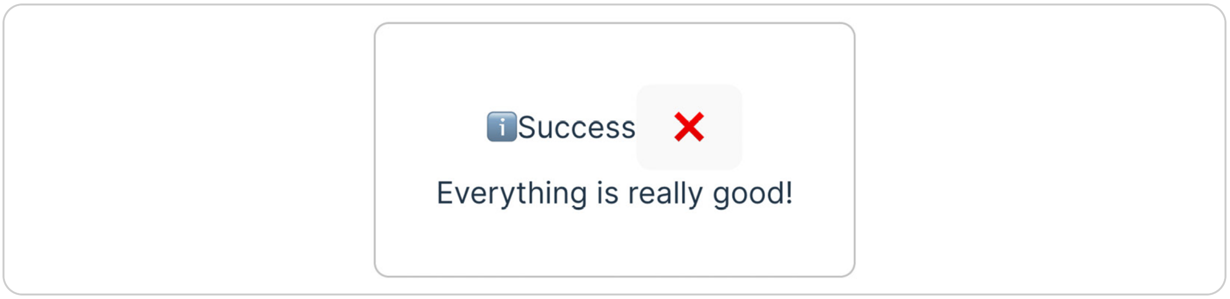


Figure 4.1 – Alert with a border around

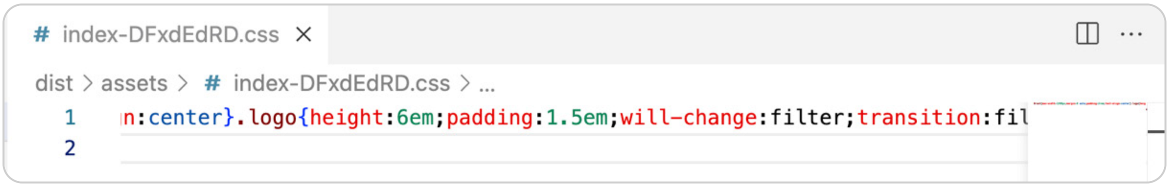
5. Now, stop the app from running by pressing *Ctrl* + *C* in the terminal.
6. Run the following command in the terminal to produce a production build:

```
npm run build
```

   After a few seconds, build artifacts will appear in a **`dist`** folder at the project's root.

7. Open **`index.html`** in the **`dist`** folder. Find the **`link`** element that references the CSS file, and note down the path – it will be something similar to **`/assets/index-DFxdEdRD.css`**.
8. 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 **logo** CSS class:



```
n:center}.logo{height:6em;padding:1.5em;will-change:filter;transition:fil
```

Figure 4.2 – The bundled CSS file, including the redundant logo CSS class

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

Next, we'll 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 to copy on GitHub at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter04/using-plain-css/src/Alert.css.
2. We are going to 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: 1em;
  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: 0.1em;
  }
```

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 **`22px`**:

```
  .header-icon {
    width: 22px;
  }
```

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 {
    margin-left: auto;
    border: none;
    display: flex;
    align-items: center;
    justify-content: center;
    background: transparent;
    height: 24px;
    width: 24px;
    padding: 0px;
    cursor: pointer;
  }
```

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

8. Add the following class for the **`content`** element:

```
  .content {
    margin: 0 1.2em 0 1.4em;
    color: #000;
  }
```

This adds some margin so that 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 at the top of the file 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:

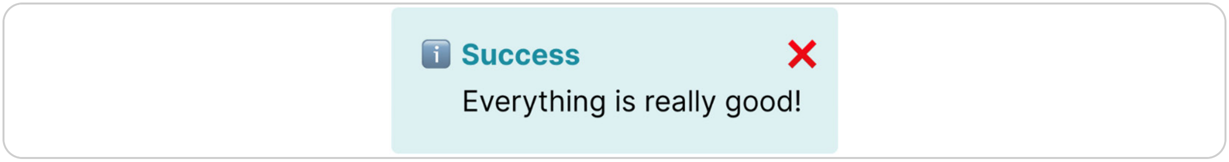
```
  <div className={`container ${type}`}>
    <div className="header">
      <span ... className="header-icon">
        {type === "warning" ? "⚠" : "ℹ"}
      </span>
```

```
        <span className="header-text">{heading}
    </span>
        {closable && (
          <button ... className="close-button">
            ...
          </button>
        )}
      </div>
      <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. Start the app in development mode by running **`npm run dev`** in the terminal.

    If you visit the app in the browser, an improved alert component will appear in the browser:



Success

Everything is really good!

Figure 4.3 – 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

Now, we'll 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 **"card"** to **"container"** on the **div** element:

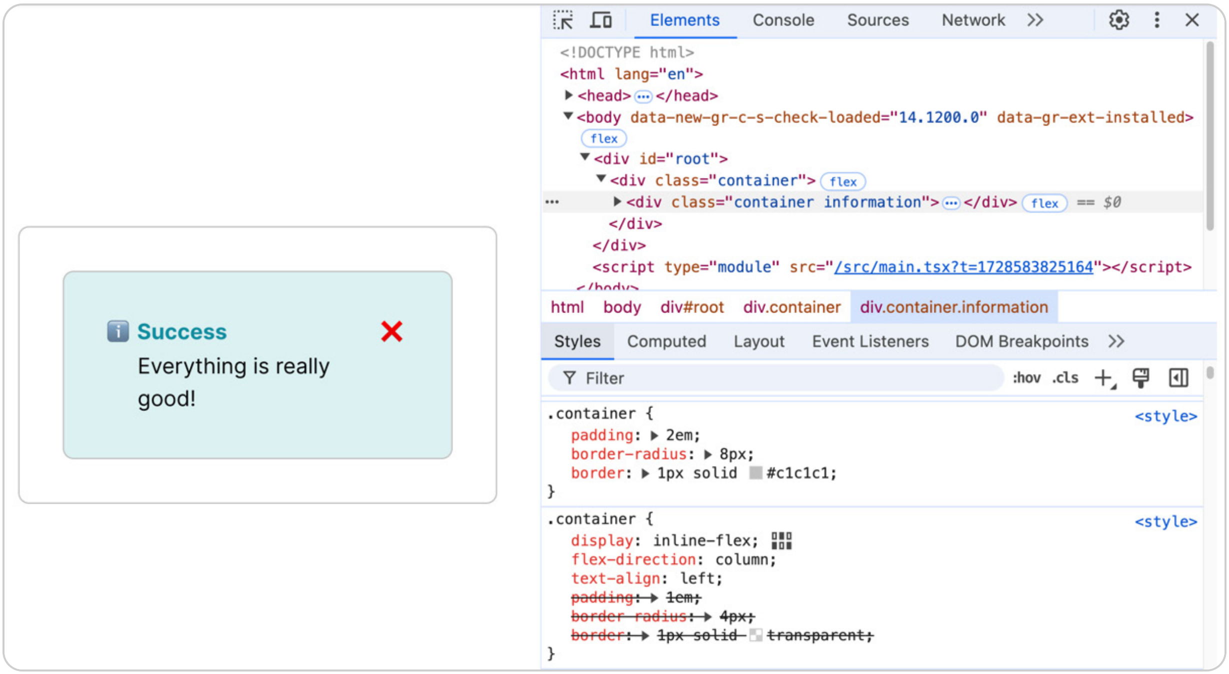
   ```
   <div className="container">
     <Alert ...>
       ...
     </Alert>
   </div>
   ```

2. Open **App.css** and change the **card** CSS class name to **container**:

   ```
   .container {
     padding: 2em;
     border-radius: 8px;
     border: 1px solid #c1c1c1;
   }
   ```

   Now, look at the running app and notice that the alert now has a gray border, its corners are a little more rounded, and there is a little more padding.

3. Inspect the elements using the browser development tools. Notice that styles from the **container** CSS class in **App.css** are leaking into the alert component, causing its padding and border to be overridden.



```
<!DOCTYPE html>
<html lang="en">
  ▶<head>…</head>
  ▼<body data-new-gr-c-s-check-loaded="14.1200.0" data-gr-ext-installed> flex
    ▼<div id="root">
      ▼<div class="container"> flex
        ▶<div class="container information">…</div> flex == $0
        </div>
      </div>
      <script type="module" src="/src/main.tsx?t=1728583825164"></script>
```

```
.container {                                   <style>
   padding: ▶ 2em;
   border-radius: ▶ 8px;
   border: ▶ 1px solid #c1c1c1;
}
.container {                                   <style>
   display: inline-flex;
   flex-direction: column;
   text-align: left;
   ~~padding: ▶ 1em;~~
   ~~border-radius: ▶ 4px;~~
   ~~border: ▶ 1px solid transparent;~~
}
```

Figure 4.4 – Clashing CSS classes

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, and 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:

- Using plain CSS is simple and familiar to most developers and works without any additional configuration in a Vite project
- CSS files can be imported in React component files using **`import 'path-to-css-file.css'`** syntax
- 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'll begin by understanding **CSS modules** and use them within the alert component 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 bundling 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 Vite 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 containing **`container`** and **`error`** CSS class names that have been imported into a component:

```
{
  container: "_container_16mbb_1",
  error: "_error_16mbb_7"
}
```

The scoped CSS class name includes 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 **`_error_16mbb_7`**. So, the styles in the running app will be the scoped style names and not the original class names.

Projects created using Vite already have CSS modules installed and configured. 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:

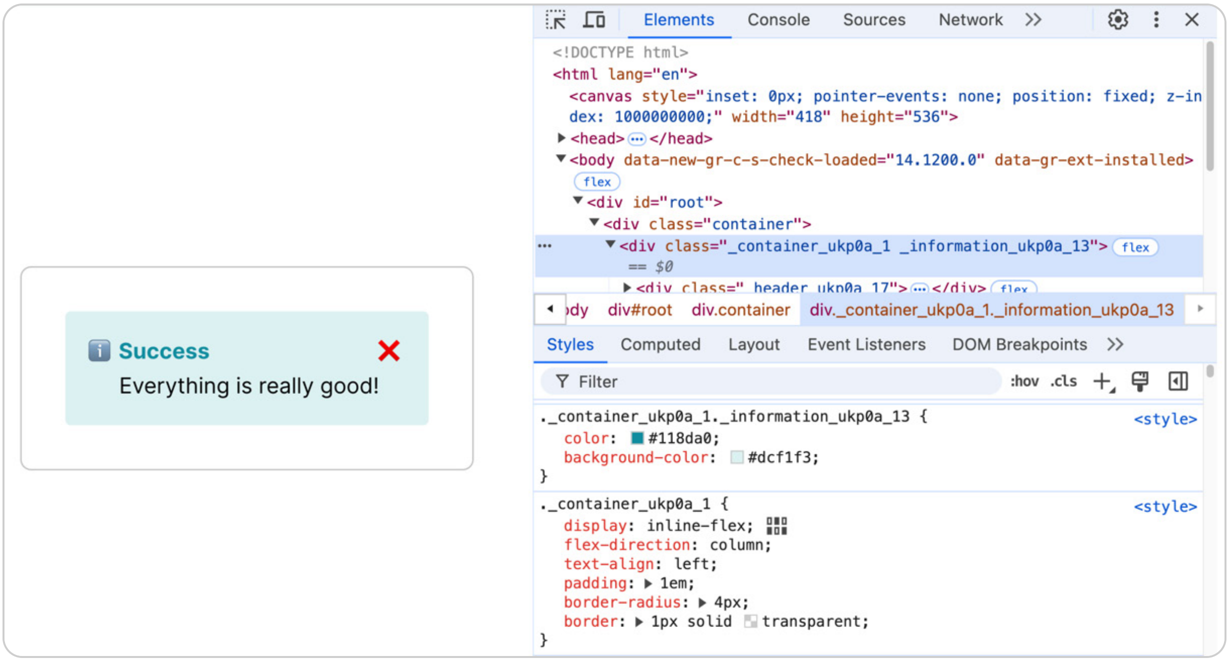
   ```
   <div className={`${styles.container}
   ```

```
${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. If the app isn't already running, start it by running **`npm run dev`** in the terminal.

   This time, the alert will no longer have its own gray border, 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.



```
<!DOCTYPE html>
<html lang="en">
  <canvas style="inset: 0px; pointer-events: none; position: fixed; z-in
  dex: 1000000000;" width="418" height="536">
 ▶<head>⋯</head>
 ▼<body data-new-gr-c-s-check-loaded="14.1200.0" data-gr-ext-installed>
   flex
   ▼<div id="root">
     ▼<div class="container">
       ▼<div class="_container_ukp0a_1 _information_ukp0a_13"> flex
          == $0
        ▶<div class="_header_ukp0a_17">⋯</div> flex
```

```
ody  div#root  div.container  div._container_ukp0a_1._information_ukp0a_13
```

```
._container_ukp0a_1._information_ukp0a_13 {                      <style>
   color: #118da0;
   background-color: #dcf1f3;
}
._container_ukp0a_1 {                                            <style>
   display: inline-flex;
   flex-direction: column;
   text-align: left;
   padding: ▶ 1em;
   border-radius: ▶ 4px;
   border: ▶ 1px solid transparent;
}
```

Figure 4.5 – 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`**:

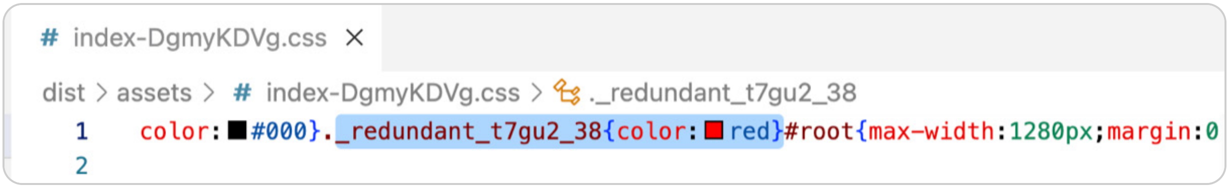
```
...
.content {
  margin-left: 1.4em;
  color: #000;
}
.redundant {
  color: red;
}
```

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

After a few seconds, build artifacts are created in the **`dist`** 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, like in development mode.
    - It contains the redundant CSS class name from the CSS module.



```
color:■#000}._redundant_t7gu2_38{color:■red}#root{max-width:1280px;margin:0
```

Figure 4.6 – The redundant CSS class included in the 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 bundling process. However, it is pre-installed and preconfigured in projects created with Vite.
- Within a CSS module file, you write plain CSS, which is familiar to most developers.
- 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 Tailwind CSS

In this section, we will start by understanding **Tailwind CSS** and its benefits. Next, we'll 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. **`md`** stands for *medium*. Alternatively, you could use **`lg`** (large) 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 over:

```
<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 @tailwindcss/vite
   ```

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

2. Open **`vite.config.ts`** and add the **`tailwindcss`** plugin:

   ```
   ...
   import tailwindcss from "@tailwindcss/vite";
   export default defineConfig({
     plugins: [react(), tailwindcss()],
   });
   ```

3. Now, open **`index.css`** in the **`src`** folder and replace the contents with the following line to add the base Tailwind CSS:

   ```
   @import 'tailwindcss';
   ```

4. Next, we will add a Visual Studio Code extension that enables IntelliSense for Tailwind CSS classes. Open up the Visual Studio Code extensions, search for the **Tailwind CSS IntelliSense** extension, and install it.

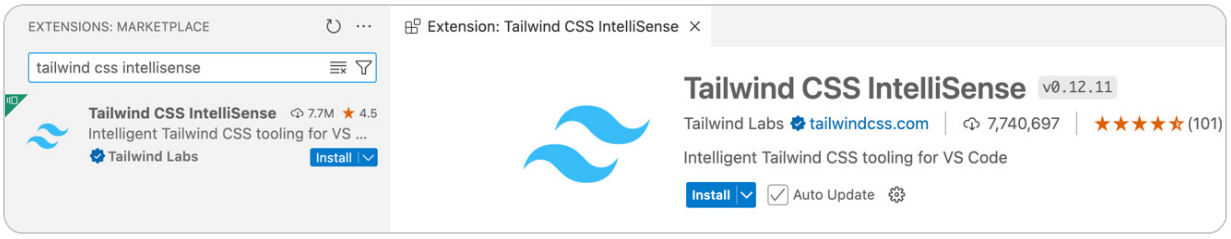

EXTENSIONS: MARKETPLACE
tailwind css intellisense
Tailwind CSS IntelliSense 7.7M ★ 4.5
Intelligent Tailwind CSS tooling for VS ...
Tailwind Labs
Install
Extension: Tailwind CSS IntelliSense
Tailwind CSS IntelliSense v0.12.11
Tailwind Labs tailwindcss.com 7,740,697 ★★★★★ (101)
Intelligent Tailwind CSS tooling for VS Code
Install
Auto Update

Figure 4.7 – Tailwind Visual Studio Code extension

The next two steps are only relevant if you are using Prettier to format code.

5. Let's add a Prettier plugin to sensibly sort CSS class names referenced on the **className** attribute during code formatting. Start by installing **prettier-plugin-tailwindcss** by executing the following command in the terminal:

```
npm i -D prettier-plugin-tailwindcss
```

6. Open the Prettier configuration file and add the following highlighted line to configure Prettier to use this plugin:

```
{
  ...,
  "plugins": ["prettier-plugin-tailwindcss"]
}
```

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 replace the use of CSS modules with Tailwind utility class names in the JSX **`className`** attribute. To do this, carry out the following steps:

1. Open **`App.tsx`** and remove the **`className`** attribute on the outermost **`div`** element. This removes the plain CSS from the **`App`** component.

2. Open **`Alert.tsx`** and start by removing the **`Alert.module.css`** import statement from the top of the file.

3. Update the **`className`** attribute on the outermost **`div`** element as follows:

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

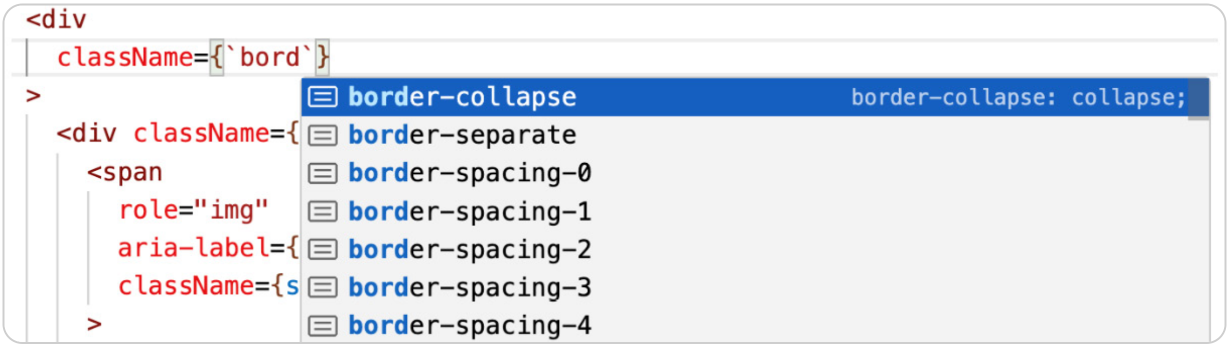
   Here is an explanation of the utility classes that were just used:

   - **`inline-flex`** and **`flex-col`** create an inline flexbox that flows vertically
   - **`text-left`** aligns items to the left
   - **`p-3`** adds three spacing units of padding
   - We have encountered **`rounded-md`** before – this rounds the corners of the **`div`** element

- **`border-1`** and **`border-transparent`** add a transparent **`1px`** 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`****.*

Notice the helpful IntelliSense as the class names are entered:



```
<div
  className={`bord`}
>
  <div className={
    <span
      role="img"
      aria-label={
      className={s
    >
```

```
border-collapse        border-collapse: collapse;
border-separate
border-spacing-0
border-spacing-1
border-spacing-2
border-spacing-3
border-spacing-4
```

Figure 4.8 – IntelliSense for Tailwind class names

Notice also that the class names will be sorted as per the preceding code snippet, regardless of the order in which they are entered.

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

```
<div
  className={`border-1 inline-flex flex-col
rounded-md border-transparent p-3 text-left ${
    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.

5. Next, update the **className** attribute on the header container as follows:

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

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

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

6. Update the **className** attribute on the icon as follows:

```
  <span role="img" ... className="w-7">
```

```
    {type === ‘warning’ ? ‘⚠’ : ‘ℹ’}
  </span>
```

The **`w-7`** instance sets the element to a width of seven spacing units.

7. Update the **`className`** attribute on the heading as follows:

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

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

8. Update the **`className`** attribute on the close button as follows:

```
  {closable && (
    <button
      ...
      className="ml-auto flex h-6 w-6 cursor-
  pointer  items-center     justify-center
  border-none bg-transparent p-0"
    >
      ...
    </button>
  )}
```

Here, **`border-none`** removes the element border, and **`bg-transparent`** makes the element background transparent. The **`ml-auto`** instance sets the left margin to automatic, which right-aligns the element. The **`flex`**, **`items-center`**, and **`justify-center`** instances center the content within the button. The **`h-6`** and **`w-6`** instances size the button, and **`p-0`** removes its padding. The **`cursor-pointer`** instance sets the mouse cursor to a pointer.

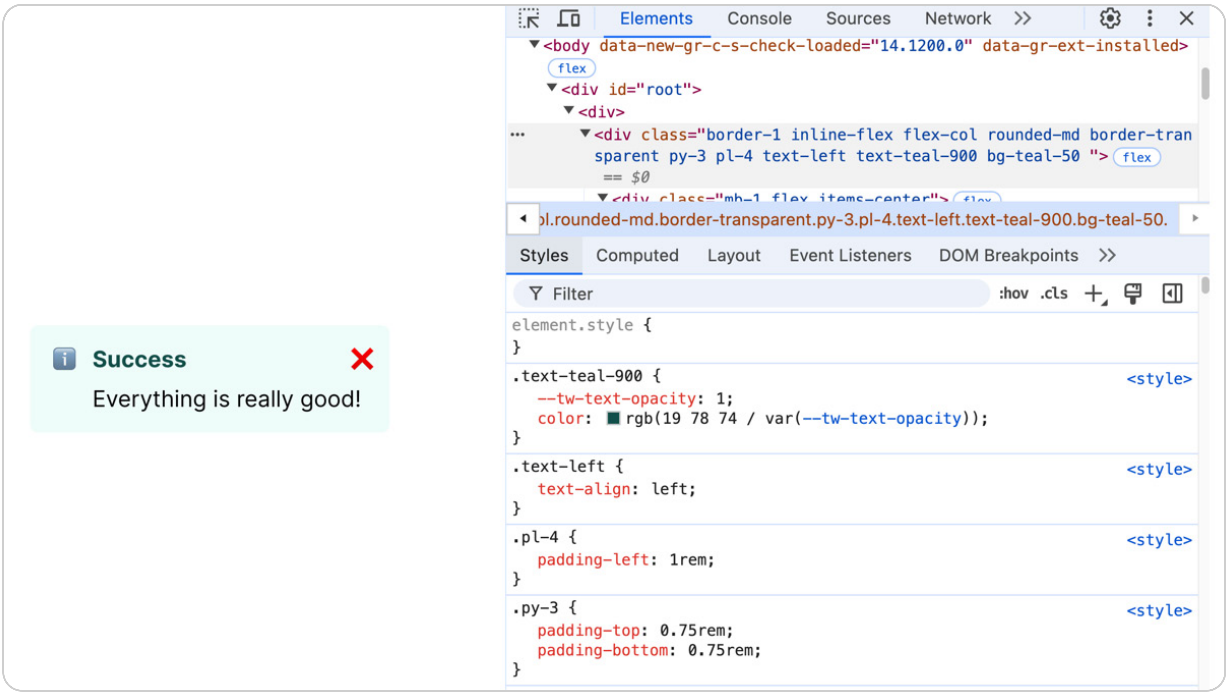
9. Finally, update the **`className`** attribute on the message container as follows:

```
<div className="ml-7 pr-5 text-black">
  {children}
</div>
```

   The **`ml-7`** instance sets the left margin on the element to seven spacing units, **`pr-5`** sets the right padding to five spacing units, and **`text-black`** sets the text color to black.

10. Run the app by running **`npm run dev`** in the terminal. After a few seconds, the app will appear in the browser.
11. Inspect the elements in the DOM using the browser's DevTools. Notice the Tailwind utility classes being used.

    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 component.



```
▼<body data-new-gr-c-s-check-loaded="14.1200.0" data-gr-ext-installed>
  flex
  ▼<div id="root">
    ▼<div>
      ▼<div class="border-1 inline-flex flex-col rounded-md border-tran
       sparent py-3 pl-4 text-left text-teal-900 bg-teal-50 "> flex
        == $0
       ▼<div class="mb-1 flex items-center"> flex
```

```
ol.rounded-md.border-transparent.py-3.pl-4.text-left.text-teal-900.bg-teal-50.
```

```
element.style {
}
.text-teal-900 {                                          <style>
   --tw-text-opacity: 1;
   color: ■rgb(19 78 74 / var(--tw-text-opacity));
}
.text-left {                                              <style>
   text-align: left;
}
.pl-4 {                                                   <style>
   padding-left: 1rem;
}
.py-3 {                                                   <style>
   padding-top: 0.75rem;
   padding-bottom: 0.75rem;
}
```

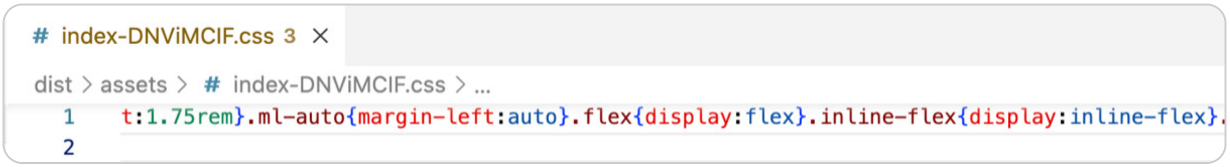
Figure 4.9 – A styled alert using Tailwind

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

13. 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, build artifacts are created in the **`dist`** folder.

14. Open the bundled CSS file from the **`dist/assets`** folder. Notice the base Tailwind styles and all the Tailwind classes we used in this file.



```
t:1.75rem}.ml-auto{margin-left:auto}.flex{display:flex}.inline-flex{display:inline-flex}.
```

Figure 4.10 – 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 requires installing and configuring in a Vite project.
- Tailwind is a well-thought-through collection of reusable CSS classes that can be applied to React elements. There is a lot to learn, but the documentation is very helpful.
- One downside of Tailwind is that the JSX can be harder to read when it contains many Tailwind CSS class references.
- Tailwind has a nice default color palette and a `4px` spacing scale, which gives a nice look and feel.
- Only classes used on React elements are included in the CSS build bundle. For this reason, and the high reusability, the CSS bundle is generally very small.

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.

The simplest way to use SVGs in React is to add an **`svg`** element directly in JSX. The following example is an upload button containing an up-arrow SVG:

```
function Upload() {
  return (
    <div className=... >
      <button className=... >
        <svg
          width="24"
          height="24"
          viewBox="0 0 24 24"
          fill="none"
          xmlns="http://www.w3.org/2000/svg"
        >
        <path
          d="M12 4L4 12H9V20H15V12H20L12 4Z"
          fill="white"
        />
        </svg>
      </button>
    </div>
  );
}
```

The downside of this approach is that **`svg`** elements can contain a lot of markup, making the React component hard to read. It's more convenient to reference an SVG file that is located outside the React component file. Vite allows SVG files to be imported without any additional configuration. In fact, a couple of SVGs were referenced in the Vite React template **`App`** component as follows:

```
import reactLogo from './assets/react.svg'
import viteLogo from './vite.svg';
...
function App() {
  ...
  return (
    ...
    <img src={viteLogo} ... />
    ...
    <img src={reactLogo} ... />
    ...
  );
}
export default App;
```

In the preceding example, **`viteLogo`** 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.

Notice the path in the **`viteLogo`** import statement and where the **`vite.svg`** file is in the project – it's in the **`public`** folder. The **`public`** folder is a special folder where assets are served at the root path (`/`), and it is intended for assets that aren't referenced in source code or for assets whose names should remain unchanged during the bundling process.

A downside of referencing an SVG in an **`img`** element is that you lose the ability to style the SVG. Another approach is to create a reusable SVG React component with the SVG embedded. The following example is a React component for an up-arrow icon:

```
export function UpArrow() {
  return (
    <svg
      width="24"
      height="24"
      viewBox="0 0 24 24"
      fill="none"
      xmlns="http://www.w3.org/2000/svg"
    >
    <path
      d="M12 4L4 12H9V20H15V12H20L12 4Z"
      fill="white"
    />
    </svg>
  );
}
```

The SVG React component can then be consumed in other components without the SVG bloating the markup, as follows:

```
import { UpArrow } from "/icons/UpArrow";
function Upload() {
  return (
     <div className=... >
        <button className=... >
           <UpArrow />
        </button>
    </div>
  );
}
```

In fact, there is a Vite plugin called **`vite-plugin-svgr`** that allows you to import SVG files as React components, as in the following example:

```
import UpArrow from "/icons/uparrow.svg?react";
function Upload() {
  return (
    <div className=... >
      <button className=... >
        <UpArrow />
      </button>
    </div>
  );
}
```

See the **`vite-plugin-svgr`** GitHub repository for more details: https://github.com/pd4d10/vite-plugin-svgr.

Next, we will 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 empty files called **`cross.svg`**, **`info.svg`**, and **`warning.svg`** in the **`src/assets`** folder. Then, copy and paste the content of these from the GitHub repository at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter04/using-svgs/src/assets.

2. Open **`Alert.tsx`** and add the following import statements to import the SVGs:

   ```
   import crossIcon from './assets/cross.svg';
   import infoIcon from './assets/info.svg';
   import warningIcon from
   './assets/warning.svg';
   ```

   We have given the SVG components appropriately named aliases.

3. Replace the **`span`** element containing the emoji icons with an **`img`** element conditionally referencing the SVGs as follows:

   ```
   <div className="mb-1 flex items-center">
     <img
       src={type === 'warning' ? warningIcon :
   infoIcon}
       alt={type === 'warning' ? 'Warning' :
   'Information'}
       className="mr-1 h-6 w-6"
   ```

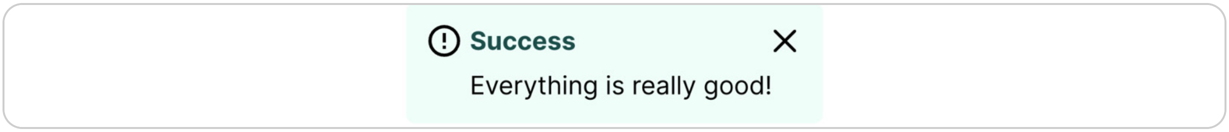
```
      />
      <span className="font-bold">{heading}
      </span>
    </div>
```

We have used Tailwind to size icons appropriately and add a gap between them and the heading.

4. Next, replace the emoji close icon with the SVG close icon as follows:

```
    <button ... >
      <img src={crossIcon} alt="Close" />
    </button>
```

5. Run the app by running **`npm run dev`**. The app in the browser will contain the alert component with the SVG icons.



Success

Everything is really good!

Figure 4.11 – 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:

- SVGs are commonly used for icons in an app to ensure a professional look
- SVG files can be imported into a React component and then referenced within an **`img`** element

Next, we touch on a few other styling approaches.

Other styling approaches

In this section, we'll cover other popular styling approaches.

Using inline styles

The **`style`** attribute can be used to style JSX elements. It is set to a JavaScript object, where CSS properties are written in camelCase with string or number values.

Here is an example of the alert container styled using this approach:

```
<div
  style={{
    display: "inline-flex",
    flexDirection: "column",
    textAlign: "left",
    padding: "1em",
    borderRadius: "4px",
    border: "1px solid transparent",
    backgroundColor:
      type === "warning" ? "#f3e8da" : "#dcf1f3",
      color: type === "warning" ? "#e7650f" :
"#118da0",
  }}
>
  ...
</div>;
```

The benefits of this approach are the following:

- It's simple and familiar to most developers
- There's no build step required
- The styles are scoped to the component

However, there are some drawbacks:

- CSS pseudo classes (such as **`:hover`**) and pseudo elements (such as **`::before`**) can't be used.
- It's generally less performant because no style sheet can be cached. It can also lead to unnecessary style duplication, increasing the DOM size.

Here's a link to the complete code for the alert component styled using inline styles: https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter04/using-inline-styles.

Using SCSS

SCSS is a superset of CSS that has historically been very popular because of features such as variables, nesting, and mixins. However, variables and nesting have now become available in native CSS, so SCSS is a less popular option for new React projects.

Here's the alert container styled using the nesting feature in SCSS:

```
container {
    display: inline-flex;
    flex-direction: column;
    text-align: left;
    padding: 1em;
    border-radius: 4px;
    border: 1px solid transparent;
    &.warning {
      color: #e7650f;
      background-color: #f3e8da;
    }
    &.information {
      color: #118da0;
      background-color: #dcf1f3;
    }
  }
```

Vite does support SCSS, but a **`sass-embedded`** package does need to be installed. The SCSS is placed in a file with a **`.scss`** file extension and imported into a component in the same way as a **`.css`** file.

SCSS doesn't automatically scope styles, so it suffers from the same clashing problem as plain CSS.

Here's a link to the code for the alert component styled using SCSS: https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter04/using-scss.

For more information on SCSS, see the documentation at https://sass-lang.com.

Using CSS-in-JS

CSS-in-JS is a styling technique where the styles are written directly in JSX. So, it's similar to inline CSS, but a build step moves the inline styles from the HTML elements into a **`style`** element. The HTML elements then reference the styles using a scoped CSS class name.

The **`styled-components`** library is a very popular CSS-in-JS library. Here's the alert container using **`styled-components`**:

```
const Container = styled.div<{ type: string }>`
  display: inline-flex;
  flex-direction: column;
  text-align: left;
  padding: 1em;
  border-radius: 4px;
  border: 1px solid transparent;
  ${(props) =>
    props.type === "warning" &&
    `
    color: #e7650f;
    background-color: #f3e8da;
  `}
  ${(props) =>
    props.type === "information" &&
    `
    color: #118da0;
    background-color: #dcf1f3;
  `}
`;
```

The **`Container`** component is a regular React component that can take in props – in this case, a **`type`** prop applies the container for the specified alert type. It uses a template literal to define a styled **`div`** element with CSS directly in the React component.

Here's a link to the code for the alert component styled using **`styled-components`**: https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter04/using-css-in-js.

A downside of **`styled-components`** is that styles are applied at runtime rather than at build time, resulting in a minor performance penalty. Another downside is it can't be used in React Server Components, which we'll cover in *Chapter 5*, *Using React Server and Client Components*. The project has also been recently placed in *maintenance mode*.

For more information on **`styled-components`**, see its documentation at https://styled-components.com.

Newer CSS-in-JS libraries, such as **StyleX** (https://stylexjs.com) and **Panda** (https://panda-css.com), address the downsides of **`styled-components`** but haven't gained as much traction at the time of writing this book.

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

Summary

In this chapter, we learned three methods of styling and their pros and cons.

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 allow us to write plain CSS files imported in a way that scopes styles to the component, allowing it to be used anywhere in the app. We learned that CSS modules is an open source library pre-installed and preconfigured in projects created with Vite. We saw how this resolved the CSS clashing problem but didn't remove redundant styles.

We looked at styling with the popular Tailwind CSS library. 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 **`4px`** spacing scale, both of which can be customized. We learned that only the used Tailwind classes are included in the production build.

We touched on a few other styling approaches. Inline CSS is convenient but has performance penalties and doesn't allow pseudo classes or pseudo elements. SCSS and CSS-in-JS were very popular in the past, so you may use them in existing code bases.

Finally, we learned that Vite enables the use of SVG files. SVGs can be imported and referenced as a path in an **`img`** element.

An important note is that styling doesn't impact component reusability. Our alert component can still be used in many parts of an app and even in different apps.

In the next chapter, we will look at React Server Components.

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 './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?

3. 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?

4. A logo SVG is referenced as follows:

```
import Logo from './logo.svg';
function SomeComponent() {
  return (
    <div>
      <Logo />
    </div>
  );
}
```

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

5. 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 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>
}
```

3. 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
```

4. The **Logo** instance will hold the path to the SVG rather than a component. The import statement can be adjusted as follows to import a React component if the **vite-plugin-svgr** plugin is used:

```
import Logo from './logo.svg?react';
function SomeComponent() {
  return (
    <div>
      <Logo />
    </div>
  );
}
```

5. The **bg-${color}-500** class name is problematic as 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.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://packt.link/GxSkC

5
Using React Server and Client Components

In this chapter, we will learn all about React Server and Client Components. We'll start by understanding the problem that Server Components aim to solve before exploring their capabilities and when to use them. Additionally, we will learn how to nest them inside each other to create a performant app with the capabilities we need.

We'll also start to use a React-based app framework called Next.js and gain hands-on experience creating both Server and Client Components.

We'll cover the following topics:

- Understanding SPAs
- Understanding Server Components
- Creating Server Components
- Exploring Client Components
- Composing Server and Client Components

Technical requirements

We will be using 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-Third-Edition/tree/main/Chapter05.

Understanding SPAs

Before diving into Server Components, let's understand the problems they aim to solve, all of which stem from **single-page applications** (**SPAs**).

Understanding the SPA problem

The SPA problem occurs when the first page is loaded. The following diagram visualizes the steps to load the first page in an SPA:

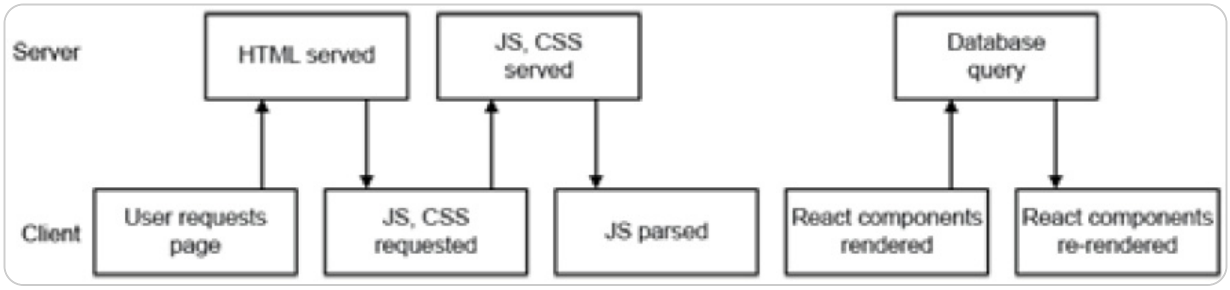


Figure 5.1 – React SPA loading steps

Here's a description of the steps:

1. The process starts with the user requesting a page from the server and an HTML page being downloaded. A typical SPA loads an HTML page with very little content other than references to JavaScript and CSS files. Usually, the HTML won't contain any meaningful content to the user.
2. The JavaScript and CSS referenced in the HTML are requested and downloaded from the server. The JavaScript contains code to render the app because the elements are rendered on the client. In a React app, this JavaScript will contain both React and the app, meaning there may be a lot of JavaScript. The CSS contains styles for the rendered elements.
3. Once the JavaScript has been downloaded, it is parsed and executed, which results in the component tree being rendered in a React app. The rendering process is often referred to as **client-side rendering** (**CSR**) in an SPA.
4. Generally, an app will fetch some data from a database to display on the first page. So, after the initial rendering in a React app, a data fetching process will occur, followed by component re-rendering to display the data on the page.

The main problem with SPAs is that a lot happens before the user sees meaningful content on the page – typically, at least three network requests! This inefficient process is a potential performance problem. If any of the steps take a long time, the problem can be significant. For example, downloading, parsing, and executing a large amount of JavaScript can be problematic. Slow loading performance not only frustrates users but can impact SEO.

It is worth noting that bundlers such as Vite can split the app into separate bundles to mitigate the first load performance problem. Bundles are loaded on demand as the user interacts with the app. However, this solution can cause a knock-on problem – sluggish user interactions as the bundles are lazily loaded.

Understanding the benefits of SPAs

Before we start to cover **React Server Components** (**RSCs**), it is important to understand that SPAs aren't all bad – it's been a popular architecture for well over a decade, and they still offer significant benefits today. In fact, the apps we've built so far in this book are SPAs. Here are some of the good parts of SPAs:

- Once the initial JavaScript is loaded and executed, SPAs are fast. UI elements are updated on the client, and any data fetching happens without a full page reload. In addition, page navigation happens on the client without a server request.

- It scales well because the server load doesn't contain any HTML or JavaScript processing – that work is distributed to each client.
- Frontend-backend separation is easy, meaning the backend can be a technology other than JavaScript.
- Hosting an SPA is simple and cheap because it's just a set of static resources. A CDN can be used, enabling global delivery and potentially reducing costs further because of the reduced server requests and reduced bandwidth usage.

Next, it's time to learn about RSCs and how they address the SPA problem.

Understanding Server Components

In this section, we will learn what RSCs are and their benefits.

Understanding what a Server Component is

RSCs were first introduced in the experimental React 18 version and fully released in React 19. React components before this version were all Client Components.

The React components we have built so far in this book aren't RSCs – instead, they are Client Components. We'll dive deeper into how Client Components can work alongside RSCs later in this chapter.

Here's an example of an RSC:

```
export async function Person({ id }: { id: string
}) {
    const result = await db
        .select({ name: people.name, notes:
people.notes })
        .from(people)
        .where(eq(people.id, id));
    if (result.length === 0) {
        return null;
    }
    const person = result[0];
    return (
        <main>
             <h1>{person.name}</h1>
            <p>{person.notes}</p>
        </main>
    );
}
```

At first glance, an RSC looks like a regular React component, similar to what we have already been building. Note the following in particular:

- The component is a function that takes in props – in this case, **`id`**
- The component returns JSX

However, there are some obvious differences:

- The function is asynchronous. Client Components can only be synchronous – RSCs can be asynchronous or synchronous.

- The function is accessing a database on the server. Client Components can't access server resources directly.

Here are some other key points about RSCs that aren't apparent from the preceding code snippet:

- They run *exclusively* on the server.
- They don't re-render. They are run once on the server to generate the UI, which is then downloaded to the client.
- They can't be very interactive – React state, effects, and events aren't allowed. However, HTML-native interactivity, such as basic forms, can be used. Interactions that require JavaScript can't be used.
- The JavaScript bundle doesn't include the component's JavaScript code, which is executed in the rendering process. Only the result of the RSC function call is downloaded to the client, which is the generated UI.

RSCs have taken a couple of years to be fully released and available after being introduced in an experimental version. This is because they require pieces outside of React, such as a bundler and a web server that supports RSCs.

This brings us to an architectural requirement for RSCs – a web server capable of executing an RSC is required. So, non-JavaScript-based servers will require an additional server for RSCs. In addition, even if the server is JavaScript-based, the web framework that's used needs to support RSCs, and not many do at the time of writing this book.

Now that we have started to understand what RSCs are, next, we'll cover their benefits in depth.

Understanding how RSCs address the SPA problem

The following diagram visualizes the steps to load the first page where an RSC is used:

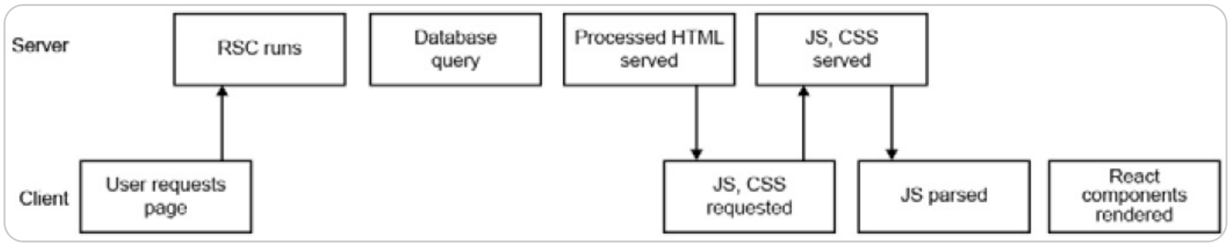


Figure 5.2 – RSC loading steps

Here's a description of the steps:

1. The process starts with the user requesting a page from the server. The server runs the RSC, executing any database queries. The processed HTML page is then downloaded. So, the HTML will probably contain meaningful content for the user, unlike an SPA.
2. The JavaScript and CSS referenced in the HTML are requested and downloaded from the server. The amount of JavaScript will generally be less than what you'd receive from an SPA because some of it will already

have been executed and not needed on the client. The JavaScript will contain Client Components, something we'll cover later in this chapter.

3. Once the JavaScript has been downloaded, it is parsed and executed, which results in Client Components being hydrated. We'll cover hydration later in this chapter, but it's important to note that it's less work than fully rendering the components.

Now that we have started to understand RSCs and how they address the SPA problem, let's explore their benefits further.

Understanding the benefits of Server Components

RSCs offer significant benefits. Here are some valuable tasks an RSC can do:

- Fetch data directly from a database
- Call another web service that requires secret credentials
- Check whether a user is authorized to access a server resource
- Convert markdown content into HTML without having to download the converter code to the browser

Broadly speaking, there are three categories of benefits:

- **Performance**: Performance is positively impacted in the following ways:
    - The amount of JavaScript that's downloaded to the browser is reduced

    - The amount of network calls from the browser is reduced when the app loads
    - Component rendering is reduced when the app loads

- **Developer productivity**: Let's look at how developer productivity is impacted:
    - We've always been able to write server code to query databases and perform user authorization checks. However, RSCs allow you to do this using the React paradigm.
    - RSCs allow frontend and backend code to be colocated, as well as strong typing to be available across the network boundary.
    - RSCs reduce the React state that is needed in an app because it's not needed for server data – this data is just rendered directly on the server. State is one of the most complex aspects of React, so reducing the state reduces complexity.

- **Security**: Since server components never run on the client, they can safely contain sensitive logic such as conditional access rules or API keys

This concludes this section on understanding RSCs. To recap, RSCs run exclusively on the server and can be asynchronous, unlike Client Components. They can fetch data directly from databases and interact with server resources but can't perform JavaScript user interactions. RSCs offer good performance, productivity, and security benefits compared to Client Components.

Next, we will implement our first RSC.

Creating Server Components

In this section, we are going to create a new project in Next.js, a popular React-based framework that supports RSCs. Then, we'll create some RSCs and cement our understanding of them.

Creating the project

Next.js is a mature framework that was an early adopter of RSCs. We will use it in this chapter and throughout the rest of this book. Carry out the following steps to create a Next.js project:

1. In a terminal, execute the following command to create the project:

   ```
   npx create-next-app@latest app --ts --tailwind
   --eslint --app --src-dir --import-alias "@/*"
   --turbopack
   ```

   Here's an explanation of the command:

   - **`create-next-app`**: The tool that creates the project
   - **`app`**: The app's name
   - **`--ts`**: An option that allows you to configure the project so that it can use TypeScript
   - **`--tailwind`**: An option that configures the project so that it uses Tailwind styling

- `--eslint`: An option that configures ESLint for linting
- `--app`: An option that configures the project so that it uses the latest router variant, which is called the app router
- `--src-dir`: An option that configures whether the source code is in a `src` folder
- `--import-alias`: An option that configures import path aliases. We have set this to the default of `"@/*"`
- `--turbopack`: An option that specifies whether Turbopack should be used for development

The command also installs the project dependencies, something that will take a few seconds.

More information on creating a Next.js app can be found at https://nextjs.org/docs/app/api-reference/cli/create-next-app.

2. Still in the terminal, move to the project folder and open Visual Studio Code by running the following commands:

```
cd app
code .
```

If you aren't using Visual Studio Code, then the `code .` command won't open your editor – you can omit that command and open your editor manually.

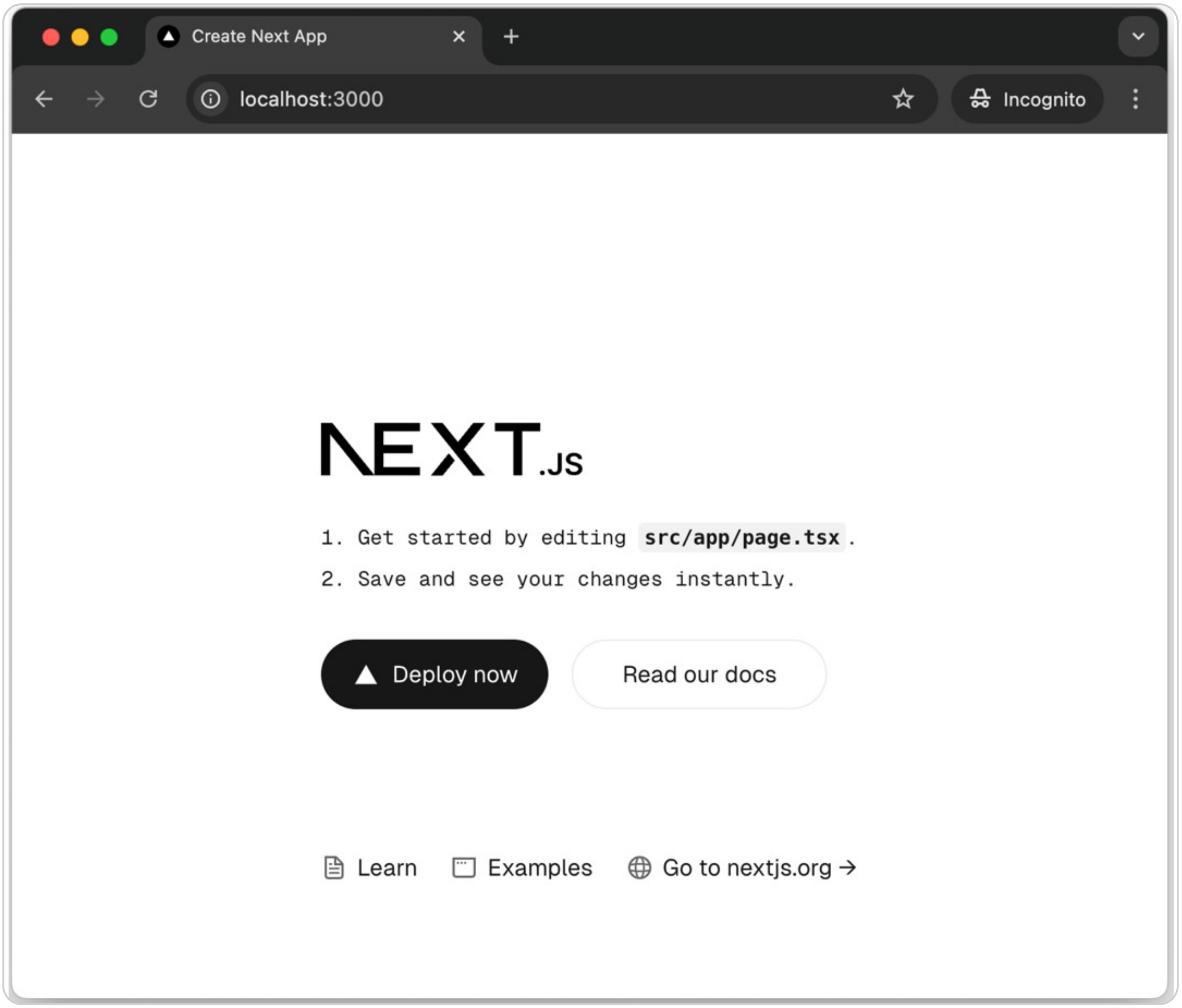
3. Prettier can be set up in the same manner as Vite, something we learned in *Chapter 1*, *Getting Started with React*. Feel free to add

automatic code formatting to this project.

4. In the terminal, execute the following command to run the app in development mode:

```
npm run dev
```

The app will be available in your browser at http://localhost:3000:



1. Get started by editing `src/app/page.tsx`.
2. Save and see your changes instantly.

Figure 5.3 – Next.js template app home page

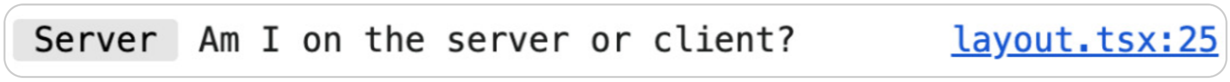
5. Next, go to the code in Visual Studio Code and open **`layout.tsx`** in the **`src/app`** folder. This is the root React component, which renders at all paths of the app. Add the following highlighted line before the **`return`** statement:

```
export default function RootLayout( ... ) {
    console.log('Am I on the server or
client?');
    return ...
}
```

   The console's output will help us determine whether the component is an RSC.

   As we previously experienced with Vite, the app will be automatically rebuilt and reloaded into the browser efficiently.

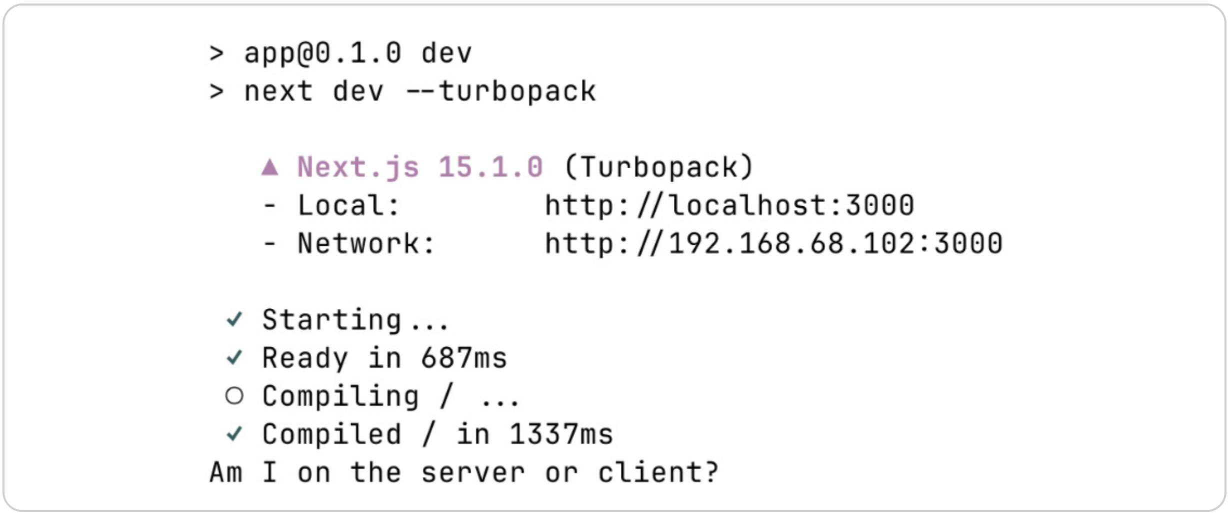
6. Open the browser console in the browser development tools area. You will see the following message, alongside a **`Server`** badge, indicating it came from the server:



```
Server  Am I on the server or client?        layout.tsx:25
```

Figure 5.4 – Browser console output

7. Now, open the terminal that is running the Next.js app. We'll see the following message:



```
> app@0.1.0 dev
> next dev --turbopack

   ▲ Next.js 15.1.0 (Turbopack)
   - Local:        http://localhost:3000
   - Network:      http://192.168.68.102:3000

 ✓ Starting...
 ✓ Ready in 687ms
 ○ Compiling / ...
 ✓ Compiled / in 1337ms
Am I on the server or client?
```

Figure 5.5 – Console output from the Next.js server

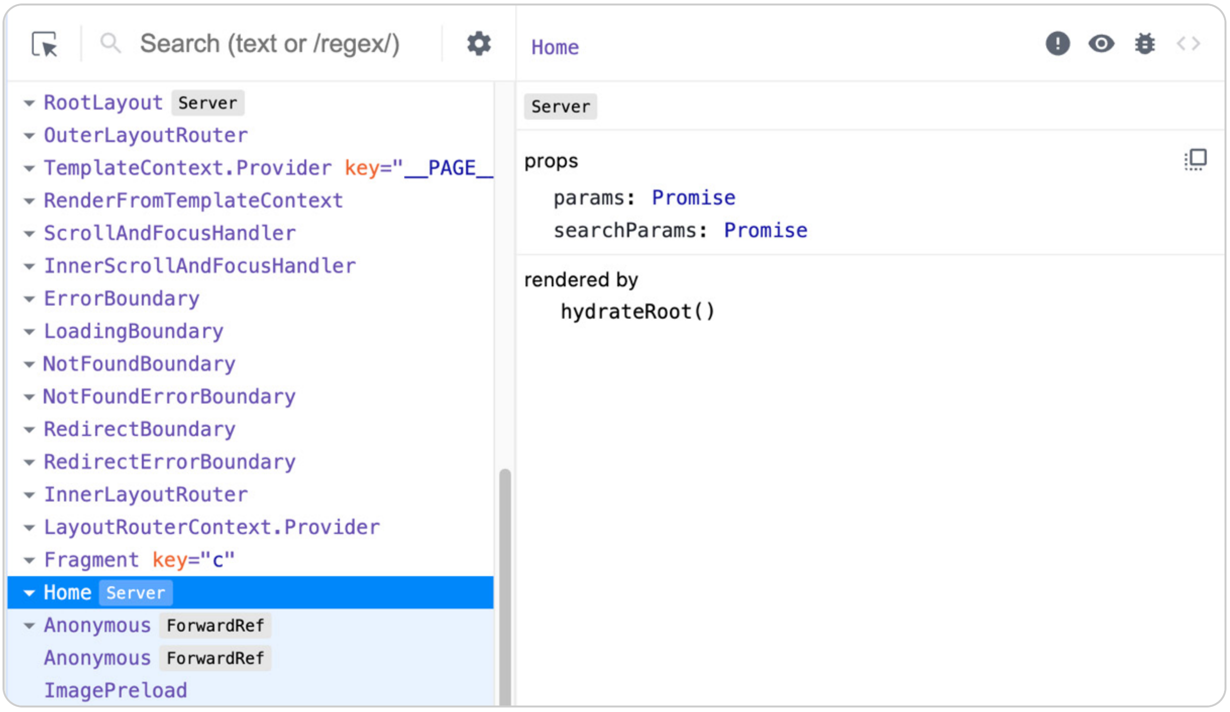
From this, we can see that the **`Layout`** component is an RSC.

8. Now, open **`Page.tsx`**. This is the component that renders inside **`Layout`** in the root path of the app. Add a similar console output to this component:

```
export default function Home() {
    console.log('Is Home on the server or
client?');
    return ...
}
```

You will determine that this is an RSC as well.

9. In the browser development tools area, go to the **Components** panel and look at the React components list. You will see a **`Server`** badge against the RSCs. This is another way of determining whether a component is an RSC:



```
RootLayout Server
OuterLayoutRouter
TemplateContext.Provider key="__PAGE__
RenderFromTemplateContext
ScrollAndFocusHandler
InnerScrollAndFocusHandler
ErrorBoundary
LoadingBoundary
NotFoundBoundary
NotFoundErrorBoundary
RedirectBoundary
RedirectErrorBoundary
InnerLayoutRouter
LayoutRouterContext.Provider
Fragment key="c"
Home Server
Anonymous ForwardRef
Anonymous ForwardRef
ImagePreload
```

```
Home
Server
props
  params: Promise
  searchParams: Promise
rendered by
  hydrateRoot()
```

Figure 5.6 – Server badge in the React components list

In Next.js, components are RSCs by default – that is, you don't have to do anything to declare that a component is an RSC. In fact, you can't specify that a component is an RSC – you can only specify that a component is a Client Component. Later in this chapter, we'll explore how this is done.

In the next section, we'll create our own RSC in the Next.js project.

Creating an RSC

In this section, we will create a **`Header`** RSC and nest it inside the **`Home`** RSC we observed in the previous section. The **`Header`** component will simply contain the app's name. Carry out the following steps:

1. Create a folder in the **src** folder called **components**. Then, create a file called **Header.tsx** in that **components** folder.

> **NOTE**
>
> *It is common practice to isolate routing components in the* **app** *folder from other components using a* **components** *folder. We will learn more about routing components in Chapter 6, Creating a Multi-Page App with Next.js.*

2. Add the following content to **Header.tsx**:

```
export function Header() {
    console.log("Is Header an RSC?");
    return (
        <header className="flex w-full items-
center justify-between">
            <span className="text-lg font-
bold">
                My app
            </span>
        </header>
    );
}
```

This outputs a **header** element with the text **My app** inside.

3. Open **page.tsx** and add an **import** statement to import **Header**:

```
import { Header } from '@/components/Header';
```

Here, **@/** is an import path alias for the **src** folder. Using an import path alias is an alternative to using a relative path, which would look as follows:

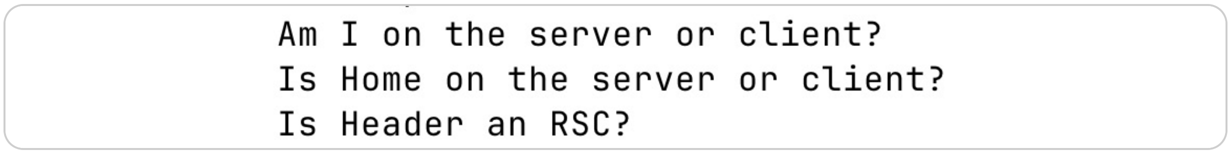
```
import { Header } from '../components/Header';
```

The benefits of import path aliases over relative paths include readability and reduced changes when files are moved.

4. Add **`Header`** inside the **`div`** element, just above the **`main`** element:

```
return (
        <div ... >
            <Header />
            <main ... >
            ...
        </div>
    );
```

Looking at the browser console and the terminal will reveal that **`Header`** is an RSC:



```
Am I on the server or client?
Is Home on the server or client?
Is Header an RSC?
```

Figure 5.7 – Confirmation in the terminal that Header is an RSC

That's our first RSC created. Nice!

In the next section, we'll start to understand what happens under the hood of an RSC.

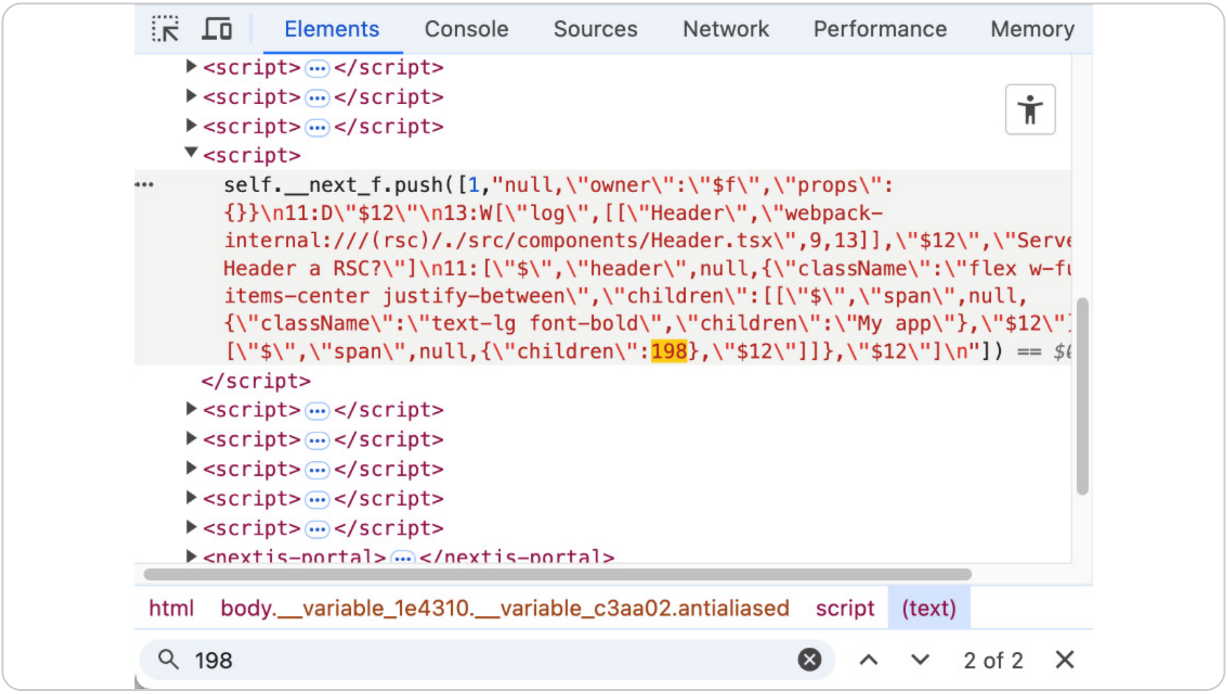
Understanding how Server Components work

In this section, we are going to take the time to understand how the **`Header`** component runs on the server and is then displayed in the browser. Carry out the following steps:

1. First, we will extend the **`Header`** component so that it includes some computation. Once we've done this, we'll be able to determine whether the computation was carried out on the server or the client. Open **`Header.tsx`** and add a calculated total, as follows:

   ```
   export function Header() {
       console.log('Is Header an RSC?');
       const total = 99 + 99;
       return (
           <header ... >
               <span ... >My app</span>
               <span>{total}</span>
           </header>
       );
   }
   ```

2. In the browser development tools area, in the **Elements** panel, search for **`198`**. There will be two instances of **`198`**. The first will be inside the **`span`** element, while the second will be inside a **`script`** element, inside a JSON structure, as shown in the following screenshot:



```
▶<script>…</script>
▶<script>…</script>
▶<script>…</script>
▼<script>
   self.__next_f.push([1,"null,\"owner\":\"$f\",\"props\":
   {}}\n11:D\"$12\"\n13:W[\"log\",[[\"Header\",\"webpack-
   internal:///(rsc)/./src/components/Header.tsx\",9,13]],\"$12\",\"Serve
   Header a RSC?\"]\n11:[\"$\",\"header\",null,{\"className\":\"flex w-fu
   items-center justify-between\",\"children\":[[\"$\",\"span\",null,
   {\"className\":\"text-lg font-bold\",\"children\":\"My app\"},\"$12\"
   [\"$\",\"span\",null,{\"children\":198},\"$12\"]]},\"$12\"]\n"]) == $
 </script>
▶<script>…</script>
▶<script>…</script>
▶<script>…</script>
▶<script>…</script>
▶<script>…</script>
▶<nextjs-portal>…</nextjs-portal>
```

```
html  body.__variable_1e4310.__variable_c3aa02.antialiased  script  (text)
```

```
198   2 of 2
```

Figure 5.8 – Serialized RSC

Note that the **`script`** element's JSON may be different for you compared to what's shown in the preceding screenshot.

The JSON in the **`script`** element starts to reveal how RSCs work. Once the RSCs have been run on the server, the RSC output is serialized along with information to render them. It is important to note that only the result of the RSC is serialized, not the function code. In this example, only **`198`** is serialized, not the calculation. So, sensitive details in the RSC will never be leaked to the client unless they are included in the RSC output.

That brings us to the end of this section on creating RSCs. Here's a quick recap:

- Next.js is an app framework that supports RSCs. We used this to create our React project.
- In a Next.js project, components are RSCs by default, so **`Layout`** and **`Page`** are RSCs. We created a **`Header`** RSC inside the **`Page`** RSC.
- We used the console to confirm that the components were RSCs and inspected HTML elements in the browser development tools area to see their serialized format.

Now that we have a good understanding of RSCs, next, we will learn about Client Components.

Exploring Client Components

In this section, we will start by understanding Client Components concerning RSCs. Then, we will create a Client Component in our project that will toggle the app between dark and light mode.

Understanding Client Components

For interactive apps, Client Components are essential. In fact, every React component was a Client Component before the introduction of RSCs.

Here are some of the features that Client Components have that RSCs don't:

- They have React Hooks, such as **`useState`**, **`useRef`**, and **`useEffect`**
- They provide browser event handling, such as **`onClick`** on a button
- State and functions are shared with the React context
- Access to browser APIs such as **`window.localStorage`** is provided

Next, we'll learn how Client Components are rendered in Next.js

Understanding Client Component rendering

As the name suggests, Client Components run on the client. However, they can also run on the server during the initial load to pre-render the parts that can be pre-rendered, which is what happens in Next.js. In Next.js, after the initial load, Client Components are **hydrated** with the parts not covered by pre-rendering, which is generally interactivity. This process is complex, but it speeds up the initial rendering of Client Components.

To better understand the Next.js Client Component rendering process a little more, consider the following **`Button`** Client Component:

```
function Button() {
    return (
        <button onClick={() => {
console.log("click"); }}>
```

```
                Click
            </button>
        );
    }
```

Here are the steps that take place in the rendering process:

1. First, the component is rendered on the server. However, it can't be fully rendered because the server can't attach the click handler. So, the result of this first rendering process is an HTML button without a click handler.
2. The HTML button is sent to the client, along with a hydration script.
3. Finally, the client adds the HTML button to the DOM and runs the hydration script to attach the button click handler.

Before we create a Client Component in our project, we'll learn how to specify a Client Component in Next.js.

Specifying Client Components

As mentioned earlier in this chapter, in Next.js, components are RSCs by default. To specify a Client Component, the file must contain **`'use client'`** at the top. However, components that are imported into a Client Component file automatically become Client Components, meaning **`'use client'`** isn't required.

Don't worry if this is a bit confusing at this stage! In the next section, we will add a Client Component to the app to cement this knowledge.

Creating Client Components

In this section, we're going to create a Client Component that will toggle our app between dark and light mode. We'll take the time to experience what happens if the component isn't marked as a Client Component and also prove it runs on the server as well as the client. Carry out the following steps:

1. Create a new file called **`ColorModeToggle.tsx`** in the **`src/components`** folder containing the following content:

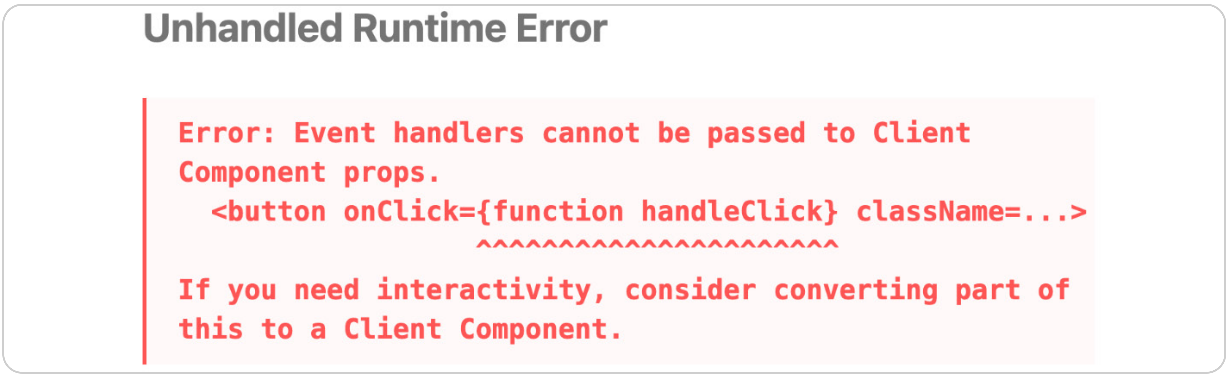
   ```
   export function ColorModeToggle() {
       console.log('Does ColorModeToggle run on
   the Server and     Client?');
       function handleClick() {}
       return (
           <button onClick={handleClick}
   className="flex rounded         bg-blue-500
   px-4 py-2 text-white">
           </button>
       );
   }
   ```

   The component renders a blue button with a click handler that doesn't do anything yet.

2. Open **`Header.tsx`** and add **`ColorModeToggle`** after the **`<span>{total}</span>`** element:

```
import { ColorModeToggle } from
'./ColorModeToggle';
export function Header() {
    ...
    return (
        <header ...>
            ...
             <span>{total}</span>
            <ColorModeToggle />
        </header>
    );
}
```

A runtime error will occur:


Unhandled Runtime Error

```
Error: Event handlers cannot be passed to Client
Component props.
  <button onClick={function handleClick} className=...>
                  ^^^^^^^^^^^^^^^^^^^^^^^^
If you need interactivity, consider converting part of
this to a Client Component.
```

Figure 5.9 – Missing "use client" error

Currently, `ColorModeToggle` is an RSC, and we have an event handler, which isn't allowed.

3. Back in `ColorModeToggle.tsx`, resolve the error by specifying that the component is a Client Component at the top of the file:

```
'use client';
```

4. Still in **`ColorModeToggle.tsx`**, we will use state to store information regarding whether we are in dark mode or not. We'll update this in the button click handler and render the current mode as the button text. Add the following highlighted lines of code:

```
'use client';
import { useState } from 'react';
export function ColorModeToggle() {
    ...
    const [colorMode, setColorMode] =
        useState<'dark' | 'light'>('light');
    function handleClick() {
        const newColorMode =
            colorMode === 'dark' ? 'light' :
'dark';
        setColorMode(newColorMode);
    }
    return (
        <button ... >
            {colorMode === 'dark' ? 'Light' :
'Dark'}
        </button>
    );
}
```

The color mode toggle button now appears to the right of the app header:

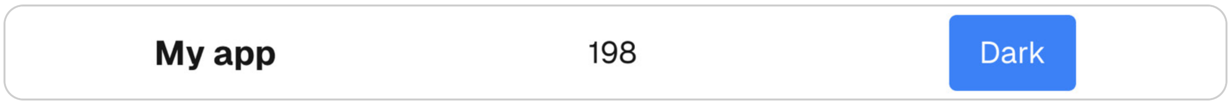


Figure 5.10 – The color mode toggle button on the right

5. Clicking the color mode toggle button doesn't toggle between dark and light mode at the moment. To make this work, we need to add a **`dark`** CSS class name to the **`body`** element. We also need to update the **`--background`** and **`--background`** CSS variables. We can use an effect to synchronize the **`colorMode`** state with these:

```
import { ..., useEffect } from 'react';
export function ColorModeToggle() {
    ...
    useEffect(() => {
        if (colorMode === 'dark') {
            document.body.classList.add('dark'
);
            document.documentElement.style.set
Property(
                '--background',
                '#0a0a0a',
            );
            document.documentElement.style.set
Property(
                '--foreground',
                '#ededed',
            );
        } else {
             document.body.classList.remove('d
ark');
             document.documentElement.style.se
tProperty(
                '--background',
                '#ffffff',
            );
            document.documentElement.style.set
Property(
                    '--foreground',
                    '#171717',
                );
            }
        }, [colorMode]);

        return ...
    }
```

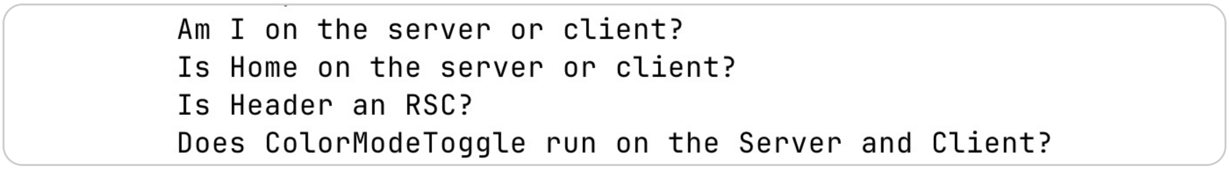
6. The color mode toggle button now works when clicked. Let's make one more improvement that will default the color mode to what is specified in the operating system or browser. Add the following effect to do this:

```
export function ColorModeToggle() {
    ...
    useEffect(() => {
        const mediaQuery = window.matchMedia(
            '(prefers-color-scheme: dark)',
        );
        setColorMode(
             mediaQuery.matches ? 'dark' :
'light'
        );
    }, []);
    return ...
}
```

We are using an effect here because we only want this code to run on the client where **`window.matchMedia`** can be accessed. Remember that Client Components can initially run on the server, but the server doesn't have access to the browser **`window`** object. Effects are never run on the server – they only run on the client.

This pattern is handy for DOM access, which needs to happen when the component is loading.

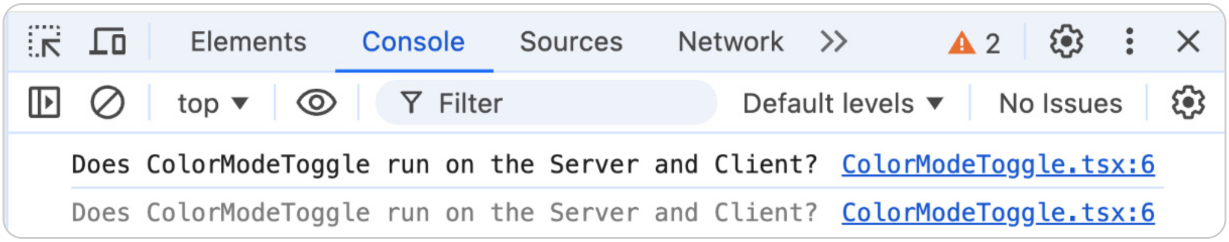
7. Now, let's check where the `ColorModeToggle` component is run. First, check the terminal to confirm that is it initially run on the server:



```
Am I on the server or client?
Is Home on the server or client?
Is Header an RSC?
Does ColorModeToggle run on the Server and Client?
```

Figure 5.11 – Confirmation that ColorModeToggle runs on the server

8. Finally, let's check the browser console:



```
Does ColorModeToggle run on the Server and Client?  ColorModeToggle.tsx:6
Does ColorModeToggle run on the Server and Client?  ColorModeToggle.tsx:6
```

Figure 5.12 – Confirmation that ColorModeToggle runs on the client

That brings us to the end of this section on Client Components. Here's a recap:

- Client Components are necessary for interactive apps as they support features such as React Hooks, event handling, and DOM access. They initially run on the server and are later hydrated with interactivity on the client.

- A Client Component requires a `'use client'` directive at the top of its file. Other components that are imported into a Client Component file automatically become Client Components, meaning `'use client'` isn't required for subsequent Client Components.
- Finally, `ColorModeToggle` is the first Client Component in our app. It contains a state, an effect, and event handlers. So, it must be a Client Component, rather than an RSC.

That completes our exploration of Client Components. Next, we'll learn how Server and Client Components can be composed.

Composing Server and Client Components

This section will explain the thought process behind choosing between Server and Client Components. Then, we will understand what a client boundary is and how to inject an RSC into it.

RSCs versus Client Components

We have already covered the benefits and different capabilities of RSCs and Client Components. As a recap, here is a succinct comparison table:

Feature/Aspect	**RSC**	**Client Component**
First page load	Fast	Slower
Access to secure resources (databases, third-party APIs)	Direct	Via an HTTP call to a secure server
Interactivity	Not supported	Full interactivity
React Hooks	Very limited	Full support
Access to browser APIs	Not supported	Full access

Table 5.1: Key Differences Between RSCs and Client Components

Next, we'll cover a strategy for choosing an RSC or Client Component.

Understanding when to use an RSC or Client Component

If you are creating a new Next.js page, you should start the component tree with RSCs. This is because it's the default in Next.js, and RSCs tend to be simpler and more performant. Switch a component to a Client Component if it requires functionality outside of RSCs, such as interactivity. To maximize performance, isolate only the parts that RSCs can't do in Client Components. This is exactly what we did with `ColorModeToggle` in the previous section.

Next, we'll understand exactly when we need to use the `'use client'` directive.

Understanding client boundaries

Every file that's imported after a `'use client'` directive is automatically a Client Component. This is referred to as a **client boundary**.

The following diagram visualizes a client boundary on a page in an app:

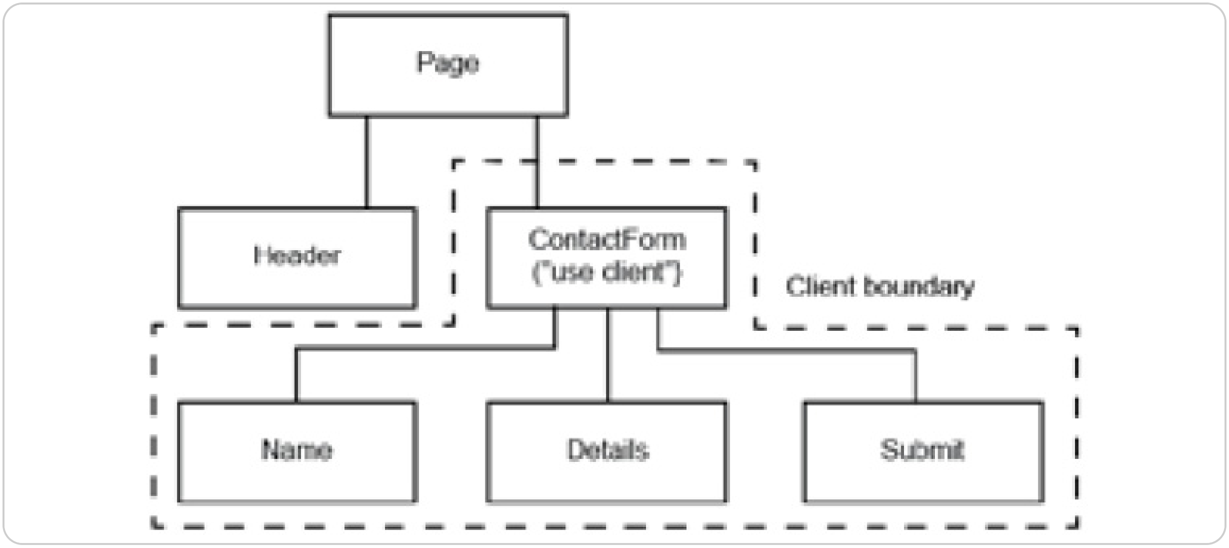


Figure 5.13 – Client boundary in an app

The **`Page`** and **`Header`** components are RSCs. Note that **`ContactForm`** contains a **`'use client'`** directive, so it is a Client Component. Here, **`Name`**, **`Details`**, and **`Submit`** are all Client Components and don't need a **`'use client'`** directive. So, a client boundary is formed around everything that is imported into **`ContactForm`**.

You might think it isn't possible to render an RSC inside a Client Component. In the next section, we will learn how to do this.

Rendering an RSC in ColorModeToggle

In this section, we're going to add an icon to our **`ColorModeToggle`** component, which, as you may recall, is a Client Component. We'll learn how the icon can be turned into an RSC.

Client Components can't import an RSC. However, an RSC can be passed to a Client Component as a prop.

Using this key concept, we'll add an icon RSC to our **`ColorModeToggle`** component. Follow these steps:

1. Start by creating a file for the SVG icon called **`colorModeIcon.svg`** in the **`src/components`** folder. Copy the contents of the SVG at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter05/rendering-rsc-in-client-component/src/components/colorModeIcon.svg and paste it into this file.

2. Next, create a React component for the icon. Create a file called **`ColorModeIcon.tsx`** with the following contents:

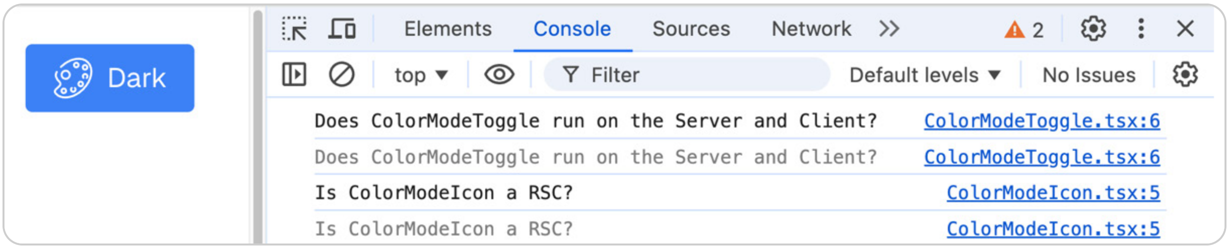
```
import Image from "next/image";
import colorModeSvg from
"./colorModeIcon.svg";
export function ColorModeIcon() {
    console.log("Is ColorModeIcon an RSC?");
    return (
        <Image
             src={colorModeSvg}
             alt="Color mode icon"
            className="mr-2 h-6 w-6"
        />
    );
}
```

   The component renders the SVG in a Next.js **`Image`** component with a fixed size.

3. Now, let's put the icon inside **ColorModeToggle**. First, we will try to import it and use it in **ColorModeToggle.tsx**:

```
import { ColorModeIcon } from
"./ColorModeIcon";
export function ColorModeToggle() {
    ...
    return (
        <button ...>
            <ColorModeIcon />
             {colorMode === 'dark' ? 'Light' :
'Dark'}
        </button>
    );
}
```

4. If the app isn't running, you can make it run by entering **npm run dev** in the terminal. Have a look in the browser console to determine whether **ColorModeIcon** is a Client Component:



```
Does ColorModeToggle run on the Server and Client?   ColorModeToggle.tsx:6
Does ColorModeToggle run on the Server and Client?   ColorModeToggle.tsx:6
Is ColorModeIcon a RSC?                                ColorModeIcon.tsx:5
Is ColorModeIcon a RSC?                                ColorModeIcon.tsx:5
```

Figure 5.14 – Confirmation that ColorModeIcon is a Client Component

The fact that the message **Is ColorModeIcon an RSC?** is output to the browser console without a **Server** badge confirms it is a Client Component. This confirms what we learned in the previous section

– a component that's imported into a Client Component will automatically be a Client Component.

5. Next, we will try a different approach for adding **`ColorModeIcon`** to **`ColorModeToggle`**. This time, we'll pass it as a prop. Start by removing the **`ColorModeIcon`** import statement.
6. Now, add an **`icon`** prop to **`ColorModeToggle`** and output this instead of **`ColorModeIcon`**:

```
import { ..., type ReactNode } from 'react';
export function ColorModeToggle({
    icon,
}: {
    icon: ReactNode;
}) {
    ...
    return (
        <button ...>
            {icon}
            {colorMode === 'dark' ? 'Light' :
'Dark'}
        </button>
    );
}
```

In *Chapter 1*, *Getting Started with React*, we learned that components can be passed to other components using a special **`children`** prop. We can create our own prop that accepts a component, something we have done here with the **`icon`** prop.

We have set its type to **`ReactNode`** so that it accepts any React component.

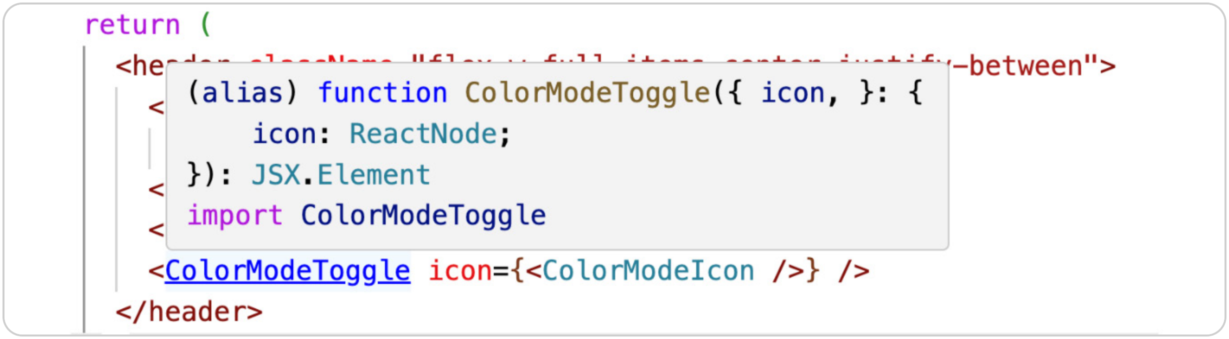
7. Lastly, open **`Header.tsx`** and pass **`ColorModeIcon`** into **`ColorModeToggle`** using the **`icon`** prop:

```
...
import { ColorModeIcon } from
'./ColorModeIcon';
export function Header() {
    ...
    return (
        <header ...>
             ...
             <ColorModeToggle icon=
{<ColorModeIcon />} />
        </header>
    );
}
```

   Now, if you look in the browser console, you'll see that the message **Is ColorModeIcon an RSC?** has a **`Server`** badge because **`ColorModeIcon`** is an RSC.

Before completing this section, let's appreciate some productivity benefits of using RSCs with Client Components. Even though these components run on completely different computers, we develop them using the same paradigm. Open **`Header.tsx`** and carry out the following steps:

1. Hover over **`ColorModeToggle`** in the JSX. Note that we get IntelliSense across the server/client boundary:



```
return (
  <header className="flex w-full items-center justify-between">
    <
    <
    <
    <ColorModeToggle icon={<ColorModeIcon />} />
  </header>
```

```
(alias) function ColorModeToggle({ icon, }: {
    icon: ReactNode;
}): JSX.Element
import ColorModeToggle
```

Figure 5.15 – IntelliSense across the server/client boundary

2. Temporarily remove the **`icon`** attribute from the JSX. Note that we get a type error.

3. Right-click on **`ColorModeToggle`** and choose the **Go to Definition** option. You will be taken to the definition for **`ColorModeToggle`**.

Amazingly, these kinds of features work across these two worlds – they give us great productivity benefits. It doesn't feel like two different worlds at all!

That concludes this section on composing RSCs with Client Components. Here's a recap:

- It is recommended to start a new Next.js page with RSCs for simplicity and performance. Use Client Components only when additional functionality, such as interactivity, is needed.
- A client boundary is created when a component has a **`'use client'`** directive. All files that are imported after become Client

Components.

- While a Client Component cannot directly import an RSC, an RSC can be passed as a prop.

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

Summary

In this chapter, we learned that RSCs were fully released in React 19 and run exclusively on the server. Unlike Client Components, RSCs can be asynchronous and interact directly with server resources such as databases.

The primary benefits of RSCs include improved performance since they reduce the amount of JavaScript that's sent to the client and lower the number of network calls that are made from the browser. Developer productivity is also enhanced because RSCs allow frontend and backend code to be colocated, eliminating the need to shift between different paradigms.

Client Components are essential for interactivity as they support React Hooks and browser API access. They run on the client after initial server rendering. Developers can declare a component as a Client Component by adding the **`'use client'`** directive. RSCs can be nested inside Client Components by passing them as props.

Next.js is a popular React-based framework that supports RSCs, where components are RSCs by default. We created a Next.js app and created a **`Header`** RSC with a **`ColorModeToggle`** Client Component nested inside. We then nested a **`ColorModeIcon`** RSC inside **`ColorModeToggle`** by passing it as a prop.

In the next chapter, we will learn more about Next.js so that we can create a multi-page app.

Further reading

For more information on RSCs, see the React documentation at https://react.dev/reference/rsc/server-components.

Questions

Answer the following questions to check what you have learned in this chapter:

1. We have a Next.js app that has the following RSC:

```
import Counter from “@/components/Counter”;
export default function Home() {
    return (
        <div>
             <Counter />
        </div>
    );
}
```

The **`Counter`** component reads as follows:

```
import { useState } from "react";
export default function Counter() {
    const [count, setCount] = useState(1);
    return (
        <button
             onClick={() => setCount((prev) =>
prev + 1)}
        >
             {count}
        </button>
    );
}
```

A compile error occurs. What's the problem?

2. Does an RSC run on the client? Does a client component run on the server?

3. We have the following component, but an error occurs when it renders. What's the problem?

```
"use client";
export async function PeopleList() {
    const people = await getPeople();
    return (
        <ul>
             {people.map((person) => (
                 <li key={person}>
                     <span>{person}</span>
                </li>
            ))}
        </ul>
        );
    }
```

4. In a Next.js app, is it possible for an RSC to be nested inside a Client Component?
5. In Next.js, which of the following is not allowed inside an RSC?
    - **`fetch`**
    - **`useEffect`**
    - **`async / await`**

Answers

1. Here, **`Counter.tsx`** needs to be a Client Component because it requires state. However, it's an RSC because it's imported into an RSC file. So, the following directive needs to be placed at the top of the file to make it a Client Component:

    ```
    "use client";
    ```

2. An RSC only runs on the server – it doesn't run on the client. A client component can run on the server and then be hydrated on the client.
3. Client Components can't be asynchronous. So, the component can be changed to an RSC if it's imported into an RSC. If it needs to be a Client Component, the **`getPeople`** call can be made using the **`useEffect`** Hook. A simple example is shown here:

    ```
    export function PeopleList() {
        const [people, setPeople] =
    ```

```
useState<string[]>([]);
    useEffect(() => {
        getPeople().then((p) => {
            setPeople(p);
        });
    }, []);
    return ...;
}
```

4. Yes – if the RSC is passed into the Client Component as a prop. You can't import an RSC into a Client Component file.
5. The **`useEffect`** Hook – because **`useEffect`** only runs on the client.

6
Creating a Multi-Page App with Next.js

So far in this book, we've created apps with a single screen. In this chapter, we'll learn how to create apps with multiple screens with Next.js, which was introduced in the last chapter. React doesn't include much of the functionality required to build multi-page apps, and Next.js is a popular option that fills that gap.

We will create a blog post app containing three screens – a **Home** page, a **Posts** page, and a **Post** page. We will create a shared header on the screens and navigation options between them. We will also create a search feature and learn all about search parameters.

By the end of this chapter, you'll know how to build multi-page apps using Next.js.

We'll cover the following topics:

- Creating routes
- Creating navigation
- Creating shared layout
- Creating dynamic routes
- Using search parameters

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-Third-Edition/tree/main/Chapter06.

Creating routes

In this section, we will create a Next.js project. We will understand what routes are and how to create them. We'll use this knowledge to create a route that lists all our blog posts.

Creating the project

Let's start by creating a Next.js project. This was covered in the previous chapter, so the step explanations are brief:

1. In a terminal, execute the following command to create the project:

```
npx create-next-app@latest blog --ts --eslint
```

```
--app --src-dir --import-alias "@/*" --no-
tailwind --turbopack
```

2. Still in the terminal, move to the project folder and open Visual Studio Code using the following commands:

```
cd blog
code .
```

3. Prettier can be set up in the same manner as we learned with Vite in *Chapter 1*, *Getting Started with React*. Feel free to add automatic code formatting to this project.

4. In the terminal, execute the following command to run the app in development mode:

```
npm run dev
```

The app will be available in a browser at http://localhost:3000.

5. Open `src/app/global.css` and overwrite the content with the CSS from the following file in the GitHub repository:

https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter06/creating-project/src/app/globals.css

This will nicely style our app.

6. Lastly, let's clean up the **Home** page. Open `page.tsx` in the `src/app` folder and replace its content with the following:

```
export default function Home() {
        return (
            <main>
                 <h2>
                     Welcome to my blog!
                </h2>
            </main>
        );
    }
```

The page outputs a nice header centered at the top of the page.

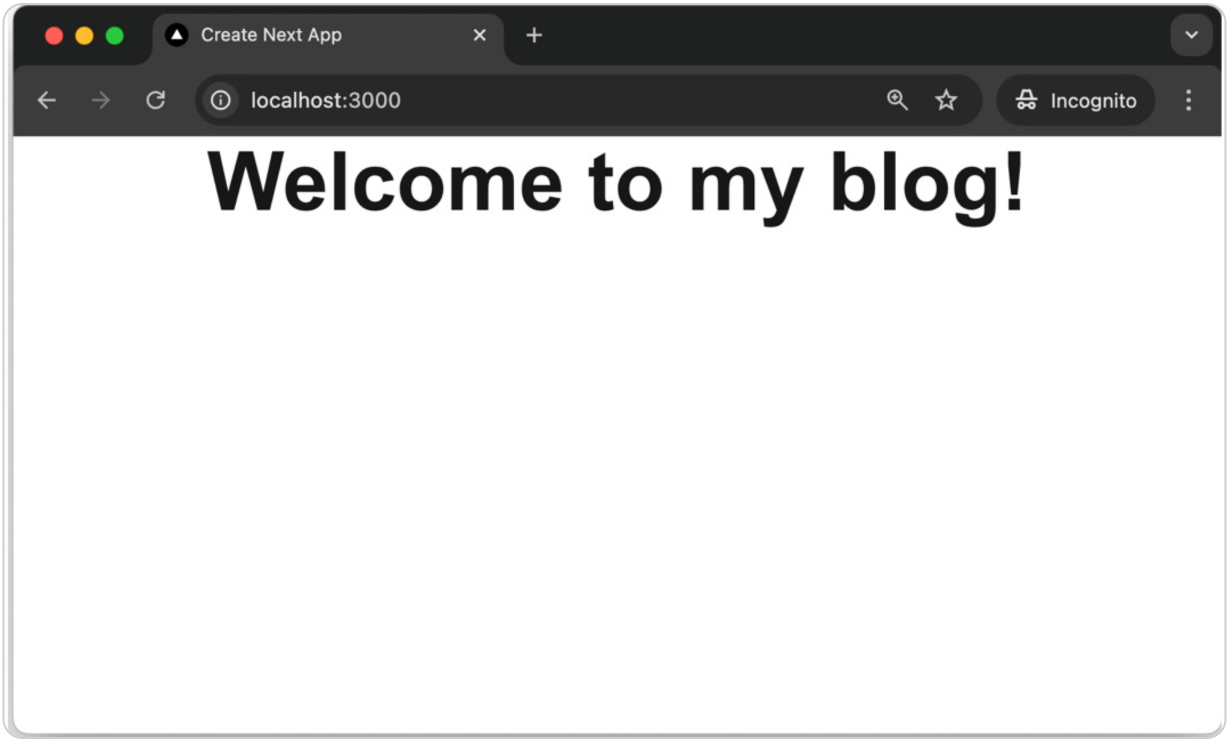


Figure 6.1 – Home page

That completes the project setup. Next, we'll understand how to create new pages.

Understanding routes

A route is a path in a web app that corresponds to a URL. Next.js has two types of routing systems: the **page router** and the **app router**. In this book, we'll focus on the more recently released app router, which our project has been configured with.

Routes are defined using the folder structure in the `app` folder and a special file called `page.tsx`. We already have a home route defined for path `/`, which is the `src/app/page.tsx` file. To define a `settings` route, we would implement a `src/app/settings/page.tsx` file.

The contents of `page.tsx` is a React component. It can be an RSC or a Client Component. The component name can be a meaningful name of our choice; however, it's important that it is the default export. It must be the default export to allow Next.js to automatically detect and render the right component for a route – if it was a named export, it wouldn't know which one to use.

Now that we understand routes and how to create them, we'll create a `posts` route.

Creating a posts route

Carry out the following steps to create a `posts` route in our app:

1. Create a folder in the `src/app` folder called `posts` with a `page.tsx` file inside.

2. Add the following content to the **`page.tsx`** file that was just created:

```
export default function Posts() {
    return (
        <main>
             <h2>Posts</h2>
            <ul></ul>
        </main>
    );
}
```

   The component outputs a heading and an empty list at this stage.

3. Go to the http://localhost:3000/posts URL in a browser. This will hit the route we just created, displaying the **`Posts`** component.

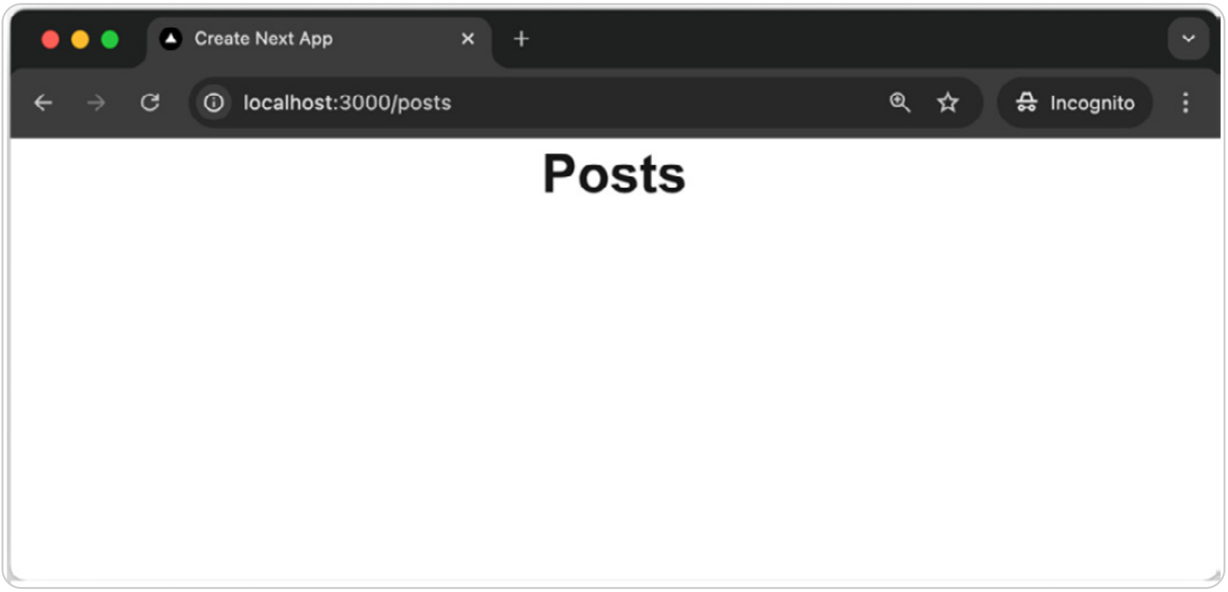


Figure 6.2 – posts route

4. We will now add some blog post data to display on this page. Start by creating a **`data`** folder in the **`src`** folder with a file called **`posts.ts`**

in it. Copy the content from the file in the GitHub repository at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter06/creating-routes/src/data/posts.ts and paste it into **posts.ts**. It contains an exported variable called **posts** containing an array of blog posts:

```
export const posts = [
    {
        Id: 1,
        title: 'Understanding React Hooks',
        description:
            'A comprehensive guide to React
Hooks and how they              simplify state
management in functional components',
    },
    ...
];
```

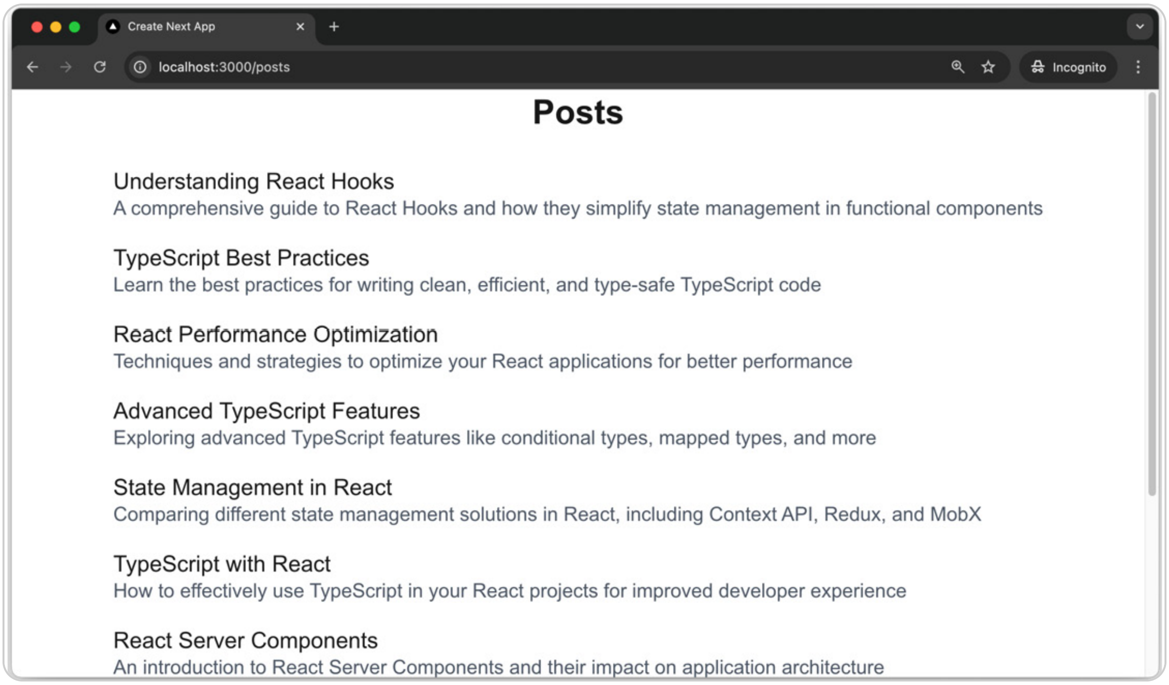
5. We will now use the **posts** variable in the **Posts** page to output a list of blog posts. Open **app/posts/page.tsx** and add the following highlighted lines:

```
import { posts } from '@/data/posts';
export default function Posts() {
    return (
        <main>
             <h2>Posts</h2>
             <ul>
                {posts.map((post) => (
                      <li key={post.id}>
                          <span>{post.title}
</span>
                         <p>{post.description}
</p>
                    </li>
                ))}
            </ul>
        </main>
    );
}
```

The **`posts`** variable is imported and used in the JSX to loop through its array to output list items. Using **`Array.map`** is a common practice for JSX looping logic.

Notice the **`key`** attribute 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`** attribute must be unique and stable within the array, so we have used the post ID.

If you look at the **`posts`** route in the browser, it now contains a list of blog posts.



Posts

Understanding React Hooks
A comprehensive guide to React Hooks and how they simplify state management in functional components

TypeScript Best Practices
Learn the best practices for writing clean, efficient, and type-safe TypeScript code

React Performance Optimization
Techniques and strategies to optimize your React applications for better performance

Advanced TypeScript Features
Exploring advanced TypeScript features like conditional types, mapped types, and more

State Management in React
Comparing different state management solutions in React, including Context API, Redux, and MobX

TypeScript with React
How to effectively use TypeScript in your React projects for improved developer experience

React Server Components
An introduction to React Server Components and their impact on application architecture

Figure 6.3 – Updated posts route

That completes this section on creating routes. The key point is that routes in Next.js are created using the **`app`** folder structure and **`page.tsx`** files.

For more information on routes, see the following page in the Next.js documentation: https://nextjs.org/docs/app/getting-started/layouts-and-pages.

Next, we will learn about a Next.js component that can perform navigation.

Creating navigation

Next.js has two ways to implement navigation, which we'll cover in this section. As part of the learning process, we will update the blog posts list to include links to the associated **Post** page.

Using the Link component

The `Link` component is the recommended way to perform navigation in Next.js. Carry out the following steps to use this component in the **Posts** page:

1. Open `app/posts/page.tsx` and add an `import` statement at the top of the file to import `Link`:

   ```
   import Link from 'next/link';
   ```

2. In the `Posts` component's JSX, change the element containing the blog post title to `Link` as follows:

   ```
   <li key={post.id}>
       <Link href={`/posts/${post.id}`}>
           {post.title}
       </Link>
       <p>{post.description}</p>
   </li>
   ```

   We've also defined the route that the link should go to using the `href` attribute.

3. The route we have just referenced doesn't exist yet, so let's create it. Inside the `posts` folder, create a folder named `1` (the number 1).

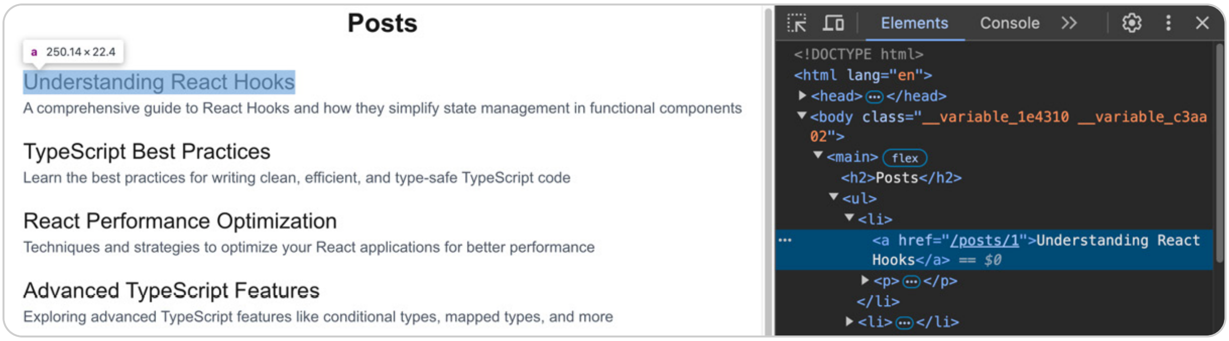
Then, create a **`page.tsx`** file inside it containing the following:

```
export default function Post() {
    return <main>Blog post one</main>;
}
```

This implements route **`posts/1`**, which will satisfy the first blog post list item. We will improve this route later in this chapter so that all blog posts are linked.

4. Run the app by executing **`npm run dev`** in a terminal if it isn't already running.

5. Now, inspect the blog post list using the browser development tools when on the **Posts** page.



Posts

Understanding React Hooks
A comprehensive guide to React Hooks and how they simplify state management in functional components

TypeScript Best Practices
Learn the best practices for writing clean, efficient, and type-safe TypeScript code

React Performance Optimization
Techniques and strategies to optimize your React applications for better performance

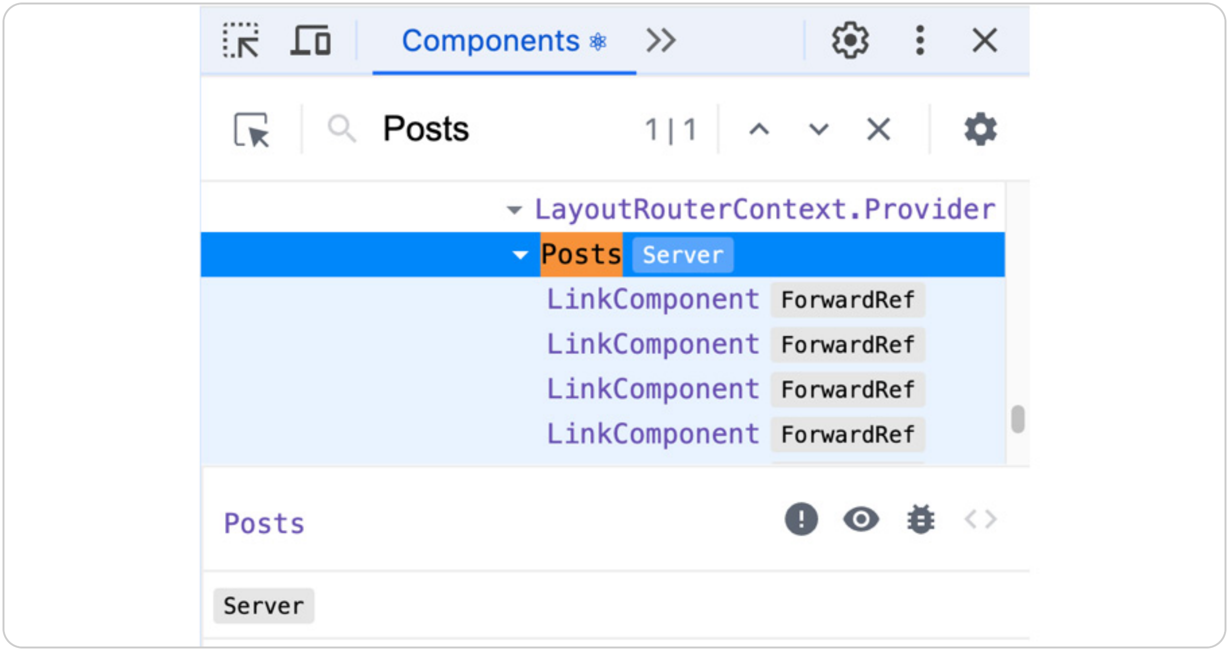
Advanced TypeScript Features
Exploring advanced TypeScript features like conditional types, mapped types, and more

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body class="__variable_1e4310 __variable_c3aa
  02">
   ▼<main> flex
     <h2>Posts</h2>
    ▼<ul>
     ▼<li>
        <a href="/posts/1">Understanding React
        Hooks</a> == $0
       ▶<p>…</p>
      </li>
     ▶<li>…</li>
```

Figure 6.4 – Link component inspection

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

6. While still in the browser development tools, go to the **Components** panel in the React developer tools. Find the **`Posts`** component and confirm that it is an RSC:

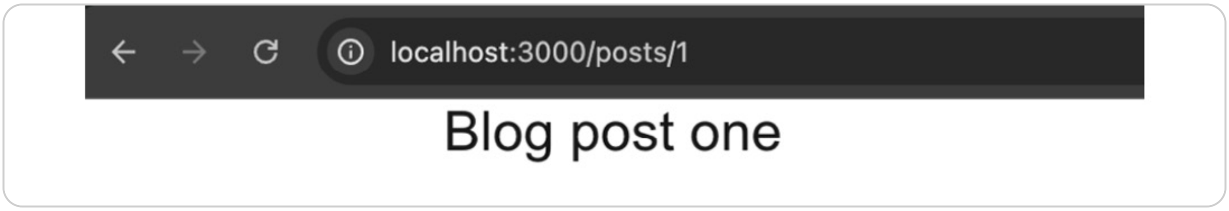


```
LayoutRouterContext.Provider
  Posts Server
    LinkComponent ForwardRef
    LinkComponent ForwardRef
    LinkComponent ForwardRef
    LinkComponent ForwardRef
Posts
Server
```

Figure 6.5 – Posts RSC

So, the key point is that **`Link`** can be used in RSCs.

7. Click on the first blog post item link in the app – it will direct you to the **Post** page.



Blog post one

Figure 6.6 – Link navigation

You may notice that a full-page reload didn't happen. Instead, client-side navigation is performed to maximize performance.

That completes the use of the **`Link`** component for now. We'll use it again later in this chapter. For more information on the **`Link`** component, see the following page in the Next.js documentation: https://nextjs.org/docs/app/api-reference/components/link.

Next, we will understand a different way of navigating.

Using useRouter

The Next.js **`useRouter`** hook allows programmatic navigation. Unlike **`Link`**, it can't be used in an RSC, so the consuming component must be a Client Component.

The hook returns an object containing useful routing functions such as the following:

- **`Push`**: Performs client-side navigation, adding a new entry to the browser history
- **`replace`**: Performs client-side navigation, without adding a new entry to the browser history
- **`refresh`**: Refreshes the current route without losing any state

The following is a code snippet that includes navigation using **`useRouter`**:

```
import { useRouter } from 'next/navigation';
function SomeComponent() {
    const router = useRouter();
    ...
    function handleClick() {
        if (someCheck()) {
                router.push('/some-path');
            } else
                router.push('/some-other-path');
            }
        }
        return <button onClick=
{handleClick}>Action</button>;
}
```

Here's a breakdown of the code snippet:

- The **`useRouter`** hook is called at the start of the component and assigned to a **`router`** variable. So, all the routing functions are available in the **`router`** variable.
- The router's **`push`** function is called in the button-click handler with the path being passed. There is a call to the function, **`someCheck`**, to determine the path to navigate to.

For more information on the **`useRouter`** hook, see the following page in the Next.js documentation:

https://nextjs.org/docs/app/api-reference/functions/use-router

That completes this section on navigation. To recap, the **`Link`** component is the recommended way of navigating in Next.js and can be used in RSCs as well as Client Components. The **`useRouter`** hook allows programmatic navigation in Client Components.

Next, we will cover how to implement shared layout components.

Creating shared layout

In this section, we will create a header for our app containing links to the **Home** and **Posts** pages. We will use the shared layout capabilities of Next.js to implement this.

Understanding layout components

In Next.js, a shared layout is defined in a special file called **`layout.tsx`**. Shared layouts can be defined on any route. Our current app has a shared layout at the root path in the **`src/app`** folder. Following are some of the contents of this file:

```
...
export default function RootLayout({ children }:
...) {
    return (
        <html lang="en">
             <body ... >
                 {children}
            </body>
        </html>
    );
}
```

The file's content is a React component set as the default export. The component's name can be a meaningful name of our choice. This component is sensibly named **`RootLayout`** because it will be rendered for every route.

Layout components can be RSCs or Client Components. In our app, **`RootLayout`** is an RSC.

The page component for the rendered route will be passed into the layout component's **`children`** prop. So, **`Home`** will be passed for route **`/`** and **`Posts`** for route **`posts/`**.

Now that we understand how shared layout components work, we will create a shared header.

Creating a header

We will create a **`Header`** component in our app containing links to the **Home** and **Posts** pages. We will then add **`Header`** to **`RootLayout`** to be visible in all routes. Carry out the following steps to do this:

1. Create a **`components`** folder in the **`src`** folder and then a file called **`Header.tsx`** inside it. Add the following content inside **`Header.tsx`**:

```
import Link from 'next/link';
export function Header() {
    return (
        <header>
             <Link href="/">Home</Link>
             <Link href="/posts">Posts</Link>
        </header>
    );
}
```

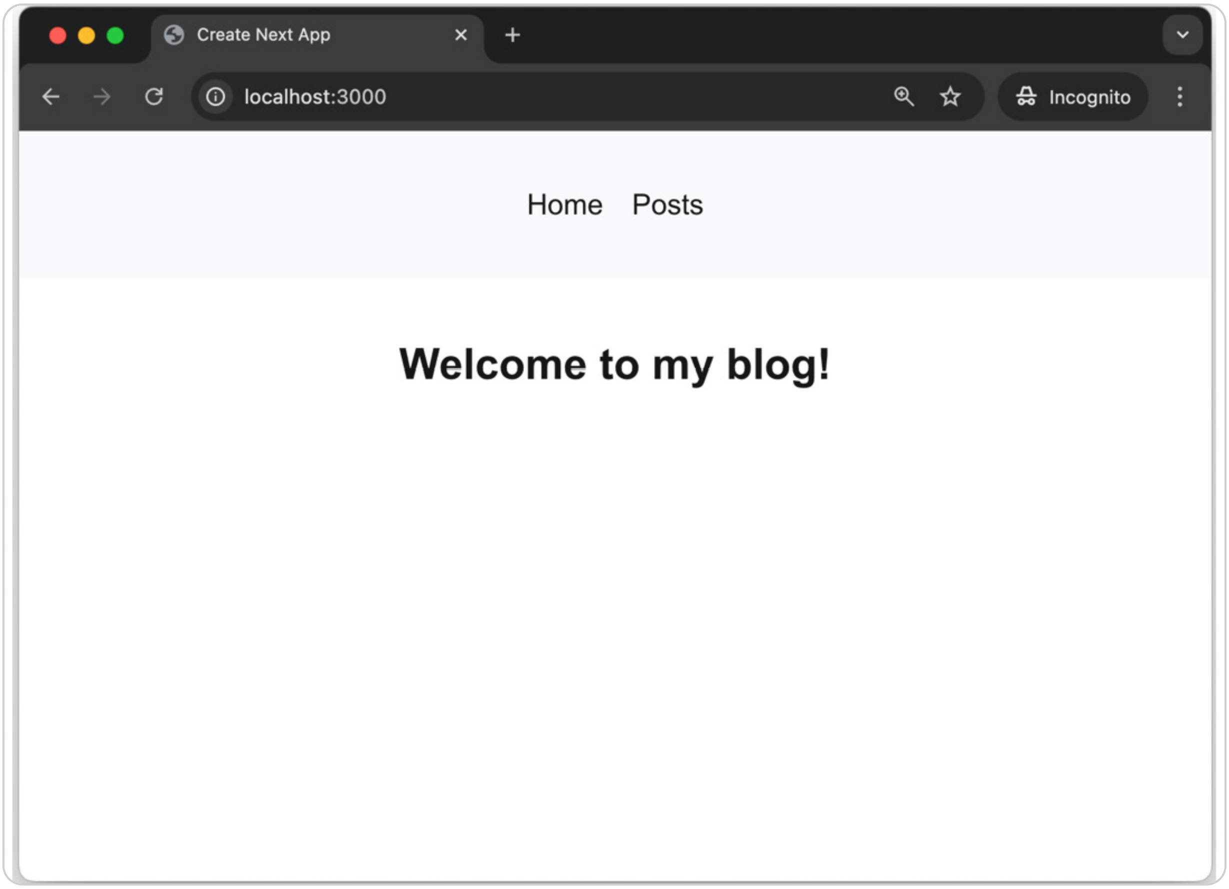
Inside a **`header`** element, we've added links to the home and posts routes using the **`Link`** component.

2. Open **`src/app/layout.tsx`** and add **`Header`** inside the **`body`** element as follows:

```
...
import { Header } from '@/components/Header';
...
export default function RootLayout( ... ) {
    return (
        <html ... >
             <body ... >
                 <Header />
                  {children}
            </body>
        </html>
    );
}
```

3. Run the app by executing **`npm run dev`** in a terminal if it isn't already running.

   Look at the app in the browser – you will see the header we've just added on both the **Home** and **Posts** pages:



Welcome to my blog!

Figure 6.7 – Header on the Home page

4. We are going to improve the styling of the header links a little. We will style the active link so that it stands out. We will use a hook called **`usePathname`** from Next.js to get the active path to check against the link's path to determine whether it is active. To start with, make the following highlighted changes in **`Header.tsx`** to get the active path:

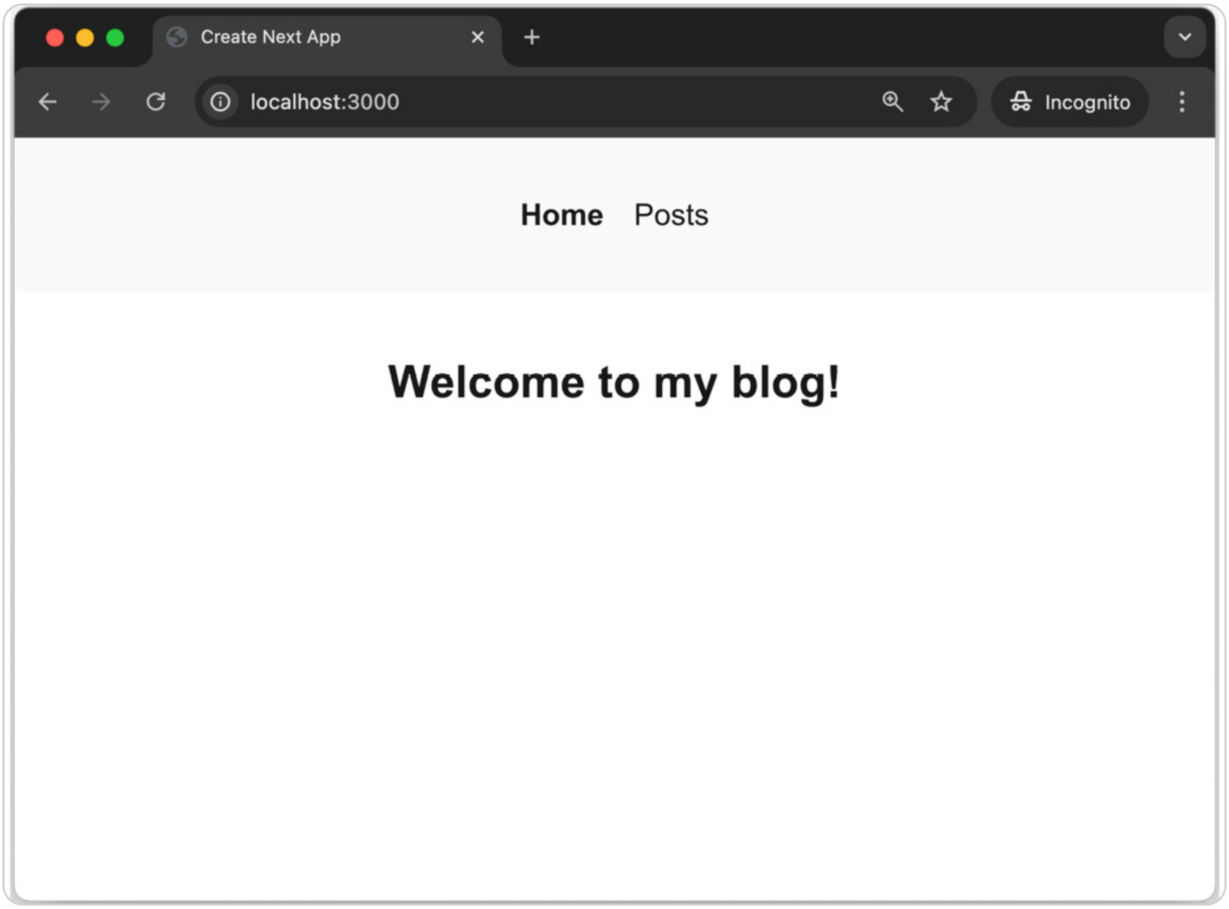
```
'use client';
import Link from 'next/link';
import { usePathname } from 'next/navigation';
export function Header() {
    const pathname = usePathname();
    return ...
}
```

We needed to turn **`Header`** into a Client Component because **`usePathname`** can't be used in an RSC.

5. Now use the **`pathname`** variable to set an **`active`** CSS class on the links conditionally:

```
<header>
    <Link
        href="/"
        className={pathname === '/' ? 'active'
: ''}
    >
        Home
    </Link>
    <Link
        href="/posts"
        className={
            pathname === '/posts' ? 'active' :
''
        }
    >
        Posts
    </Link>
</header>
```

If you look at the running app, the active link will be bold:



Welcome to my blog!

Figure 6.8 – Active header link

That completes our shared header and this section on shared layout.

To recap, a shared layout component is defined in **`layout.tsx`**. The root layout is shared across all paths and is in the **`app`** folder.

For more information on the **`layout.tsx`** file, see the following page in the Next.js documentation:

https://nextjs.org/docs/app/api-reference/file-conventions/layout

Currently, the blog post route is only working for the first one. We'll address this in the next section.

Creating dynamic routes

We will learn about dynamic routes in this section and use it to fully implement the blog post route.

Understanding dynamic routes

In Next.js, a **dynamic route** allows you to create pages that can respond to different URL parameters. This allows page content to be displayed from varying data in the URL. The dynamic part of the URL is defined in square brackets.

An example is our **Post** page at route **`posts/1`**, **`posts/2`**, and so on, which is **`posts/[id]`**, in its dynamic form. In this dynamic route, **`id`** is referred to as a **route parameter**.

Route parameters are passed into the page component in a **`params`** prop. For example, an **`id`** parameter could be used in the **Post** page as follows:

```
export default async function Post({
    params,
}: {
    params: Promise<{ id: string }>;
}) {
    const id = (await params).id;
    return <main>Blog post {id}</main>;
}
```

A type annotation is used to strongly type the route parameters. Notice that the **`params`** prop is asynchronous, which means the component must be declared with the **`async`** keyword.

If you want to use a route parameter lower in the component tree, Next.js has a **`useParams`** hook that can be used. This is only available in Client Components and not RSCs. **`useParams`** has a generic parameter to strongly type the route parameters. Here's an example of an **`id`** parameter being used:

```
'use client';
import { useParams } from 'next/navigation';
export function SomeComponent() {
    const params = useParams<{id: string}>();
    return <h3>Blog post {params.id}</h3>
}
```

Now that we understand dynamic routes and how to implement them, we'll fully implement our blog post route.

Creating a blog post dynamic route

Carry out the following steps to make the blog post route dynamic. We will also display the blog post title and description in its page component.

1. Start by renaming the **`1`** folder inside the **`src/app/posts`** folder to **`[id]`**. This makes the route dynamic with an **`id`** route parameter.
2. Open up **`page.tsx`** inside the **`[id]`** folder and add the highlighted changes:

   ```
   export default async function Post({
       params,
   }: {
       params: Promise<{ id: string }>;
   }) {
       const id = (await params).id;
       return <main>Blog post {id}</main>;
   }
   ```

   The component is now using the route parameter.
3. Run the app by executing **`npm run dev`** in a terminal if it isn't already running.
4. Go to the app in the browser, go to the blog post list, and click on different blog post links. You will see the **Post** page correctly displaying the **`id`** route parameter:

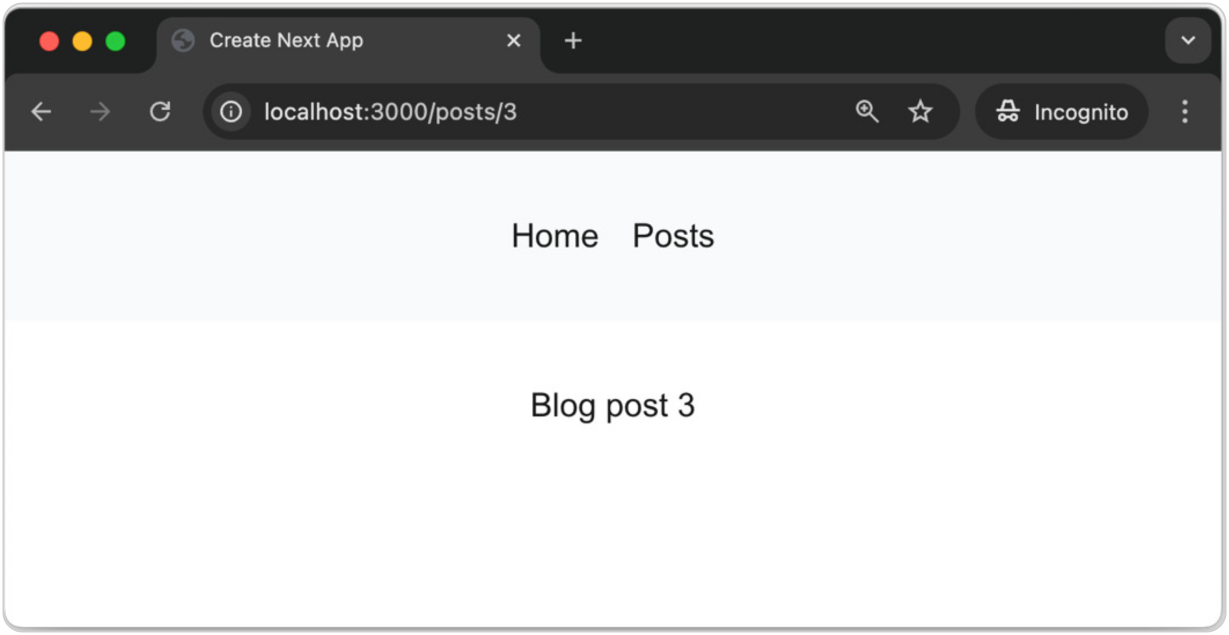


Figure 6.9 – Post page displaying its id route parameter

5. Let's add some validation to ensure the **id** route parameter is numeric. If it isn't numeric, we'll inform the user that the page isn't found:

```
import { notFound } from "next/navigation";
export default async function Post( ... ) {
    const id = Number((await params).id);
    if (!Number.isInteger(id)) {
        notFound();
    }
    return <main>Blog post {id}</main>;
}
```

The Next.js **notFound** function will display the default Next.js not found component.

6. We want to display the blog post title and description on the **Post** page. First, let's access the **posts** variable and find the relevant post using

the **id** route parameter. Make the following additions in **src/app/posts/[id].page.tsx**:

```
import { posts } from '@/data/posts';
export default async function Post( ... ) {
    ...
    const post = posts.find(
        (post) => post.id === Number(id),
    );
    return <main>Blog post {params.id}</main>;
}
```

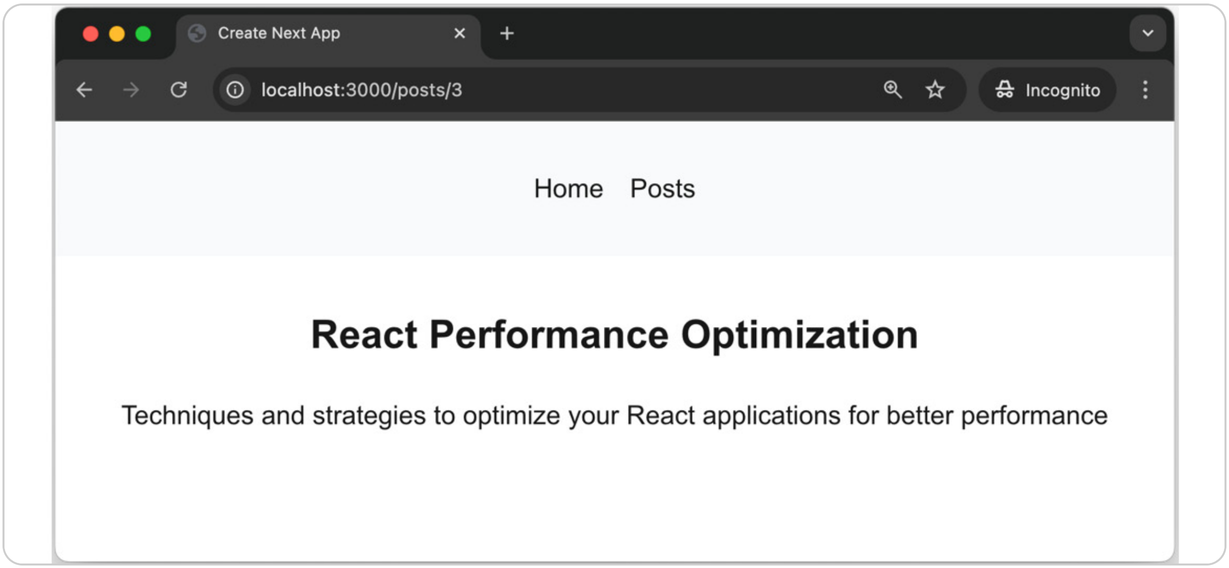
7. If the post isn't found, output a helpful message by adding the highlighted lines:

```
export default async function Post( ... ) {
    ...
    const post = posts.find( ... );
    if (!post) {
        notFound();
    }
    return <main>Blog post {params.id}</main>;
}
```

8. Finally, replace the content of the main element with the blog post title and description as follows:

```
export default async function Post( ... ) {
    ...
    return (
        <main>
            <h2>{post.title}</h2>
             <p>{post.description}</p>
        </main>
    );
}
```

9. Now go to different blog posts in the running app – you'll see the title and description displayed:



React Performance Optimization

Techniques and strategies to optimize your React applications for better performance

Figure 6.10 – Post page with title and description

10. If you enter an invalid **id** route parameter in the browser's URL, a message will be displayed that you have not found it.

That completes this section on dynamic routes. Here's a quick recap:

- Dynamic routes in Next.js allow you to create pages that respond to URL parameters. We used this feature to display different blog posts based on the **id** route parameter.
- The **params** prop or the **useParams** hook in Client Components can access the route parameters and dynamically update the content based on the URL.

For more information on dynamic routes, see the following page in the Next.js documentation:

https://nextjs.org/docs/app/building-your-application/routing/dynamic-routes

Next, we will learn about the other type of URL parameter – a search parameter.

Using search parameters

In this section, we will learn about search parameters in Next.js 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.

In Next.js, search parameters can be accessed via a `searchParams` prop as follows:

```
export default async function Page({
    searchParams,
}: {
    searchParams: Promise<{
        [key: string]: string | string[] |
undefined;
    }>;
}) {
    const params = await searchParams;
    return (
        <main>
            Searching: {params.type},
{params.when}
        </main>
    );
}
```

Notice that **`searchParams`** is asynchronous, so accessing it must be awaited and the component must be declared as asynchronous.

The type annotation for **`searchParams`** is a little complex, so let's break it down:

- **`[key: string]`** is an **index signature** representing any property name. This is because any search parameter can be added to the URL even though our components only use **`type`** and **`when`**.
- The union that follows the index signature is all the types that search parameters can have. Again, we can't fully control what a user puts in the URL, so this represents what could happen.
- The type is wrapped in the **`Promise`** type because **`searchParams`** is asynchronous.

If you want to use a search parameter lower in the component tree, Next.js has a **`useSearchParams`** hook that can be used. This is only available in Client Components and not RSCs. Here's an example of `type` and `when` parameters being accessed:

```
'use client';
import { useSearchParams } from
'next/navigation';
export function SomeComponent() {
    const searchParams = useSearchParams();
    const type = searchParams.get('type');
    const when = searchParams.get('when');
    ...
}
```

So, `useSearchParams` is a little different to `useParams` for route parameters. It returns a standard JavaScript `URLSearchParams` interface, which provides functions to access search parameters. A `get` function on the interface allows access to a particular search parameter value. For more information on `URLSearchParams`, see the following link: https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams.

Next, we will add search functionality to our app.

Adding search functionality to the app

We will add a search input to the header of the app. Submitting a search will take the user to the **Posts** page with a filtered set of blog posts matching the search criteria. Carry out the following steps:

1. First, open **`src/components/Header.tsx`** and add a search form as shown here:

   ```
   import Form from 'next/form';
   export function Header() {
       ...
       return (
           <header>
               <Link ... >Home</Link>
               <Link ... >Posts</Link>
                <Form action="/posts">
                    <input
                        type="search"
                        name="criteria"
                        placeholder="Search"
                        aria-label="Search blog
   posts"
                   />
               </Form>
           </header>
       );
   }
   ```

   `Form` is a Next.js component that extends the native HTML **`form`** element. It allows the form to be submitted without a full page reload.

   The **`action`** attribute on **`Form`** will cause a navigation to the **`posts`** route. The navigation will include a **`criteria`** search parameter with the value from the **`criteria`** input.

2. Run the app by executing **`npm run dev`** in a terminal if it isn't already running.

3. Go to the app in the browser, enter some criteria in the search input in the header, and press *Enter*. Navigation occurs to the **Posts** page with a **`criteria`** search parameter. The blog post list isn't filtered yet – we'll start to implement this in the next step.

4. Now open **`src/app/posts/page.tsx`**. Add a **`searchParams`** prop to the **`Posts`** component and make the component asynchronous as follows:

```
export default async function Posts({
    searchParams,
}: {
    searchParams: Promise<{
        [key: string]: string | string[] |
undefined;
    }>;
}) {
      ...
}
```

5. We have destructured the **`criteria`** search parameter to make referencing it in the following steps a little simpler.

6. Inside the **`Posts`** component, filter the blog posts using the search criteria:

```
export default async function Posts( ... ) {
    const criteria = (await
searchParams).criteria;
    const resolvedPosts =
        typeof criteria === 'string'
            ? posts.filter((post) =>
                    post.title
```

```
                        .toLowerCase()
                        .includes(criteria.toLow
erCase()),
                )
            : posts;
    return ...
}
```

If a search criterion is defined, a case-insensitive filter occurs. Otherwise, unfiltered posts are used.

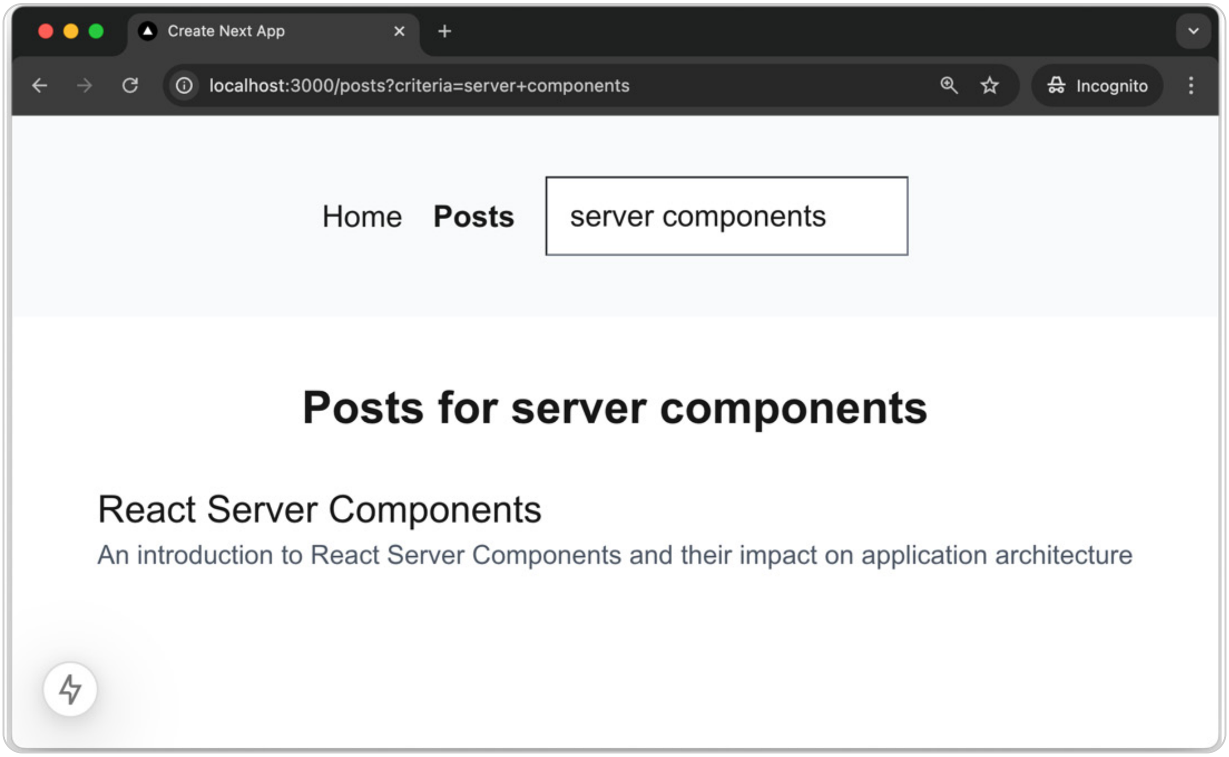
7. Let's also include search criteria in the heading if defined. We'll create a variable to hold the heading:

```
export default async function Posts( ... ) {
    const criteria = ...
    const resolvedPosts = ...
    const resolvedHeading =
        typeof criteria === 'string'
            ? `Posts for ${criteria}`
            : 'Posts';
    return ...
}
```

8. Finally, we can use these variables in the JSX as follows:

```
export default async function Posts( ... ) {
    ...
    return (
        <main>
            <h2>{resolvedHeading}</h2>
            <ul>
                {resolvedPosts.map((post) =>
(
                    ...
                ))}
           </ul>
        </main>
    );
}
```

9. In the running app, if we do a search, the blog post list is filtered:



Posts for server components

React Server Components

An introduction to React Server Components and their impact on application architecture

Figure 6.11 – Filtered blog post list

10. That completes the app. Stop the app from running using *Ctrl* + *C* in the terminal.

That completes this section on search parameters. To recap, search parameters in Next.js are accessed through a **`searchParams`** prop or the **`useSearchParams`** hook.

For more information on search parameters, see the following page in the Next.js documentation:

https://nextjs.org/docs/app/api-reference/functions/use-search-params

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

Summary

Next.js gives us a comprehensive set of features to create multi-page apps. Different routes are defined using folders and a special **`page.tsx`** file. Dynamic routes are defined using square brackets containing the route parameter. We created a static route for a blog post list and a dynamic route for each blog post.

A shared layout component is defined in **`layout.tsx`** in the relevant folder. We used the root layout to share a **`Header`** component.

The **`Link`** component is the recommended way of navigating in Next.js and can be used in RSCs as well as Client Components. We used this in the blog post list as well as the app header. The **`useRouter`** hook allows programmatic navigation in Client Components.

Route and search parameters can be accessed via **`params`** and **`searchParams`** props, respectively. They can also be accessed in

Client Components via **`useParams`** and **`useSearchParams`** hooks. We used an **`id`** route parameter in the blog post dynamic route. We also used a **`criteria`** search parameter in the blog post list route.

The knowledge you have learned in this chapter has given you the skills to write apps with multiple pages.

In the next chapter, we will enhance the app we created in this chapter to use data from a database, as we learn how React can interact with server data.

Questions

Answer the following questions to check what you have learned in this chapter:

1. In the Next.js app router, what file defines a route segment's layout that persists across multiple pages?
2. When navigating to the **`/home`** URL in a Next.js app, a 404 error message is returned. Here's the contents of the **`src/app`** folder:

   ```
   app/
   ├── globals.css
   ├── layout.tsx
   ├── page.tsx
   └── home.tsx
   ```

   What could be the problem?
3. Can a Next.js **`Link`** component be used in an RSC?

4. In Next.js, a **`Header`** component needs to be placed on every page. What approach would you recommend?
5. When navigating to the **`/customers/10`** URL in a Next.js app, a 404 error message is returned. Here's the contents of the **`src/app/customers`** folder:

```
app/
├── customers
      └── id
                └── page.tsx
```

   What's the problem?

Answers

1. **`Layout.tsx`**.
2. The **`/home`** path will render the **`app/home/page.tsx`** file, not **`app/home.tsx`**.
3. Yes, it can.
4. **`Header`** can be added to the root layout as follows:

```
export default function RootLayout( ... ) {
    return (
        <html lang="en">
             <body ... >
                 <Header />
                   {children}
            </body>
        </html>
        );
    }
```

5. The **`id`** folder name needs to have square brackets to form a dynamic route (i.e., **`[id]`**).

Part 3:Data

This part covers different approaches for interacting with database data on a server with its benefits and trade-offs. It includes fetching data using React Server Components and React Client Components. It also includes mutating data using React Server Functions and various form approaches to capture the mutation data.

This part has the following chapters:

7
Server Component Data Fetching and Server Function Mutations

In this chapter, we will learn how React can fetch data from the server using a Server Component and take a look at the benefits of doing so. We will enhance the blog post app we created in the last chapter to use data from a database to make the data interactions realistic. We'll implement loading indicators and error handlers along the way to ensure the user experience is great.

We will learn about React Server Functions to mutate server data and use this knowledge to create new blog post data in our app. Again, we'll ensure a great user experience by implementing a mutating indicator and handling errors.

By the end of this chapter, you will have the skills to implement web pages with fast load times and the knowledge to create actions on those pages in a simple and robust manner.

We'll cover the following topics:

- Understanding server and client data fetching
- Getting set up
- Fetching data using a Server Component

- Adding loading indicators using React Suspense
- Handling errors with React error boundaries
- Mutating data using a Server Function

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-Third-Edition/tree/main/Chapter07.

Understanding server-side and client-side data fetching

Before implementing data fetching in our app, we will learn about the differences between client-side and server-side data fetching and the benefits of each approach.

Client-side data fetching

Client-side data fetching happens after a Client Component is initially rendered. We touched on the client-side data fetching process in *Chapter 5*, *Using React Server and Client Components*. As a recap, here's a visualization of the steps involved:

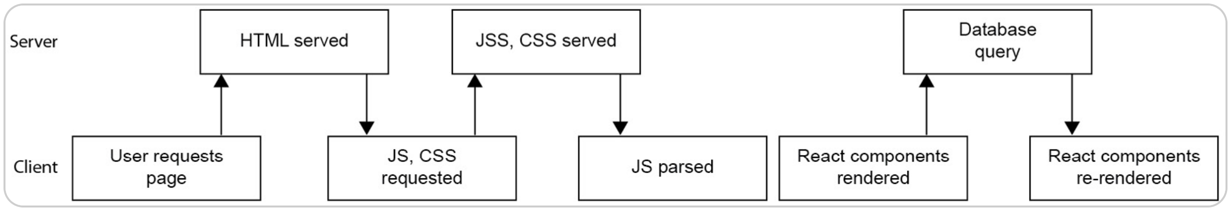


Figure 7.1 – Client-side data fetching steps

Here are the key points:

- A lot happens before data is fetched – the HTML, JavaScript, and CSS are downloaded from the server, the JavaScript is parsed, and an initial rendering of the React components happens.
- Many HTTP requests happen – one to get the HTML, one for the JavaScript, one for the CSS, and another to perform the database query. If multiple components fetch data, more HTTP requests happen; this is often referred to as a **network waterfall**.
- The React components need to be re-rendered before the user sees data.
- Data can be refreshed at a later point using an HTTP request from the client.

Server-side data fetching

Server-side data fetching happens when a **React Server Component** (**RSC**) is run on the server. Again, we touched on this process in *Chapter 5*, *Using React Server and Client Components*. As a recap, here's a visualization of the steps involved:

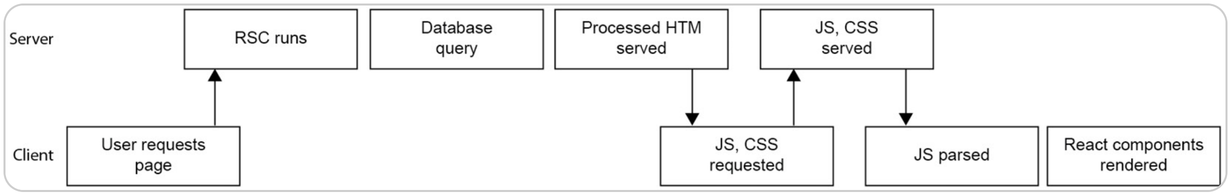


Figure 7.2 – Server-side data fetching steps

Here are the key points:

- The database query happens early in the process while the RSC is run on the server.
- RSCs generally result in less JavaScript being downloaded than Client Components. This not only means that the JavaScript is downloaded faster, but also that it is parsed faster.
- Fewer HTTP requests happen. If multiple RSCs fetch data, this all happens on the server in a single HTTP request. For example, in the following **Home** RSC, a user is fetched from the database, and then the user's tasks are fetched in the **TaskList** child RSC. So, both these database queries happen in the same HTTP request:

```
async function Home() {
    const user = await fetchUser();
    return (
        <main>
            <h2>Welcome, {user.name}</h2>
            <TaskList userId={user.id} />
        </main>
    );
}
async function TaskList({ userId }: ...) {
    const tasks = await fetchTasks(userId);
    return <ul> ... </ul>;
}
```

- React components are able to display data in their initial render.
- Data can't be refreshed at a later point in an RSC without a full page reload.

> ***NOTE***
>
> *It's worth noting that the data could be fetched from another server instead of a direct database query for both client-side and server-side data fetching.*

Understanding the benefits

Both server-side and client-side data fetching have benefits and downsides. Here are the benefits of server-side data fetching:

- The user will see the data on the page quicker than client-side data fetching.
- RSCs have access to cookies, allowing cookie-based authentication and authorization checks. Client Components don't have access to cookies, so an alternative authentication and authorization approach will need to be taken.

Here are the benefits of client-side data fetching:

- Data can be refreshed without a full page load. A typical use case where data refreshing is required is a dashboard that needs to show up-to-date data.
- Data paging can be done without a full page load.
- Infinite scroll can only be done in a Client Component. This is where additional data is fetched as the user scrolls down a list.
- Component composition and reuse are easier in a Client Component. This is because Client Components can fetch their own data, reducing the need for data to be passed in through props. As a result, developers can more easily compose high-level components from low-level ones without tightly coupling them to a specific data-fetching hierarchy. In contrast, when using RSCs to fetch data, the data is passed to a Client Component through props. This introduces a rigid top-down data flow, where high-level RSCs must be aware of specific data requirements of their nested children. This breaks encapsulation and makes it harder to reuse components in different contexts since their data dependencies are no longer internal.
- Client-side data fetching can be used with any web server framework – it doesn't have to support RSCs or even be based on JavaScript.

That concludes this section on how client-side and server-side data fetching works. To recap, client-side data fetching occurs after the initial rendering and involves multiple HTTP requests, resulting in a slower user experience due to re-rendering. Server-side data fetching is more efficient, showing data during the initial render and using fewer HTTP requests, but it does not allow data refreshing without a full page reload.

Next, we will set up the project for our app and then implement a server-side data fetch.

Getting set up

In this section, we will set up the code for the app we will work on in this chapter, which is the blog app from the last chapter. We will also connect the app to a SQLite database.

Creating the project

Carry out the following steps to set up the project. It is the project we ended with in the last chapter, plus some additional styles needed for this chapter:

1. In a terminal, execute the following command to create the project:

   ```
   npx create-next-app@latest blog --ts --eslint
   --app --src-dir --import-alias "@/*" --no-
   tailwind --turbopack
   ```

2. Still in the terminal, move to the project folder and open Visual Studio Code using the following commands:

   ```
   cd blog
   ```

```
code .
```

3. Prettier can be set up in the same manner as Vite, as we covered in *Chapter 1*, *Getting Started with React*. Feel free to add automatic code formatting to this project.

4. Copy the following files from the **`src`** folder in the GitHub repository at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter07/start/src. Add them into the project in the same folder structure, replacing any existing files:

    - **`app/layout.tsx`**
    - **`app/page.tsx`**
    - **`app/posts/page.tsx`**
    - **`app/posts/[id]/page.tsx`**
    - **`app/global.css`**
    - **`components/Header.tsx`**
    - **`data/posts.ts`**

The project is now as it was at the end of the last chapter, with some additional styles.

Setting up the database

SQLite is a SQL-based database engine that is simple to set up. We will use a popular fork of SQLite called **libSQL** that works nicely with Next.js. The dependency for libSQL is already installed.

NOTE

More information on libSQL can be found at https://turso.tech/libsql.

Carry out the following steps to create our blog database and view the data:

1. Install SQLite by executing the following in a terminal:

   ```
   npm i @libsql/client
   ```

2. Create a script that we'll eventually run to create our database. Create a folder called **`scripts`** in the **`src`** folder and then a file called **`createDatabase.mjs`** in this folder. Copy the script from the GitHub repository at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter07/start/src/scripts/createDatabase.mjs and paste it into the file.

3. To create the database, run the following command in the terminal:

   ```
   node src/scripts/createDatabase.mjs
   ```

   This command executes the script using Node.js. After the command has completed a **`blog.db`** database file will appear in the **`data`** folder.

4. The data inside **`blog.db`** can be viewed using a Visual Studio Code extension called **SQLite3 Editor**.

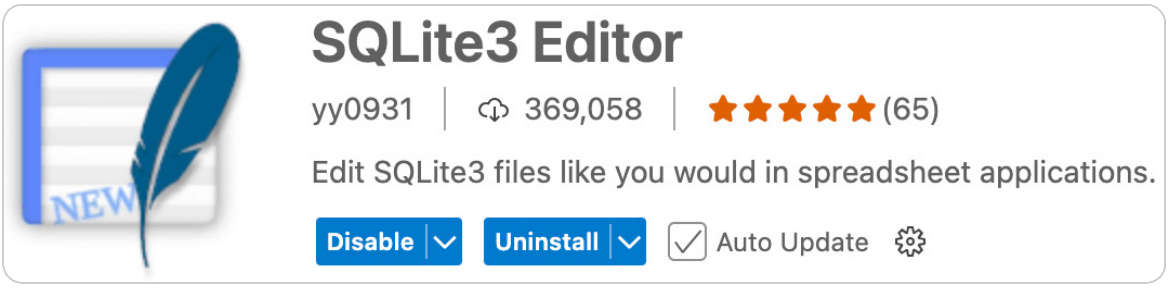
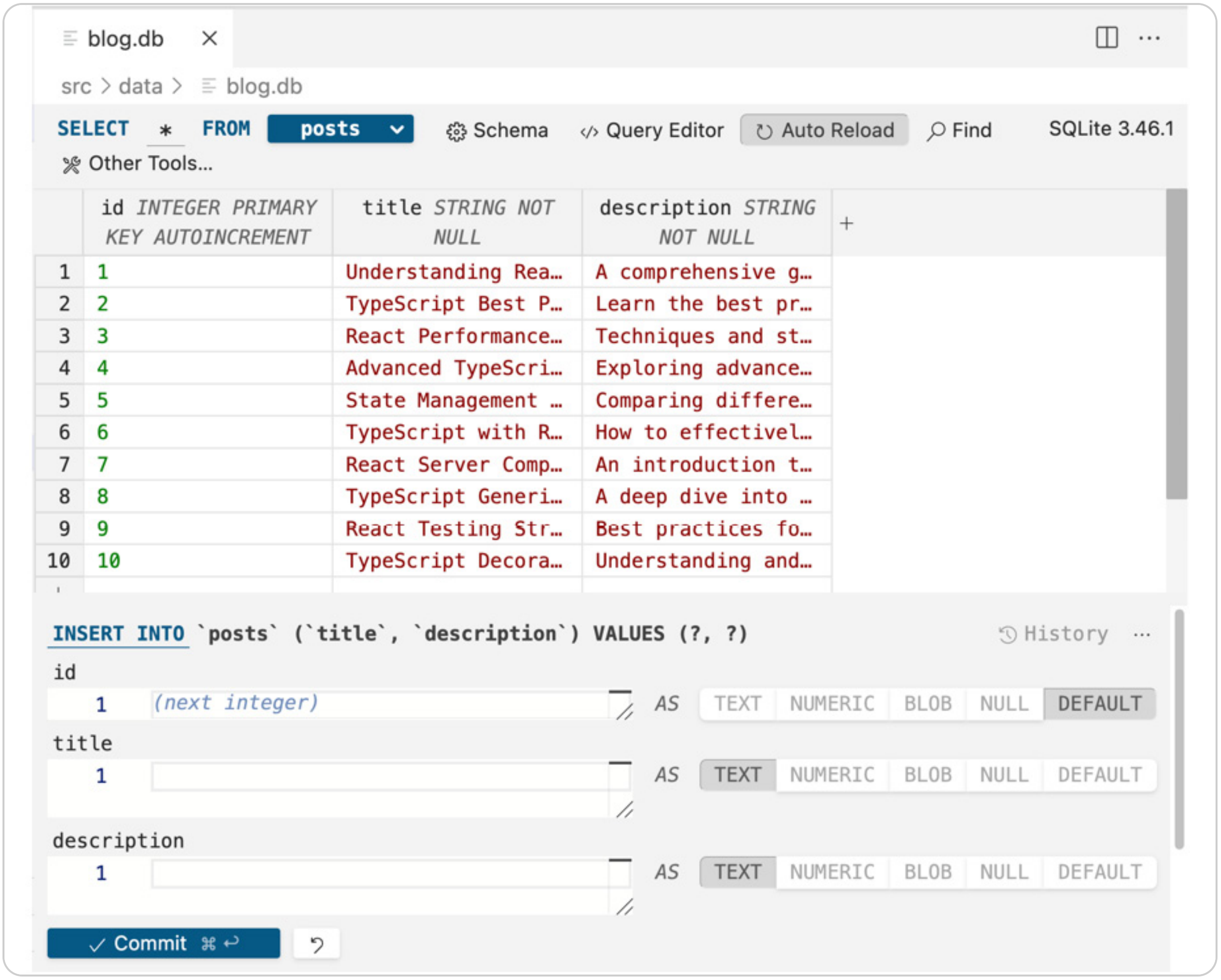

SQLite3 Editor
yy0931
369,058
(65)
Edit SQLite3 files like you would in spreadsheet applications.
Disable
Uninstall
Auto Update

Figure 7.3 – SQLite3 Editor extension

This extension also allows us to edit data, which will be useful later in the chapter. So, install this extension and you'll then be able to click on the file in the **Explorer** panel to view the data.



```
SELECT * FROM posts
```

	id INTEGER PRIMARY KEY AUTOINCREMENT	title STRING NOT NULL	description STRING NOT NULL
1	1	Understanding Rea…	A comprehensive g…
2	2	TypeScript Best P…	Learn the best pr…
3	3	React Performance…	Techniques and st…
4	4	Advanced TypeScri…	Exploring advance…
5	5	State Management …	Comparing differe…
6	6	TypeScript with R…	How to effectivel…
7	7	React Server Comp…	An introduction t…
8	8	TypeScript Generi…	A deep dive into …
9	9	React Testing Str…	Best practices fo…
10	10	TypeScript Decora…	Understanding and…

```
INSERT INTO `posts` (`title`, `description`) VALUES (?, ?)
```

id	1 (next integer)	AS	TEXT NUMERIC BLOB NULL [✓ DEFAULT]
title	1	AS	[✓ TEXT] NUMERIC BLOB NULL DEFAULT
description	1	AS	[✓ TEXT] NUMERIC BLOB NULL DEFAULT

Figure 7.4 – Blog data in SQLite3 Editor

Excellent! That completes this section on setting up the project for this chapter. Here's a quick recap:

- We created a Next.js project using the `create-next-app` tool and copied code from the last chapter's app with some additional styles.
- We created a SQLite database for the blog post data and installed a library that we will eventually use to connect to it.

Next, we'll implement some data-fetching code in an RSC.

Fetching data using an RSC

In this section, we will implement data fetching in the blog post list and blog post detail RSCs to get the data from our new database. We will structure the code that interacts with the database in separate functions – we will call them **query functions**.

Implementing query functions

First, we will implement functions to connect to the database and query it to get the necessary data. Follow these steps:

1. We will start by storing the path to the database in an environment variable. This is common practice because it allows the app in different environments to connect to different databases. Create a file called `.env` in the project root with the following content:

```
DB_URL=file:src/data/blog.db
```

2. Create a file called **`queries.ts`** in the **`src/data`** folder. Add the following content to **`queries.ts`** to import a function from libSQL that will allow us to connect to the database:

   ```
   import { createClient } from '@libsql/client';
   ```

3. Create a type for the post data as follows:

   ```
   type Post = {
       id: number;
       title: string;
       description: string;
   };
   ```

   We will use this to strongly type the data returned from the function queries in the subsequent steps.

4. Create a function to get all the blog posts as follows:

   ```
   export async function getAllPosts() {
       const client = createClient({
           url: process.env.DB_URL ?? '',
       });
       const data = await client.execute(
           'SELECT id, title, description FROM
   posts',
       );
       client.close();
       return data.rows as unknown as Post[];
   }
   ```

First, we use the **`createClient`** function to connect to the database. The database URL environment variable is accessed via the **`process.env`** object, which is the standard way of accessing environment variables in Next.js. The function returns an object containing an **`execute`** function, which allows the database to be queried.

We use the **`execute`** function to select the **`id`**, **`title`**, and **`description`** fields from the **`posts`** table, which is where our blog post data is.

We then close the database connection and return the rows returned in the query.

Notice that a type assertion is used in the **`return`** statement to strongly type the returned data. This is a little messy because we need to assert to **`unknown`** before asserting **`post[]`**. We will clean this up later in this chapter, but we'll stick with this for now.

5. Implement a similarly structured function to get filtered blog posts:

```
export async function getFilteredPosts(
    criteria: string,
) {
    const client = createClient({
          url: process.env.DB_URL ?? '',
    });
    const data = await client.execute({
        sql: ‘SELECT id, title, description
FROM posts WHERE               title LIKE ?’,
        args: [`%${criteria}%`],
    });
    client.close();
    return data.rows as unknown as Post[];
}
```

The main difference this time is that we pass a SQL statement and a SQL parameter to the database client’s **`execute`** function. The SQL statement contains a **`WHERE`** clause that filters the data using the **`title`** field and a parameter (**`?`** is a parameter in SQLite). The SQL parameter value is passed via the **`args`** property in an array. We set the SQL parameter value to the **`criteria`** parameter that was passed into the function inside **`%`** symbols, which are wildcards in SQLite. The wildcards and the SQL **`LIKE`** operator mean rows with matching **`criteria`** inside **`title`** will be returned by the query. It is worth noting that SQLite is case insensitive, so the **`criteria`** case doesn’t need to match the **`title`** field.

6. The last query function is to get a single blog post:

```
export async function getPost(id: number) {
    const client = createClient({
          url: process.env.DB_URL ?? ‘’,
    });
    const data = await client.execute({
        sql: ‘SELECT id, title, description
FROM posts WHERE id              = ?’,
        args: [id],
    });
    client.close();
    if (data.rows.length === 0) {
        return undefined;
    }
    return data.rows[0] as unknown as Post;
}
```

This time, we pass the blog post **`id`** as a SQL parameter. If no rows are found, we return **`undefined`**; otherwise, we return the first row found.

That's the query functions implemented. Next, we will call these functions in the relevant RSCs.

Calling query functions from RSCs

We will now call the query functions we just implemented in the blog post list and blog post detail RSCs. Follow these steps:

1. Start by opening **`src/app/posts/page.tsx`** and replacing the **`posts`** **`import`** statement with an **`import`** statement to import the required query functions:

```
import Link from 'next/link';
import {
    getAllPosts,
    getFilteredPosts,
} from '@/data/queries';
```

2. Within the **`Posts`** component, update the assignment of **`resolvedPosts`** to call the query functions as follows:

```
const resolvedPosts =
        typeof criteria === 'string'
            ? await getFilteredPosts(criteria)
            : await getAllPosts();
    const resolvedHeading =
        typeof criteria === 'string'
            ? `Posts for ${criteria}`
            : 'Posts';
```

3. Now, open **`src/app/posts/[id]/page.tsx`** and make similar changes. Start by replacing the existing **`import`** statement with an **`import`** statement for **`getPost`**.

```
import { getPost } from '@/data/queries';
```

4. Next, update the **`posts`** variable assignment with a call to **`getPost`**:

```
const post = await getPost(id);
```

5. Run the app by executing **`npm run dev`** in the terminal. The app will behave as it did in the last chapter, but now it is using a real database. Try going to the post list page, doing a search, and then going to a particular blog post to make sure everything is working.

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

We've now implemented data fetching from a database in two RSCs. Next, we will clean up the TypeScript typing of the query function data.

Adding type safety to a database query

At the moment, we are trusting that the data from the database queries is of type `Post[]` for the `getAllPosts` and `getFilterPosts` functions, and also type `Post` for the `getPost` function. We know this is the case in our example, but in the real world, database schemas may change without connecting code being updated accordingly. This can happen when different teams own the database, and the code, and a database change isn’t properly communicated.

If the type representing query data is incorrect, then code consuming wouldn’t work as expected and may result in an unexpected runtime error. In our app, this would result in the post list and detail pages not rendering correctly. In larger apps, these kinds of bugs can be hard to pinpoint and take a while to fix.

To protect the code against unexpected database changes, the type of the data can be checked at runtime to see if it is as expected. A popular library called **Zod** can elegantly do schema validation checks. If the checks fail, an error is thrown. This might not seem ideal, but it gives clarity as to where the problem is, which helps ensure a quick resolution.

Carry out the following steps to add schema validation with Zod to our database queries:

1. First, install Zod by executing the following command in a terminal:

   ```
   npm i zod
   ```

2. Create a new file in the **`src/data`** folder called **`schema.ts`**. This will contain the Zod schemas for the database queries. Add the following content to **`schema.ts`**:

   ```
   import { z } from 'zod';
   export const postSchema = z.object({
       id: z.number(),
       title: z.string(),
       description: z.string(),
   });
   export const postsSchema =
   z.array(postSchema);
   ```

   We have defined two schemas:

   - The first, **`postSchema`**, represents a single blog post. The schema specifies an object with a numeric **`id`** property, and string **`title`** and **`description`** properties.
   - The second, **`postsSchema`**, represents multiple blog posts. It builds on **`postSchema`**, simply specifying that it is an array of **`postSchema`**.

   Both schemas are exported so that we can use them in our function queries.

3. Open **`queries.ts`** and remove the existing **`Post`** type as it will become redundant after the subsequent steps.

4. Import the schemas into **`queries.ts`**:

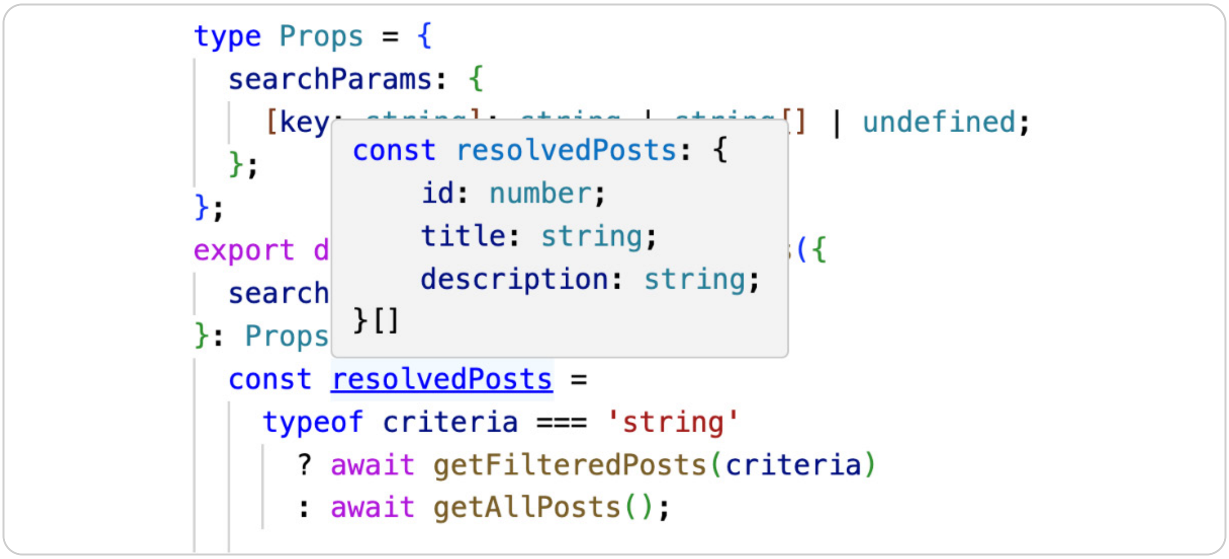
   ```
   import { postsSchema, postSchema } from
   './schema';
   ```

5. In the query functions, validate the query data using the schemas just imported, removing the type assertions:

   ```
   export async function getAllPosts() {
       ...
       return postsSchema.parse(data.rows);
   }
   export async function getFilteredPosts( ... )
   {
       ...
       return postsSchema.parse(data.rows);
   }
   export async function getPost( ... ) {
       ...
       return postSchema.parse(data.rows[0]);
   }
   ```

   The **`parse`** function in the Zod schemas performs the validation check. An error is thrown if the check fails. If the check is successful, the data passed into **`parse`** is returned and typed as per the schema.

6. Open **`src/app/posts/page.tsx`** and hover over the **`resolvedPosts`** variable where two of the query functions are called in its assignment. You'll see that it has been correctly typed from the Zod schema.



```
type Props = {
  searchParams: {
    [key:                              [] | undefined;
  };
};
export d                              ({
  search
}: Props
  const resolvedPosts =
    typeof criteria === 'string'
      ? await getFilteredPosts(criteria)
      : await getAllPosts();
```

```
const resolvedPosts: {
    id: number;
    title: string;
    description: string;
}[]
```

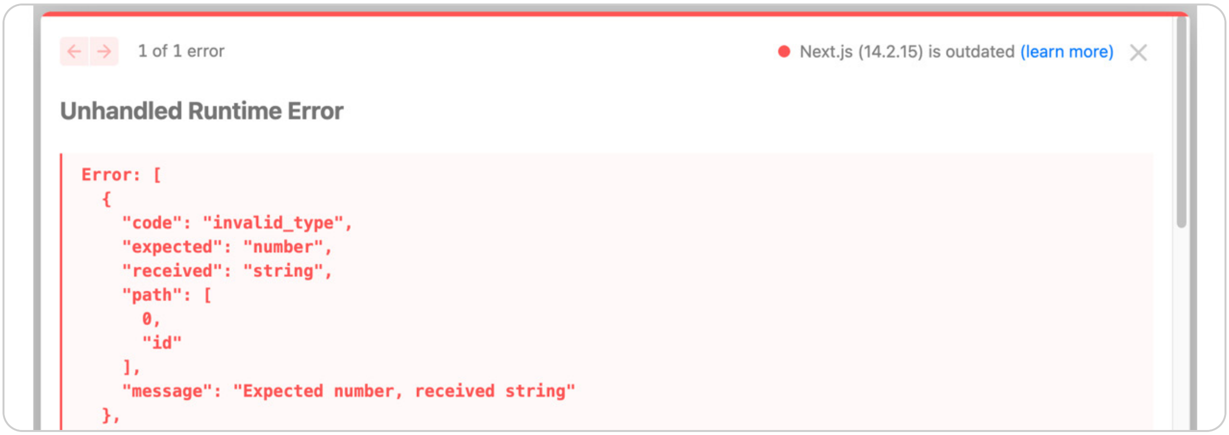
Figure 7.5 – Correctly typed resolvedPosts variable

7. Run the app by executing **`npm run dev`** in the terminal. The app will behave as it did before.

8. To see the query schema validation in action, open **`queries.ts`** and temporarily cast **`id`** to text in the SQL statement in the **`getAllPosts`** function. This simulates a potentially breaking database change:

```
export async function getAllPosts() {
    ...
    const data = await client.execute(
        'SELECT CAST(id as text) as id, ...',
    );
    ...
}
```

9. Visit the posts list page in the running app and you'll see a runtime error created by Zod:



```
1 of 1 error                                   Next.js (14.2.15) is outdated (learn more)

Unhandled Runtime Error

Error: [
  {
    "code": "invalid_type",
    "expected": "number",
    "received": "string",
    "path": [
      0,
      "id"
    ],
    "message": "Expected number, received string"
  },
```

Figure 7.6 – Runtime Zod error

The error message is very clear, telling us that the **`id`** field in every row is a string rather than an expected number. This is very useful for developers but not for users – we'll improve the user error experience later in this chapter.

10. Revert the change to **`getAllPosts`** before continuing so that the error is resolved.

That completes the improved typing of query data. More information on Zod can be found at https://zod.dev/.

Here's a quick recap of this section:

- Data can be fetched in an RSC by simply calling the query in the RSC function body before the **`return`** statement. The query can be awaited, and the RSC can be marked as **`async`** if the query is asynchronous.
- Zod can be used to ensure data from database queries is type safe.

Next, we will improve the data-fetching user experience by adding a loading indicator.

Adding loading indicators using React Suspense

In this section, we will learn about **React Suspense** and use it to implement a loading indicator in the RSCs that fetch data in our app. This will improve the loading user experience.

Understanding the need for loading indicators

Currently, the data-fetching user experience in our app is reasonable because the process is quick. This is because everything is running locally, and so the latency is low. This is also because the database is small, and the queries are simple, so they execute fast. Lastly, we are the only user using the app.

When apps run on real servers with larger, more complex databases, queries will be a little slower – particularly when many users use the app.

Loading indicators let the user know that the app is loading the page, and they prevent the app from feeling laggy.

Adding a delay

Before implementing a loading indicator, we will simulate a more significant data fetching delay. This will enable us to feel the difference loading indicators make. Carry out the following steps:

1. Open **`queries.ts`** and add a **`delay`** function as follows:

   ```
   async function delay() {
       await new Promise((resolve) =>
           setTimeout(resolve, 1000),
       );
   }
   ```

   The delay is 1 second, which is longer than we would expect the typical query execution to take. However, this allows us to experience what a slow load would be like.

2. Add a call to **`delay`** at the top of the three queries:

   ```
   export async function getAllPosts() {
       await delay();
       ...
   }
   export async function getFilteredPosts( ... )
   {
       await delay();
       ...
   }
   export async function getPost(id: number) {
       await delay();
       ...
   }
   ```

3. Trying navigating to different pages in the app. Clicking links that navigate to pages containing data will feel laggy.

We will eventually improve the experience so that the page immediately loads with a loading indicator informing the user that some content is still loading. Before this, we will learn about a critical React feature needed to do this.

Understanding React Suspense

React Suspense enables components to wait for asynchronous tasks during rendering. A common asynchronous task that Suspense is used for is *data fetching*. It allows the rendering of some JSX elements to be *suspended* while data is being fetched, allowing other elements to be rendered normally. In addition, a fallback can be specified for the suspended elements, which can be a loading indicator.

The Suspense fallback can be shown to the user before suspended elements because RSCs containing suspended elements are streamed to the browser. So, the Suspense fallback will be sent to the browser in the first chunk, with suspended elements following when they are ready. The chunks of the RSC are added to the DOM and shown to the user as they are downloaded to the browser.

In the following example, the **`Page`** RSC will immediately display a **Loading …** message. After the name has been fetched, the loading indicator will be replaced with a **Hello, {fetched name}** message:

```
export default function Page() {
    return (
        <Suspense fallback={<div>Loading...
</div>}>
            <Name />
        </Suspense>
    );
}
async function Name() {
    const name = await getName();
    return <div>Hello, {name}</div>;
}
```

The **`Suspense`** component is from React, which forms a **Suspense boundary**. Its children are suspended from rendering until their asynchronous tasks are complete. Its **`fallback`** attribute allows an alternative component to be rendered while the suspended children's asynchronous tasks run.

> ### NOTE
> *Next.js has a built-in loading indicator convention that uses React Suspense under the hood. However, the whole page component is replaced with the loading component. More information can be found at https://nextjs.org/docs/app/api-reference/file-conventions/loading.*

Next, we will add loading indicators on the blog post list and detail pages.

Implementing loading indicators

We will use React Suspense to add loading indicators to parts of the page components for the blog post list and details. We will extract the asynchronous parts of the components into child components so that they can be wrapped with a **Suspense** component. Carry out the following steps:

1. Start by creating a loading indicator component that can be reused in both page components. Create a file called **Loading.tsx** in **src/components** with the following content:

   ```
   export function Loading() {
       return (
           <div className="skeleton">
               <div className="skeleton-item-
   title"></div>
               <div className="skeleton-item-
   desc"></div>
           </div>
       );
   }
   ```

   The component renders **div** elements that visually represent placeholders for the blog post title and description. The referenced styling can be found in **global.css**.

2. We will extract the data fetching and rendering of the blog post list from **Posts** into a **PostList** component. Start by adding a new file called **PostList.tsx** to the **src/components** folder. Copy the content into this file from the GitHub repository at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter07/fetching-rsc/src/components/PostList.tsx. Here's an extract of this code:

```
export async function PostList({ criteria }:
...) {
    const resolvedPosts = ...
    return (
        <ul>
            {resolvedPosts.map((post) => (
                <li key={post.id}>
                    <Link href=
{`/posts/${post.id}`}>
                        {post.title}
                    </Link>
                    <p>{post.description}</p>
                </li>
            ))}
        </ul>
    );
}
```

3. Open **src/app/posts/page.tsx** and remove the **Link** and query imports. Add imports for the **Suspense**, **Loading**, and **PostList** components as follows:

```
import { Suspense } from 'react';
import { Loading } from
```

```
'@/components/Loading';
import { PostList } from
'@/components/PostList';
```

4. Remove the **`resolvedPosts`** variable in the **`Posts`** component.

5. Replace the **`ul`** and **`li`** elements in the JSX with **`PostList`** inside a **`Suspense`** component as follows:

```
export default async function Posts( ... ) {
    const criteria = ...
    const resolvedHeading = ...
    return (
        <main>
            <h2>{resolvedHeading}</h2>
            <Suspense fallback={<Loading />}>
                <PostList criteria={criteria}
/>
            </Suspense>
        </main>
    );
}
```

That completes the reworking of the **`Posts`** component.

6. Open **`src/app/posts/[id]/page.tsx`**. We will extract the data fetching and rendering from **`Post`** into a **`PostDetail`** component. Start by adding a new file at **`src/components/PostDetail.tsx`** with the following content, which can be largely copied from the **`Post`** component:

```
import { getPost } from '@/data/queries';
export async function PostDetail({
    id,
}: {
    id: number;
}) {
    const post = await getPost(id);
    if (!post) {
        return <p>Post not found</p>;
    }
    return (
        <>
            <h2>{post.title}</h2>
            <p>{post.description}</p>
        </>
    );
}
```

7. Open **`src/app/posts/[id]/page.tsx`** and remove the **`getPost import`** statement. Add imports for the **`Suspense`**, **`Loading`**, and **`PostDetail`** components as follows:

```
import { Suspense } from 'react';
import { Loading } from
'@/components/Loading';
import { PostDetail } from
'@/components/PostDetail';
```

8. Inside the **`Post`** component, remove the **`post`** variable and guard clause for the blog post not being found. Keep the **`id`** variable in place, and it's casting to a number. Also, keep the guard clause in place to handle cases where the ID isn't numeric.

9. Replace the JSX inside the **`main`** element with **`PostDetail`** inside a **`Suspense`** as follows:

```
<main>
    <Suspense fallback={<Loading />}>
        <PostDetail id={id} />
    </Suspense>
</main>
```

That completes the reworking of the **`Post`** component. The full component file is available on GitHub at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter07/fetching-rsc/src/app/posts/%5Bid%5D/page.tsx.

10. In the running app, try clicking links to navigate to different pages in the app. The pages will immediately appear with a loading indicator if data needs to be fetched:

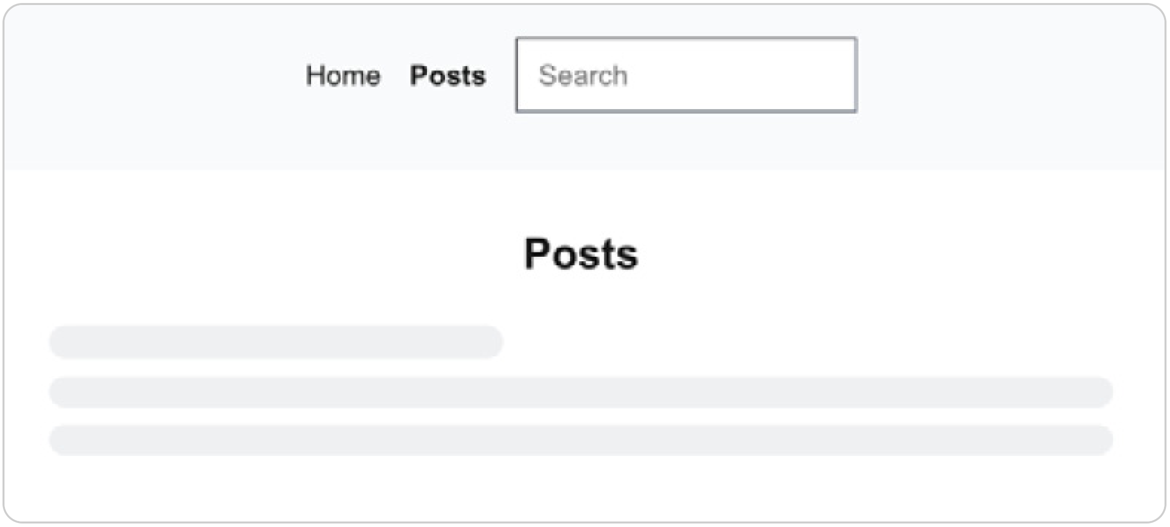


Figure 7.7 – Loading indicator on post list page

11. Stop the app from running by pressing *Ctrl* + *C* in the terminal.

12. Remove the simulated delays in the queries now that we are happy with the loading indicators.

That completes the implementation of the loading indicators. For more information on React Suspense, see the page in the React documentation: https://react.dev/reference/react/Suspense.

Here's a quick recap:

- React Suspense is a React component that suspends its children from rendering while an asynchronous database query function executes.
- When children are suspended, the Suspense component allows a loading indicator component to be rendered instead.
- When the database query function executed has been completed, the suspended children will render, replacing the loading indicator.
- This pattern works in an RSC because React Suspense causes RSCs to be streamed to the browser.

Next, we will learn how to improve the handling of errors during data fetching.

Handling errors with React error boundaries

In this section, we will learn about error handling using React error boundaries. With this knowledge, we will improve the error handling in our app.

Understanding React error boundaries

A React error boundary is a component that catches errors in its children during rendering. React error boundaries are available in React class components. To use them in function components we can use a package called **`react-error-boundary`** that exposes an **`ErrorBoundary`** component.

Here's an example use of the **`ErrorBoundary`** component:

```
export function SomeComponent() {
    return (
        <main>
            <h2>Some heading</h2>
            <ErrorBoundary
                FallbackComponent={ErrorAlert}
                onError={(error, info) => {
                    // Log to error service
                }}
            >
            <SomeChildComponent />
        </ErrorBoundary>
        </main>
    );
}
```

The **`FallbackComponent`** attribute allows an error component to replace the children that failed to render.

The **`onError`** attribute allows errors to be captured, allowing them to be logged. It's worth noting that this attribute can only be used if the consuming component is a Client Component and not an RSC because it is an event.

Here's the definition of the error fallback from the preceding code snippet:

```
"use client";
export function ErrorAlert({
    error,
    resetErrorBoundary,
}: {
    error: Error;
    resetErrorBoundary: () => void;
}) {
    return (
        <div role="alert">
            <h3>Something went wrong</h3>
            <p>{error.message}</p>
            <button onClick=
{resetErrorBoundary}>Retry</button>
          </div>
    );
}
```

The error fallback component must be a Client Component. It takes in the following props:

- **`error`**. This is the raised **`Error`** object that contains all the information about the error.
- **`resetErrorBoundary`**. This allows the state in the error boundary to be reset to reattempt rendering.

NOTE

Next.js has a built-in error boundary indicator convention, but the whole page component is replaced with the error component. More information can be found at https://nextjs.org/docs/app/api-reference/file-conventions/error.

Next, we will add error boundaries on the blog post list and detail pages.

Implementing error boundaries

Will use **`ErrorBoundary`** from the **`react-error-boundary`** package. We'll use this in the blog post list and detail pages. Carry out these steps:

1. Install the **`react-error-boundary`** package by executing the following command in a terminal:

   ```
   npm i react-error-boundary
   ```

2. We will start by adding an error fallback component that displays the error to the user. Create a file called **`ErrorAlert.tsx`** in the **`src/components`** folder containing the following:

   ```
   "use client";
   export function ErrorAlert({
       error,
       resetErrorBoundary,
   }: {
   ```

```
    error: Error;
    resetErrorBoundary: () => void;
}) {
    return (
        <div role="alert">
            <h3>Something went wrong</h3>
            <p>{error.message}</p>
            <button onClick=
{resetErrorBoundary}>
                Retry
            </button>
        </div>
    );
}
```

The component displays the error message with a **Retry** button underneath that resets the error boundary.

3. We will wrap the **`ErrorBoundary`** from **`react-error-boundary`**, so that the fallback and error reporting are centralized. Create a new file called **`ErrorBoundary.tsx`** in **`src/components`** with the following content:

```
'use client';
import type { ReactNode } from 'react';
import {
    ErrorBoundary as ReactErrorBoundary,
} from 'react-error-boundary';
import { ErrorAlert } from './ErrorAlert';
export function ErrorBoundary({
    children,
}: {
    children: ReactNode;
}) {
    return (
        <ReactErrorBoundary
            FallbackComponent={ErrorAlert}
            onError={(error, info) => {
                console.error('Unexpected
error', {
                    error,
                    info,
                });
            }}
        >
        {children}
        </ReactErrorBoundary>
    );
}
```

We called our component **ErrorBoundary** and aliased **ErrorBoundary** from **react-error-boundary** as **ReactErrorBoundary** to prevent them from colliding.

We've specified **ErrorAlert** as the error fallback, and we are outputting errors to the console.

4. Open **src/pages/posts/page.tsx** and wrap our **ErrorBoundary** around **PostList**:

```
import {
    ErrorBoundary
} from '@/components/ErrorBoundary';
...
export default function Posts( ... ) {
    ...
    return (
        <main>
```

```
            ...
            <Suspense ... >
                <ErrorBoundary>
                    <PostList ... />
                </ErrorBoundary>
            </Suspense>
            </main>
        );
    }
```

5. Open **`src/pages/posts/[id]/page.tsx`** and wrap our **`ErrorBoundary`** around **`PostDetail`**:

```
import {
    ErrorBoundary
} from '@/components/ErrorBoundary';
...
export default async function Post( ... ) {
    ...
    return (
        <main>
            <Suspense ...>
                <ErrorBoundary>
                    <PostDetail ... />
                </ErrorBoundary>
            </Suspense>
        </main>
    );
}
```

The error boundaries are now implemented.

6. Run the app by executing **`npm run dev`** in a terminal, navigate to the blog post list page, open the browser developer tools, and go to the **Components** panel. We can force the **`PostList`** into an error state by

searching for the component in the tree, selecting **`PostList`** in the tree, and selecting the option that contains the exclamation mark icon.

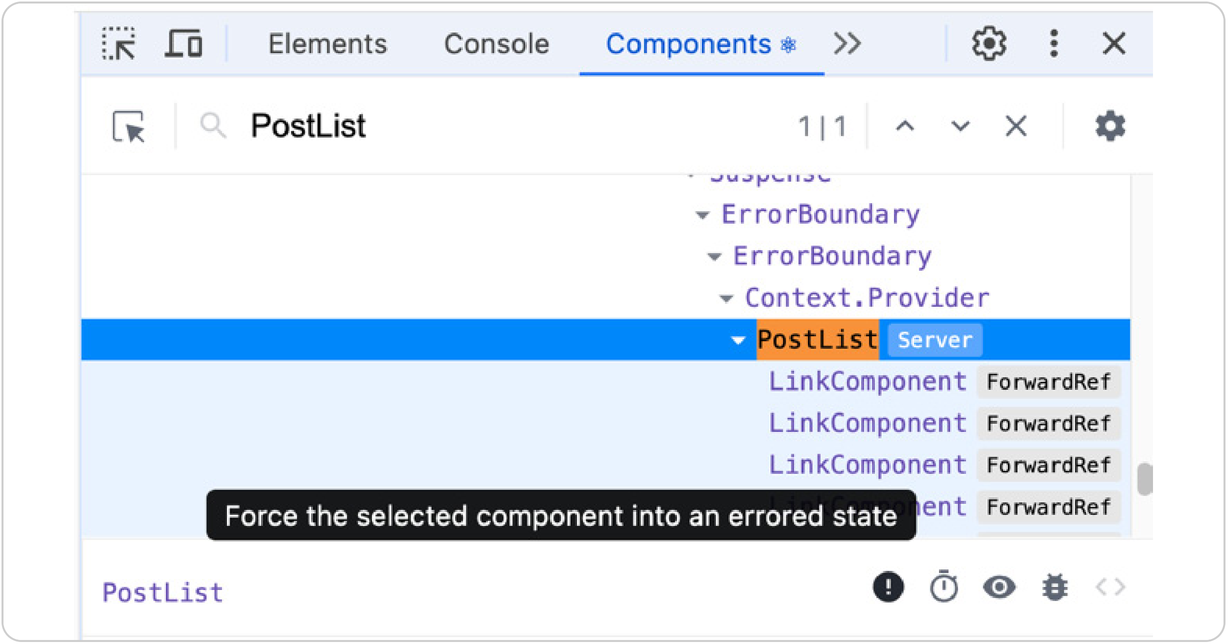


Figure 7.8 – Forcing a component into an error state

The error will then appear, replacing the blog post list:

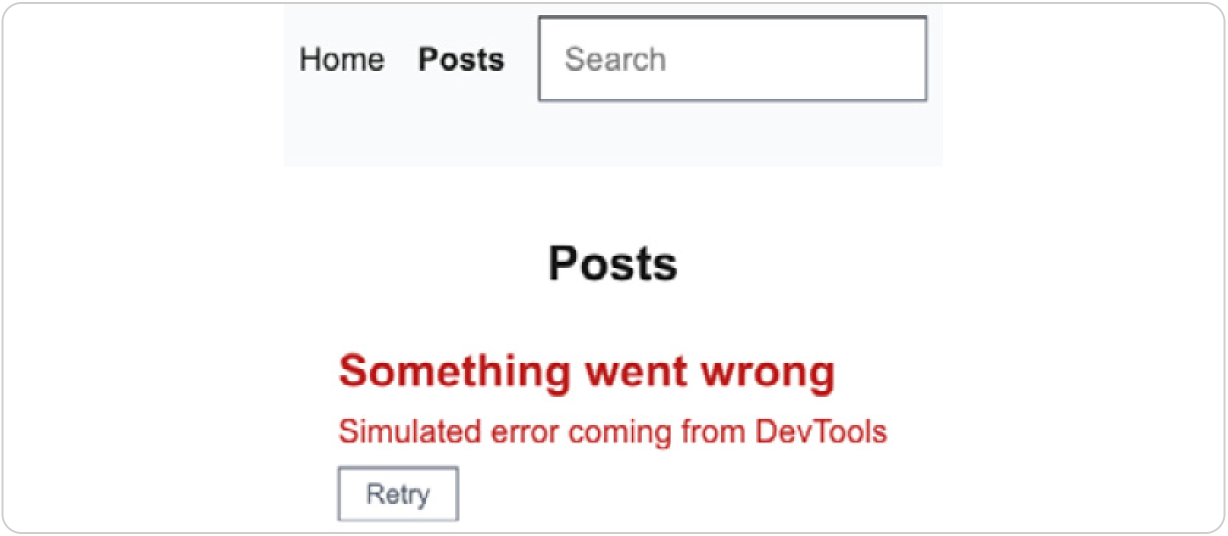


Figure 7.9 – PostList component in an error state

7. Click the **Retry** button to reset the error state. The `PostList` component will successfully render.
8. Repeat the previous two steps for the `PostDetail` component on the blog post details page. You will find that the error boundary works nicely there as well.

That completes the error boundaries in our app and this section. Here's a quick recap:

- A React error boundary is a component that catches errors from its child components.
- The error boundary component can render a custom error alert component when in an error state.
- The error boundary component can also log errors to an error service, which helps production troubleshooting.

For more information on React error boundaries, see the following page in the React documentation: https://react.dev/reference/react/Component#catching-rendering-errors-with-an-error-boundary.

Next, we will learn how to mutate data.

Mutating data using a Server Function

In this section, we will learn how to mutate database data via a React Server Function. We will create a button in our app to create a new blog post leveraging a Server Function. We will also learn how to implement a progress indicator for the Server Function and handle any errors.

Understanding a Server Function

A React Server Function is exactly as the name suggests – it's a function that executes on the server. A common use case is to execute a database mutation.

Traditionally, a React app would call a web API to execute code on the server. Server Functions simplify this task.

Here is an example of a Server Function used within an RSC. It renders a button that, when clicked, deletes a product from a database:

```
export function DeleteButton({id}: {id: number})
{
    async function deleteProduct() {
        'use server';
        const client = createClient({
            url: process.env.DB_URL ?? ''
        });
        await client.execute({
            sql: 'DELETE FROM products WHERE id =
?',
            args: [id],
        });
        client.close();
    }
    return (
        <button type="button" onClick=
{deleteProduct}>
            Delete
        </button>
    );
}
```

The **'use server'** directive marks the **deleteProduct** function as a Server Function. This function is never downloaded to the browser – it only exists on the server. This means sensitive information about the database, such as its connection credentials, won't leak into the browser.

You may be thinking that the preceding code snippet can't be an RSC because it contains a button click handler. We previously learned that RSCs aren't capable of handling events because they need to run on the client. In Next.js, this code works because RSCs can pass Server Functions to the client in certain event handlers, and **onClick** is one that supports this capability – **onSubmit** is another that is commonly used. As previously stated, the whole Server Function isn't sent to the client – just a reference to it.

Server Functions can be called from Client Components as well, but they must be in a different file. Here's the same example, but this time **DeleteButton** is a Client Component:

- Here's the **DeleteButton.tsx** file:

  ```
  'use client';
  import { deleteProduct } from
  '@/data/deleteProduct';
  export function DeleteButton({ id }: { id:
  number }) {
      return (
          <button
              type="button"
              onClick={() => deleteProduct(id)}
          >
          Delete
          </button>
      );
  }
  ```

  The **deleteProduct** Server Function is imported and called as a regular function.

- Here's **deleteProduct.ts**:

  ```
  'use server';
  import { createClient } from '@libsql/client';
  export async function deleteProduct(id:
  number) {
      const client = createClient({
          url: process.env.DB_URL ?? ''
      });
      await client.execute({
          sql: 'DELETE FROM posts WHERE id = ?',
          args: [id],
      });
  ```

```
        client.close();
    }
```

The **'use server'** directive at the top of the file marks the exported functions in the file as Server Functions.

Now that we understand how to implement a Server Function, we'll use one to create new blog posts in our app.

Creating a Server Function

In this section, we will create a button on the blog posts list page that creates a new blog post. We'll use a Server Function to implement this. Carry out the following steps:

1. We'll start by implementing the Server Function that inserts a new blog post into the database. Create a new file called **createPost.ts** in the **src/data** folder and add the following content to it:

```
'use server';
import { revalidatePath } from 'next/cache';
import { createClient } from '@libsql/client';
export async function createPost(
    title: string,
    description: string,
) {
    const client = createClient({
        url: process.env.DB_URL ?? '',
    });
    await client.execute({
        sql: 'INSERT INTO posts(title, 
description) VALUES (?, ?)',
        args: [title, description],
    });
    client.close();
    revalidatePath('/posts');
}
```

The function connects to the database and runs a SQL query to insert a new record into the **`posts`** table with the **`title`** and **`description`** passed to the function.

The **`revalidatePath`** function invalidates the Next.js cache so that the new post appears in the list.

The **`'use server'`** directive means that the function will be available as a Server Function.

2. We will now create a **`button`** Client Component that calls **`createPost`**. Create a file called **`NewPost.tsx`** in the **`src/components`** folder with the following content:

```
    'use client';
    import { createPost } from
    '@/data/createPost';
    export function NewPost() {
        async function handleClick() {
            await createPost(
                'New Post',
                'New Post Description',
            );
        }
        return (
            <div className="actions">
                <button type="button" onClick=
  {handleClick}>
                  Create New Post
              </button>
          </div>
      );
  }
```

The component renders a button that calls the **`createPost`** Server Function in its click handler. We have hardcoded the blog post title and description **`New Post`** and **`New Post Description`** respectively.

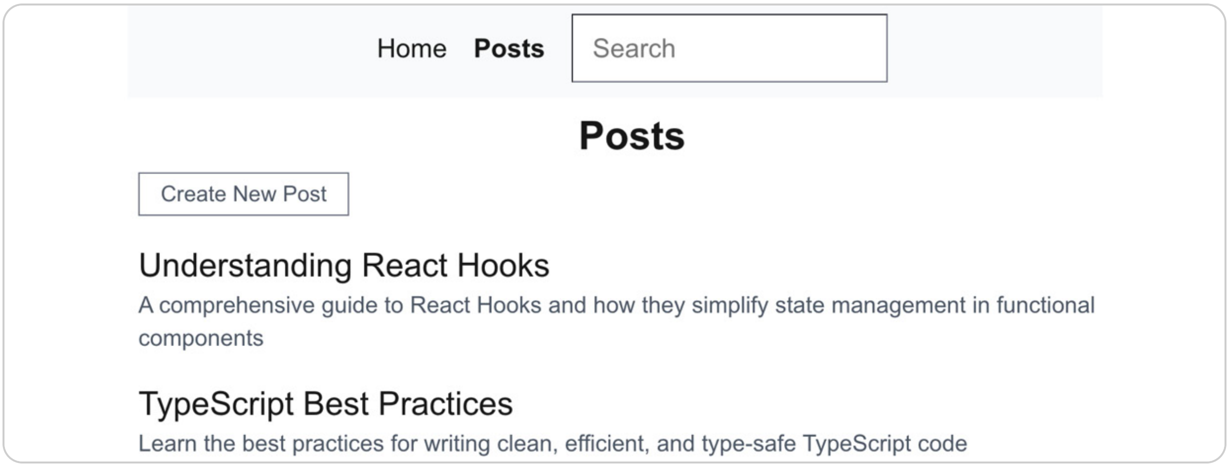
A form could be used to capture the title and description from the user before submitting it to the Server Function. We will cover this in detail in *Chapter 9*, *Working with Forms*.

As you write the code to call **`createPost`**, take a moment to appreciate the IntelliSense. Try also to pass non-string arguments and experience the type checking. These features work across the client/server network boundary, just like there is no boundary at all!

3. The last implementation step is referencing the **`NewPost`** component in the **`Posts`** page. Open **`src/app/posts/page.tsx`** and add **`NewPost`** as follows:

```
  import { NewPost } from
  '@/components/NewPost';
  ...
  export default async function Posts( ... ) {
        const criteria = ...
        const resolvedHeading = ...
        return (
            <main>
                <h2>{resolvedHeading}</h2>
                <NewPost />
                <Suspense ...>
                ...
                </Suspense>
            </main>
        );
    }
```

Run the app by executing **`npm run dev`** in a terminal, then navigate to the blog post list page. The **Create New Post** button appears above the blog post list:



Home Posts Search

Posts

Create New Post

Understanding React Hooks

A comprehensive guide to React Hooks and how they simplify state management in functional components

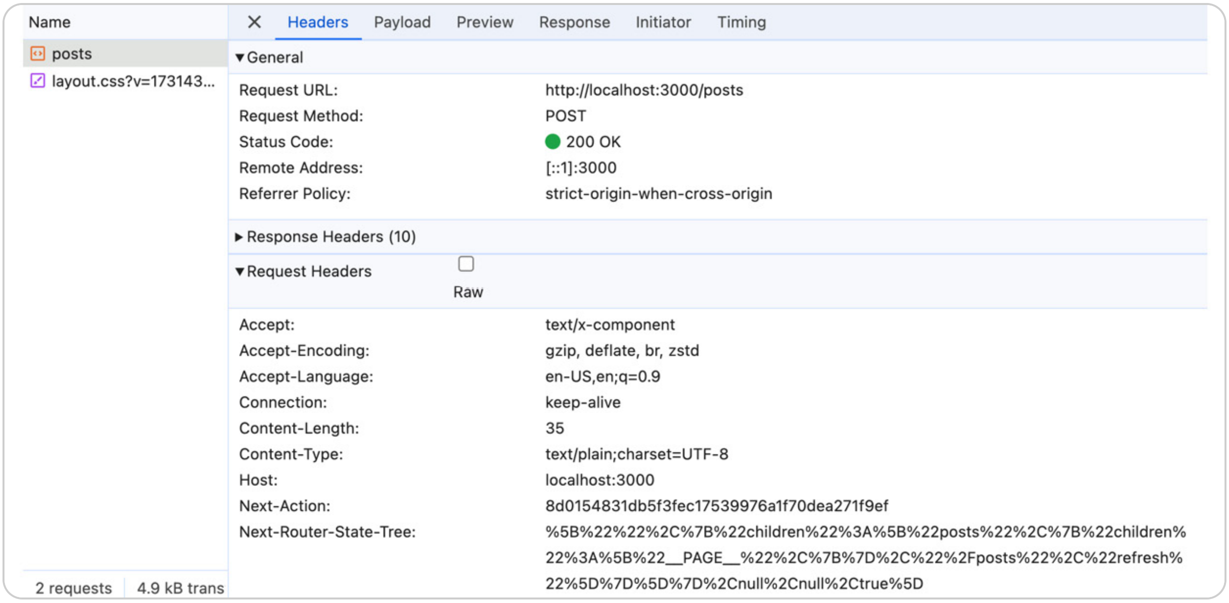
TypeScript Best Practices

Learn the best practices for writing clean, efficient, and type-safe TypeScript code

Figure 7.10 – Create New Post button on blog post list page

4. Open the browser developer tools, go to the **Network** panel, and clear any existing network calls shown. We will use this panel to better understand how the Server Function is called in *step 6*.

5. Click the **Create New Post** button, and the new blog post will appear at the bottom of the list. You can use the Visual Studio Code SQLite extension to verify the record is added to the `posts` table in the database.
6. In the browser developer tools **Network** panel, find the request to the `posts` path and look at the HTTP request method, an HTTP request header called `Next-Action`, and the request payload.



```
Request URL:              http://localhost:3000/posts
Request Method:           POST
Status Code:              200 OK
Remote Address:           [::1]:3000
Referrer Policy:          strict-origin-when-cross-origin

Accept:                   text/x-component
Accept-Encoding:          gzip, deflate, br, zstd
Accept-Language:          en-US,en;q=0.9
Connection:               keep-alive
Content-Length:           35
Content-Type:             text/plain;charset=UTF-8
Host:                     localhost:3000
Next-Action:              8d0154831db5f3fec17539976a1f70dea271f9ef
Next-Router-State-Tree:   %5B%22%22%2C%7B%22children%22%3A%5B%22posts%22%2C%7B%22children%
                          22%3A%5B%22__PAGE__%22%2C%7B%7D%2C%22%2Fposts%22%2C%22refresh%
                          22%5D%7D%5D%7D%2Cnull%2Cnull%2Ctrue%5D
```

Figure 7.11 – Server Function HTTP request

Here are the key points on how the Server Function is invoked:

- It is invoked using an HTTP `POST` request to the page the component is rendered on.
- The request `Next-Action` HTTP header is an indirect reference to the `createPost` function on the server. This is set to a GUID-like string in this example.

- The payload is an array containing the arguments to pass to the Server Function.

This completes the implementation and consumption of our Server Function. For more information on Server Functions, see the following page in the React documentation: https://react.dev/reference/rsc/server-functions.

Next, we will add a progress indicator to the **NewPost** component.

Adding a progress indicator

In the **NewPost** component, we will use an **isMutating** React state variable to render a progress indicator. Carry out the following steps:

1. Open **NewPost.tsx** and add the following highlighted code:

```
import { useState } from 'react';
...
export function NewPost() {
    const [isMutating, setIsMutating] =
useState(false);
    async function handleClick() {
        setIsMutating(true);
        await createPost( ... );
        setIsMutating(false);
    }
    return (
        <div ... >
            <button ... >
```

```
                    {isMutating
                        ? 'Creating...'
                        : 'Create New Post'}
                    </button>
                </div>
            );
        }
```

We set **`isMutating`** to **`true`** before the Server Function call and then **`false`** when it has finished executing. The button content changes to **Creating…** during the mutation, forming the progress indicator.

2. Click the **Create New Post** button to see the progress indicator. It will probably appear and disappear quickly, so you can use the browser developer tools **Network** panel to throttle the network to make the progress indicator appear for longer.

That completes the progress indicator. Next, we'll improve the error handling around the Server Function.

Handling errors

If the Server Function errors, no error notification is shown to the user. Instead, the operation will appear to hang. As developers, we'll see an error alert that Next.js displays, but this won't be the case in a production build.

Carry out the following steps to improve the error handling:

1. We will start by enhancing the Server Function to catch errors in the database calls. Open **createPost.ts** and import the following LibSQL types:

```
...
import {
    ...,
    type Client,
    type ResultSet,
} from '@libsql/client';
```

2. Inside **createPost**, add a **try**-**catch** statement around the creation of the database client and execution of the SQL query as follows:

```
export async function createPost( ... ) {
    let client: Client | undefined;
    let result: ResultSet | undefined;
    try {
        client = createClient( ... );
        await client.execute( ... );
    } catch {
        return { ok: false };
    } finally {
        if (client) {
            client.close();
        }
    }
    revalidatePath('/posts');
}
```

3. Add a **return** statement to return whether the mutation is successful, along with the **id** of the new post:

```
export async function createPost( ... ) {
    ...
    revalidatePath('/posts');
    return {
        ok: true,
        id: result
            ? result.lastInsertRowid
            : undefined,
    };
}
```

4. We can now use the **return** object from **createPost** in the **NewPost** component to set some state, which in turn will display an error or success message. Open **NewPost.tsx** and start by setting a **status** state from the **createPost** call:

```
export function NewPost() {
    ...
    const [status, setStatus] = useState<
        'pending' | 'error' | 'success'
    >('pending');
    async function handleClick() {
        setIsMutating(true);
        const result = await createPost(
            'New Post',
            'New Post Description',
        );
        setStatus(result.ok ? 'success' :
'error');
        setIsMutating(false);
    }
    ...
}
```

5. Lastly, we can use **`status`** to render a success or error message as follows:

```
export function NewPost() {
    ...
    return (
        <div ... >
            <button ...>
            ...
            </button>
            {status === 'error' && (
                <span role="alert">
                    An unexpected error
occurred
                </span>
            )}
            {status === 'success' && (
                <span role="alert"
className="success">
                    Post successfully created
                </span>
            )}
        </div>
    );
}
```

   Note that we are rendering a general error message in this situation. In *Chapter 9*, *Working with Forms*, we'll learn how to render more specific error messages from a Server Function.

6. In the running app, a success message now appears when the **Create New Post** button is clicked and the **`createPost`** Server Function is

successful. The table name in the SQL statement in **`createPost`** can be adjusted to test an error.

Figure 7.12 – Error message when createPost errors

Don't forget to correct the SQL statement before continuing.

That completes the error handling and this section on Server Functions. Here's a quick recap:

- Server Functions allow server code to be executed from a React component in a simple and type-safe manner.
- A Client Component progress indicator can be implemented using React state that is updated before and after the execution of the Server Function.
- A **`try-catch`** statement can be used to handle errors in a Server Function. Whether an error occurred can be returned to the React component to render in a success or error state.

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

Summary

In this chapter, we learned that server-side data fetching can improve the performance of the initial loading of a page because of reduced network calls and reduced client-side JavaScript.

We used server-side data fetching in our app, in an RSC, querying a SQLite database for the data. We used React Suspense to implement a loading indicator and a React error boundary to handle and report errors.

We learned that Server Functions are a simple and type-safe approach to mutating data. Finally, we used a Server Function in our app to add a new blog post to our database.

You now have the skills to make web pages load blazingly fast and the knowledge to quickly implement maintainable actions on those pages.

In the next chapter, we learn how to implement client-side data fetching and mutations using a popular third-party library.

Questions

Answer the following questions to check what you have learned in this chapter:

1. We have the following RSC for a page. The page takes a while because the **`getPeople`** function is a bit slow. What can we do to improve the user experience other than improve the performance of the **`getPeople`** function?

```
export default async function People() {
    const people = await getPeople();
```

```
        return (
            <ul>
                {people.map((person) => (
                    <li key={person}>
                        <span>{person}</span>
                    </li>
                ))}
            </ul>
        );
    }
```

2. Consider the following RSC. When an error is thrown in the **`getPeople`** function, the whole React app fails to render. How can we improve this situation so that only the **`PeopleList`** component fails to render?

```
    export async function PeopleList() {
        const people = await getPeople();
        return (
            <ul>
                {people.map((person) => (
                    <li key={person}>
                        <span>{person}</span>
                    </li>
                ))}
            </ul>
        );
    }
```

3. What is the advantage of using React Server Functions over traditional API routes in Next.js?
4. Why isn't it recommended to use React Server Functions in Next.js for data fetching?

5. The following component raises a build error in Next.js. What's the problem?

```
"use client";
import { useState } from "react";
export function Counter() {
    const [count, setCount] = useState(1);
    async function saveCount(count: number) {
        use server";
        db.count.save(count);
    }
    function handleClick() {
        setCount((prev) => {
            const newCount = prev + 1;
            saveCount(newCount);
            return newCount;
        });
    }
    return (
        <button onClick={handleClick}>
            {count}
        </button>
    );
}
```

Answers

1. React Suspense can be used with the **`people`** list to provide a loading indicator while its data is being fetched. First, the **`people`** list needs to be extracted into its own component:

```
export async function PeopleList() {
    const people = await getPeople();
    return (
        <ul>
            {people.map((person) => (
                <li key={person}>
                    <span>{person}</span>
                </li>
            ))}
        </ul>
    );
}
```

The **`People`** component can then reference **`PeopleList`** inside a **`Suspense`** component with a loading indicator fallback.

```
export default function People() {
    return (
        <main>
            <Suspense fallback=
{<div>Loading...</div>}>
                <PeopleList />
            </Suspense>
        </main>
    );
}
```

If using Next.js and **`People`** is a page-level component, the conventional **`loading.tsx`** file can be used as an alternative solution.

2. An **`ErrorBoundary`** component (from the **`react-error-boundary`** package) can wrap **`PeopleList`** in the component tree to catch errors and render a fallback component:

```
<ErrorBoundary FallbackComponent=
{ErrorFallback}>
    <PeopleList />
</ErrorBoundary>
```

The fallback component must be a Client Component.

```
"use client";
export function ErrorFallback() {
    return <div role="alert">An error
occurred</div>;
}
```

3. React Server Functions allow server code to be executed from a React component using less code and better type safety than a Next.js API route handler.

4. React Server Functions in Next.js use a HTTP **`Post`** rather than a HTTP **`GET`** so they won't use the browser or CDN cache.

5. Client Components can't contain inline Server Functions. So, the **`saveCount`** function needs to be extracted into a separate file with the **`"use server"`** directive at the top:

```
// saveCount.ts
"use server";
export async function saveCount(count: number)
{
    db.count.save(count);
}
// Counter.tsx
"use client";
import { useState } from "react";
import { saveCount } from "./saveCount";
export function Counter() {
    const [count, setCount] = useState(1);
    function handleClick() {
        setCount((prev) => {
            const newCount = prev + 1;
            saveCount(newCount);
            return newCount;
        });
    }
    return (
        <button onClick={handleClick}>
            {count}
        </button>
    );
}
```

8
Client Component Data Fetching and Mutations with TanStack Query

In this chapter, we will learn how React can fetch data from a Client Component, reworking the blog post app from the last chapter.

We will start by exploring the challenges of using `useEffect` for client-side data fetching. We'll move on to use a popular library called **TanStack Query** for client-side data fetching and experience how it simplifies this task. We will maintain the great UX created in the last chapter, using the capabilities in TanStack Query to implement loading indicators and error handling. We will also use TanStack Query to rework the mutation code.

We will cover using React Server Functions for client-side data fetching and understand the downsides of this approach. We'll also learn how a Next.js Route Handler compares with React Server Functions for client-side data fetching.

By the end of the chapter, you'll have the knowledge to implement maintainable, robust client-side data fetching and mutations.

We'll cover the following topics:

- Fetching data using TanStack Query
- Using a Route Handler with TanStack Query
- Mutating data using a TanStack Query mutation

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-Third-Edition/tree/main/Chapter08.

Fetching data using TanStack Query

In this section, we will start by understanding how to fetch data using React's `useEffect` Hook and the challenges with this approach. We will then explore how to fetch data using a popular library called **TanStack Query** and use it in our app, experiencing the benefits.

Understanding the challenges with useEffect for data fetching

In this section, we will explore fetching data using React's **`useEffect`** Hook and the challenges with this approach.

We used the **`useEffect`** Hook in *Chapter 3*, *Using React Hooks* to fetch a person's name. Here's a reminder of the code:

```
const [name, setName] = useState<string |
undefined>();
const [loading, setLoading] = useState(true);
useEffect(() => {
  getPerson().then((person) => {
    setLoading(false);
    setName(person.name);
  });
}, []);
if (loading) {
  return <div>Loading ...</div>;
}
```

Notice that we need to manage the state for the fetched data and the loading indicator. If error handling is added, more state is required, and the code gets a little more complex:

```
...
const [error, setError] = useState<Error |
undefined>();
useEffect(() => {
  getPerson()
    .then((person) => {
      ...
    })
    .catch(e => {
      setError(e);
      setLoading(false);
    });
}, []);
...
```

In reality, we'd need to pass the person ID to the fetching function, making things even more complex:

```
  useEffect(() => {
    getPerson(personId)
      ...
  }, [personId]);
```

Now, a refetch will occur whenever `personId` changes. A race condition can occur if `personId` changes while a fetch is still in progress. Also, if the component is unmounted while a fetch is still in progress, an error will occur when the fetch is completed and the `data` state is set.

The complexity of implementing this code quickly escalates as more edge cases are implemented. The following blog post from Dominik Forgmeister, one of the TanStack Query maintainers, dives into the complexity a little deeper: https://tkdodo.eu/blog/why-you-want-react-query.

That completes this section on fetching data with the `useEffect` Hook. In summary, you can fetch data with the `useEffect` Hook, but it isn't recommended.

Luckily, TanStack Query reduces the complexity of data fetching and is a very popular choice in the React community. We'll learn about this next.

Understanding TanStack Query

TanStack Query simplifies data fetching code by managing the different states for us, such as loading and error states. At its core is a smart data cache for the fetched data. When cached data becomes stale, it automatically refetches data.

Here's a simple code snippet of TanStack Query in a component that fetches data for a product and displays its name and description:

```
function Product({ id }: { id: number }) {
  const { data, error, isPending } = useQuery({
    queryKey: ['products', id],
    queryFn: () =>
      fetch(
        `https://some-server.com/products/${id}`,
      ).then((res) => res.json()),
  });
  if (isPending) return 'Loading... ';
  if (error) return 'Error: ' + error.message;
  return (
    <div>
      <h2>{data.name}</h2>
```

```
        <p>{data.description}</p>
      </div>
    );
  }
```

TanStack Query contains a **`useQuery`** Hook that manages the data fetching process and the different states. Options are passed into **`useQuery`** in an object, and the two most important options are the following:

- **`queryKey`**: A unique key for the data. The TanStack Query cache is capable of storing data from many different queries, so this key identifies the data in the cache. The key in this example is an array containing the word "products" and the numeric product ID in this case.

> **NOTE**
>
> *Using an array for a key is the best practice because it allows TanStack Query to easily differentiate between individual elements with the key and make decisions based on them. For example, if the data for key* ***`['products']`*** *is invalidated, the data for keys* ***`['products',1]`****,* ***`['products',2]`****, … would automatically be invalidated.*

- **`queryFn`**: A function that actually does the data fetching. The browser **`fetch`** function gets the product from the API in this example.

The **`useQuery`** Hook returns an object containing lots of useful variables, including the following:

- **`data`**: The fetched data.
- **`isPending`**: Whether the data has been fetched yet. This can be used to implement a loading indicator, as in the preceding example.
- **`isSuccess`**: When the data has been successfully fetched.
- **`isError`**: When the data fetching errored. This can be used to implement an error alert, as in the preceding example.
- **`error`**: The **`Error`** instance if data fetching errored.

The **`useQuery`** Hook can only be used within a **`QueryClientProvider`** component in the React component tree. This allows the same data cache to be used throughout the app. The **`QueryClientProvider`** component takes in a **`QueryClient`** instance, as in the following example:

```
function App() {
  const [queryClient] = useState(() => new
QueryClient());
  return (
    <QueryClientProvider client={queryClient}>
      ...
    </QueryClientProvider>
  );
}
```

The **QueryClient** instance is held in state so that the same instance is reused after a re-render.

QueryClient can also be used to access the data cache; a common use case is invalidating it so that fresh data is fetched. The following code snippet invalidates the cache for the **[‘products’]** key:

```
queryClient.invalidateQueries({ queryKey:
['products'] });
```

Next, we will set up the project we will be using in this chapter.

Setting up the project

The project we need for this chapter is the one we finished at the end of the last chapter. A copy of this can be found in the GitHub repository at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter08/start.

Next, we will rework our app to use TanStack Query for data fetching.

Fetching blog post data

We will start by reworking the data fetching on the post list page in our app to use TanStack Query. Along the way, we will create a loading indicator and handle errors. Carry out the following steps:

1. Install TanStack Query by executing the following command in the terminal:

   ```
   npm i @tanstack/react-query
   ```

2. We will create a **`Providers`** Client Component containing TanStack Query's **`QueryClientProvider`**. We will then reference this in the root layout, high in the React component tree, so that all our Client Components can access the cache. Create a new file called **`Providers.tsx`** in the **`src/components`** folder with the following content:

   ```
   'use client';
   import {
     QueryClient,
     QueryClientProvider,
   } from '@tanstack/react-query';
   import { ReactNode, useState } from 'react';
   export function Providers({
     children,
   }: {
     children: ReactNode;
   }) {
     const [queryClient] = useState(
       () => new QueryClient(),
     );
     return (
       <QueryClientProvider client={queryClient}>
         {children}
       </QueryClientProvider>
     );
   }
   ```

3. Open **`src/app/layout.tsx`** and add **`Providers.tsx`** to **`RootLayout`** as follows:

```
import { Providers } from
'@/components/Providers';
...
export default function RootLayout( ... ) {
  return (
    <html ... >
      <body ... >
        <Providers>
          <Header />
          {children}
        </Providers>
      </body>
    </html>
  );
}
```

   All the Client Components have access to the cache now.

4. We will expose the database queries as Server Functions so they are callable from Client Components. Open **`queries.ts`** and add a **`'use server'`** directive at the top of the file.

5. Open **`ErrorAlert.tsx`**. We will simplify the **`ErrorAlert`** component because retries are automatic in TanStack Query. Remove the **`Retry`** button and **`resetErrorBoundary`** prop. The component should now be as follows:

```
export function ErrorAlert({
  error,
}: {
  error: Error;
}) {
  return (
    <div role="alert">
      <h3>Something went wrong</h3>
      <p>{error.message}</p>
    </div>
  );
}
```

6. We will now turn **`PostLists`** into a Client Component and rework data fetching to use TanStack Query. Open **`PostList.tsx`** and make the following highlighted changes:

```
'use client';
import { useQuery } from '@tanstack/react-
query';
...
export function PostList( ... ) {
  const { data: resolvedPosts } = useQuery({
    queryKey: ['posts', criteria],
    queryFn: () => {
      if (typeof criteria === 'string') {
        return getFilteredPosts(criteria);
      }
      return getAllPosts();
    },
  });
  return ...
}
```

Don't forget to remove the **`async`** keyword from the function.

We have set the query key to **`['posts', criteria]`** so that all posts and filtered posts are cached separately. We aliased the fetched data to be **`resolvedPosts`** so that no changes are required in the JSX.

7. Add a loading indicator and error handling to **`PostList`** as follows:

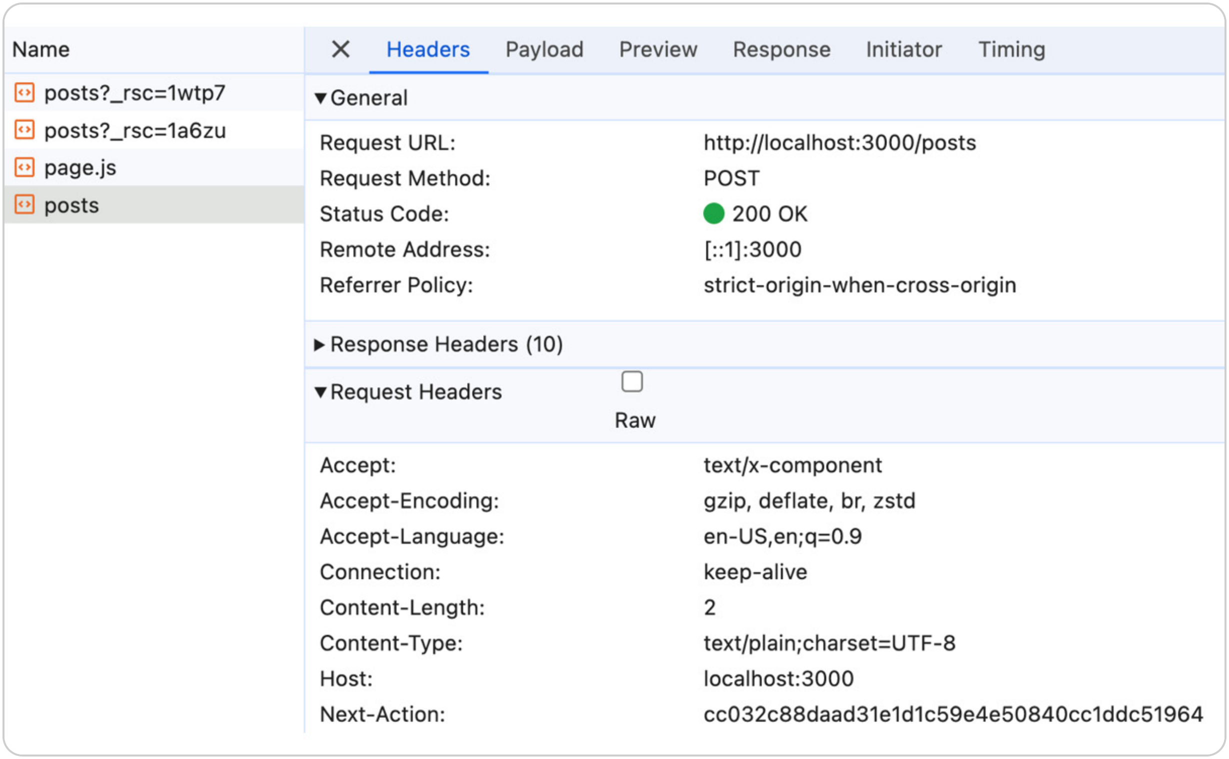
```
import { Loading } from './Loading';
import { ErrorAlert } from './ErrorAlert';
export function PostList( ... ) {
  const {
    ...,
    isPending,
    error,
  } = useQuery( ... );
  if (isPending) {
    return <Loading />;
  }
  if (error) {
    return <ErrorAlert error={error} />;
  }
  return ...
}
```

Open **`src/app/posts/page.tsx`** and remove the React Suspense and error boundaries because data fetching is no longer happening from RSCs. The loading indicator import can also be removed. The JSX in **`Posts`** should now be as follows:

```
<main>
  <h2>{resolvedHeading}</h2>
  <NewPost />
```

```
    <PostList criteria={criteria} />
  </main>
```

8. Run the app by executing **`npm run dev`** in the terminal, if it's not already running. Open the browser developer tools, go to the **Network** panel and navigate to the **Posts** page. The last HTTP request will be the data fetching request.



```
Request URL:        http://localhost:3000/posts
Request Method:     POST
Status Code:        ● 200 OK
Remote Address:     [::1]:3000
Referrer Policy:    strict-origin-when-cross-origin

Accept:             text/x-component
Accept-Encoding:    gzip, deflate, br, zstd
Accept-Language:    en-US,en;q=0.9
Connection:         keep-alive
Content-Length:     2
Content-Type:       text/plain;charset=UTF-8
Host:               localhost:3000
Next-Action:        cc032c88daad31e1d1c59e4e50840cc1ddc51964
```

Figure 8.1 – Client-side data fetching HTTP request

Notice that the request is an HTTP POST rather than a GET request. This is because all Server Functions in Next.js are HTTP POST requests.

> ***NOTE***

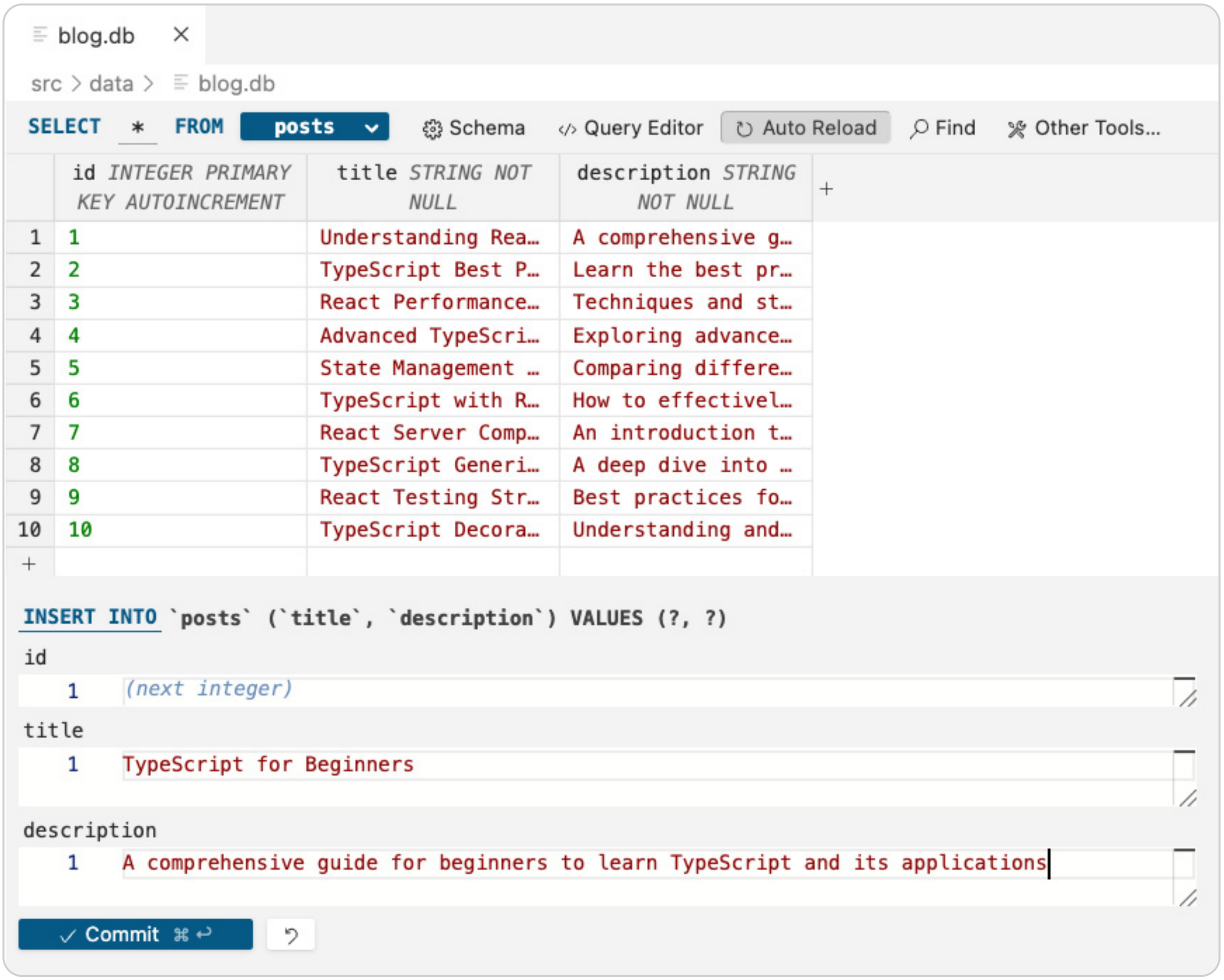
> *You may miss the loading indicator because the data fetching is quick, with everything running locally. You can change the* ***Throttle*** *setting in the browser developer tool* ***Network*** *panel to slow the data fetching process.*

9. We will temporarily cause an error to check that the error handling code is working. Open **`queries.ts`**, and in the **`getAllPosts`** function, change **`posts`** to **`postsX`** in the SQL statement. Refresh the **Posts** page in the running app, and you will see several HTTP requests to get the data that returns status 500 before the error is displayed. This is the default retrying behavior in TanStack Query.

   Revert the change before continuing so that the query is successful again.

10. We will experience another nice default feature in TanStack Query. First, ensure the **Posts** page is active in the app and then switch to a different tab in the browser.

11. Now let's simulate a different user adding a blog post. Insert a record into the **`posts`** table using the SQLite Visual Studio Code extension.



SELECT * FROM posts

	id INTEGER PRIMARY KEY AUTOINCREMENT	title STRING NOT NULL	description STRING NOT NULL
1	1	Understanding Rea…	A comprehensive g…
2	2	TypeScript Best P…	Learn the best pr…
3	3	React Performance…	Techniques and st…
4	4	Advanced TypeScri…	Exploring advance…
5	5	State Management …	Comparing differe…
6	6	TypeScript with R…	How to effectivel…
7	7	React Server Comp…	An introduction t…
8	8	TypeScript Generi…	A deep dive into …
9	9	React Testing Str…	Best practices fo…
10	10	TypeScript Decora…	Understanding and…

```
INSERT INTO `posts` (`title`, `description`) VALUES (?, ?)
```

id

(next integer)

title

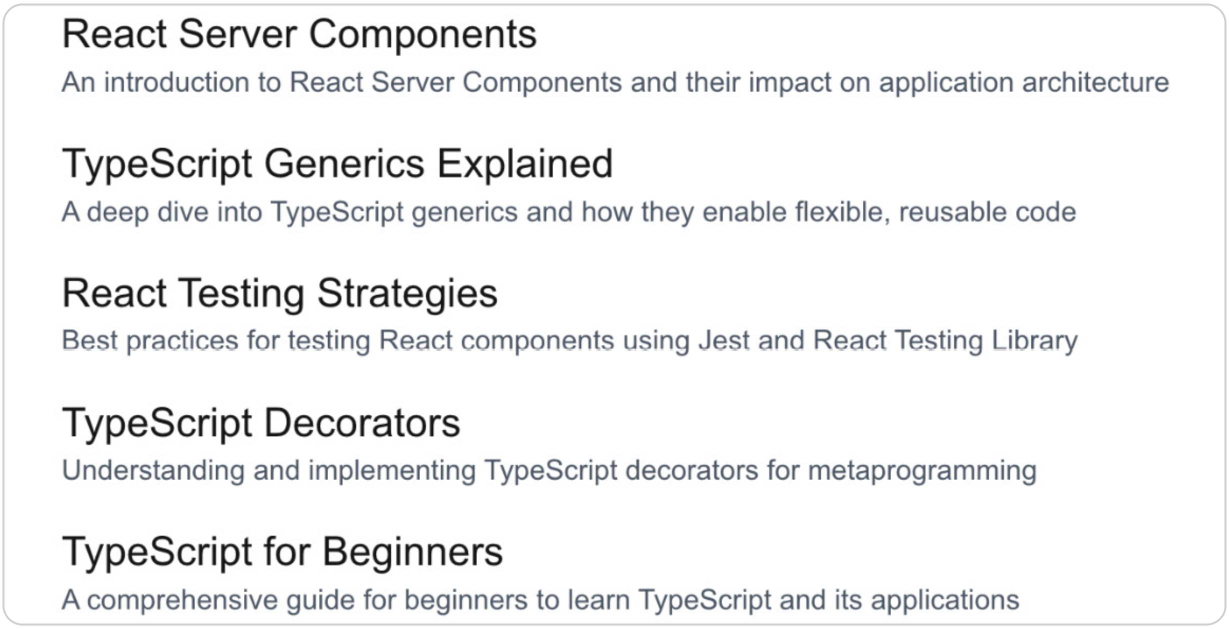
TypeScript for Beginners

description

A comprehensive guide for beginners to learn TypeScript and its applications

Figure 8.2 – Adding a record into the posts table

12. Go back to the app in the browser, and you'll see the new blog post in the list. TanStack Query automatically fetches a fresh copy of the data if the browser window loses and regains focus.


React Server Components

An introduction to React Server Components and their impact on application architecture

TypeScript Generics Explained

A deep dive into TypeScript generics and how they enable flexible, reusable code

React Testing Strategies

Best practices for testing React components using Jest and React Testing Library

TypeScript Decorators

Understanding and implementing TypeScript decorators for metaprogramming

TypeScript for Beginners

A comprehensive guide for beginners to learn TypeScript and its applications

Figure 8.3 – New post automatically added to the bottom of the Posts page

We will now rework the data fetching the **Post** page in our app to use TanStack Query. We will again create a loading indicator and handle errors. Carry out the following steps:

1. We will start by turning `PostDetail` into a Client Component and rework its data fetching to use TanStack Query. Open `PostDetail.tsx` and make the following highlighted changes:

```
'use client';
import { useQuery } from '@tanstack/react-
query';
...
export function PostDetail( ... ) {
  const { data: post } = useQuery({
    queryKey: ['post', id],
    queryFn: () => {
        return getPost(id);
      },
    });
    if (!post) {
      ...
    }
    return ...
  }
```

Don't forget to remove the **async** keyword from the function.

We have set the query key to **['post', id]** so that different posts are cached separately. We aliased the fetched data to be **posts** so that no changes are required in the JSX.

2. Add a loading indicator and error handling to **PostDetail** as follows:

```
  import { Loading } from './Loading';
  import { ErrorAlert } from './ErrorAlert';
  ...
  export function PostDetail( ... ) {
    const { ..., isPending, error } = useQuery(
  ... );
    if (isPending) {
      return <Loading />;
    }
    if (error) {
      return <ErrorAlert error={error} />;
    }
    if (!post) {
      ...
    }
    return ...
  }
```

3. Lastly, open **`src/app/posts/[id]/page.tsx`** and remove the React Suspense, error boundary, and loading indicator because data fetching is no longer happening from RSCs. The JSX in **`Post`** should now be as follows:

```
  <main>
    <PostDetail id={id} />
  </main>
```

4. The **Post** page will now use client-side data fetching in the running app. You can experience the error handling on the page using the same approach as before.

You may have noticed that new blog posts created using the **Create New Post** button don't appear in the list. We'll resolve this later in this chapter when we look at TanStack mutations.

That completes the reworking of data fetching to use TanStack Query. Here's a recap of this section:

- TanStack Query simplifies client-side data fetching with features such as automatic management of loading and error states, caching, stale data refetching, and retrying failed queries.
- To use TanStack Query, **`QueryClientProvider`** is placed high in the React component tree, enabling global data cache access and providing Hooks such as **`useQuery`** for fetching and managing query states.

- TanStack Query default settings mean that erroneous queries are retried and data is refetched when the browser window regains focus.

Next, we'll rework the data fetching to use a Next.js Route Handler.

Using a Route Handler with TanStack Query

In this section, we will learn about Next.js Route Handlers and the benefits of using them for data fetching over Server Functions. We will then rework our app to use Route Handlers.

Understanding the benefits of Route Handlers

A Next.js Route Handler allows an API endpoint to be created. Although the Server Function approach is simple and provides type safety across the network boundary, it has some downsides, such as the following:

- Server Functions run sequentially. So, if a component calls a Server Function to fetch data, and a child component calls another Server Function, the second call must wait for the first to complete. In contrast, Route Handlers run in parallel.
- Server Functions use an HTTP POST rather than a GET, which is commonly used for fetching data. HTTP POST is generally used for mutations.

- The Next.js or React teams don't recommend Server Functions for data fetching.

Now that we understand the reasons to use Route Handlers, we will use them in our app.

Using Route Handlers

We will replace the current use of Server Functions in our app with Route Handlers. Carry out the following steps to create Route Handlers for **`api/posts`** and **`api/posts/[id]`** paths:

1. Start by removing the **`'use server'`** directive at the top of **`queries.ts`** so that its exported functions are no longer available as Server Functions.
2. Route Handlers are defined in the **`src/app/api`** folder. The handlers for different paths are defined in a similar manner to pages with a **`route.ts`** file instead of a **`page.tsx`** file. Create a **`route.ts`** file in the **`src/app/api/posts`** folder and add the following content:

```
import { type NextRequest } from
'next/server';
import {
  getAllPosts,
  getFilteredPosts,
} from '@/data/queries';
export async function GET(request:
NextRequest) {
  const criteria =
    request.nextUrl.searchParams.get('criteria
');
  if (typeof criteria === 'string') {
    return Response.json(
      await getFilteredPosts(criteria),
    );
  }
  return Response.json(await getAllPosts());
}
```

The code handles GET requests to the **`api/posts`** path. It extracts **`criteria`** from the search parameters and returns the data from the **`getFilteredPosts`** query if it is defined. If no criteria are specified, the **`getAllPosts`** query is called for the data to return.

3. Try hitting the API in a browser by entering http://localhost:3000/api/posts as the browser address. The JSON data for all the posts will be returned.

4. Create a **`route.ts`** file in the **`src/app/api/posts/[id]`** folder and add the following content:

```
  import { getPost } from '@/data/queries';
  export async function GET(
    _: Request,
    { params }: { params: Promise<{ id: string
  }> }
  ) {
    const id = Number((await params).id);
    if (!Number.isInteger(id)) {
      return Response.json(
        { message: 'Post not found' }, { status:
```

```
404 }
      );
    }
    const data = await getPost(id);
    if (!data) {
      return Response.json(
        { message: 'Post not found' }, { status:
404 }
      );
    }
    return Response.json(data);
  }
```

The code handles GET requests to **`api/posts/[id]`** paths. It extracts the **`id`** route parameter and returns the data from the **`getPost`** query. If **`id`** isn't numeric or no rows are returned from the query, HTTP status code 404 is returned.

5. Try hitting the API in a browser by entering http://localhost:3000/api/posts/1 as the browser address. The JSON data for post 1 will be returned.

6. We will now integrate the Route Handlers into the **`PostList`** and **`PostDetail`** components. Start with **`PostList`** by opening **`PostList.tsx`** and remove the imported **`getAllPosts`** and **`getFilteredPosts`** queries. Lastly, update **`useQuery`** to use the Route Handler as follows:

```
const { ... } = useQuery({
  queryKey: ...,
  queryFn: async () => {
      const path =
          typeof criteria === 'string'
            ? `/api/posts/?criteria=${
                encodeURIComponent(criteria)}`
            : '/api/posts/';
        const response = await fetch(path);
        if (!response.ok) {
          throw new Error('Problem fetching
  data');
        }
        return await response.json();
      },
  });
```

We call the browser **fetch** function to make an HTTP request to the relevant path depending on whether there is a **criteria** search parameter. If the response returns an HTTP error status code, we throw an error to put the query in an error state.

7. You'll notice a type error in the **resolvedPosts** variable **map** function because it can only be inferred to have the **any** type. We will resolve this error later in this chapter. However, the code will function fine if we navigate to the **Posts** page in the running app.

8. We will rework the **PostDetail** component now. Open **PostDetail.tsx** and remove the imported **getPost** function. Lastly, update **useQuery** to use the Route Handler as follows:

```
  const { ... } = useQuery({
      queryKey: ...,
      queryFn: async () => {
        const response = await fetch(
          `/api/posts/${id}`,
        );
        if (!response.ok) {
            throw new Error(
            response.status === 404
              ? "Blog post not found"
              : "Problem fetching data",
          );
        }
        return await response.json();
      },
    });
```

We call the browser **`fetch`** function to make an HTTP request to the relevant path using the **`id`** route parameter. If the response returns an HTTP error status code, we throw an error to put the query in an error state. We specifically catch the **`404`** status code so that we can throw an appropriate error message.

9. Again, the data from TanStack Query is inferred to have the **`any`** type, which isn't ideal. However, the code will function fine if we navigate to **Post** page in the running app.

That completes the rework to switch to using Next.js Route Handlers. For more information on Route Handlers, see the following page: https://nextjs.org/docs/app/building-your-application/routing/route-handlers.

Next, we will resolve the type safety issue with the TanStack Query data.

Adding type safety to the API response

We could use a TypeScript type assertion to make the data fetching code a little more type-safe as follows:

```
return response.json() as Promise<Posts>
```

However, we are assuming the schema of the API response body. This is fine if we control the API, like in our app. However, what if a separate team develops that API? What if it's a third-party API that a different company controls? This assumption can be risky.

Instead of a type assertion, we will use Zod to validate the schema of the API response body – as we did for the SQL query results earlier in this chapter. We will use the same Zod schema as we did for the SQL query validation because the schema is unchanged after it is returned from the query. Carry out the following steps:

1. Open **PostList.tsx** and add a Zod schema check as follows:

```
import { postsSchema } from '@/data/schema';
export function PostList( ... ) {
  const { ... } = useQuery({
    ...,
    queryFn: async () => {
      ...
      return postsSchema.parse(
        await response.json(),
      );
    },
  });
```

```
    ...
  }
```

The type error is now resolved, and if you hover over the **`resolvedPosts`** variable, its type will now be correctly inferred.

2. Open **`PostDetail.tsx`** and add a Zod schema check as follows:

```
  import { postSchema } from '@/data/schema';
  export function PostDetail( ... ) {
    const { ... } = useQuery({
      ...,
      queryFn: async () => {
        ...
        return postSchema.parse(
          await response.json(),
        );
      },
    });
    ...
  }
```

If you hover over the **`posts`** variable, its type will now be correctly inferred.

That's the type-safety improvement complete, and the end of this section on using Route Handlers. Here's a recap:

- Route Handlers are generally preferred to Server Functions for data fetching. This is because they improve performance by allowing parallel API requests and using HTTP GET requests.

- One disadvantage of using Route Handlers for data fetching is the lack of type safety across the network boundary. Zod can be used to bridge this gap with a schema that validates the API response body.

Next, we will use TanStack Query for mutations.

Mutating data using a TanStack Query mutation

In this section, we will understand and then use a TanStack Query mutation in the **`NewPost`** component. We will also resolve the problem with new posts not appearing in the **Posts** page.

Understanding TanStack Query mutations

The TanStack Query **`useMutation`** Hook manages the mutation process, including helpful state variables. It also provides a mechanism to update the cache after a mutation. Here's an example:

```
const { mutate, isPending, isError } =
useMutation({
  mutationFn: (newProduct) =>
createProduct(newProduct),
  onSuccess: async () => {
    await queryClient.invalidateQueries({
      queryKey: [‘products’],
    }),
  },
});
async function handleClick() {
  mutate({
    name: 'New product',
    description: 'New product description',
  });
}
if (isPending) return 'Mutating... ';
if (isError) return 'An unexpected error
occurred';
```

The **`useMutation`** Hook takes in an options object as an argument. Here's a description of the options used in this example:

- **`mutationFn`**: The function that calls the server to do the mutation. In this example, we call a Server Function called **`createProduct`**.
- **`onSuccess`**: A function to call when the mutation is successful. In this example, we invalidate the **`products`** cache.

The **`useMutation`** Hook returns an object. Here's a description of the object members used in this example:

- **`mutate`**: This is the function that starts the mutation process. In this example, this function is used in a click handler.
- **`isPending`**: This indicates whether the mutation is currently executing. We use it in this example to render a mutating indicator.
- **`isError`**: This indicates the mutation has errored. We use it in this example to render an error alert.

Next, we will use **`useMutation`** in our app.

Using useMutation

We will simplify the state management in the **`NewPost`** component by using **`useMutation`**. We will also invalidate the **`posts`** cache after the mutation has been successful. Carry out the following steps:

1. Open **`NewPost.tsx`** and start by removing the existing **`isMutating`** and **`status`** **`useState`** calls. The React **`useState`** import can also be removed.
2. Import the **`useMutation`** Hook and call it as follows:

```
import { useMutation } from '@tanstack/react-
query';
...
export function NewPost() {
  const {
    mutate,
    isPending,
    isError,
    isSuccess
  } = useMutation({
    mutationFn: ({
      title,
      description,
    }: {
      title: string;
      description: string;
    }) => createPost(title, description),
  });
  async function handleClick() {
    ...
  }
  return ...
}
```

The mutation function calls the **`createPost`** Server Function, passing in the title and description for the new blog post. We have destructured the **`mutate`** function as well as the **`isPending`**, **`isError`**, and **`isSuccess`** state variables.

> **NOTE**
>
> *We could have used a Route Handler for the mutation instead of a Server Function. However, Server Functions are simpler to implement and give more type safety.*

3. Call **`mutate`** in the click handler and remove the old state variable references as well. Also, the function no longer needs to be marked with the **`async`** keyword. The click handler should now be as follows:

```
  function handleClick() {
    mutate({
      title: 'New Post',
      description: 'New Post Description',
    });
  }
```

4. In the JSX, replace **`isMutating`** with **`isPending`**. Also, replace the conditionals on the old status variable with **`isError`** and **`isSuccess`**:

```
  <div ... >
    <button ... >
      {isPending ? 'Creating...' : 'Create New
  Post'}
    </button>
    {isError && (
      <span ... >An unexpected error
  occurred</span>
    )}
    {isSuccess && (
      <span ... >Post successfully
  created</span>
    )}
  </div>
```

5. If you try the **Create New Post** button in the running app, it'll behave as it did before. This means the new post still doesn't appear in the list after the mutation has been successfully completed.

6. Let's resolve this problem by invalidating the cache in the **`onSuccess`** function option as follows:

```
import {
  ...,
  useQueryClient,
} from '@tanstack/react-query';
...
export function NewPost() {
  const queryClient = useQueryClient();
  const { ... } = useMutation({
    mutationFn: ...,
    onSuccess: async () => {
      queryClient.invalidateQueries({
        queryKey: ['posts'],
      });
    },
    });
    ...
  }
```

   We use the **`useQueryClient`** Hook to access **`queryClient`** so that we can call its **`invalidateQueries`** function. We pass the **`invalidateQueries`** function the key we want to invalidate.

7. If you try the **Create New Post** button in the running app, the new post will now appear in the list after the mutation has been successfully completed.

That completes this section on mutations with TanStack Query. Here's a quick recap:

- The **`useMutation`** Hook from TanStack Query simplifies state management by handling mutations, providing helper state variables such as **`isPending`**, **`isSuccess`**, and **`isError`**.
- By calling **`QueryClient.invalidateQueries`** with the appropriate query key in the **`onSuccess`** function, data in the cache impacted by the mutation can be invalidated.

> ***NOTE***
>
> *For more information on TanStack Query, see its documentation at https://tanstack.com/query/latest.*

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

Summary

In this chapter, we started by exploring the use of the `useEffect` Hook for client-side data fetching and how challenging writing robust code is. We quickly switched to using TanStack Query and experienced how it simplified the code and enabled features like the retrying of a data fetch when it errors. We experienced how it automatically refreshed data when our app's browser tab regained focus – something not possible with server-side data fetching.

We learned that using a Server Function has some downsides for client-side data fetching, such as using an HTTP POST request rather than an HTTP GET. We switched to using a Next.js Route Handler because of the downsides.

We learned that Server Functions are ideal for mutating data because of their simplicity and type safety. We used a Server Function in our app to add a new blog post to our database. We used TanStack Query with a Server Function for mutating data and learned that the TanStack Query cache needs to be invalidated after a successful mutation.

You now have the skills to implement robust client-side data fetching and mutations in a very maintainable fashion.

In the next chapter, we will learn more about mutations as we explore forms in depth.

Questions

Answer the following questions to check what you have learned in this chapter:

1. What is the problem with the following data fetching code?

```
useEffect(() => {
  fetch("/api/data")
    .then((res) => res.json())
    .then((data) => setData(data));
}, []);
```

2. Why isn't it recommended to use a Server Function for data fetching in Next.js?

3. What will be the output of the following Zod schema validation?

```
const userSchema = z.object({
  name: z.string(),
  age: z.number().min(18),
});
const result = userSchema.safeParse({
  name: "Alice",
  age: 16,
});
console.log(result.success);
```

4. What does the following mutation do when the form is submitted?

```
const mutation = useMutation({
  mutationFn: (newUser) =>
    fetch("/api/users", {
      method: "POST",
      body: JSON.stringify(newUser),
    }),
  onSuccess: () => {
    console.log("User created!");
  },
});
<form
  onSubmit={(e) => {
    e.preventDefault();
    mutation.mutate({ name: "Jane" });
  }}
>
  <button type="submit">Create User</button>
</form>;
```

5. What does the **`staleTime`** option do in the following code?

```
useQuery(["todos"], fetchTodos, { staleTime: 
10000 });
```

Answers

1. If the component unmounts before the fetch finishes, it may try to update the state, causing a warning. An **`AbortController`** object that is aborted when the component unmounts can be used to fix the problem:

```
useEffect(() => {
  const controller = new AbortController();
  fetch("/api/data", { signal: 
controller.signal })
    .then((res) => res.json())
    .then((data) => setData(data))
```

```
      .catch((err) => {
        if (err.name !== "AbortError") {
          console.error(err);
        }
      });
    return () => controller.abort();
  }, []);
```

2. Server Functions run in sequence rather than in parallel, which can cause a performance problem when multiple Server Functions are called. They also use an HTTP POST rather than an HTTP GET.
3. The validation will fail because the schema requires **`age`** to be a number greater than or equal to **`18`**, but **`16`** is provided. So, **`safeParse`** will return an object with a **`success`** property set to **`false`**. Therefore, the output of the **`console.log`** statement will be **`false`**.
4. It sends an HTTP POST request and outputs **User Created!** to the console when successful.
5. The **`staleTime`** option is the number of milliseconds that a query's data is considered fresh. During this time, TanStack Query won't refetch the data automatically in the background, even if the component remounts or the window refocuses. So, in the code snippet, the data fetched will be considered fresh for ten seconds.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases

– follow the QR code below:

https://packt.link/GxSkC

9
Working with Forms

Forms are extremely common in apps, so it's essential to be able to implement them efficiently. In this chapter, we'll build a contact form that you often see on company websites in Next.js. It will contain a handful of fields and some validation logic.

We will start with a basic form implementation using native browser capabilities. We'll revise the form submission to use a Route Handler and then a React Server Action introducing server validation. We'll then implement a submission indicator and better error handling using React form status and action state. We'll use a popular library called React Hook Form to add client-side validation. Lastly, we'll implement optimistic updates for an even better user experience.

By the end of this chapter, you'll be able to build user-friendly and robust forms in React.

We'll cover the following topics:

- Using basic forms
- Using a Route Handler for submission
- Using a Server Action for submission

- Using **`useFormStatus`**
- Using **`useActionState`**
- Using React Hook Form
- Implementing optimistic updates

Technical requirements

We will use the following tools 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-Third-Edition/tree/main/Chapter09.

Using basic forms

In this section, we will create a Next.js project and the first iteration of the contact form. We will also create a database that will eventually hold the form data.

Creating the project

Let's start by creating a Next.js project and a SQLite database. Carry out the following steps:

1. In a terminal, execute the following command to create the project:

   ```
   npx create-next-app@latest forms --ts --eslint
   --app --src-dir --import-alias "@/*" --no-
   tailwind --no-turbopack
   ```

2. Still in the terminal, move to the project folder and open Visual Studio Code using the following commands:

   ```
   cd forms
   code .
   ```

3. Prettier can be set up in the same manner as we learned with Vite in *Chapter 1*, *Getting Started with React*. Feel free to add automatic code formatting to this project.

4. Install the libSQL dependency by running the following command in a terminal:

   ```
   npm i @libsql/client
   ```

5. Create a script that we'll eventually run to create our database. Create a folder called **`scripts`** in the **`src`** folder and then a file called **`createDatabase.mjs`** in this folder. Copy the script from the GitHub repository at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-

Edition/blob/main/Chapter09/start/src/scripts/createDatabase.mjs and paste it into the file.

6. Create a **.env** file in the project root and add the URL to the database into it:

   ```
   DB_URL=file:src/data/forms.db
   ```

7. Create a folder in **src** called **data** for the location of the database. To create the database, run the following command in the terminal:

   ```
   node src/scripts/createDatabase.mjs
   ```

   This command executes the script using Node.js. After the command has completed, a **forms.db** database file will appear in the **data** folder.

8. Open **src/app/global.css** and overwrite the content with the CSS from the following file in the GitHub repository:

   https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter09/start/src/app/globals.css.

   This will nicely style our app.

9. Let's clean up the home page. Open **page.tsx** in the **src/app** folder and replace its content with the following:

   ```
   export default function Home() {
     return (
       <main>
   ```

```
    </main>
  );
}
```

10. In the terminal, execute the following command to run the app in development mode:

    ```
    npm run dev
    ```

    The app will be available in a browser at http://localhost:3000 and will show a blank screen.

That completes the project setup. Next, we will create a basic contact form in the app.

Creating a native form

Carry out the following steps to create a native HTML contact form:

1. Open **src/app/page.tsx** and add the following elements inside the **main** element:

    ```
    import {
      ContactForm
    } from '@/components/ContactForm';
    export default function Home() {
      return (
        <main>
          <h2>Contact Us</h2>
          <p>
            If you enter your details we will get
    ```

```
back
        to you as soon as we can.
      </p>
      <ContactForm />
    </main>
  );
}
```

The page displays a title, some instructions, and a **ContactForm** component.

The **ContactForm** component hasn't been implemented yet, so a compile error occurs. We'll create this in the next step.

2. Create a folder called **components** in the **src** folder and a file called **ContactForm.tsx** in it. Add the following content to the file:

```
export function ContactForm() {
  return <form></form>;
}
```

3. Add the following fields inside the **form** element:

```
<form>
  <div className="field">
    <label htmlFor="name">Your name</label>
    <input type="text" id="name" name="name"
/>
  </div>
  <div className="field">
    <label htmlFor="email">Your email
address</label>
    <input type="email" id="email"
name="email" />
```

```
    </div>
    <div className="field">
      <label htmlFor="reason">Reason you need to
  contact us     </label>
      <select id="reason" name="reason">
        <option value=""></option>
        <option value="Support">Support</option>
        <option
  value="Feedback">Feedback</option>
        <option value="Other">Other</option>
      </select>
    </div>
    <div className="field">
      <label htmlFor="notes">Additional
  notes</label>
      <textarea id="notes" name="notes" />
    </div>
  </form>
```

NOTE

A reminder that the code can be copied from *https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter09/basic-form/src/components/ContactForm.tsx*.

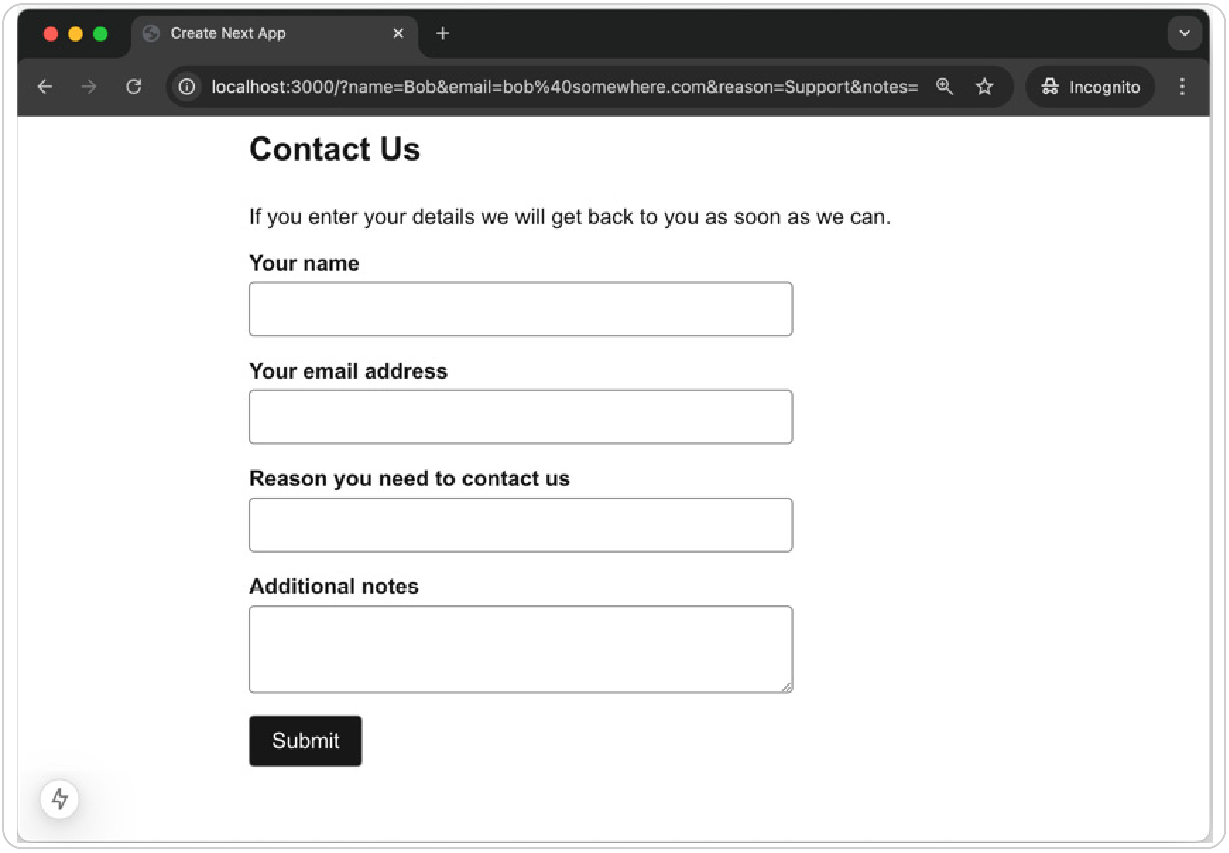
We have added fields for the user's name, email address, contact reason, and additional notes. Each field label is associated with its editor by setting the **htmlFor** attribute to the editor's **id** value. This helps assistive technology such as screen readers read out labels when fields gain focus.

The **name** attribute on the field editors allows field values to be extracted in the form submission process.

4. Add a **Submit** button to the bottom of the **form** element as follows:

```
<form>
  ...
  <button type="submit">Submit</button>
</form>
```

5. That completes our basic form. In the running app, fill the form in and submit it. You'll see the values added to the URL as search parameters. You may also notice a full page reload occurs.


Create Next App
localhost:3000/?name=Bob&email=bob%40somewhere.com&reason=Support&notes=
Incognito

Contact Us

If you enter your details we will get back to you as soon as we can.

Your name

Your email address

Reason you need to contact us

Additional notes

Figure 9.1 – Submitted form

6. After the form is submitted, there is no indication that the process was successful for the user. Add the following **`action`** attribute to the **`form`** element to navigate to a Thanks page after the form is submitted:

```
<form action="thanks">
```

7. Let's create the Thanks page by creating a new **`thanks`** folder in the **`src/app`** folder. Create a **`page.tsx`** file within this folder with the following content:

```
export default async function Thanks({
  searchParams,
}: {
  searchParams: Promise<{
    [key: string]: string | string[] |
undefined;
  }>;
}) {
  return (
    <main>
      <h2>Form successfully submitted</h2>
      Thank you {(await searchParams).name},
we will be in touch soon.
    </main>
  );
}
```

The component informs the user that the form has been successfully submitted and thanks them. Their name is obtained from the **`name`** search parameter.

8. Fill out the form again and submit it. You'll be navigated to the **Thanks** page. We are still getting a full page reload though – we'll address this in the next step.
9. Back in **`ContactForm.tsx`**, we can use the **`Form`** component from Next.js to prevent the full page reload, as follows:

```
import Form from 'next/form';
export function ContactForm() {
  return (
    <Form ... >
      ...
    </Form>
  );
}
```

10. Fill out the form once again and submit it. A full page reload no longer occurs when navigating to the **Thanks** page.

That completes our basic form, which largely uses native HTML features. We used the **`Form`** component from Next.js to optimize the form submission performance.

Next, we will integrate the form with a Next.js Route Handler.

Using a Route Handler for submission

In this section, we will change the form to submit the data to a web API. We will use a Route Handler to create the API, which we learned about in *Chapter 8*, *Client Component Data Fetching and Mutations with TanStack Query*. Submitting data to APIs has been common practice for many years because it prevents a full page reload.

Creating a database mutation

We will create a function that adds the form data to our database. This will eventually be used in the Route Handler. Carry out the following steps:

1. First, we will create a type to represent the data. Create a file called **`schema.ts`** in the **`data`** folder with the following content:

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

2. Create a new file called **`insertContact.ts`** in the **`src/data`** folder containing the following content:

   ```
   import {
     createClient,
     type Client,
   } from '@libsql/client';
   import { Contact } from './schema';
   export async function insertContact({
     name,
   ```

```
    email,
    reason,
    notes,
  }: Contact) {
    let client: Client | undefined;
    let ok = true;
  }
```

The file imports the createClient function and Client type needed from the libSQL package as well as the Contact type we just created. The insertContact function implementation has been started by defining a parameter for the form data. Variables for the database connection and whether the database mutation is successful have also been declared.

3. After the variable declarations, connect to the database and execute SQL to insert the record into the **`contact`** table:

```
export async function insertContact( ... ) {
  ...
  try {
    client = createClient({
      url: process.env.DB_URL ?? '',
    });
    await client.execute({
      sql: 'INSERT INTO contact(name, email,
reason, notes)             VALUES (?, ?, ?,
?)',
      args: [name, email, reason, notes],
    });
  } catch {
    ok = false;
    }
  }
```

4. Finish the **`insertContact`** implementation by ensuring the database connection is closed and the success of the operation is returned. The following is the updated code:

```
  export async function insertContact( ... ) {
    ...
    if (client) {
      client.close();
    }
    return {
      ok
    };
  }
```

That's the function to mutate the database completed. Next, we'll implement the Route Handler.

Creating a Route Handler

Carry out the following steps to create the Route Handler:

1. Create an **`api`** folder in the **`src/app`** folder and then a **`route.ts`** file inside it, so the Route Handler will handle HTTP requests to the **`/api`** path.

2. Add the following code to **`route.ts`**:

```
  import { type NextRequest } from
  'next/server';
```

```
import { insertContact } from
'@/data/insertContact';
export async function POST(request:
NextRequest) {
  const data = await request.json();
  const result = await insertContact(data);
  if (result.ok) {
    return Response.json(
      {}, { status: 201 },
    );
  }
  return Response.json({}, { status: 500 });
}
```

The Route Handler handles HTTP `POST` requests. The handler gets the form data from the request body and calls the `insertContact` database mutation, passing in the form data. If the mutation is successful, the HTTP status code **Created** (status code `201`) is returned. If the mutation errored, the HTTP status code **Internal Server Error** (status code `500`) is returned.

That completes the Route Handler. Next, we will integrate this Route Handler into our form.

Integrating the form submission with the Route Handler

To integrate the Route Handler into the form, we will write client-side code for the form submission handler, replacing the form action. We will also remove the use of the Next.js **`Form`** component, as this is no longer required. Open **`ContactForm.tsx`** and carry out the following steps:

1. Remove the **`Form`** import from Next.js and replace the **`Form`** component in the JSX with a **`form`** element.
2. Make the component a Client Component by adding **`'use client'`** at the top of the file. This is because we are going to implement an event handler.
3. Change the **`action`** attribute on the **`form`** element to reference an action handler, as follows:

   ```
   export function ContactForm() {
     async function handleAction( formData:
   FormData) {}
     return (
       <form action={handleAction}>
       </form>
     );
   }
   ```

   The **`handleAction`** function will now be invoked when the form is submitted. It is now our responsibility to submit the form data to the server.

   The **`handleAction`** function parameter is of the **`FormData`** type. **`FormData`** is a native browser interface that allows access to values in a form and takes in a **`form`** element in its constructor parameter. For more information on **`FormData`**, see https://developer.mozilla.org/en-US/docs/Web/API/FormData.

4. Add the following lines to the action handler to get the data into an object compatible with the Route Handler:

```
import { Contact } from '@/data/schema';
...
export function ContactForm() {
  async function handleAction( ... ) {
    const contact = Object.fromEntries(
      formData,
    ) as Contact;
  }
}
```

   We use the **`Object.fromEntries`** method to turn **`formData`** into a simple object with properties containing the field values.

   We also use a type assertion to type the **`contact`** object, which isn't ideal. It isn't ideal because it doesn't actually verify that **`formData`** contains the properties and value types that **`Contact`** should have. We'll improve this later in this chapter.

5. On the next lines in the submit handler, make an HTTP request to the Route Handler's API, and if it errors, output a console error:

```
async function handleAction( ... ) {
  ...
  const response = await fetch('api', {
    method: 'POST',
    body: JSON.stringify(contact),
  });
  if (!response.ok) {
    console.error('Something went wrong');
    return;
  }
}
```

6. The final step in the submission handler is to navigate to the **Thanks** page. We can perform client-side navigation using the Next.js **useRouter** Hook, as follows:

```
  import { useRouter } from 'next/navigation';
  ...
  export function ContactForm() {
    const { push } = useRouter();
    async function handleAction( ... ) {
      ...
      push(
        '/thanks/?name=' +
        encodeURIComponent(contact.name),
      );
    }
    return ...
  }
```

   We use the **push** function from the **useRouter** Hook to navigate, passing in the path to the **Thanks** page. We include **name** as a search parameter using **encodeURIComponent** to escape special characters that the user may have entered in their name.

7. In the running app, fill out the form and submit it. You can verify the data was saved into the database using the Visual Studio Code SQLite

extension we used in Chapter 8, *Client Component Data Fetching and Mutations with TanStack Query*.

That completes the integration of the form with a Route Handler. Here's a quick recap:

- A full page reload can be avoided by handling the **`form`** element's action event
- The action can submit the form data via a web API, which can be implemented using a Next.js Route Handler
- The Next.js **`useRouter`** Hook can be used to navigate to a submission success page

There is another, more modern way of submitting forms. We'll cover this next.

Using a Server Action for submission

In this section, we'll understand what a Server Action is. We'll then use a Server Action in our form.

Understanding Server Actions

In Chapter 7, *Server Component Data Fetching and Server Function Mutations*, we learned how to mutate data with Server Functions. We can use a Server Function for form submission. In fact, Server

Functions used for form submission are so common, they have a special name called **Server Actions**.

A Server Action can be passed to a `form` element in its `action` prop as follows:

```
<form action={serverAction}>
```

Earlier in this chapter, we used the `action` prop for the page path to submit to. We then used the `action` prop to implement a client-side submission handler. Here, we are submitting directly to a Server Action. So, the Server Action will need to contain any required page navigation.

A benefit of using a React Server Action for form submission is that it works without JavaScript. This is handy for web apps used on mobile phones in places where the network connection is slow. In this situation, the form may be available for the user to fill in and submit before it has been hydrated with JavaScript.

A Server Action must have a `function` parameter of the `FormData` type and shouldn't return anything:

```
export async function serverAction(formData:
FormData) {
  // Save the data
  // Don't return anything
}
```

Next, we will revise our form to use a Server Action.

Using a Server Action in ContactForm

We will turn **`insertContact`** into a Server Function and use it as a Server Action from our form. We will also add some server-side validation to the form. Carry out the following steps:

1. Start by deleting the Route Handler at **`src/app/api/route.ts`** because this will no longer be used in this chapter, and the code will break with the changes we will make to the **`insertContact`** function.
2. Open **`insertContact.ts`** and add a **`'use server'`** directive at the top of the file. This marks the **`insertContact`** function as a Server Function.
3. Adjust the parameter of the **`insertContact`** function to take in **`FormData`** and then extract the field values from it:

   ```
   export async function insertContact(
    formData: FormData
   ) {
     const { name, email, reason, notes } =
       Object.fromEntries(formData) as Contact;
     let client: Client | undefined;
     ...
   }
   ```

   So, we are interacting with **`FormData`** on the server now, instead of the client.

4. We also need to remove the **`return`** statement in the **`insertContact`** function.

5. We need to perform navigation on the server now. We can use the **`redirect`** function from Next.js to do this. Add a call to **`redirect`** at the bottom of the **`insertContact`** function if the mutation is successful:

```
import { redirect } from 'next/navigation';
...
export async function insertContact( ... ) {
  ...
  if (client) {
    client.close();
  }
  if (ok) {
    redirect(
      `/thanks/?
name=${encodeURIComponent(name)}`,
    );
  }
}
```

6. Open **`ContactForm.tsx`**, remove the use of the **`useRouter`** Hook, and set the **`action`** attribute on the **`form`** element to the **`insertContact`** Server Function. The **`handleAction`** function and importing of the **`Contact`** type can now be removed. The file should now be as follows:

```
'use client';
import { insertContact } from
'@/data/insertContact';
```

```
export function ContactForm() {
  return (
    <form action={insertContact}>
      ...
    </form>
  )
}
```

The `'use client'` directive could also be removed because there is no event handler now. However, we'll leave this in place because we'll use React Hooks in this component in the next section.

In the running app, if you fill in and submit the form, it'll behave as before.

7. Try turning off JavaScript in the browser and fill in and submit the form again. JavaScript can be turned off in Chrome by opening the command window and choosing the **Disable Javascript** command. The command window can be opened by pressing *Shift* + *Command* + *P* on a Mac and *Shift* + *Ctrl* + *P* on Windows.

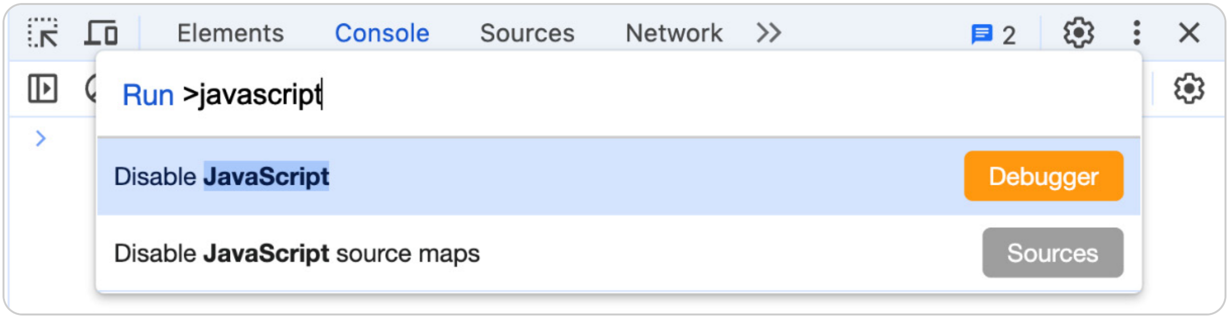


Figure 9.2 – Disabling JavaScript in Chrome

You will find the form is submitted and saved into the database fine.

Re-enable JavaScript in the browser before continuing using the **Enable JavaScript** command in the command window.

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

That completes the reworking of the form to use a Server Action. The implementation is a lot simpler than the previous Route Handler approach.

At the moment, there is no validation on our form. For example, we can submit it without entering any information. We'll address this next.

Adding server validation

Server-side validation not only helps users fill in forms correctly, but it also helps protect against security attacks such as SQL injection and malformed requests. This is because client-side input can be tampered with, bypassed, or disabled entirely by malicious users.

We will add server-side validation using the Zod library. We used Zod in *Chapter 8*, *Client Component Data Fetching and Mutations with TanStack Query*, to validate database and web API data. We can use it to validate form data as well:

1. Install Zod into the project by running the following command in a terminal:

```
npm i zod
```

2. Open up the **`schema.ts`** file. Replace the **`Contact`** type with the following Zod schema:

```
import { z } from 'zod';
export const contactSchema = z.object({
  name: z
    .string()
    .min(1, {
      message: 'You must enter your name',
    })
    .max(50, {
      message:
        'The name must be less than 50
characters',
    }),
  email: z.string().email({
    message:
      'You must enter a valid email address',
  }),
  reason: z.string().min(1, {
    message: 'You must enter a reason',
  }),
  notes: z.string().optional(),
});
```

The schema defines the fields in our form as strings. All the fields are defined as being required, apart from **`notes`**. The **`name`** field will take a maximum of 50 characters, and the **`email`** field must be a valid email format.

3. In **insertContact.ts**, remove the **Contact** import statement and the type assertion that uses **Contact** in the **insertContact** function.

4. In the **insertContact** function, validate the field values as follows:

```
import { contactSchema } from './schema';
...
export async function insertContact( ... ) {
  const parsedResult =
contactSchema.safeParse(
    Object.fromEntries(formData),
  );
  if (!parsedResult.success) {
    return;
  }
  const { name, email, reason, notes } =
    parsedResult.data;
  ...
}
```

We call the schema **safeParse** function so that an error isn't automatically thrown if the form is invalid. This is because we'll eventually handle errors better. At the moment, if the form is invalid, we short-circuit the function and just return nothing.

We then extract the field values from the **data** property from the **safeParse** return object.

5. We now get a type error on the SQL command because the database doesn't accept **undefined** into the **notes** field. To resolve this, coalesce **undefined** to **null** on the **notes** argument as follows:

```
await client.execute({
  sql: ...,
  args: [name, email, reason, notes ?? null],
});
```

6. Run the app by entering the **`npm run dev`** command in a terminal. If you don't fill in the form correctly and submit it, the data won't be saved into the database. However, no error is shown to the user. Also, we are making the user re-enter the whole form. We will resolve these issues later in the chapter.

That completes the implementation of server validation and also this section on using Server Actions for form submission. Here's a quick recap:

- Server Actions streamline form submission by enabling server-side handling of form data, removing the need for client-side submit handlers. These can function without JavaScript, making them a robust solution for low-connectivity scenarios.
- The Server Action must have a parameter of the **`FormData`** type, which will contain the field values.
- The **`action`** attribute on the **`form`** element is set to the Server Action for the form to submit to it.
- Zod is commonly used to validate form data before being saved to a database.

Although we’ve enhanced the form by adding server validation, the user experience still isn’t great. We will improve this in the next section.

Using useFormStatus

In this section, we will understand how the React **`useFormStatus`** Hook can improve the user experience and then use it within our form.

Understanding useFormStatus

The **`useFormStatus`** Hook allows access to information from a form submission in a Client Component. Its main uses are to implement a submission indicator and to disable certain form elements, such as the **Submit** button, during submission. Unlike other React Hooks, this one comes from the React DOM package.

The syntax for **`useFormStatus`** is as follows:

```
const { pending, data } = useFormStatus();
```

The Hook returns an object containing the following properties:

- **`pending`**: Whether form submission is in progress.
- **`data`**: Gives access to field values for the form being submitted in a **`FormData`** structure. **`data`** is **`null`** if the submission is not in

progress.

A restriction of the **`useFormStatus`** Hook is that it must be called in a child component of the component containing a **`form`** element. So, the following code won't work:

```
function Form() {
  const { pending } = useFormStatus();
  return (
    <form>
      ...
      {pending && <p>Submitting ...</p>}
      <button type="submit" disabled={pending}>
        Submit
      </button>
    </form>
  );
}
```

Instead, the use of **`useFormStatus`** must be extracted and placed as a child:

```
function Form() {
  return (
    <form>
      ...
      <SubmitButtion />
    </form>
  );
}
function SubmitButtion {
  const { pending } = useFormStatus();
  return (
    <>
      {pending && <p>Submitting ...</p>}
      <button type="submit" disabled={pending}>
        Submit
      </button>
    </>
  );
}
```

Now that we understand how to use **`useFormStatus`**, we'll use this Hook in our form.

Using useFormStatus

We will add a submission indicator to our form using **`useFormStatus`**. Open **`ContactForm.tsx`** and carry out the following steps:

1. Import **`useFormStatus`** from React DOM:

   ```
   import { useFormStatus } from 'react-dom';
   ```

2. Create a new component at the bottom of **`ContactForm.tsx`**, as follows:

   ```
   function SubmitButton() {
     const { pending } = useFormStatus();
     return (
       <>
         {pending && <p role="alert">Saving ...
   </p>}
         <button type="submit" disabled=
   {pending}>
   ```

```
          Submit
        </button>
      </>
    );
  }
```

The component contains a submission indicator and a **Submit** button. We use **`pending`** from **`useFormStatus`** to show the submission indicator and disable the **Submit** button during form submission.

3. In the **`ContactForm`** component, replace the current **Submit** button with our enhanced version:

```
  export function ContactForm() {
    return (
      <form ... >
        ...
        <SubmitButton />
      </form>
    );
  }
```

4. Try filling in the form in the running app. To see the submission indicator, throttle the network using the browser development tools. The submission process will then be slow enough for you to see the submission indicator.

That completes the implementation of the submission indicator and this section on **`useFormStatus`**. Here's a quick recap:

- The **`useFormStatus`** Hook can be used for implementing submission indicators in a form
- A downside of **`useFormStatus`** is that it must be in a child component of the **`form`** element

There is an alternative Hook we can use to implement a form submission indicator, which we will learn about in the next section.

Using useActionState

Our form still doesn't show validation errors and still wipes field values after an invalid submission. In this section, we will understand how the React **`useFormStatus`** Hook can address these problems. We will then use this Hook within our form.

Understanding useActionState

The **`useActionState`** Hook allows a Client Component to access the result of a Server Action. This enables error messages to be shown to the user. It also enables field values to be persisted in the form after submission.

The syntax for **`useActionState`** is as follows:

```
const [
  actionState,
  formAction,
  isPending,
] = useActionState(serverAction, initialState);
```

The Hook is passed a reference to the Server Action and an initial state value as arguments.

The Hook returns an array containing the following ordered elements:

- **`actionState`**: The current action state value
- **`formAction`**: An action to bind to the **`form`** element
- **`isPending`**: Whether the Server Action is in progress

The array element names can be any meaningful names of our choice. The action state's structure is also our choice, but it typically contains at least a message and a copy of the field values.

The Server Action must have particular parameters and must return the new state value when used with **`useActionState`**:

```
  export async function serverAction(
    previousState: { message: string, formData:
  FormData },
    formData: FormData,
  ) {
    ...
    return { message, formData }
  }
```

The parameters must be the previous state, followed by the form data. Often, the previous state isn't needed in the Server Action implementation, but React requires this parameter.

Next, we'll use this Hook in our form.

Using useActionState

We will use the **`useActionState`** Hook in our form to better handle errors and prevent the loss of field values already entered. We will make the necessary changes to the **`insertContact`** Server Action before reworking the **`ContactForm`** component.

Returning state from the Server Action

Open **`insertContact.ts`** and carry out the following steps to return the state from the **`insertContact`** Server Action:

1. Start by adding a type to represent the action state:

   ```
   type ActionState = {
     ok: boolean;
     error: string;
     formData: FormData;
   };
   ```

   The **`ok`** property determines whether the mutation was successful. The **`error`** property will contain the error message if the validation or mutation fails. The **`formData`** property is a copy of the form's **`FormData`** object, which contains the current field values.

2. Add a **`previousState`** parameter to the **`insertContact`** function:

```
export async function insertContact(
  previousState: ActionState,
  formData: FormData,
) { ... }
```

3. Return the relevant action state if the form data is invalid:

```
export async function insertContact( ... ) {
  const parsedResult =
contactSchema.safeParse( ... );
  if (!parsedResult.success) {
    return {
      ok: false,
      error:
        'Unable to save - invalid field
values',
        formData,
    };
  }
  ...
}
```

4. Set an **`error`** variable if the database mutation fails:

```
let error = '';
try {
  client = createClient( ... );
  await client.execute( ... );
} catch {
  ok = false;
  error = 'Problem saving form';
}
```

5. Lastly, return the action state at the end of the **insertContact** function:

```
if (ok) {
  redirect( ... );
}
return { ok, error, formData };
```

That completes the changes to the Server Action.

Adding action state to the form

We will make changes to **ContactForm.tsx** now, making use of the **useActionState** Hook. We are going to remove the use of **useFormStatus** and instead use the **pending** variable from **useActionState**. Open **ContactForm.tsx** and make the following changes:

1. Start by removing the **useFormStatus** import statement and importing the **useActionState** Hook from React:

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

2. Remove the **SubmitButton** component from the bottom of the file and replace the reference to it in **ContactForm** with the following **button** element:

```
<form ...>
  ...
  <button type="submit">Submit</button>
</form>
```

3. In the **`ContactForm`** component, call **`useActionState`**, as follows, before the **`return`** statement:

```
export function ContactForm() {
  const [
    { ok, error, formData },
    formAction,
    isPending,
  ] = useActionState(insertContact, {
    ok: false,
    error: '',
    formData: new FormData(),
  });
  return ...
}
```

   The Server Action and initial state is passed into the Hook. The returned action state is destructured into **`ok`**, **`error`**, and **`formData`** variables. We also destructure the **`form`** action into a **`formAction`** variable and the **`pending`** flag into an **`isPending`** variable.

4. The **`formAction`** variable can now be set on the **`form`** element:

```
<form action={formAction}>
```

5. To retain field values, we set the **`defaultValue`** attribute on the editor elements to the relevant value from the **`formData`** value from the action state:

```
<input ... name="name"
  defaultValue={
      (formData.get('name') ?? '') as string
    }
  />
  ...
  <input ... name="email"
    defaultValue={
      (formData.get('email') ?? '') as string
    }
  />
  ...
  <select ... name="reason"
    defaultValue={
      (formData.get('reason') ?? '') as string
    }
  > ... </select>
  ...
  <textarea ... name="notes"
    defaultValue={
      (formData.get('notes') ?? '') as string
    }
  />
```

We use the `get` function in the `formData` action state object to get the relevant field value. If a field value hasn't been entered, `null` will be returned, so we coalesce this to an empty string. We use a `string` type assertion to keep the TypeScript compiler happy.

6. Add an error alert after the fields, just above the **Submit** button:

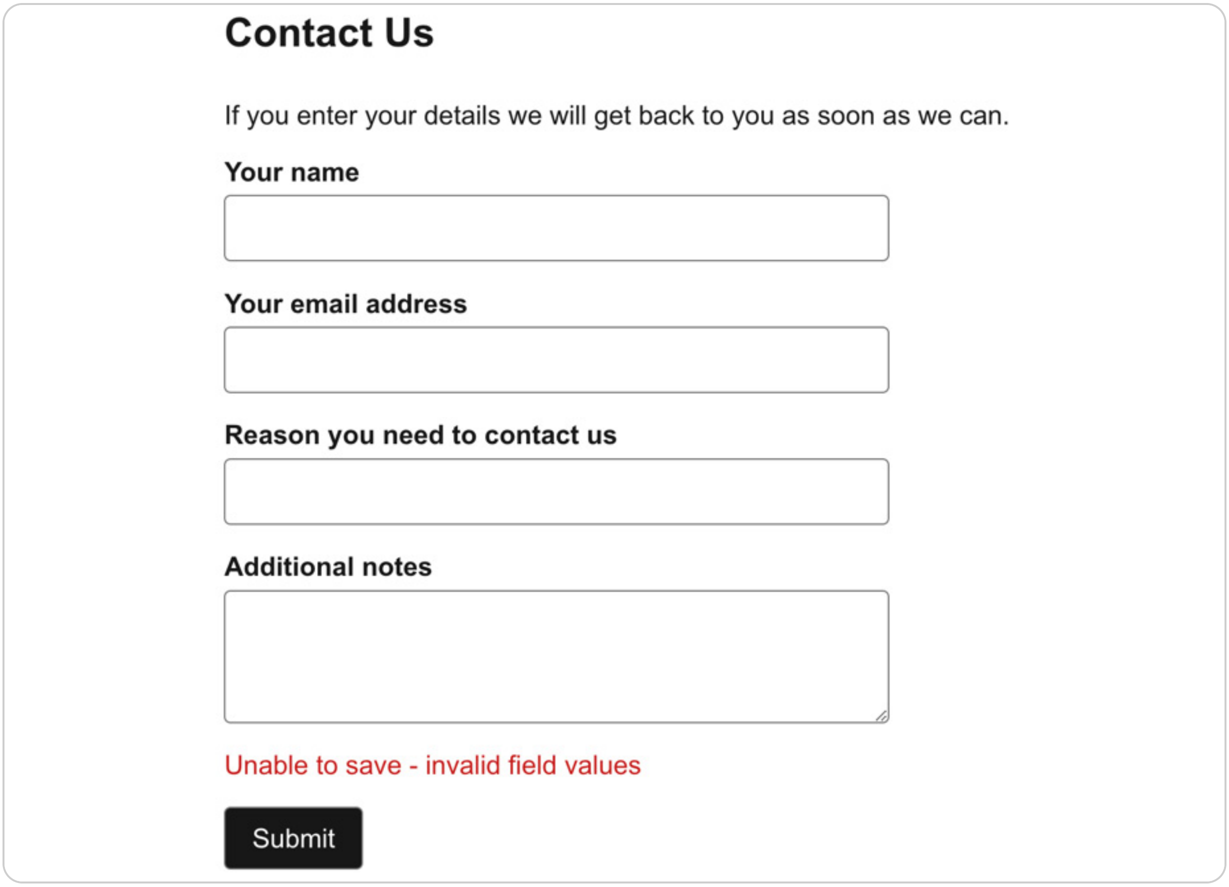
```
  {!ok && (
    <p role="alert" className="error">{error}
  </p>
```

```
)}
<button type="submit">Submit</button>
```

7. Lastly, add a submission indicator after the error alert, just above the **Submit** button. Also, disable the button while the component is in a **`pending`** state:

```
{isPending && (<p role="alert">Saving ...
</p>)}
<button type="submit" disabled={isPending}>
  Submit
</button>
```

8. In the running app, submit the form without entering anything. The validation error is returned:


Contact Us

If you enter your details we will get back to you as soon as we can.

Your name

Your email address

Reason you need to contact us

Additional notes

Unable to save - invalid field values

Figure 9.3 – Validation error

9. Try partially filling in the form. You'll find that field values are no longer lost.
10. To see the submission indicator, throttle the network using the browser development tools. The submission process will then be slow enough for you to see the submission indicator.
11. It is also worth checking that the form still works with JavaScript disabled – you'll find that it does!
12. The final check is to submit a valid form. You will find this navigates to the **Thanks** page as it did before.

That's improved the validation user experience. We'll continue to improve this even more.

Adding field errors

Currently, we aren't showing specific errors for each field. We will extract the field errors from the Zod error and return this in a new piece of action state. We will then be able to render these errors under each field.

We will start with a function to extract the field errors from the Zod error. We'll place this function in `insertContact.ts`. Carry out the following steps:

1. Open `insertContact.ts` and add an `import` statement for Zod:

```
import { z } from 'zod';
```

2. Add an **`errors`** property to the **`ActionState`** type, as follows. This will hold the field validation errors. We'll keep the **`error`** property for general form errors:

   ```
   type Err = { message: string };
   type FieldErrors = {
     name: Err | null;
     email: Err | null;
     reason: Err | null;
   };
   type ActionState = {
     ...
     errors: FieldErrors;
   };
   ```

   The field errors will be stored in an object with the field name as the property name. The error value will be set to **`null`** if there is no error. If there is an error, the error message will be stored in an object in a **`message`** property.

3. Add a function at the bottom of **`insertContact.ts`** to extract field errors from a Zod error:

   ```
   function formatZodErrors(error: z.ZodError) {
     const formattedErrors: FieldErrors = {
       name: null,
       email: null,
       reason: null,
     };
   }
   ```

At the moment, the function initializes the field errors to **`null`**.

4. Carry on with the function implementation and iterate through the flattened Zod errors. Inside the loop, open a conditional branch for errors in an array structure because this is where the field errors will be:

```
function formatZodErrors( ... ) {
  ...
  for (const [key, value] of Object.entries(
    error.flatten().fieldErrors,
  )) {
    if (Array.isArray(value)) {
    }
  }
}
```

5. The final implementation step in the function is to set the errors in the relevant properties in the **`formattedErrors`** variable:

```
function formatZodErrors(error: unknown) {
  ...
  for ( ... ) {
    if ( ... ) {
      if (key === 'name') {
        formattedErrors.name = {
          message: value[0],
        };
      } else if (key === 'email') {
        formattedErrors.email = {
          message: value[0],
        };
      } else if (key === 'reason') {
        formattedErrors.reason = {
          message: value[0],
        };
      }
    }
  }
  return formattedErrors;
}
```

We also return the **formattedErrors** variable containing the field errors.

That completes the function to extract field errors. We will now update the **insertContact** function to return field errors in the action state:

1. Still in **insertContact.ts**, use the **formatZodErrors** function to return field errors when a validation error occurs:

```
export async function insertContact( ... ) {
  ...
  if (!parsedResult.success) {
    return {
      ok: false,
      error: ...,
      formData: ...,
      errors:
formatZodErrors(parsedResult.error),
    };
  }
  ...
}
```

We have left the returned error property with the general error message so that this message is rendered above the **Submit** button.

2. At the bottom of the **insertContact** function, include the **errors** property with no errors in the returned object:

```
export async function insertContact( ... ) {
  ...
  return {
    ok,
    error,
    formData,
    errors: {
      name: null,
      email: null,
      reason: null,
    }
  };
}
```

The Server Action is now returning field errors. Let's open **ContactForm.tsx** and make the necessary changes to render the field errors:

1. Start by passing an initial **errors** object value into the **useActionState** Hook and destructuring the **errors** variable:

```
const [
  { ok, error, errors, formData },
  formAction,
  isPending,
] = useActionState(insertContact, {
  ok: false,
  error: '',
  errors: {
    name: null,
    email: null,
    reason: null,
  },
  formData: new FormData(),
});
```

2. Underneath the **ContactForm** component, add a new **FieldError** component that will render a server validation error for a field:

```
  type Err = { message?: string } | null;
  function FieldError({ serverError, errorId }:
  {
    serverError: Err;
    errorId: string;
  }) {
    if (!serverError) {
      return null;
    }
    return (
      <div id={errorId} role="alert">
        {serverError.message}
      </div>
    );
  }
```

The validation error is conditionally rendered in a **div** element if the error object has a value. The **div** element has a role attribute of **"alert"** so that a screen reader will announce it.

3. Moving back to the **ContactForm** component, we will render the validation errors using the **errors** variable and the **FieldError**

component. Start by adding a validation error under the **`name`** input:

```
<div ... >
  <label ... >Your name</label>
  <input ...
    aria-invalid={errors.name ? 'true' :
'false'}
    aria-describedby="name-error"
  />
  <FieldError
    serverError={errors.name}
    errorId="name-error"
  />
</div>
```

The **`aria-describedby`** attribute on the **`input`** element associates it with the **`div`** element containing the error message, allowing a screen reader to announce the error when appropriate. The **`aria-invalid`** attribute informs a screen reader whether the input is in an invalid state or not.

4. We will use the same pattern on the **`email`** and **`reason`** fields:

```
<div ... >
  <label ... >Your email address</label>
  <input ...
    aria-invalid={errors.email ? 'true' :
'false'}
    aria-describedby="email-error"
  />
  <FieldError
    serverError={errors.email}
    errorId="email-error"
  />
</div>
<div ... >
  <label ... >Reason you need to contact
us</label>
  <select ...
    aria-invalid={
      errors.reason ? 'true' : 'false'
     }
    aria-describedby="reason-error"
  > ... </select>
  <FieldError
    serverError={errors.reason}
    errorId="reason-error"
  />
</div>
```

5. In the running app, submit the form without entering anything.

   Validation errors under each invalid field are now returned:

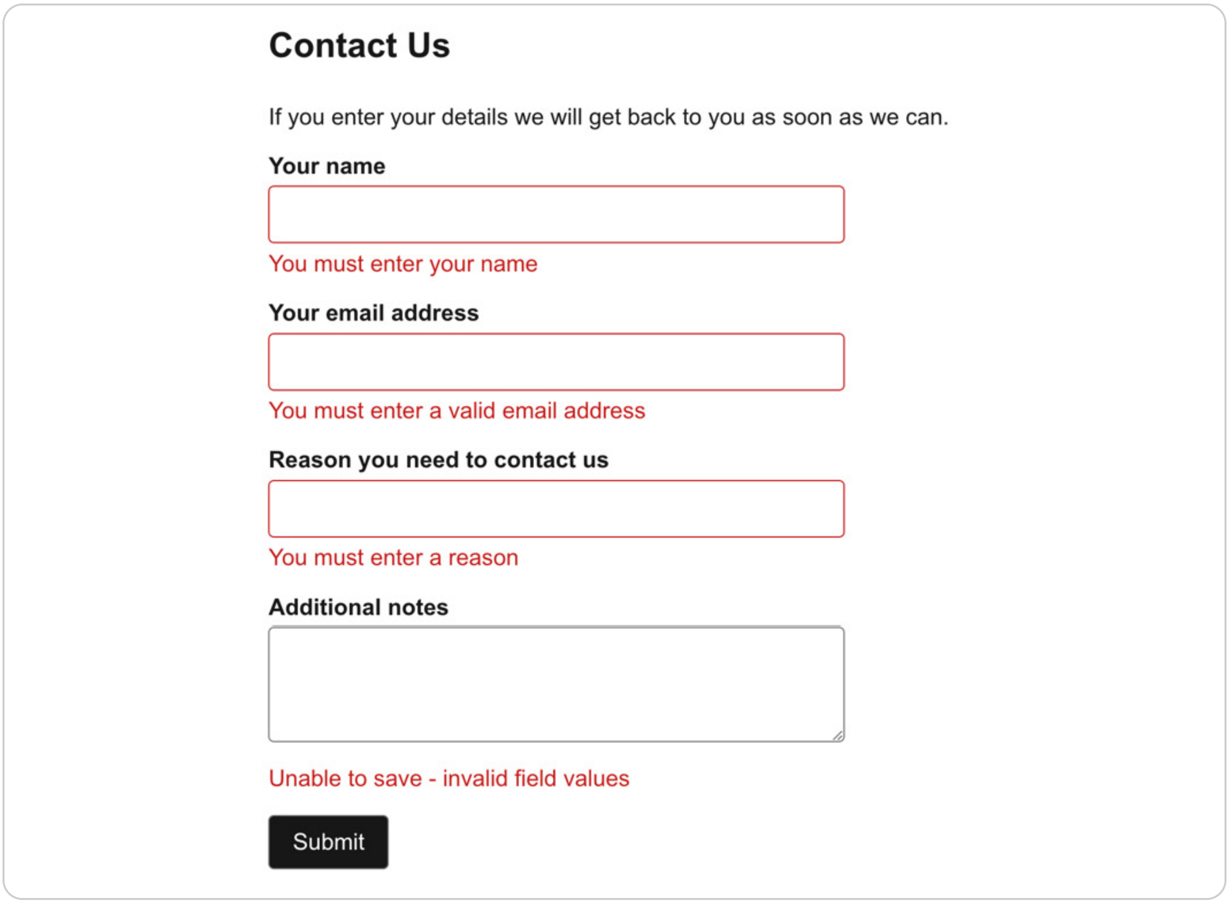

Contact Us
If you enter your details we will get back to you as soon as we can.
Your name
You must enter your name
Your email address
You must enter a valid email address
Reason you need to contact us
You must enter a reason
Additional notes
Unable to save - invalid field values
Submit

Figure 9.4 – Field validation errors

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

That completes the rendering of the field validation errors and this section on **`useActionState`**. Here's a quick recap:

- The **`useActionState`** Hook integrates with a Server Action and enables validation errors to be rendered and field values to be retained after submission.
- The Hook also enables the implementation of a submission indicator and the disabling of **`form`** elements during submission.

- The Server Action must contain parameters for the previous action state followed by the form data. It must also return the new action state.

So, both `useActionState` and `useFormStatus` have overlapping capabilities for implementing submission indicators and disabling `form` elements during submission. However, `useFormStatus` can only be used in a child component of a form, whereas `useActionState` can only be used in the same component as the form. So, the `useActionState` approach is preferable when `form` elements are in the same component, and the `useFormStatus` approach is ideal when the `form` elements are separated into multiple components.

Next, we will learn about a popular form library that can enhance the user experience of our form even further.

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. We will also learn about its benefits compared to native HTML form validation.

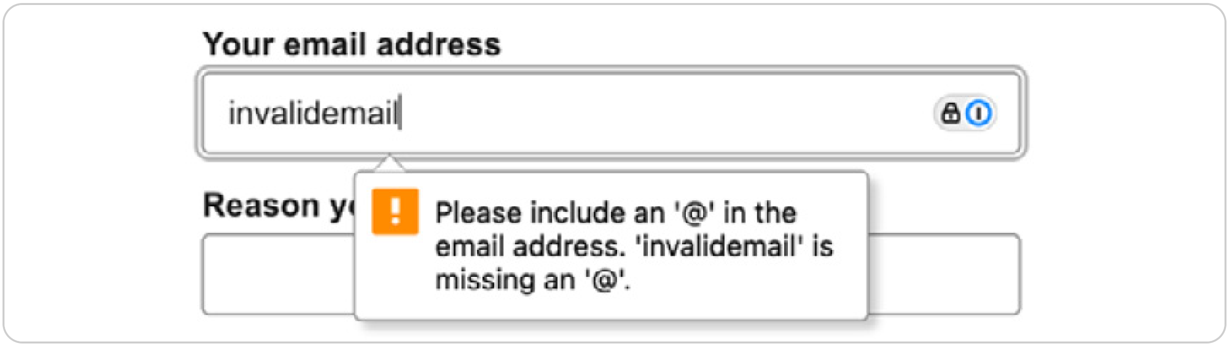
Understanding React Hook Form

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 optimized not to cause unnecessary re-renders. It is very popular with tens of thousands of GitHub stars and maturing nicely, having been first released in 2019.

Understanding client-side validation

A key feature that React Hook Form enables is client-side validation. Client-side validation improves the user experience by informing the user of a problematic entry before a server submission. It also helps server scalability because it won't deal with as many invalid form submissions.

There are native browser client validation capabilities – in fact, you may have spotted that the **`email`** field currently uses client-side email validation.



Please include an '@' in the email address. 'invalidemail' is missing an '@'.

Figure 9.5 – Native email validation

Email client-side validation is automatically included in **`input`** elements of the **`email`** type. We could have added a **`required`** attribute to the **`name`**, **`email`**, and **`reason`** inputs to add required client-side field validation rules to them.

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. So, common requirements such as customizing the error message styling or when the validation is triggered require quite a bit of code. This is why libraries such as React Hook Form are often preferred to the standard HTML form validation.

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.

Understanding the useForm Hook

The key part of using React Hook 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<FieldValues>();
```

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

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 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, the **`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.

A Zod schema can also be used to specify validation rules using a **`resolver`** option in the **`useForm`** Hook:

```
const { ... } = useForm({
  resolver: zodResolver(schema)
});
```

The schema variable in the preceding code snippet is a Zod schema definition and **`zodResolver`** is a function from a companion package, **`@hookform/resolvers`**.

Obtaining validation errors

The **`useForm`** Hook returns a state variable called **`errors`**, which contains the form validation errors. The **`errors`** state variable 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>}
```

Our form already uses this rendering pattern, so it should feel familiar.

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 while 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 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 how to use React Hook Form, we will use it in our contact form.

Using React Hook Form

We will use React Hook Form in the contact form we have been working on. We will still use **`useActionState`** and the Server Action for the form submission. We will be primarily using React Hook Form for client validation. Carry out the following steps:

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

   ```
   npm i react-hook-form @hookform/resolvers
   ```

2. Open **ContactForm.tsx** and import the React Hook Form Hook, the Zod resolver function, and our Zod schema. Also, import **useRef** from React, which we'll need to integrate React Hook Form's submission with the Server Action:

   ```
   import { ..., useRef } from 'react';
   import { useForm } from 'react-hook-form';
   import { zodResolver } from
   '@hookform/resolvers/zod';
   import { contactSchema } from '@/data/schema';
   ```

3. Call the **useForm** Hook after the **useActionState** Hook is called, as follows:

   ```
   export function ContactForm() {
     const [ ... ] = useActionState( ... );
     const {
       handleSubmit,
       register,
       formState: { errors: clientErrors },
     } = useForm({
       resolver: zodResolver(contactSchema),
       defaultValues: {
         name: '',
         email: '',
         reason: '',
         notes: '',
         ...(Object.fromEntries(formData) ?? {}),
   ```

```
    },
  });
```

We alias the validation errors as **`clientErrors`** so they don't collide with the server errors we get from the action state.

As well as passing **`useForm`** our Zod schema, we also pass it default values. We specify the default field values as empty strings and then overwrite these with any field values from the action state from an invalid submission.

4. After the **`useForm`** call, add a reference for the **`form`** element:

```
const { ... } = useForm( ... );
const formRef = useRef<HTMLFormElement>(null);
return (
  <form ref={formRef} ... >
    ...
  </form>
);
```

We will need the **`form`** element reference to integrate React Hook Form into the form submission.

5. Add a React Hook Form submission handler that invokes the Server Action as follows:

```
function onSubmit() {
  if (!formRef.current) {
    return;
  }
  formAction(new FormData(formRef.current));
}
```

```
return (
  <form
    ref={formRef}
    action={formAction}
    onSubmit={handleSubmit(onSubmit)}
  >
    ...
  </form>
);
```

The **`action`** attribute is left in place so that the submission still works before hydration has been completed.

The React Hook Form submission handler doesn't have the usual **`data`** parameter because we need to pass the Server Action field values in **`FormData`** format, which we get using the **`FormData`** constructor function passing in the form reference.

6. Add a **`noValidate`** attribute to the **`form`** element to suppress native HTML validation, which will remove the current email validation on the **`email`** field:

```
<form
  ...
  noValidate
>
```

7. We will update the **`name`** field to work with React Hook Form alongside the current action state. Use the **`register`** function to register the field with React Hook Form and remove the **`name`** attribute, because the **`register`** function sets this for us:

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

React Hook Form will set the default value, but it does it using JavaScript. So, the **`defaultValue`** attribute is left as it is so that we don't lose values after an invalid submission when JavaScript hasn't been hydrated.

8. Update the **`FieldError`** component to include a client error, as follows:

```
type Err = { message?: string } | null |
undefined;
function FieldError({ clientError, ...}: {
  clientError: Err;
  ...
}) {
  const error = clientError ?? serverError;
  if (!error) {
    return null;
  }
  return (
    <div id={errorId} role="alert">
      {error.message}
    </div>
  );
}
```

An **`error`** variable is assigned to the client or server error and replaces **`serverError`** in the function logic.

9. Moving back to the **`name`** field in the **`ContactForm`** component, update the error message to consider the client error from React Hook

Form:

```
<input ...
  aria-invalid={
    (clientErrors.name ?? errors.name)
      ? 'true'
      : 'false'
  }
/>
<FieldError clientError={clientErrors.name}
... />
```

10. Continuing with the **`name`** field, add an **`aria-required`** attribute to inform screen readers that it is a required field:

```
<input ...
  aria-required="true"
/>
```

11. Follow the same pattern to integrate React Hook Form into the **`email`** field:

```
<input ...
  {...register('email')}
  aria-required="true"
  aria-invalid={
    (clientErrors.email ?? errors.email)
      ? 'true'
      : 'false'
  }
/>
<FieldError clientError={clientErrors.email}
... />
```

12. Repeat the preceding for the **reason** field:

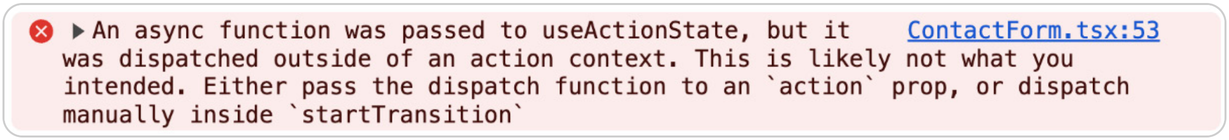
```
<select ...
  {...register('reason')}
  aria-required="true"
  aria-invalid={
    (clientErrors.reason ?? errors.reason)
      ? 'true'
      : 'false'
  }
> ... </select>
<FieldError clientError={clientErrors.reason}
... />
```

13. Integrate the **notes** field with React Hook Form:

```
<textarea {...register('notes')} />
```

14. Start the app running by running the **npm run dev** command in a terminal.

15. Fill in the form correctly and submit it. The submission will succeed and the app will navigate to the **Thanks** page. However, there is a React complaint if you look in the browser console:



```
▶ An async function was passed to useActionState, but it    ContactForm.tsx:53
was dispatched outside of an action context. This is likely not what you
intended. Either pass the dispatch function to an `action` prop, or dispatch
manually inside `startTransition`
```

Figure 9.6 – React submission warning

We've taken control of calling the Server Action from React – the React Hook Form submission handler will override the **action** attribute when JavaScript has been hydrated. This means React can't properly manage the action state. So, it's asking us to wrap the Server Action in a React Transition so that it can manage the action state.

16. To wrap the Server Action call in a React Transition, first, import the **`startTransition`** function from React and then wrap it around the call:

```
import { ..., startTransition } from 'react';
...
export function ContactForm() {
  ...
  function onSubmit() {
    startTransition(() => {
      if (!formRef.current) {
        return;
      }
        formAction(new
FormData(formRef.current));
    });
  }
  ...
}
```

17. In the running app, fill in the form correctly and submit it. This time, everything will work nicely without any warnings.

18. Try submitting an invalid form. An error will be displayed without any request to the server – it all happens in the browser. Notice also how the focus is automatically set to the first invalid field by React Hook Form.

19. Try turning JavaScript off and submitting invalid and valid forms. The form will behave as expected, falling back to server validation.
20. Re-enable JavaScript in the browser and stop the app from running by pressing *Ctrl* + *C*.

That completes the integration of client-side validation into our form and this section on React Hook Form. Here's a quick recap:

- React Hook Form is a popular, performant, and flexible React library for building forms that include client-side validation.
- The **`useForm`** Hook allows a Zod schema to define the validation rules. The Hook returns a **`register`** function for tracking fields and an **`errors`** state variable.
- React's action state and React Hook Form's state do overlap, and in many cases, it's fine to use either approach to keep the code simple. However, you can have the best of both worlds following the approach in this section.

Next, we will learn about optimistic updates after a form submission.

Implementing optimistic updates

In this section, we will learn what optimistic UI updates are and how to use React's **`useOptimistic`** Hook to implement them. We will then use this pattern on a new page in our app that allows users to mark a contact item as done.

Understanding useOptimistic

An optimistic UI update is when the UI is updated immediately after a user action is invoked, before the action is fully complete. The pattern makes the app faster and more responsive. The **`useOptimistic`** Hook can be used to manage a variable that is expected to change during an action. Typically, the variable will hold some data from the server. Here's the syntax:

```
const [optimisticValue, setOptimisticValue] =
  useOptimistic(initialValue);
```

The variable we want to set optimistically is passed into **`useOptimistic`**. A tuple containing the optimistic value and a setter function is returned. Here's a component that uses **`useOptimistic`**:

```
export function Name() {
  const [name, setName] = useState('');
  const [optName, setOptName] =
useOptimistic(name);
  async function updateName(formData: FormData) {
    const newName = formData.get('name');
    setOptName(newName);
    // Send newName to server to update db
    setName(newName);
  }
  return (
    <form action={updateName}>
      <input type="text" name="name"
defaultValue={name} />
      <p>New name: {optName}</p>
    </form>
  );
}
```

The component contains a form containing a **`name`** input. When the form is submitted, the entered name is sent to the server before the **`name`** state is updated. The **`useOptimistic`** Hook is used to optimistically show the new name beneath the **`name`** input.

You might be wondering why we wouldn't just use a normal state variable for the optimistic state. Well, **`useOptimistic`** automatically falls back to the old state if an error occurs, and it automatically deals with race conditions when multiple actions are triggered quickly.

> ***NOTE***
>
> *The **`useOptimistic`** setter function can be used for actions not invoked from a form. However, if it's not used in a form action, it must be inside a React Transition.*

Now that we understand the **`useOptimistic`** Hook, we'll use it inside our app.

Using useOptimistic

We will create a page in our app listing all the contact items from the database. Each list item will contain a form to make additional notes and mark the item as done. We will use **`useOptimistic`** for an optimistic done state.

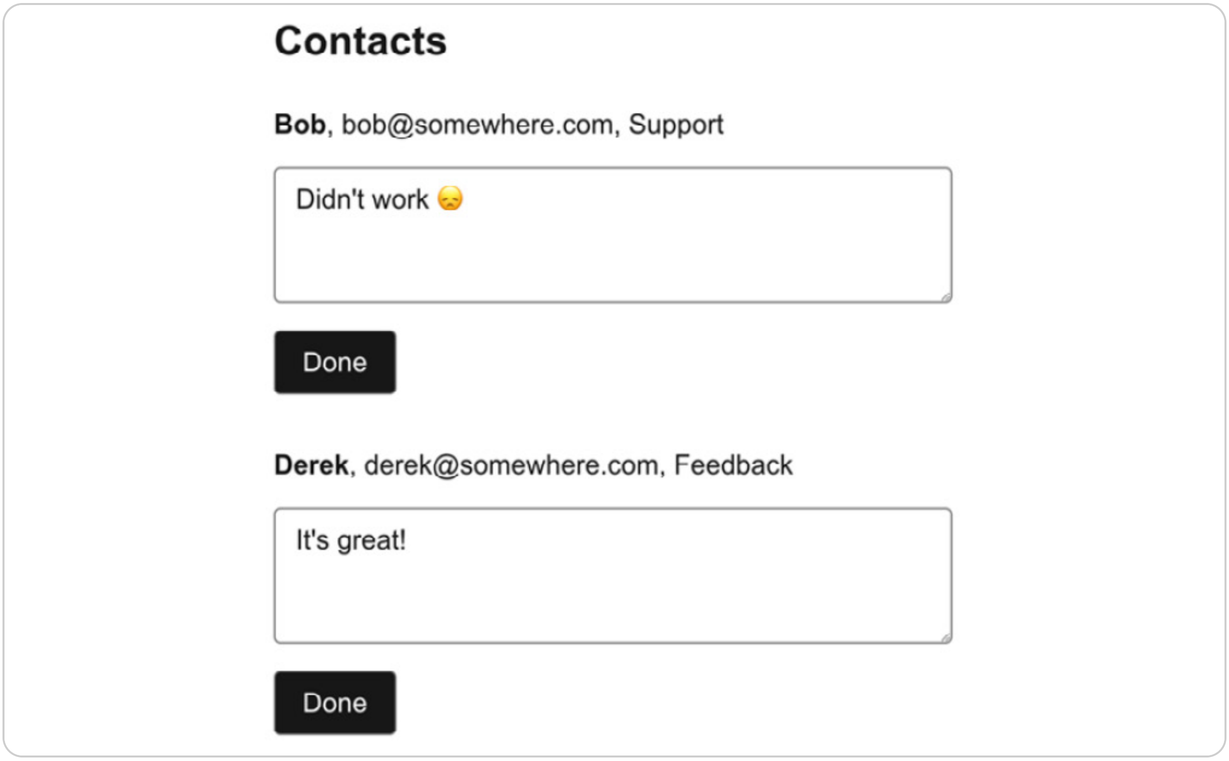
Adding an unoptimistic contacts page

We will start by adding a version of a contacts page that doesn't implement an optimistic done state. This uses previously covered knowledge, so we will copy the relevant file content from the GitHub repository at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter09/use-optimistic. Add the following files from the repository along with their folder structure:

- **`src/data/getContacts.ts`**: This contains a function to get all the contact items.
- **`src/data/completeContact.ts`**: This contains a Server Action to update the contact item notes and mark it as done in the database.
- **`src/data/schema.ts`**: This contains additional Zod schema for the list of contact items and the Server Action for completing an item. This file already exists, but it can be replaced with the content from the repository.

- **`src/app/contacts/page.tsx`**: This is the new page for the contact items.
- **`src/components/ContactItem.tsx`**: This is the component we will be working on, which is referenced in the new page. It renders details of a contact item with a form below it. The form contains a **`notes`** field and a **Submit** button. The form submission calls the **`completeContact`** Server Action. You may also notice a hidden input holding the contact item ID, which is a common pattern for passing additional data to a Server Action.

Start the app running and navigate to the **`/contacts`** path to see the new page rendered.



Contacts

Bob, bob@somewhere.com, Support

Didn't work 😞

Derek, derek@somewhere.com, Feedback

It's great!

Figure 9.7 – Contacts page

You can add some contact submissions using the root page if no contact items appear.

Try clicking the **Done** button to mark an item as done. It will work but with an annoying delay.

Making done optimistic

Open **`ContactItem.tsx`** and carry out the following steps to enhance the **`ContactItem`** component to render the **`done`** state optimistically:

1. Add **`useOptimistic`** to the React **`import`** statement:

   ```
   import { ..., useOptimistic } from 'react';
   ```

2. Call the **`useOptimistic`** Hook after the **`useActionState`** Hook is called, as follows:

   ```
   export function ContactItem( ... ) {
     const [ ... ] = useActionState( ... );
     const [optimisticDone, setOptimisticDone] =
       useOptimistic(done);
     return ...
   }
   ```

   The **`done`** variable is passed into **`useOptimistic`**, which comes from the database from the page RSC. The returned **`optimisticDone`** will contain the optimistic **`done`** value, and **`setOptimisticDone`** is a setter function to set it.

3. Update the paragraph element to use **optimisticDone** and reduce the opacity when the action is not actually done:

```
<p
  style={{
    textDecoration: optimisticDone
      ? 'line-through'
      : 'none',
    opacity: !done && optimisticDone ? 0.5 :
1,
  }}
>
  <b>{name}</b>, {email}, {reason}
</p>
```

4. Change the conditional rendering of the form to use the optimistic **done** value:

```
{!optimisticDone && (
  <form ... >
    ...
  </form>
)}
```

5. Rework the action handler to set **optimisticDone** before calling the Server Action:

```
<form
  action={(formData) => {
    setOptimisticDone(true);
    return formAction(formData);
  }}
>
```

6. In the running app, try clicking the **Done** button to mark an item as done. The form will disappear, and the item will immediately be crossed out with reduced opacity. After a second, when the Server Action is complete, the item will revert to full opacity.

That completes the enhancement to the `ContactItem` component and this section on optimistic updates. To recap, optimistic UI updates improve app responsiveness by immediately reflecting user actions in the UI, and React's `useOptimistic` Hook helps implement this pattern while handling errors and race conditions effectively.

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

Summary

In this chapter, we learned how to build forms in React, starting with a basic HTML form with the `Form` component from Next.js to prevent a full page reload during form submission.

We learned that Server Actions are special Server Functions used for form submissions using a `form` element's `action` attribute. The nice thing about this submission pattern is that it works without JavaScript.

We covered how to use the `useFormStatus` Hook for a submission indicator and disabling `form` elements, and understood its requirement for being in a child component of a `form` element. We learned that the `useActionState` Hook is an alternative way of implementing a submission indicator and disabling `form` elements when the form is in the same component. The `useActionState` Hook also allows the rendering of server-side validation errors.

We introduced ourselves to a popular forms library called React Hook Form to provide client-side validation with a Zod schema. This contains a `useForm` Hook that returns a `register` function to register fields and an `errors` state variable containing validation error messages.

Lastly, we covered using React's `useOptimistic` Hook to implement optimistic UI updates.

In this chapter, we used form state for field values, error messages, and whether submission indicators are visible. In the next chapter, we will learn about other types of state and also how to share state between components.

Questions

Answer the following questions to check what you have learned in this chapter:

1. How does the built-in Next.js `Form` component enhance form handling compared to a native `form` element?

2. What argument can be placed in the **console.log** statement to output the entered name during form submission?

```
<form action={(data) => console.log()}>
  <input type="text" name="name" />
  <button type="submit">Submit</button>
</form>
```

3. What's the problem with the following form using a React Server Action for submission?

```
<form onSubmit={someServerAction}>
  ...
</form>
```

4. What's the benefit of using the **form** element **action** attribute rather than **onSubmit** for form submission?

5. Consider the following form that captures and submits a name. When a name is entered and submitted, the result of the **console.log(name)** statement is **null**. Why is this the case?

```
function App() {
  const [name, formAction] = useActionState(
    updateName,
    "",
  );
  return (
    <form action={formAction}>
      <input
        type="text"
        defaultValue={(name ?? "") as string}
      />
      <button type="submit">Submit</button>
    </form>
  );
}
async function updateName(
  _: FormDataEntryValue | null,
  formData: FormData,
) {
  const name = formData.get("name");
  console.log(name);
  return name;
}
```

Answers

1. It uses client-side navigation rather than full-page navigation but still continues to function if JavaScript is disabled.
2. The **`data`** parameter in the action handler is an object with the native **`FormData`** interface. So, the **`get`** method can be used as follows to get the **`name`** field from the form:

   ```
   <form
     action={(data) =>
   console.log(data.get("name"))}
   >
     <input type="text" name="name" />
     <button type="submit">Submit</button>
   </form>
   ```
3. The **`action`** attribute should be used rather than **`onSubmit`**, as follows:

```
<form action={someServerAction}>
```

4. **`onSubmit`** requires JavaScript, whereas **`action`** can work without JavaScript.

5. The **`name`** attribute is missing on the **`input`** element, which means it isn't captured in the form submission:

```
<input
  name="name"
  type="text"
  defaultValue={(name ?? "") as string}
/>
```

Part 4:Advanced React

This part covers advanced React topics. It starts with different approaches to sharing state between components in an app with their benefits. Different patterns for implementing highly reusable type-safe components are then explored. The part finishes with how to implement automated tests on components, giving us the confidence to ship new features of applications quickly.

This part has the following chapters:

10
State Management

We've already used React state many times throughout this book. In this chapter, we'll cover React state in depth, starting by understanding the different types of state and situations when we can avoid writing code to manage state ourselves.

The chapter then focuses on **shared state**, which is the trickiest type of state to manage. We'll explore different approaches to managing shared state, discussing the pros and cons of each approach. To experience the different approaches, we will build a simple app containing a header that displays the user's name, with the main content also referencing the user's name. The user's name will be stored in a state that needs to be accessed by several components.

As such, we'll cover the following main topics in the chapter:

- Understanding the types of state
- Using prop drilling
- Using React context
- Using Zustand
- Using TanStack Query and URL parameters

Technical requirements

We will use the following technologies in this chapter:

- **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-Third-Edition/tree/main/Chapter10.

Understanding the types of state

In this section, we'll cover some of the different types of state and how it can be managed. React state increases code complexity, so it's helpful to be able to categorize it and potentially offload its management.

Server state

Server state is data fetched from an external server/API to render in the UI. It's also referred to as **remote state** or **data state**.

We learned in *Chapter 7*, *Server Component Data Fetching and Server Function Mutations*, how RSCs remove the need for server state because data is fetched and rendered all on the server.

Sometimes we do need to fetch data in Client Components, and in Chapter 8, *Client Component Data Fetching and Mutations with TanStack Query*, we learned how TanStack Query manages server state for us and how managing this ourselves using **`useEffect`** and **`useState`** is problematic.

Form state

Form state includes field values, validation error messages, and whether submission indicators are rendered. In Chapter 9, *Working with Forms*, we learned how this state can be managed using React's **`useActionState`** and **`useFormStatus`** Hooks, as well as the popular React Hook Form library. It's much simpler to use those approaches than to manage the state with our own code using **`useState`**.

URL state

URL parameters are a great way to store small bits of UI state. We used search parameters to store search criteria in previous chapters, but route parameters can be used as well. A key benefit of this approach is that the URL containing the state can be shared with another user to open the app rendered with that state.

Local state

Local state is managed by a single component using React's `useState` or `useReducer` Hooks, which we have used throughout this book.

An example of local state is the `visible` state in the alert component we built over several chapters. As a reminder, here's a snippet of the alert component:

```
export function Alert( ... ) {
  const [visible, setVisible] = useState(true);
  if (!visible) {
    return null;
  }
  function handleCloseClick() {
    setVisible(false);
    ...
  }
  return (
    <div>
      ...
      {closable && (
        <button ... onClick=
{handleCloseClick}>...</button>
      )}
      ...
    </div>
  );
}
```

Derived state

Derived state is computed from other state rather than being stored as state directly. It's a pattern that keeps code simpler and less error-prone by avoiding unnecessary duplication of state.

Here's a common example of state duplication where **`filteredItems`** is the **`active`** elements from the **`items`** state:

```
const [items, setItems] = useState([ ... ]);
const [filteredItems, setFilteredItems] =
useState([]);
useEffect(() => {
  setFilteredItems(items.filter((item) =>
item.active));
}, [items]);
```

So, **`filteredItems`** is duplicated state, which has to be synchronized using a **`useEffect`** Hook.

Here's the same example using derived state:

```
const [items, setItems] = useState([ ... ]);
const filteredItems = items.filter((item) =>
item.active);
```

It's much simpler because there is no synchronization logic.

The **`useMemo`** Hook can be used to reduce unnecessary derived state computations:

```
const filteredItems = useMemo(() => {
  return items.filter((item) => item.active);
}, [items]);
```

Shared state

As the name suggests, **shared state** is shared across multiple components. This is sometimes referred to as **global state**.

Shared state can get tricky fast if not handled carefully. Here are some reasons why:

- When lots of components read and write from some shared state, changes in one place can cause unintended side effects elsewhere. It can become hard to determine which component caused a change or why a component re-rendered.
- When some shared state changes, many components may re-render—even those that don't use the shared state. All this re-rendering can cause performance challenges.
- Too much can be placed in shared state—even state that could have stayed local. The shared state then becomes bloated and hard to maintain.
- A component that uses shared state is harder to unit test because the shared state is an external dependency that needs to be set up in the test.

That completes this section on state categories. Here's a quick summary:

- Server state is data from the server to render in the UI. Ideally, an RSC should be used to avoid server state. Alternatively, a library such as TanStack Query should be used to robustly manage server state.

- Form state is for state required in a form. It's recommended to use the React form Hooks and/or a library such as React Hook Form to manage it.
- URL state is when state is stored in a URL route or search parameters. It's a simple way to store bits of UI state.
- Local state is for a single component, whereas shared state is for multiple components.
- Derived state is a pattern to avoid state duplication and synchronization.

The remainder of this chapter focuses primarily on different approaches to implementing shared state. It's worth noting that server state and URL state can be used as approaches for shared state – this chapter covers these approaches as well.

Next, we'll cover the simplest approach for implementing shared state.

Using prop drilling

In this section, we will create the first iteration of the app, which will use a technique called **prop drilling** to share state between components. After we create the project, we will take the time to understand what prop drilling is and then make use of it in the app.

Creating the project

The app we will build will contain a header and some content beneath it. Here is the component structure we will create:

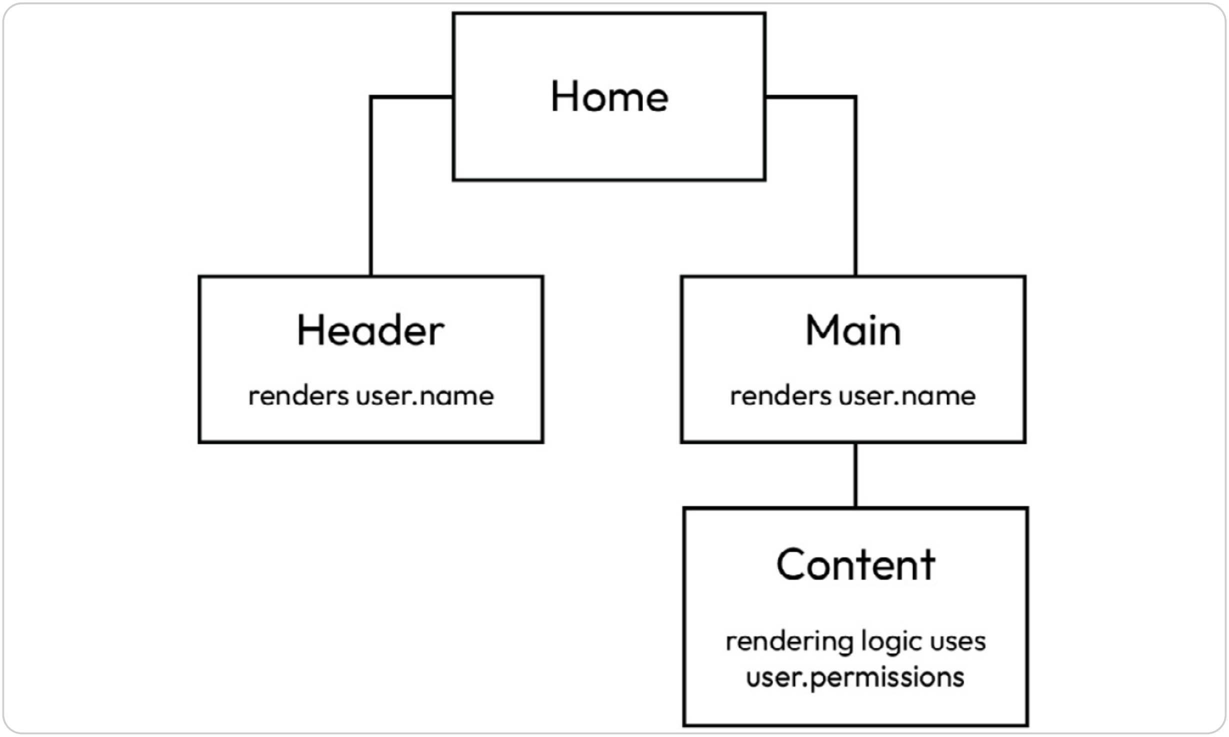


Figure 10.1 – App component structure

The header will have a **Sign in** button to authenticate and authorize a user to get their name and permissions. Once authenticated, the user's name will be displayed in the app header, and the user will be welcomed in the content. If the user has admin permissions, important content will be shown.

Carry out the following steps to create the initial versions of the files that we need in the app without any state sharing. We will copy and paste the code from the GitHub repository to save time:

1. In a terminal, execute the following command to create the project:

```
npx create-next-app@latest state --ts --eslint
--app --src-dir --import-alias "@/*" --no-
tailwind --turbopack
```

2. Still in the terminal, move to the project folder and open Visual Studio Code using the following commands:

   ```
   cd state
   code .
   ```

3. Prettier can be set up in the same manner as we learned with Vite in *Chapter 1*, *Getting Started with React*. Feel free to add automatic code formatting to this project.

4. Open **`src/app/global.css`** and overwrite the content with the CSS from the following file in the GitHub repository: https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter10/start/src/app/globals.css.

   This will nicely style our app.

5. Open **`page.tsx`** in the **`src/app`** folder and replace its content with the content in the GitHub repository at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter10/start/src/app/page.tsx.

   This component manages all the state but isn't currently sharing it. It references the **`Header`** and **`Main`** components and **`signIn`** and **`signOut`** Server Functions, which we'll create in the next step.

Notice that **`Home`** is declared as a *Client Component* with the **`'use client'`** directive because it is managing state.

6. Create the following component files and copy and paste the contents from the GitHub repository at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter10/start/src/components:

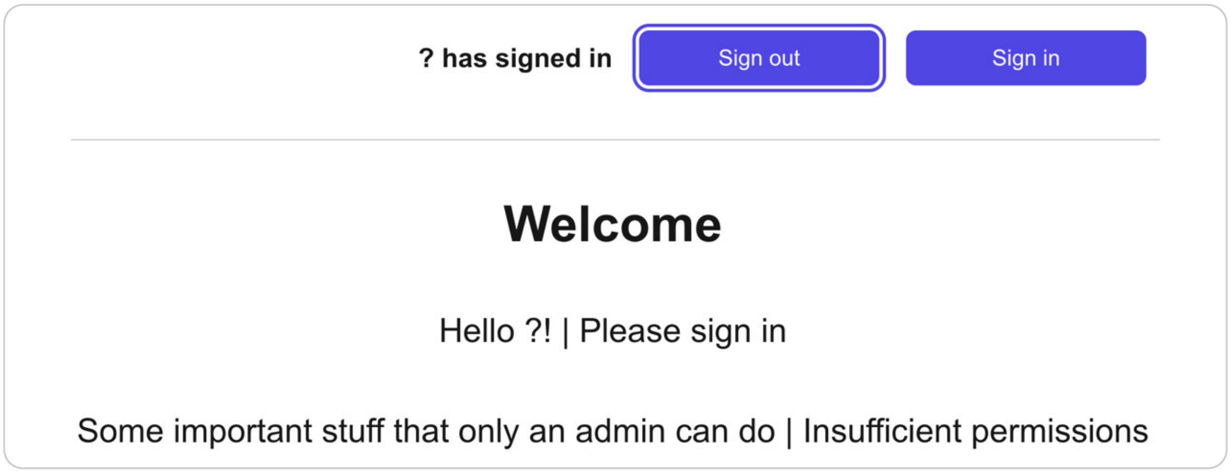
    - **`src/components/Header.tsx`**: This renders buttons to sign in and out. There is also a message to indicate who is signed in, which displays a **`?`** because it doesn't know yet.
    - **`src/components/Main.tsx`**: This renders hello and sign-in messages. Again, it doesn't know whether a user has signed in yet, so both messages are displayed. It also renders a **`Content`** component.
    - **`src/components/Content.tsx`**. This renders some important stuff and an insufficient permissions message. It doesn't know whether the user has permissions, so both messages are currently displayed.
    - **`src/data/auth.ts`**. This contains the **`signIn`** and **`signOut`** Server Functions. These functions only simulate authentication for simplicity, so that we can focus on state management in this chapter. **`signIn`** returns a user called Bob with admin permissions.

There will be ESLint errors in **`page.tsx`** because the state variables are currently not used. This will be resolved in the next section.

7. In the terminal, execute the following command to run the app in development mode:

```
npm run dev
```

The app will be available in a browser at http://localhost:3000 and will show the following:



Welcome

Hello ?! | Please sign in

Some important stuff that only an admin can do | Insufficient permissions

Figure 10.2 – Initial screen

That completes the project setup. Next, we will learn what prop drilling is.

Understanding and using prop drilling

Prop drilling is the practice of passing data via props from a parent component down through multiple levels of intermediary components to reach a nested child component. This approach uses React features that we are already aware of.

In our app, we will pass the state variables in the **`Home`** component to its child components using props. We will also pass the **`handleSignIn`** and **`handleSignOut`** functions as props so that the child components can call these to invoke the sign-in and sign-out processes.

We will make changes to the bottom of the component tree and work our way up. Carry out the following steps:

1. Open **`Content.tsx`** and add a **`permissions`** prop. Use **`permissions`** to render the appropriate message:

```
export function Content({
  permissions,
}: {
  permissions: undefined | string[];
}) {
  if (permissions === undefined) {
    return null;
  }
  return (
    <p>
      {permissions.includes(‹admin›)
        ? ‘Some important stuff that only an
admin can do’
        : ‘Insufficient permissions’}
    </p>
  );
}
```

2. Open **`Main.tsx`** and add **`userName`** and **`permissions`** props. Render an appropriate message using the **`userName`** prop and pass the

permissions prop to **Content** in the JSX:

```
export function Main({
  userName,
  permissions,
}: {
  userName: string | undefined;
  permissions: undefined | string[];
}) {
  return (
    <main>
      <h1>Welcome</h1>
      <p>
        {userName ? `Hello ${userName}!` :
          'Please sign in'}
      </p>
      <Content permissions={permissions} />
    </main>
  );
}
```

Notice that the component doesn't use **permissions** in its rendering logic – it is only passing it through to the **Content** component.

3. Open **Header.tsx** and start by adding the following props:

```
export function Header({
  userName,
  onSignInClick,
  onSignOutClick,
  loading,
}: {
  userName: string | undefined;
```

```
    onSignInClick: () => void;
    onSignOutClick: () => void;
    loading: boolean;
  }) { ... }
```

4. Update the JSX in **`Header`** to render the sign-in or sign-out buttons depending on whether **`userName`** is defined. Also, output **`userName`** in the message:

```
  <header>
    {userName ? (
      <>
        <span>{userName} has signed in</span>
        <button type="button">Sign Out</button>
      </>
    ) : (
      <button type="button">Sign in</button>
    )}
  </header>
```

5. Wire the buttons up to the **`onSignInClick`** and **`onSignOutClick`** props and use the **`loading`** prop to disable the buttons when a sign-in or sign-out is taking place. Also, use the **`loading`** prop to update the button content when a sign-in or sign-out is taking place:

```
  {userName ? (
    <>
      <span> ... </span>
      <button ...
        onClick={onSignOutClick}
        disabled={loading}
      >
      {loading ? <...> : <Sign out'}
      </button>
    </>
  ) : (
    <button ...
      onClick={onSignInClick}
      disabled={loading}
    >
      {loading ? <...> : <Sign in'}
    </button>
  )}
```

6. Open **page.tsx** and pass the state and sign-in and sign-out handlers to the child components as follows:

```
<Header
  userName={userName}
  onSignInClick={handleSignIn}
  onSignOutClick={handleSignOut}
  loading={loading}
/>
<Main userName={userName} permissions=
{permissions} />
```

The running app now appears as shown in the screenshot:

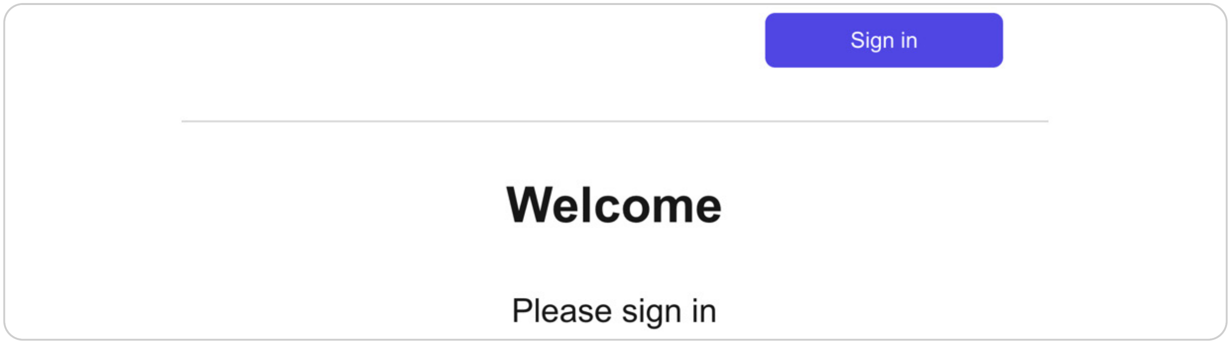


Figure 10.3 – App before signing in

7. Click the **Sign in** button. The sign-in process then happens, and after a couple of seconds, the following screen appears:

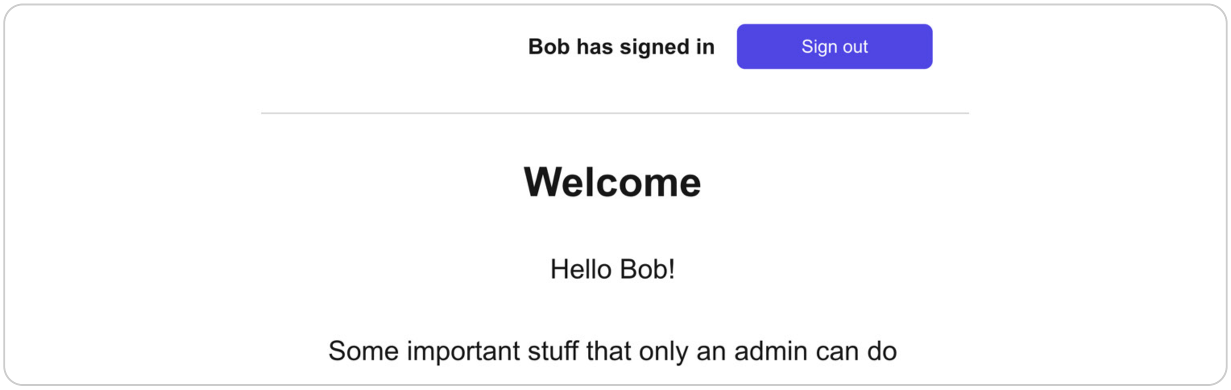


Figure 10.4 – App after signing in

A nice thing about this approach is that it is simple and uses React features we are already familiar with. A downside of this approach is that it can force components between the component providing state and components accessing the state to have a prop for that state. So, some components that do not need access to the state may be forced to access it. An example is the **`Main`** component – the **`permissions`** state is forced to pass through it to the **`Content`** component.

Using better composition

Often, better composition can resolve the issue of a state being prop-drilled through the component tree unnecessarily. For example, we can better compose the components inside the **`Home`** component to resolve the issue of the **`permissions`** state unnecessarily passing through **`Main`** to get to **`Content`**. We can achieve this by **`Main`** rendering React **`children`** instead of **`Content`**. **`Main`** can then pass **`Content`** as a child of **`Main`**.

Carry out the following steps to do this:

1. Open **`Main.tsx`** and remove the import statement for the **`Content`** component. Also, add an import statement for the **`ReactNode`** type:

   ```
   import type { ReactNode } from 'react';
   ```

2. Still in **`Main.tsx`**, remove the **`permissions`** prop and add a **`children`** prop. Also, replace **`Content`** with **`children`** in the JSX:

   ```
   export function Main({ userName, children }: {
     userName: string | undefined;
     children: ReactNode;
   }) {
     return (
       <main>
         <h1>Welcome</h1>
         <p>...</p>
         {children}
       </main>
     );
   }
   ```

3. Move to **`page.tsx`** and import the **`Content`** component:

   ```
   import { Content } from '@/components/Content';
   ```

4. Still in **`page.tsx`**, in the **`Home`** component JSX, remove the permissions attribute from **`Main`** and pass **`Content`** as a child of **`Main`** as follows:

   ```
   <Main userName={userName}>
     <Content permissions={permissions} />
   </Main>
   ```

The app will function as before.

That completes this section on prop drilling. Here's a recap:

- Prop drilling is when state is passed through multiple layers of components via props.
- It's the simplest state sharing approach – it uses very basic React features, and it's easy to trace how state flows through the app.
- It works well for sharing state across a few adjacent components. Good component composition can help prevent state from being unnecessarily passed through components.
- However, sharing state across deeply nested components becomes cumbersome because it involves state being unnecessarily passed through components. Components are also harder to refactor in this situation because of unnecessary dependencies.

Next, keep the app running, and we will look at a more appropriate solution for sharing state across many components.

Using React context

In this section, we will learn about a feature in React called **context**. We will then refactor the app from the last section to use React context.

Understanding React context

React context is an object that can be accessed by components. This object can contain state values, so it provides a mechanism for sharing state across components.

A context is created using a **`createContext`** function as follows:

```
const Context = createContext<ContextType>
(defaultValue);
```

A default value for the context must be passed into **`createContext`**. It also has a generic type parameter for the type that represents the object created by **`createContext`**.

A context provider component needs to be placed in the component tree above the components requiring access to it. A provider wrapper component can be created that stores the shared state and passes it to the context component as follows:

```
export function Provider({ children }: Props) {
  const [someState, setSomeState] =
useState(initialState);
  return (
    <Context value={{ someState }}>
      {children}
    </Context >
  );
}
```

The preceding code snippet uses the short variant of the context provider component, **`Context`**. The longer version is **`Context.Provider`**.

The **`useState`** Hook has been used for the state in the preceding example, but **`useReducer`** could also be used.

The provider wrapper component can then be placed appropriately in the component tree, above components requiring the shared state:

```
function App() {
  return (
    <Provider>
      <Header />
      <Main />
    </Provider>
  );
}
```

React also contains a **`use`** Hook that can be used to get values from the context:

```
const { someState } = use(Context);
```

The context must be passed into the **`use`** Hook and properties from the context object can be destructured from its result.

NOTE

There is an alternative **`useContext`** *Hook that can be used to get state that has the same syntax as the* **`use`** *Hook. However, unlike* **`useContext`**, *the* **`use`** *Hook can be used conditionally.*

So, components that want access to the shared state can access it using the **`use`** Hook as follows:

```
export function SomeComponent() {
  const { someState } = use(Context);
  return <div>I have access to {someState}</div>;
}
```

For more information on React context, see the following link: https://react.dev/reference/react/createContext.

Now that we understand React context, we will use it in the app we created in the previous section.

Using React context

We will rework the app from the last section to use React context. We will start by creating the context and a provider component. Then, we will move the state from the **Home** component to the provider component. We will also add the ability to toggle permissions and observe the impact on re-rendering.

So, to do this, carry out the following steps:

1. Start by creating a file called **types.ts** in a new **state** folder in the **src** folder. Add the following type to represent the state and functions we will need in our context:

```
export type UserState = {
  userName: undefined | string;
  permissions: undefined | string[];
  loading: boolean;
  handleSignIn: () => Promise<void>;
  handleSignOut: () => Promise<void>;
```

```
    togglePermissions: () => void;
  };
```

2. Create another file in the **`state`** folder called **`UserContext.ts`**. Add the following content to the file to create a React context that will contain the state and functions:

   ```
   import { createContext } from 'react';
   import type { UserState } from './types';
   export const UserContext =
   createContext<UserState>({
     userName: undefined,
     permissions: undefined,
     loading: false,
     handleSignIn: () => new Promise(() => {}),
     handleSignOut: () => new Promise(() => {}),
     togglePermissions: () => {},
   });
   ```

   The **`createContext`** function requires an initial state, so we pass an object containing undefined state values and empty functions.

> **NOTE**
>
> *React context can share functions as well as state. We will use this feature to share the **`handleSignIn`**, **`handleSignOut`**, and **`togglePermissions`** functions.*

3. Create another file called **`UserProvider.tsx`** in the **`state`** folder. Add the start of the provider component as follows:

   ```
   'use client';
   ```

```
import { type ReactNode } from 'react';
import { signIn, signOut } from '@/data/auth';
import { UserContext } from './UserContext';
export function UserProvider({
  children,
}: {
  children: ReactNode;
}) {
  return (
    <UserContext
      value={{}}
    >
      {children}
    </UserContext>
  );
}
```

At the moment, the **UserProvider** component is just rendering the **UserContext** provider component.

We have marked it as a *Client Component* because we will eventually use it in the **RootLayout** RSC.

4. Move the state and **handleSignIn** and **handleSignout** functions from the **Home** component to the **UserProvider** component:

```
import { ..., useCallback, useState } from
'react';
...
export function UserProvider() {
  const [userName, setUserName] = useState<
    string | undefined
  >();
  const [permissions, setPermissions] =
useState<
```

```
    string[] | undefined
  >();
  const [loading, setLoading] =
useState(false);
  const handleSignIn = useCallback(async () =>
{
    setLoading(true);
    const user = await signIn();
    setUserName(user.name);
    setPermissions(user.permissions);
    setLoading(false);
  }, []);
  const handleSignOut = useCallback(async () =>
{
    setLoading(true);
    await signOut();
    setUserName(undefined);
    setPermissions(undefined);
    setLoading(false);
  }, []);
  return ...
}
```

5. Add a **togglePermission** function to the context as well:

```
export function UserProvider( ... ) {
  ...
  const togglePermissions = useCallback(
    () =>
      setPermissions((currPermissions) =>
        currPermissions?.length === 0
          ? [‘admin’]
          : [],
      ),
    [],
  );
}
```

The function toggles between admin and no permissions.

6. Still in **`UserProvider.tsx`**, pass the state, **`handleSignIn`**, **`handleSignout`**, and **`togglePermissions`** functions into **`UserContext`** in the JSX:

```
<UserContext
  value={{
    userName,
    permissions,
    loading,
    handleSignIn,
    handleSignOut,
    togglePermissions
  }}
>
  {children}
</UserContext>
```

7. Open **`layout.tsx`** and wrap **`UserProvider`** around the layout content:

```
import { UserProvider } from
'@/state/UserProvider';
...
export default function RootLayout( ... ) {
  return (
    <html ... >
      <body ... >
        <UserProvider>{children}</UserProvider>
      </body>
    </html>
    );
  }
```

This means the whole component tree will have access to the context.

8. Open **`page.tsx`**. The state and **`handleSignIn`** and **`handleSignout`** functions should be removed. The React and **`auth`** import statements should also be removed, and remove the passing of the props in the JSX. Complete the work in this file by removing the **`'use client'`** directive because it can now be an RSC. The file contents should now be as follows:

```
import { Header } from '@/components/Header';
import { Main } from '@/components/Main';
export default function Home() {
  return (
    <>
      <Header />
      <Main>
        <Content />
      </Main>
    </>
  );
}
```

9. Open **`Header.tsx`** and mark it as a *Client Component* because it will be using React context. Also, import the **`use`** Hook from React and the **`UserContext`** context we created earlier. Remove the props and get the state and functions from the context instead:

```
'use client';
import { use } from 'react';
import { UserContext } from
'@/state/UserContext';
export function Header() {
  const {
    userName,
    handleSignIn,
    handleSignOut,
    loading,
  } = use(UserContext);
  return ...
}
```

10. Still in **Header.tsx**, extract **togglePermissions** from the context and use it when the sign-in message is clicked:

```
export function Header() {
  const { ..., togglePermissions } =
use(UserContext);
  return (
    <header>
      {userName ? (
        <>
          <span onClick={togglePermissions}>
            {userName} has signed in
          </span>
          ...
        </>
      ) : ... }
    </header>
  );
}
```

Adding a click handler on a span element isn't good practice, but it's a simple way to experience React context re-renders.

11. Update the button **onClick** handlers to reference sign-in and sign-out handlers from the context:

```
return (
    <header>
      {userName ? (
        <>
        ...
          <button ... onClick={handleSignOut}>
          ...
          </button>
        </>
      ) : (
        <button ... onClick={handleSignIn}>
          ...
        </button>
      )}
    </header>
  );
```

12. Open **Main.tsx** and follow the same pattern as the previous step. Don't pass **permissions** to **Content** – it'll cause a compile error, but we'll resolve this in the next step:

```
'use client';
import { use, type ReactNode } from 'react';
import { UserContext } from
'@/state/UserContext';
...
export function Main({
  children,
}: {
  children: ReactNode;
}) {
    const { userName } = use(UserContext);
    return ...
  }
```

13. Open **Content.tsx** and follow the same pattern again:

```
  'use client';
  import { use } from 'react';
  import { UserContext } from
  '@/state/UserContext';
  export function Content() {
    const { permissions } = use(UserContext);
    ...
  }
```

The compile errors in all the files should now be resolved, and the running app will look and behave like before.

14. Open the React development tools in your browser and make sure re-rendering highlights are on (**Components** | **Settings** | **Highlight updates when components render**). In the app, sign in and click the **Bob has signed in** message. Notice that all the components under the context provider re-render.

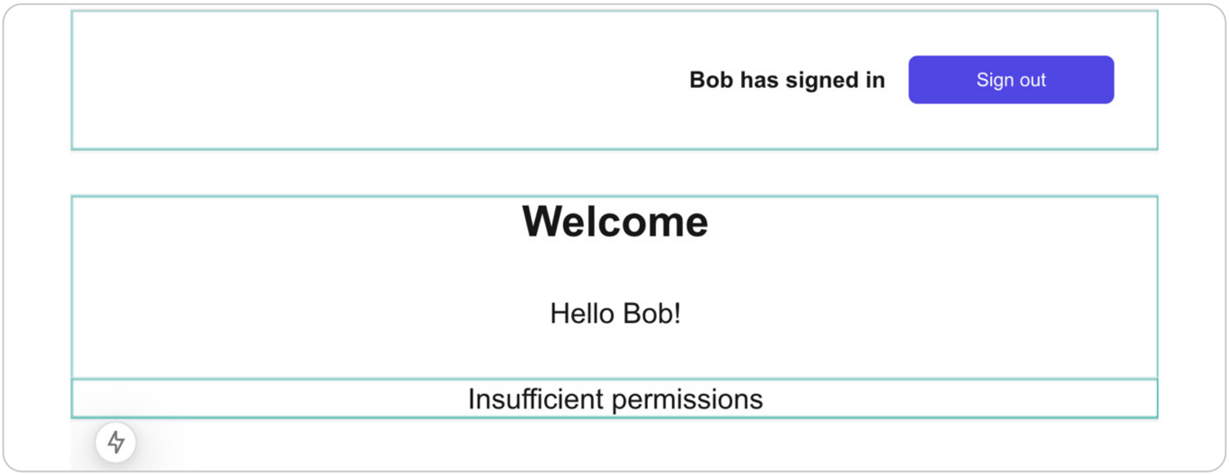


Figure 10.5 – Components re-rendering

15. Stop the app running by pressing *Ctrl* + *C*.

That completes the reworking of the app to use React context instead of prop drilling.

In comparison to prop drilling, React context requires more code to be initially written. However, it allows components to access shared state using a Hook rather than passing it through components using props. It's an elegant, shared-state solution, particularly when many components share state. However, when state changes, all the components beneath the context provider component re-render.

Next, we will learn about a popular third-party library that can be used to share state.

Using Zustand

In this section, we will learn about Zustand before using it to refactor the app we have been working on to use it.

Understanding Zustand

Zustand is a popular, performant, and scalable state management library for React that is incredibly simple to use.

The state lives in a centralized **store**, which is created using Zustand's **`create`** function:

```
const useCountStore = create((set) => ({
  count: 0,
  inc: () => set((state) => ({ count: state.count
+ 1 })),
  dec: () => set((state) => ({ count: state.count
- 1 })),
}));
```

Like React context, a Zustand store can hold functions as well as state values. The preceding example store contains **`count`** state with functions to increment and decrement it.

The **`create`** function returns a Hook that can be used to access the store. In the preceding example, we called this Hook, **`useCountStore`**.

The **`create`** function has an optional generic parameter for the type of the store:

```
export const useCountStore = create<{
  count: number;
  inc: () => void;
  dec: () => void;
}>( ... );
```

Unlike React context, no provider component is required. You simply access the store where you need it using the store Hook:

```
function Add() {
    const count = useCountStore((state) =>
  state.count);
    const inc = useCountStore((state) =>
  state.inc);
    return (
      <button type="button" onClick={inc}>
        {count}
      </button>
    );
  }
```

For more information on Zustand, see the following link: https://zustand.docs.pmnd.rs/getting-started/introduction.

Now that we understand Zustand, we will use it in the app we created in the previous section.

Using Zustand

We will refactor the app to use Zustand instead of React context. First, we will create a store and then consume it in the **`Header`**, **`Main`**, and **`Content`** components. Carry out the following steps:

1. Start by installing Zustand by executing the following command in a terminal:

   ```
   npm i zustand
   ```

2. Create a new file in the state folder called **`useUserStore.tsx`**. Add the following content to start the implementation of the store:

   ```
   import { create } from 'zustand';
   import { UserState } from './types';
   ```

```
export const useUserStore = create<UserState>(
  (set) => ({
    userName: undefined,
    permissions: undefined,
    loading: false
  }),
);
```

We have created a store to hold the state values, which we have initialized.

3. Add the sign-in and sign-out handler functions to the store:

```
import { signIn, signOut } from '@/data/auth';
export const useUserStore = create<UserState>(
  (set) => ({
    ...,
    handleSignIn: async () => {
      set({ loading: true });
      const user = await signIn();
      set({
        userName: user.name,
        permissions: user.permissions,
        loading: false,
      });
    },
    handleSignOut: async () => {
      await signOut();
      set({
        userName: undefined,
        permissions: undefined,
        loading: false,
      });
    },
  }),
);
```

The handlers are similar to the equivalent handlers in the React context, except we use Zustand's **set** function to update state values.

4. Add the **togglePermissions** function to the store as well:

```
export const useUserStore = create<UserState>(
  (set) => ({
    ...,
    togglePermissions: () =>
      set((state) =>
        state.permissions?.length === 0
          ? { permissions: ['admin'] }
          : { permissions: [] },
      ),
  }),
);
```

That completes the Zustand store.

5. Open **Header.tsx** and replace the context calls with a call to the store we just created:

```
import { useUserStore } from
'@/state/useUserStore';
export function Header() {
  const userName = useUserStore(
    (state) => state.userName,
  );
  const loading = useUserStore(
    (state) => state.loading,
  );
  const handleSignIn = useUserStore(
    (state) => state.handleSignIn,
    );
    const handleSignOut = useUserStore(
      (state) => state.handleSignOut,
    );
    const togglePermissions = useUserStore(
      (state) => state.togglePermissions,
    );
    return ...
  }
```

The React and context imports can also be removed because they are redundant.

6. Open **`Main.tsx`** and replace the context calls with a call to the store:

```
  import { type ReactNode } from 'react';
  import { useUserStore } from
  '@/state/useUserStore';
  ...
  export function Main() {
    const userName = useUserStore(
      (state) => state.userName,
    );
    return ...
  }
```

The React and context imports can also be removed because they are redundant.

7. Lastly, open **`Content.tsx`** and replace the context calls with a call to the store:

```
  import { useUserStore } from
  '@/state/useUserStore';
  ...
```

```
export function Content() {
  const permissions = useUserStore(
    (state) => state.permissions,
  );
  if (permissions === undefined) {
    return null;
  }
  return ...
}
```

Again, the React and context imports can also be removed because they are redundant.

8. Open **`layout.tsx`** and remove **`UserProvider`** because this is no longer required.
9. Run the app by running **`npm run dev`** in the terminal. The app will look and behave just as it did before.
10. Open the React development tools in your browser and make sure re-rendering highlights are on. In the app, sign in and click the **Bob has signed in** message. Notice that only the **`Content`** component re-renders. This is because this is the only component subscribed to the **`permissions`** state.

That completes the refactoring of the app to use Zustand rather than React context. As we have experienced, it is very simple to use, and not having to use a provider component is a clear benefit over React context. When a state in a Zustand store changes, only components that have subscribed to that state will re-render. This makes it more performant than React context.

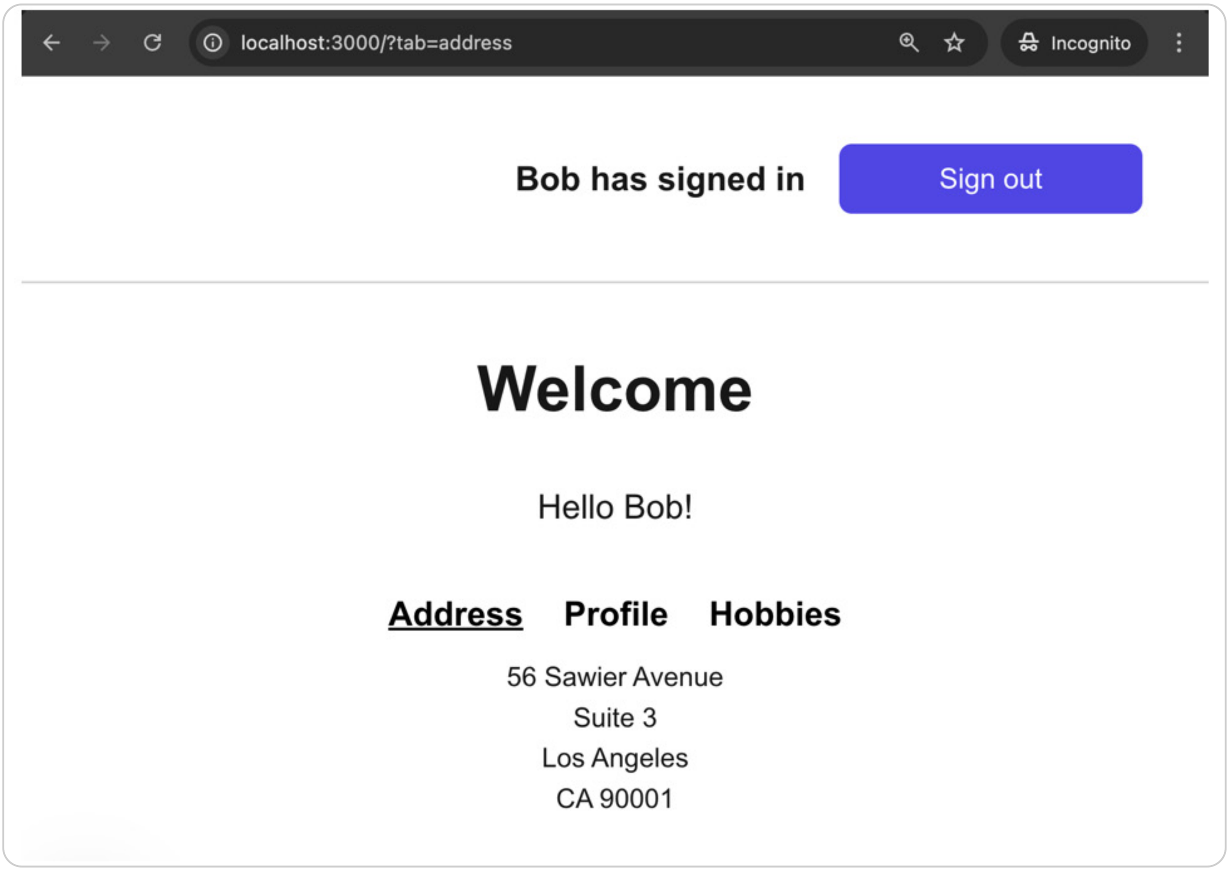
Next, we will remember URL parameters and TanStack Query and how they can be used to share state across components.

Using TanStack Query and URL parameters

In this section, we will learn how TanStack Query can be used to share state that usually comes from a server database across different React components. We will then move on to learn how state can be shared across components using URL search parameters.

We have already used TanStack Query in *Chapter 8*, *Client Component Data Fetching and Mutations with TanStack Query*, and used URL search parameters in *Chapter 6*, *Creating a Multi-Page App with Next.js*. So, we will focus on the state-sharing aspects rather than the full implementation details.

We will use an extension of the app we have been building to understand how both TanStack Query and URL search parameters can be used to share state. The app has been reworked to only return a user ID from a sign-in and to get information about the user using TanStack Query. The user information is shown in tabs, with the active tab stored as a URL search parameter.



56 Sawier Avenue
Suite 3
Los Angeles
CA 90001

Figure 10.6 – Extended app

The code for the extended app is available at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter10/url-react-query.

Using TanStack Query

As previously learned, TanStack Query maintains a client-side cache of the data it fetches. This data can be shared across different components by simply using the **`useQuery`** Hook with the same query key. When this Hook is called, the data will be fetched from the cache if it isn't stale. In the extended app, the **`Header`** and **`Main`** components use this approach to get the user name. The **`useQuery`** call to get the core user information is centralized into a **`useGetUser`** custom Hook as follows:

```
export function useGetUser(
  userId: string | undefined,
) {
  return useQuery({
    queryKey: ['user', userId],
    queryFn: async () => {
      const response = await fetch(
        `/api/users/${userId}`,
      );
      return userSchema.parse(
        await response.json(),
      );
    },
    enabled: userId !== undefined,
  });
}
```

We will learn more about custom Hooks in *Chapter 11*, *Reusable Components*.

The **`query`** function calls a Route Handler and returns the validated response body, which will be stored in the query cache. The **`enabled`** option ensures the query is only triggered when the user has signed in and a user ID is available.

After a sign-in occurs, the **`Header`** component will be re-rendered, causing the query to trigger and the data to be fetched and cached. The **`Main`** component will then be re-rendered, causing the same query to trigger. Because the cache for the key is available, the cached data will be used for the **`Main`** component.

Using TanStack Query is a natural way of sharing state from the server. We use the **`useQuery`** Hook with a **`query`** function and an appropriate key in the components that require the data. TanStack Query takes care of caching the data and whether a data fetch needs to occur.

Next, we will understand how URL search parameters can be used to share state.

Using URL parameters

In the extended app, a search parameter called **`tab`** is used to store the active tab. The implementation of the tabs is split across many components, as shown in the following diagram:

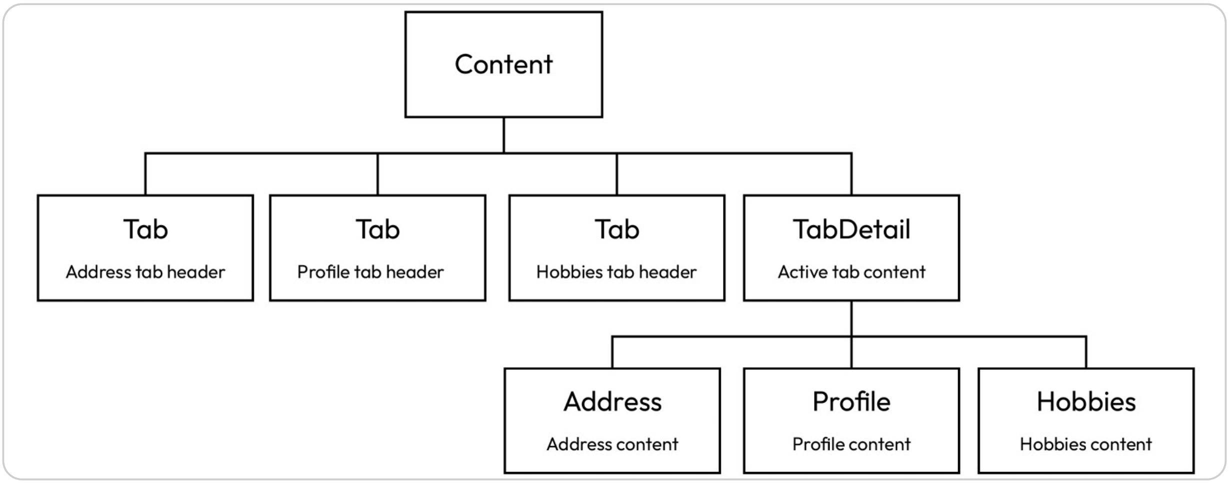


Figure 10.7 – Tab components

Both the **`Tab`** and **`TabDetail`** components access the active tab state from the URL. As previously learned, Next.js has a **`useSearchParams`** Hook to access search parameter values and a **`useRouter`** Hook to set them. The getting and setting of the **`tab`** search parameter in the **`Tab`** component is highlighted here:

```
export function Tab({ name, label }: { ... }) {
  const params = useSearchParams();
  const activeTab = params.get('tab') ??
'address';
  const router = useRouter();
  return (
    <button
      type="button"
      className={
        activeTab === name ? 'active' : ''
      }
      onClick={() => router.push(`/?
tab=${name}`)}
    >
    {label}
    </button>
  );
}
```

The getting of the tab search parameter in the **`TabDetail`** component is highlighted here:

```
export function TabDetail() {
  const params = useSearchParams();
  const activeTab = params.get('tab');
  if (activeTab === 'profile') {
    return <Profile />;
  }
  if (activeTab === 'hobbies') {
    return <Hobbies />;
  }
  return <Address />;
}
```

This approach is extremely simple but very effective. It isn't just limited to Next.js either – most frontend frameworks have mechanisms to interact with URL route and search parameters.

A common mistake is to duplicate the URL state using React state and try to synchronize them both using a React effect. However, the synchronization logic is often complex, resulting in edge case bugs.

That completes this section on using TanStack Query and the URL for sharing state. Here's a quick recap:

- TanStack Query caches server data and allows state sharing across components by using the same **`useQuery`** Hook with a shared query key
- URL search parameters can manage and share UI state, such as the active tab in a multi-tab app

Next, we will summarize the chapter.

Summary

We started this chapter by looking at different categories of state. We learned how to manage state robustly for each category. We spent most of the chapter focusing on the trickiest category of state, shared state. We built a small one-page app that contained components that needed to share state.

We started our shared state exploration by using the prop drilling approach. This is the simplest approach and ideal for a few adjacent components. However, it's cumbersome for lots of components – particularly if they are far apart in the component tree.

We moved on to learn about React context and refactored the app to use it. We learned that it's more convenient than prop drilling for sharing state between many components. However, it can cause performance issues because many components often re-render when state changes.

Next, we learned about Zustand, which is similar to React context. A difference is that no provider component is required, making it more straightforward to use. Zustand is very performant because components only re-render when the state that they subscribe to changes.

Lastly, we learned how TanStack Query is a great way to share server state. Our app used this approach to share user data. We also learned how to store certain UI state in the URL. Our app used this approach to store the active tab.

In the next chapter, we will learn how to make components highly reusable.

Questions

Answer the following questions to check what you have learned in this chapter:

1. We have a context defined as follows to hold the theme state for an app:

   ```
   type Theme = {
     name: string;
     color: ‘dark’ | ‘light’;
   };
   type ThemeContextType = Theme & {
     changeTheme: (
       name: string,
       color: ‘dark’ | ‘light’
     ) => void;
   };
   const ThemeContext =
     createContext<ThemeContextType>();
   ```

   The code doesn’t compile though. What is the problem?

2. The context from question 1 has a provider wrapper called **`ThemeProvider`**, which is added to the component tree as follows:

   ```
   <ThemeProvider>
     <Header />
     <Main />
   </ThemeProvider>
   <Footer />
   ```

The theme state is **`undefined`** when destructured from **`useContext`** in the **`Footer`** component. What is the problem?

3. Is it possible to have two React contexts in an app? Is it possible to have two Zustand stores in an app? Can you have both React context and Zustand in an app?
4. The following button updates state in a Zustand store when clicked:

```
<button
  onClick={() => {
    const changeTheme = useThemeStore(
      (state) => state.changeTheme,
    );
    changeTheme("blue", "dark");
  }}
>
  Change
</button>
```

   There is a problem with this code. What is this problem?

5. In a React component, is it possible to use React's **`useState`** as well as state from a Zustand store?

Answers

1. **`createContext`** must be passed a default value when using it with TypeScript. Here's the corrected code:

```
const ThemeContext =
createContext<ThemeContextType>({
  name: 'standard',
  color: 'light',
  changeTheme: (
    name: string, color: 'dark' | 'light') =>
{},
});
```

2. **Footer** must be nested inside **ThemeProvider** in the component tree, as follows:

```
  <ThemeProvider>
    <Header />
    <Main />
    <Footer />
  </ThemeProvider>
```

3. Yes, there is no limit on the number of React contexts in an app. There is also no limit on the number of Zustand stores in an app. In addition, apps can contain React contexts and Zustand stores.

4. The **useThemeStore** Hook can't be called in an event handler because it violates the rules of Hooks. This is the corrected code:

```
  const changeTheme = useThemeStore(
    (state) => state.changeTheme,
  );
  return (
    ...
    <button
      onClick={() => {
        changeTheme("blue", "dark");
      }}
    >
      Change
    </button>
```

```
      ...
  );
```

5. Yes, local state defined using **`useState`** or **`useReducer`** can be used alongside shared state from a Zustand store.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://packt.link/GxSkC

11
Reusable Components

In this chapter, we will build a checklist component and use various patterns to make it highly reusable but still strongly typed.

We will start by using TypeScript **generics** to strongly type the data passed to the component. Then, we will use the **props spreading** pattern to make the component API-flexible and allow consumers of the component to custom render parts of the component using the **render props** pattern. After that, we will learn how to make custom hooks and use them to extract logic for checked items and how to make the state within a component controllable to change the component's behavior.

By the end of this chapter, you'll have mastered key TypeScript and React patterns that will strengthen your ability to build flexible, maintainable, and reusable components.

We'll cover the following topics:

- Creating the project
- Using generic props
- Using prop spreading
- Using render props

- Adding checked functionality
- Creating custom hooks
- Allowing the internal state to be controlled

Technical requirements

We will use the following technologies in this chapter:

- **Node.js** and **npm**: You can install them here: https://nodejs.org/en/download/.
- **Visual Studio Code**: You can install it here: https://code.visualstudio.com/.

All the code snippets in this chapter can be found online at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter11.

Creating the project

In this short section, we will create the project for the app we will build and its folder structure. The folder structure will be straightforward because it contains a single page with the checklist component we will build.

We will develop the app using Visual Studio Code as in previous chapters, so open Visual Studio Code and carry out the following steps:

1. Create the project using Vite. See *Chapter 2*, *Getting Started with TypeScript*, if you can't remember the steps for this.
2. Copy styles from the GitHub repository at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/blob/main/Chapter11/start/src/index.css file, overwriting the existing contents.

That completes the project setup. Next, we will learn about a pattern that enables a reusable component to accept different types of data in a strongly typed manner.

Using generic props

In this section, we'll take some time to understand how to create our own generic types and also learn about the **`keyof`** TypeScript feature, which is useful for generic types. We will use this knowledge to build the first iteration of the checklist component with a generic type for its props.

Understanding generics

We have used generics throughout this book. For example, the **`useState`** hook has an optional generic parameter for the type of state variable:

```
const [visible, setVisible] = useState<boolean>()
```

Generic parameters in a function allow that function to be reusable with different types and be strongly typed. The following function returns the first element in an array, or **`null`** if the array is empty. However, the function only works with a **`string`** array:

```
function first(array: Array<string>): string |
null {
  return array.length === 0 ? null : array[0];
}
```

Generics allows us to make this function usable with any type of array.

Generic functions

Although we have used generic functions throughout this book, we haven't created our own generic function yet. Generic type parameters are defined in angled brackets before the function's parentheses:

```
function someFunc<T1, T2, ...>(...) {
 ...
}
```

The name of a generic type can be anything you like, but it should be meaningful so that it is easy to understand.

Here is a generic version of the function we saw earlier. Now, it can work with arrays containing any type of element:

```
function first<Item>(array: Array<Item>): Item |
null {
  return array.length === 0 ? null : array[0];
}
```

The function has a single generic parameter called **`Item`**, which is used in the type of the **`array`** function parameter, as well as the function's return type.

Generic types

Custom types can be generic as well. For a **`type`** alias, its generic parameters are defined in angled brackets after the type name:

```
type TypeName<T1, T2, …> = {
 ...
}
```

For example, the props of a React component can be generic. An example of a generic **`Props`** type is as follows:

```
type Props<Item> = {
  items: Item[];
  ...
};
```

The **`Props`** type has a single generic parameter called **`Item`**, which is used in the type of the **`items`** prop.

The keyof operator

The **`keyof`** operator is a TypeScript operator that returns a union of string literal types representing all the keys of a given type:

```
type User = {
  name: string;
  email: string;
};
type UserKeys = keyof User; // “name” | “email”
```

You can combine **`keyof`** with a generic type to dynamically reference the keys in the generic parameter:

```
type List<Item> = {
  items: Item[];
  id: keyof Item;
};
const users: List<User> = {
  items: [{ name: ‘user1’, email:
‘user1@somewhere.com’ }],
  id: ‘name’, // must be ‘name’ or ‘email’
};
```

In the preceding example, the **`id`** property of the **`users`** object must be a key of the **`User`** type – that is, **`‘name’`** or **`‘email’`**.

Generic React components

Generic props can be integrated into a generic function to produce a generic React component. Here’s an example of a generic **`List`** component:

```
type Props<Item> = {
  items: Item[];
};
export function List<Item>({ items }:
Props<Item>) {
  ...
}
```

The **items** prop in the **List** component can now have any type, making the component flexible and reusable.

Now that we understand how to create a component with generic props, we will create the first iteration of the checklist component.

Creating a basic list component

We will now start to create our reusable component. In this iteration, it will be a basic list containing some primary and secondary text obtained from an array of data.

Carry out the following steps:

1. Start by creating a folder for the component called **Checklist** in the **src** folder. Then, create a file called **Checklist.tsx** in this folder.
2. Open **Checklist.tsx** and add the following **Props** type:

   ```
   type Props<Data> = {
     data: Data[];
     id: keyof Data;
     primary: keyof Data;
   ```

```
  secondary: keyof Data;
};
```

Here is an explanation of each prop:

- The **`data`** prop is the data that drives the items in the list
- The **`id`** prop is the property's name in each data item that uniquely identifies the item
- The **`primary`** prop is the property's name in each data item that contains the main text to render in each item
- The **`secondary`** prop is the property's name in each data item that includes the supplementary text to render in each item

3. Next, start to implement the component function as follows:

```
export function Checklist<Data>({
  data,
  id,
  primary,
  secondary,
}: Props<Data>) {
  return (
    <ul>
      {data.map((item) => {
      })}
    </ul>
  );
}
```

The component renders an unordered list element. It maps over the data items where we will eventually render each list item.

4. Start implementing the function inside **`data.map`** as highlighted here:

```
{data.map((item) => {
  const idValue = item[id] as unknown;
  if (
    typeof idValue !== 'string' &&
    typeof idValue !== 'number'
  ) {
    return null;
  }
  const primaryText = item[primary] as
unknown;
  if (typeof primaryText !== 'string') {
    return null;
  }
  const secondaryText = item[secondary] as
unknown;
}
```

   The function checks whether the unique identifier (**`idValue`**) is a string or number, and if not, nothing is rendered. The function also checks that the primary text property (**`primaryText`**) is a string, and again, if not, nothing is rendered.

   It's important to do these runtime type checks because we want the **`Checklist`** component to be reusable with many different data sources, including data from a server.

5. Finish the implementation of the **`map`** function by rendering the list items as follows:

```
{data.map((item) => {
      ...
      return (
        <li
          key={idValue}
        >
          <div className="primary">
            {primaryText}
          </div>
          {typeof secondaryText === 'string' && (
            <div className="secondary">
              {secondaryText}
            </div>
          )}
        </li>
      );
    })}
```

The list items are rendered with a primary **`div`** element and an optional secondary **`div`** element.

6. Create a new file in the **`Checklist`** folder called **`index.ts`** and export the **`Checklist`** component into it:

```
export * from './Checklist';
```

This file will simplify **`import`** statements for the **`Checklist`** component.

7. The final step before seeing the component in action is to add it to the component tree in the app. Open **`App.tsx`** and replace the content within it with the following:

```
import { Checklist } from './Checklist';
function App() {
    return (
      <div>
        <Checklist
          data={[
            { id: 1, name: 'Lucy', role:
  'Manager' },
            { id: 2, name: 'Bob', role:
  'Developer' },
          ]}
          id="id"
          primary="name"
          secondary="role"
        />
      </div>
    );
  }
  export default App;
```

We reference the **`Checklist`** component and pass some data into it. Notice how type-safe the **`id`**, **`primary`**, and **`secondary`** attributes are – we are forced to enter a valid property name with the data items.

8. Run the app by entering **`npm run dev`** in the terminal. The checklist component appears as shown here:

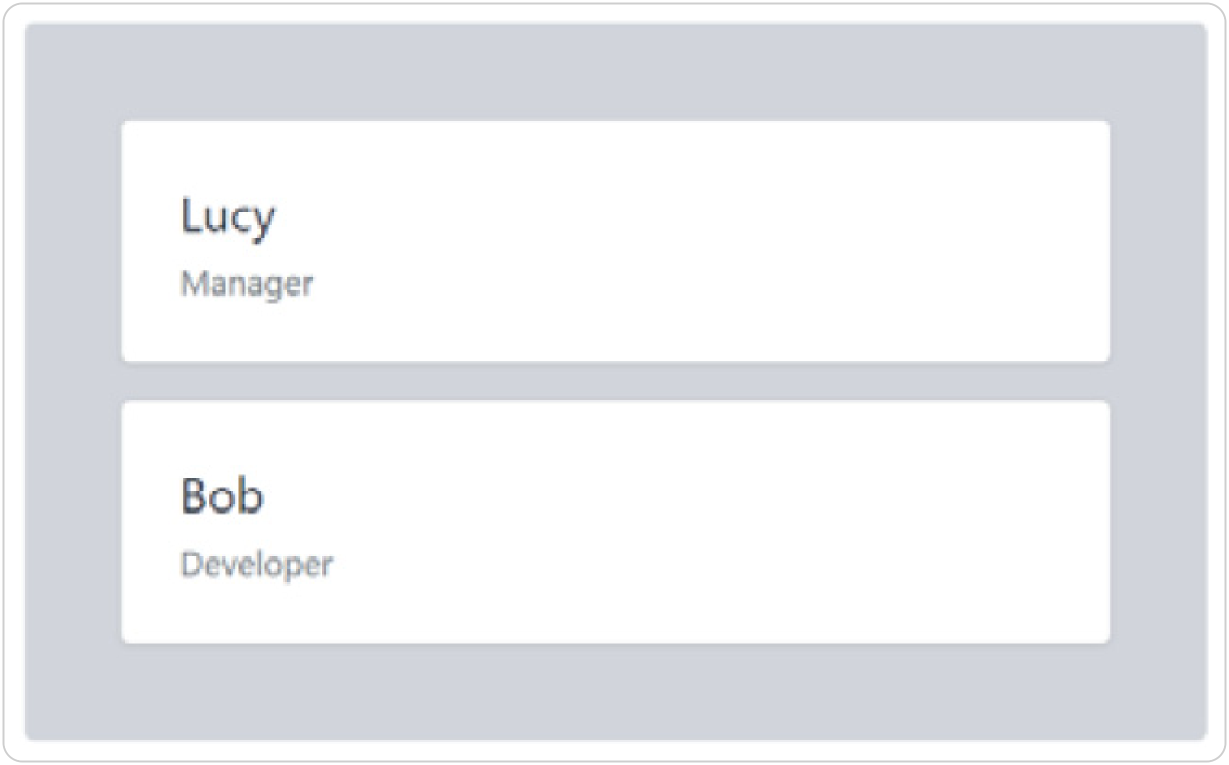


Figure 11.1 – Our basic checklist component

Currently, the component renders a basic list – we will add the checked functionality later in this chapter.

That completes this section on generic props.

To recap, here are some key points:

- TypeScript generics allow reusable code to be strongly typed.
- Functions can have generic parameters that are referenced within the implementation.
- Types can also have generic parameters that are referenced within the implementation.

- A React component can be made generic by feeding a generic props type into a generic function component. The component implementation will then be based on generic props.

Next, we will learn about a pattern that allows the prop type to inherit props from an HTML element.

Using prop spreading

In this section, we'll learn about a pattern called **prop spreading**. This pattern is useful when you want to use all the props from an HTML element in the implementation of a component. In our checklist component, we will use this pattern to add all the props for the `ul` element. This will allow consumers of the component to specify props, such as the height and width of the checklist.

So, carry out the following steps:

1. Open `Checklist.tsx` and import the following type from React:

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

   This type allows us to reference the props of an HTML element such as `ul`. It is a generic type that takes the HTML element name as a generic parameter.

2. Add the props from the `ul` element to the component props type as follows:

```
type Props<Data> = {
  data: Data[];
  id: keyof Data;
  primary: keyof Data;
  secondary: keyof Data;
} & ComponentPropsWithoutRef<'ul'>;
```

3. Add a **rest parameter** called **`ulProps`** to collect all the props for the **`ul`** element in a single **`ulProps`** variable:

   ```
   export function Checklist<Data>({
     data,
     id,
     primary,
     secondary,
     ...ulProps
   }: Props<Data>) {
     ...
   }
   ```

   This is the first time we have used rest parameters in this book. They collect multiple arguments that are passed into the function into an array, so any props that aren't called **`data`**, **`id`**, **`primary`**, or **`secondary`** will be collected into the **`ulProps`** array. For more information on rest parameters, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters.

4. Now, we can spread **`ulProps`** onto the **`ul`** element using the spread syntax:

```
export function Checklist<Data>({
  data,
  id,
  primary,
  secondary,
  ...ulProps
}: Props<Data>) {
  return (
    <ul
      {...ulProps}
    >...</ul>
  );
}
```

5. We can use this new feature of **Checklist** to specify the list height and width. Open **App.tsx** and add the following **style** attribute, as well as more data items:

```
<Checklist
  data={[
    { id: 1, name: 'Lucy', role: 'Manager' },
    { id: 2, name: 'Bob', role: 'Developer' },
    { id: 3, name: 'Bill', role: 'Developer'
},
    { id: 4, name: 'Tara', role: 'Developer'
},
    { id: 5, name: 'Sara', role: 'UX' },
    { id: 6, name: 'Derik', role: 'QA' }
  ]}
  id="id"
  primary="name"
  secondary="role"
  style={{
    width: '300px',
    maxHeight: '380px',
    overflowY: 'auto'
  }}
/>
```

6. If the app isn't running, run it by entering **npm run dev** in the terminal. The checklist component appears sized as we expect:

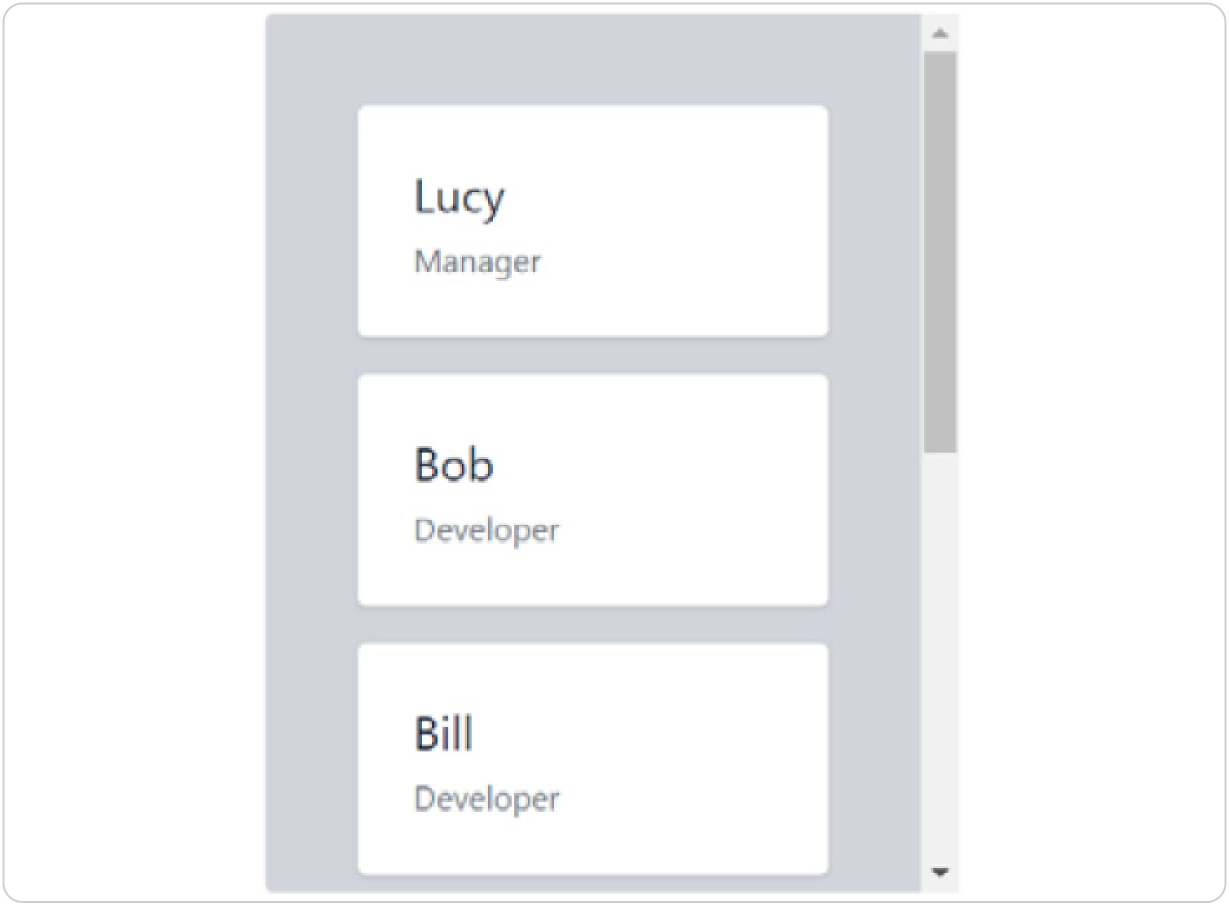


Figure 11.2 – The sized checklist component

The component now has a fixed height with a vertical scrollbar as a result of the style we passed into the component.

That completes our use of the prop spreading pattern. Here's a recap of the key points:

- We intersect the props type with **`ComponentPropsWithoutRef`** to add props for the HTML element we want to spread onto
- We use a rest parameter in the component props to collect all the HTML element props into an array
- We can then spread the rest parameter onto an HTML element in the JSX

Next, we will learn about a pattern that allows consumers to render parts of a component.

Using render props

In this section, we will learn about the **render props** pattern and use it to allow the consumer of the component to render items within the checklist component.

Understanding the render props pattern

A way of making a component highly reusable is to allow the consumer to render internal elements within it. The **`children`** prop on a **`button`** element is an example of this because it allows us to specify any button content we like:

```
<button>We can specify any content here</button>
```

The render props pattern allows us to use a prop other than **children** to provide this capability. This is useful when the **children** prop is already used for something else, as in the following example:

```
<Modal heading={<h3>Enter Details</h3>}>
  Some content
</Modal>
```

Here, **heading** is a render prop in a **Modal** component.

Render props are useful when allowing the consumer to render elements associated with the data passed into the component because the render prop can be a function:

```
<List
  data={[...]}
  renderItem={(item) => <div>{item.text}</div>}
/>
```

The preceding example has a render prop called **renderItem** that renders each list item in a **List** component. The data item is passed into it so it can include its properties in the list item. This is similar to what we will implement next for our checklist component.

Adding a renderItem prop

We will add a prop called **renderItem** to the checklist that allows consumers to take control of the rendering of the list items. Carry out the following steps:

1. Open **`Checklist.tsx`** and add the **`ReactNode`** type to the React **`import`** statement:

   ```
   import { ComponentPropsWithoutRef, type
   ReactNode } from 'react';
   ```

   `ReactNode` represents an element that React can render. Therefore, we will use **`ReactNode`** as the return type for our render prop.

2. Add a render prop called **`renderItem`** to the **`Props`** type:

   ```
   type Props<Data> = {
     data: Data[];
     id: keyof Data;
     primary: keyof Data;
     secondary: keyof Data;
     renderItem?: (item: Data) => ReactNode;
   } & ComponentPropsWithoutRef<'ul'>;
   ```

   The prop is a function that takes in the data item and returns what needs rendering. We have made the prop optional because we will provide a default implementation for list items but also allow consumers to override it.

3. Add **`renderItem`** to the component function parameters:

   ```
   export function Checklist<Data>({
     data,
     id,
   ```

```
    primary,
    secondary,
    renderItem,
    ...ulProps
  }: Props<Data>) {
    ...
  }
```

4. In the JSX, at the top of the mapping function, add an **`if`** statement to check whether the **`renderItem`** prop has been specified. If **`renderItem`** has been specified, call it with the data item, and return its result from the mapping function:

```
  <ul ...>
    {data.map((item) => {
      if (renderItem) {
        return renderItem(item);
      }
      const idValue = item[id] as unknown;
      ...
    })}
  </ul>
```

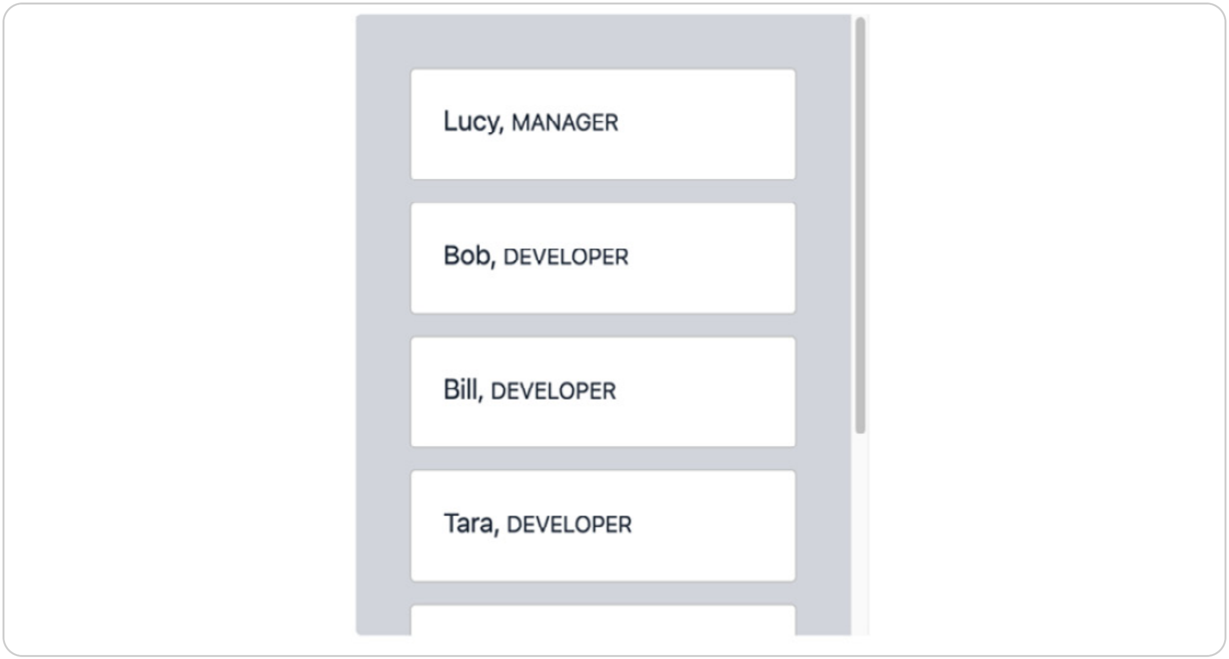
   So, if **`renderItem`** has been specified, it will be called to get the element to render as the list item. If **`renderItem`** hasn't been specified, it will render the list item as it previously did.

5. To try the new prop out, open **`App.tsx`** and add the following **`renderItem`** attribute:

```
  <Checklist
    ...
      renderItem={(item) => (
        <li key={item.id}>
          <div className="primary">
            {item.name},{" "}
            <small
              style={{
                textTransform: "uppercase"
              }}>
              {item.role}
            </small>
          </div>
        </li>
      )}
    />
```

The list items are now rendered with the role in uppercase. The **`key`** attribute on the list item elements is set to the unique item ID, allowing React to efficiently update the DOM when list items have changed, been added, or removed.

6. If the app isn't running, run it by entering **`npm run dev`** in the terminal. The checklist component appears with the overridden list items:


Lucy, MANAGER

Bob, DEVELOPER

Bill, DEVELOPER

Tara, DEVELOPER

Figure 11.3 – Overridden list items

7. Before continuing to the next section, remove the use of **`renderItem`** in the **`Checklist`** element in **`App.tsx`**. The default rendering of list items should then appear.

That completes this section on the render props pattern. To recap, here are some key points:

- The render props pattern allows a component consumer to override the rendering of parts of the component
- A render prop can either be an element or a function that returns an element
- A common use case for a render prop is a data-driven list in which the rendering of list items can be overridden

Next, we will add checked functionality to our checklist component.

Adding checked functionality

Currently, our checklist component doesn't contain the ability to check items, so we will now add checkboxes to the list of items, giving users the ability to check them. We will track the checked items using a React state.

So, carry out the following steps to add this functionality to our component:

1. Open **`Checklist.tsx`** and add **`useState`** to the React **`import`** statement:

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

   We will use the state to store the IDs of the checked items.

2. At the top of the component implementation, add the state for the IDs of the checked items:

   ```
   const [checkedIds, setCheckedIds] =
     useState<IdValue[]>([]);
   ```

   We have referenced an **`IdValue`** type that we haven't defined yet – we'll define this after we have finished the component implementation in *step 6*.

3. Add checkboxes to the list of items as follows:

```
<li key={idValue}>
  <label>
    <input
      type="checkbox"
      checked={checkedIds.includes(idValue)}
      onChange={handleCheckChange(idValue)}
    />
    <div>
      <div ... >
        {primaryText}
      </div>
        {typeof secondaryText === 'string' &&
(
      <div ... >
        {secondaryText}
      </div>
    )}
    </div>
  </label>
</li>
```

The **checkedIds** state powers the **checked** attribute of the checkbox by checking whether the list item's ID is contained within it.

The **label** element intentionally wraps the **input** element so that clicking anywhere inside the label, including the text or nested **div** elements, will toggle the checkbox. This improves usability for users with motor impairments who may struggle to click small checkboxes. Screen readers will also associate label content with the checkbox, so users using this tool will know what the checkbox represents.

We will implement the referenced **`handleCheckChange`** function in the next step. Notice that the reference calls the function passing the ID of the list item that has been checked.

4. Start to implement the **`handleCheckChange`** function in the component as follows:

```
const [checkedIds, setCheckedIds] = ...
const handleCheckChange =
  (checkedId: IdValue) => () => {};
return ...
```

This is a function that returns the handler function. This complexity is because a basic checked handler doesn't pass in the list item's ID. This approach is called **currying**, and more information on it can be found at the following link: https://javascript.info/currying-partials.

5. Complete the handler implementation as follows:

```
const handleCheckChange = (checkedId: IdValue)
=> () => {
  const isChecked =
checkedIds.includes(checkedId);
  const newCheckedIds = isChecked
    ? checkedIds.filter(
        (itemCheckedid) => itemCheckedid !==
checkedId
```

```
        )
      : checkedIds.concat(checkedId);
    setCheckedIds(newCheckedIds);
  };
```

The implementation updates the list item's ID to the **checkedIds** state if the list item has been checked and removes it if it is unchecked.

6. Next, let's define the **IdValue** type. Create a new file in the **Checklist** folder called **types.ts** with the definition of **IdValue**:

   ```
   export type IdValue = string | number;
   ```

   Here, the list item's ID can be a **string** or **number** value.

7. Move back to **Checklist.tsx** and import **IdValue**:

   ```
   import { IdValue } from './types';
   ```

   The compilation errors should now be resolved.

8. If the app isn't running, run it by entering **npm run dev** in the terminal. The checklist component appears with checkboxes for each list item:

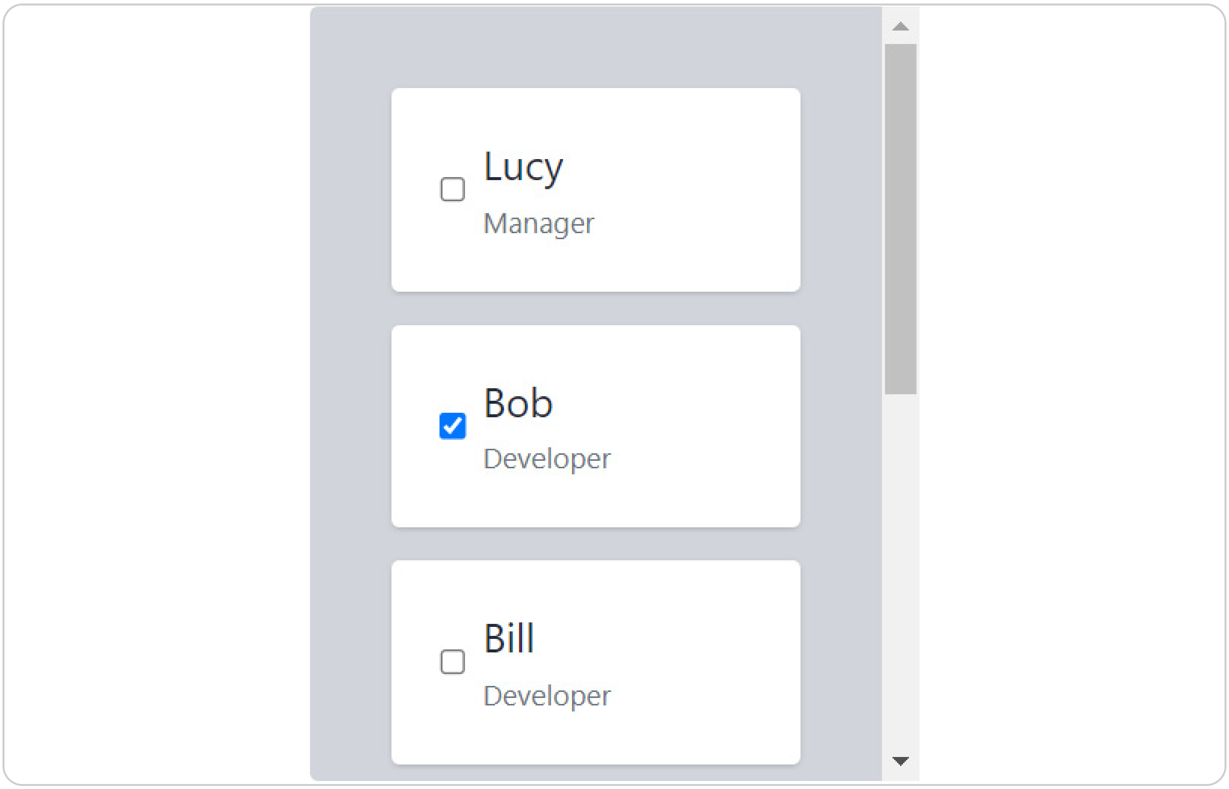


Figure 11.4 – Checkboxes for list items

The checklist component now includes checkboxes. However, there is an opportunity to make the checked logic reusable – we'll cover this in the next section.

Creating custom hooks

In this section, we'll learn about custom React hooks. Then, we will use this knowledge to extract the checked logic from the checklist component into a reusable custom hook.

Understanding custom hooks

As well as standard hooks such as **`useState`**, React allows us to create our own custom hooks. Custom hooks allow logic to be isolated and reused across multiple components.

A custom hook is defined using a function with a name that starts with the word *use*. This naming convention helps ESLint check for problems with the use of the custom hook. Here's a custom hook that provides toggling logic:

```
export function useToggle() {
  const [toggleValue, setToggleValue] =
useState(false);
  function toggle() {
    setToggleValue(!toggleValue);
  }
  return {toggleValue, toggle};
};
```

The custom hook contains the state of the current toggle value, which is either **`true`** or **`false`**. It also includes a function called **`toggle`**, which toggles the current value. The current toggle value and the **`toggle`** function are returned from the custom hook in an object structure.

Note that an object structure doesn't have to be returned. If the custom hook only returns a single item, then that item can be returned directly. If the custom hook returns two things (as in the preceding example), it can return a tuple (as **`useState`** does). An object structure is nice for more than two items because the object keys make it clear what each item is.

Another trait of a custom hook is that it uses other standard React hooks. For example, the **`useToggle`** custom hook uses **`useState`**. If the custom hook doesn't call a React hook or another custom hook, it's just a regular function rather than a custom hook.

This custom hook can then be used in the implementation of a component as follows:

```
const { toggleValue, toggle } = useToggle();
return (
  <div>
    <button onClick={toggle}>{toggleValue ? 'ON'
: 'OFF'}</button>
  </div>
);
```

The toggle value (**`toggleValue`**) and the **`toggle`** function are destructured from the return value of the custom hook. The toggle value is used to render text **ON** or **OFF** inside the button content depending on whether it is **`true`** or **`false`**. The **`toggle`** function is also assigned to the click handler of the button.

Custom hooks can take in parameters as well. In the example here, we have added a default value in the **`useToggle`** hook:

```
type Params = {
  defaultToggleValue?: boolean;
};
export function useToggle({ defaultToggleValue }:
Params) {
  const [toggleValue, setToggleValue] = useState(
    defaultToggleValue
  );
  ...
}
```

In the preceding example, the parameters are in an object structure. An object structure is nice when there are multiple parameters and nothing breaks when new parameters are added.

Arguments are passed into the custom hook in an object. Here's an example of using **`useToggle`** with its value initially being **`true`**:

```
const { toggleValue, toggle } = useToggle({
  defaultToggleValue: true
});
```

Now that we understand how to create and use custom hooks, we will put this into practice in our checklist component.

Extracting checked logic into a custom hook

We will extract the logic for checked items into a custom hook. This will allow potential future components to use the logic and clean up the code a little.

The custom hook will be called **`useChecked`** and will contain the state for the checked list item IDs. The hook will also include a handler that can be attached to the checkboxes, updating the checked list item ID's state.

To do this, carry out the following steps:

1. In the **`Checklist`** folder, create a file for the custom hook called **`useChecked.ts`**.
2. Open **`useChecked.ts`** and add the following **`import`** statements:

   ```
   import { useState } from 'react';
   import { IdValue } from './types';
   ```

   The hook will use React state that is typed using **`IdValue`**.
3. Start to implement the function for the custom hook by initializing the state:

   ```
   export function useChecked() {
     const [checkedIds, setCheckedIds] =
       useState<IdValue[]>([]);
   }
   ```

   The hook doesn't have any parameters. The **`useState`** call is exactly the same as the one currently in the **`Checklist`** component – this could be copied and pasted into the custom hook.
4. Add a checked handler to the custom hook. This can be copied from the implementation of the **`Checklist`** component:

```
export function useChecked() {
  const [checkedIds, setCheckedIds] =
    useState<IdValue[]>([]);
  const handleCheckChange = (checkedId:
IdValue) => () => {
    const isChecked =
checkedIds.includes(checkedId);
    const newCheckedIds = isChecked
      ? checkedIds.filter(
        (itemCheckedid) => itemCheckedid !==
checkedId
      )
      : checkedIds.concat(checkedId);
    setCheckedIds(newCheckedIds);
  };
}
```

5. The last task in the custom hook implementation is to return the checked IDs and the handler function:

```
export function useChecked() {
  ...
  return { handleCheckChange, checkedIds };
}
```

6. Next, open **`Checklist.tsx`** and remove the state definition and the **`handleCheckChange`** handler function. Also, remove **`useState`** and **`IdValue`** from the **`import`** statements, as they are redundant.

7. Still in **`Checklist.tsx`**, import the **`useChecked`** hook we just created:

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

8. Add a call to **`useChecked`** and destructure the checked IDs and the handler function:

   ```
   export function Checklist<Data>({ ... }:
   Props<Data>) {
     const { checkedIds, handleCheckChange } =
       useChecked();
     return ...
   }
   ```

9. If the app isn't running, run it by entering **`npm run dev`** in the terminal. The checklist component will appear and behave as it did before we made these changes.

That completes the implementation and use of the custom hook. To recap, here are some key points:

- Custom hooks make code a little cleaner and are reusable because they isolate logic, which can be complex.
- Custom hooks must start with **`use`**.
- Custom hooks must use a standard React hook or another custom hook.
- A custom hook is just a function that returns useful things for components to use. Using an object structure for the returned items is ideal when returning many items because the object keys make it clear what each item is.
- A custom hook can have parameters. Using an object structure for the parameters is ideal for many items and doesn't break anything when

new parameters are added.

Next, we will cover a pattern that will allow the consumer of a component to control some of its behavior with the state.

Allowing the internal state to be controlled

In this section, we'll learn how to allow consumers of a component to control its internal state. We will use this pattern in the checklist component so that users can check just a single item.

Understanding how the internal state can be controlled

Allowing consumers of a component to control the state allows the behavior of a component to be tweaked if that behavior is driven by the state. Let's go through an example using the **`useToggle`** custom hook we covered in the last section when learning about custom hooks:

1. Two additional props are required to allow the internal state to be controlled – one for the current state value and one for a change handler. These additional props are **`toggleValue`** and **`onToggleValueChange`** in **`useToggle`**:

   ```
   type Params = {
   ```

```
    defaultToggleValue?: boolean;
    toggleValue?: boolean;
    onToggleValueChange?: (
      toggleValue: Boolean
    ) => void;
  };
  export function useToggle({
    defaultToggleValue,
    toggleValue,
    onToggleValueChange,
  }: Params) {
    ...
  }
```

These props are marked as optional because this pattern doesn't force the consumer of the component to control the state – it's a feature they can opt in to.

> **NOTE**
>
> *The consumer of the component will never specify both* ***`defaultToggleValue`*** *and* ***`toggleValue`****. The* ***`defaultToggleValue`*** *parameter should only be used when the consumer doesn't want to control* ***`toggleValue`*** *with the state. When the consumer does want to control* ***`toggleValue`*** *with the state, they can set the initial value of their state.*

2. The **`toggleValue`** prop now clashes with the **`toggleValue`** state because they have the same name, so the state needs to be renamed:

```
  const [resolvedToggleValue,
  setResolvedToggleValue] =
    useState(defaultToggleValue);
  function toggle() {
    setResolvedToggleValue(!resolvedToggleValue)
  ;
  }
  return { resolvedToggleValue, toggle };
```

3. The default value of the internal state now needs to consider that there might be a prop controlling the state:

```
const [resolvedToggleValue,
setResolvedToggleValue] =
  useState(defaultToggleValue || toggleValue);
```

4. When the state is changed, the change handler is called, if it has been defined:

```
function toggle() {
  if (onToggleValueChange) {
    onToggleValueChange(!resolvedToggleValue);
  } else {
    setResolvedToggleValue(!resolvedToggleValu
e);
  }
}
```

   Again, it's important that we still update the internal state in case the consumer isn't controlling the state.

5. The last step in implementing this pattern is to update the internal state when the controlled state is updated. We can do this with **`useEffect`** as follows:

```
useEffect(() => {
  const isControlled = toggleValue !==
undefined;
  if (isControlled) {
    setResolvedToggleValue(toggleValue);
  }
}, [toggleValue]);
```

The effect is triggered when the state prop changes. We check whether the state prop is being controlled; if so, the internal state is updated with its value.

Here's an example of controlling **`toggleValue`** in **`useToggle`**:

```
const [toggleValue, setToggleValue] =
useState(false);
const onCount = useRef(0);
const { resolvedToggleValue, toggle } =
useToggle({
  toggleValue,
  onToggleValueChange: (value) => {
    if (onCount.current >= 3) {
      setToggleValue(false);
    } else {
      setToggleValue(value);
      if (value) {
        onCount.current++;
      }
    }
  },
});
```

This example stores the toggle value in its own state and passes it to **`useToggle`**. The **`onToggleValueChange`** parameter is handled by updating the state value. The logic for setting the state value only allows it to be **`true`** up to three times.

So, this use case has overridden the default behavior of the toggle so that it can only be set to **`true`** up to three times.

Now that we understand how to allow the internal state to be controlled, we will use it in our checklist component.

Allowing checkedIds to be controlled

At the moment, our checklist component allows many items to be selected. If we allow the **`checkedIds`** state to be controlled by the consumer, they can change the checklist component so that they can select just a single item.

So, carry out the following steps:

1. We will start in **`useChecked.ts`**. Add **`useEffect`** to the React **`import`** statement:

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

2. Add new parameters for the controlled checked IDs and the change handler:

   ```
   export function useChecked({
     checkedIds,
   ```

```
    onCheckedIdsChange,
  }: {
    checkedIds?: IdValue[];
    onCheckedIdsChange?: (checkedIds: IdValue[])
  => void;
  }) {
    ...
  }
```

3. Update the internal state name to **`resolvedCheckedIds`** and default it to the passed-in **`checkedIds`** parameter if defined:

```
  export function useChecked({
    checkedIds,
    onCheckedIdsChange,
  }: Params) {
    const [resolvedCheckedIds,
  setResolvedCheckedIds] =
      useState<IdValue[]>(checkedIds || []);
    const handleCheckChange = (checkedId:
  IdValue) => () => {
      const isChecked =
        resolvedCheckedIds.includes(checkedId);
      let newCheckedIds = isChecked
        ? resolvedCheckedIds.filter(
          (itemCheckedid) => itemCheckedid !==
  checkedId
        )
        : resolvedCheckedIds.concat(checkedId);
      setResolvedCheckedIds(newCheckedIds);
    };
    return { handleCheckChange,
  resolvedCheckedIds };
  }
```

4. Update the **handleCheckChange** handler to call the passed-in change handler if defined:

```
const handleCheckChange = (checkedId: IdValue)
=> () => {
  const isChecked =
resolvedCheckedIds.includes(checkedId);
  let newCheckedIds = isChecked
    ? resolvedCheckedIds.filter(
        (itemCheckedid) => itemCheckedid !==
checkedId
     )
     : resolvedCheckedIds.concat(checkedId);
  if (onCheckedIdsChange) {
    onCheckedIdsChange(newCheckedIds);
  }
  setResolvedCheckedIds(newCheckedIds);
};
```

5. The last task in **useCheck.ts** is to synchronize the controlled checked IDs with the internal state. Add the following **useEffect** hook to achieve this:

```
useEffect(() => {
  const isControlled = checkedIds !==
undefined;
  if (isControlled) {
    setResolvedCheckedIds(checkedIds);
  }
}, [checkedIds]);
```

6. Now, open **Checklist.tsx** and import the **IdValue** type:

```
import { IdValue } from './types';
```

7. Add the new props for the controlled checked IDs and the change handler:

```
type Props<Data> = {
  data: Data[];
  id: keyof Data;
  primary: keyof Data;
  secondary: keyof Data;
  renderItem?: (item: Data) => ReactNode;
  checkedIds?: IdValue[];
  onCheckedIdsChange?: (checkedIds: IdValue[])
=> void;
} & ComponentPropsWithoutRef<'ul'>;
export function Checklist<Data>({
  data,
  id,
  primary,
  secondary,
  renderItem,
  checkedIds,
  onCheckedIdsChange,
  ...ulProps
}: Props<Data>) {}
```

8. Pass these props to **`useChecked`** and destructure and use the **`resolvedCheckedIds`** variable:

```
const { resolvedCheckedIds, handleCheckChange
} = useChecked({
  checkedIds,
  onCheckedIdsChange,
});
return (
  <ul {...ulProps}>
```

```
      {data.map((item) => {
        ...
        return (
          <li ... >
          <label>
          <input
            type="checkbox"
            checked={
              resolvedCheckedIds.includes(idValu
e)
            }
            onChange=
{handleCheckChange(idValue)}
          />
          ..
        </label>
        </li>
      );
    })}
    </ul>
  );
```

9. Open **index.ts** in the **Checklist** folder. Export the **IdValue** type because consumers of the component can now pass in **checkedIds**, which is an array of this type:

```
export type { IdValue } from './types';
```

10. Now, open **App.tsx** and import **useState** from React, as well as the **IdValue** type:

```
import { useState } from 'react';
import {
  Checklist,
    IdValue
  } from './Checklist';
```

11. Define the state in the **`App`** component for the single checked ID:

    ```
    function App() {
      const [checkedId, setCheckedId] =
        useState<IdValue | null>(null);
      ...
    }
    ```

    The state is **`null`** when there is no checked item. This can't be set to **`undefined`** because **`Checklist`** will think **`checkedIds`** is uncontrolled.

12. Create a handler for when an item is checked:

    ```
    function handleCheckedIdsChange(newCheckedIds:
    IdValue[]) {
      const newCheckedIdArr =
    newCheckedIds.filter(
        (id) => id !== checkedId
      );
      if (newCheckedIdArr.length === 1) {
        setCheckedId(newCheckedIdArr[0]);
      } else {
        setCheckedId(null);
      }
    }
    ```

    The handler stores the checked ID in the state or sets the state to **`null`** if the checked item has been unchecked.

13. Pass the checked ID and the change handler to the **`Checklist`** element as follows:

    ```
    <Checklist
      ...
      checkedIds={checkedId === null ? [] :
    [checkedId]}
      onCheckedIdsChange={handleCheckedIdsChange}
    />;
    ```

14. Let's give this a try. If the app isn't running, run it by entering **`npm run dev`** in the terminal. You will find that only a single list item can be checked.

That completes this section on allowing the internal state to be controlled. Here's a recap:

- This pattern is useful because it changes the component's behavior
- The component must expose a prop to control the state value and another for its change handler
- Internally, the component still manages the state and synchronizes it with the consumer's using **`useEffect`**
- If the state is controlled, the consumer's change handler is called in the internal change handler

Let's summarize the chapter now!

Summary

In this chapter, we created a reusable checklist component and used many useful patterns along the way.

We started by learning how to implement generic props, which allow a component to be used with varying data types but still be strongly typed. We used this to allow varying data to be passed into the checklist component without sacrificing type safety.

We learned how to allow consumers of a component to spread props onto an internal element. A common use case is spreading props onto the internal container element to allow the consumer to size it, which is what we did with the checklist component.

The render prop pattern is one of the most useful patterns when developing reusable components. We learned that it allows the consumer to take responsibility for rendering parts of the component. We used this pattern to override the rendering of list items in our checklist component.

Custom hooks isolate logic and are useful for sharing logic across components and keeping the code within a component clean. Custom hooks must call a standard React hook directly or indirectly. We extracted the checked logic from our checklist component into a custom hook.

The last pattern we learned about was allowing a component's internal state to be controlled. This powerful pattern allows the consumer of the component to tweak its behavior. We used this to only allow a single list item to be checked in our checklist component.

In the next chapter, we will learn how to write automated tests for React components.

Questions

Answer the following questions to check what you have learned in this chapter:

1. The snippet of the following component renders a `Select` component containing options:

   ```
   type Props<TOption> = {
     options: TOption[];
     value: string;
     label: string;
   };
   export function Select({
     options,
     value,
     label,
   }: Props<TOption>) {
     return ...
   }
   ```

   The following TypeScript error is raised on the component props parameter though: **Cannot find name 'TOption'**. What is the problem?

2. The **`value`** and **`label`** props from the component in *question 1* should only be set to a property name in the **`options`** value. What type can we give **`value`** and **`label`** so that TypeScript includes them in its type checking?
3. A prop called **`option`** has been added to the **`Select`** component from the previous question as follows:

```
type Props<TOption> = {
  ...,
  option: ReactNode;
};
export function Select<TOption>({
  ...,
  option
}: Props<TOption>) {
  return (
    <div>
      <input />
      {options.map((option) => {
        if (option) {
          return option;
        }
        return ...
      })}
    </div>
  );
}
```

   The **`option`** prop is supposed to allow the consumer of the component to override the rendering of the options. Can you spot the flaw in the implementation?

4. The following is a **`Field`** component that renders a **`label`** element and an **`input`** element:

   ```
   type Props = {
     label: string;
   } & ComponentPropsWithoutRef<'input'>;
   export function Field({ ...inputProps, label
   }: Props) {
     return (
       <>
         <label>{label}</label>
         <input {...inputProps} />
       </>
     );
   }
   ```

   There is a problem with the implementation though – can you spot it?

5. How many render props can a component have?

Answers

1. The generic type must be defined in the component function as well as the prop:

   ```
   export function Select<TOption>({
     options,
     value,
     label,
   }: Props<TOption>) {
     return ...
   }
   ```

2. The **keyof** operator can be used to ensure **value** and **label** are keys in **options**:

```
type Props<TOption> = {
  options: TOption[];
  value: keyof TOption;
  label: keyof TOption;
};
```

3. The consumer is likely to need the data for the option, so the prop should be a function containing the data as a parameter:

```
type Props<TOption> = {
  ...,
  renderOption: (option: TOption) =>
ReactNode;
};
export function Select<TOption>({
  options,
  value,
  label,
  renderOption,
}: Props<TOption>) {
  return (
    <div>
      <input />
      {options.map((option) => {
        if (renderOption) {
          return renderOption(option);
        }
        return ...
    </div>
  );
}
```

4. There is a syntax error because the rest parameter is the first parameter. The rest parameter must be the last one:

   ```
   export function Field({ label, ...inputProps
   }: Props) {
     ...
   }
   ```

5. There is no limit on the number of render props a component can have.

12
Unit Testing with Vitest and the React Testing Library

In this chapter, we will learn how to use Vitest and the React Testing Library, two popular automated testing tools that can be used together in React applications. We will create tests on the checklist component we created in *Chapter 11*, *Reusable Components*.

We will start by focusing on Vitest and using it to test simple functions, learning about Vitest's common **matcher** functions for writing expectations, and how to execute tests to check whether they pass.

We will then move on to learning about component testing using the React Testing Library. We'll understand the different query types and variants and how they help us create robust tests.

After that, we will learn about the most accurate way to simulate user interactions using a React Testing Library companion package. We use this to create tests for items being checked in the checklist component.

At the end of the chapter, we will learn how to determine which code is covered by tests and, more importantly, which code is uncovered. We'll use Vitest's code coverage tool to do this and understand all the different coverage stats it gives us.

So, in this chapter, we'll cover the following topics:

- Testing pure functions
- Testing components
- Simulating user interactions
- Getting code coverage

Technical requirements

We will use the following technologies in this chapter:

- **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/

We will start with a modified version of the code we finished in the last chapter. The modified code contains logic extracted into pure functions, which will be ideal to use in the first tests we write. This code can be found online at https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter12/start.

Carry out the following steps to download this to your local computer:

1. Go to https://download-directory.github.io/ in a browser.
2. In the textbox on the web page, enter the following URL: https://github.com/PacktPublishing/Learn-React-with-TypeScript-Third-Edition/tree/main/Chapter12/start.
3. Press the *Enter* key. A ZIP file containing the **`start`** folder will now be downloaded.
4. Extract the ZIP file to a folder of your choice and open that folder in Visual Studio Code.
5. Go to this folder in a terminal and execute the following command to install all the dependencies:

   ```
   npm i
   ```

You are now ready to start writing tests for the checklist component.

Testing pure functions

In this section, we will start by understanding the fundamental parts of a Vitest test. Then, we will put this into practice by implementing tests on a pure function in the checklist component.

A pure function has a consistent output value for a given set of parameter values. These functions depend only on the function parameters and nothing outside the function, and also don't change any argument values passed into them. So, pure functions are nice for learning how to write tests because they have no tricky side effects to deal with.

In this section, we will also cover how to test exceptions, which is useful for testing type assertion functions. Finally, at the end of this section, we will learn how to run the tests in a test suite.

Understanding a test

Vitest looks for tests in files with particular extensions. These file extensions are **`.test.ts`** for tests on pure functions and **`.test.tsx`** for tests on components. Alternatively, a **`.spec.*`** file extension could be used.

A test is defined using Vitest's **`test`** function:

```
test('your test name', () => {
  // your test implementation
});
```

The **`test`** function has two parameters for the test name and implementation. It is common practice for the test implementation to be an anonymous function. The test implementation can be asynchronous by placing the **`async`** keyword in front of the anonymous function:

```
test('your test name', async () => {
  // your test implementation
});
```

The test implementation will consist of calling the function with arguments being tested and checking the result is as we expect:

```
test('your test name', async () => {
  const someResult =
yourFunction('someArgument');
  expect(someResult).toBe('something');
});
```

Vitest's **`expect`** function is used to define our expectations. The result of the function call is passed into **`expect`**, and it returns an object containing methods that we can use to define specific expectations for the result. These methods are referred to as **matchers**. If the expectation fails, Vitest will fail the test.

The preceding test uses the **`toBe`** matcher. The **`toBe`** matcher checks that primitive values are equal, and the preceding test uses it to check that the **`someResult`** variable is equal to **`'something'`**. Other common matchers are as follows:

- **`toStrictEqual`** for checking the values in an object or array. This recursively checks every property in the object or array. Here's an example:

```
expect(someResult).toStrictEqual({
  field1: 'something',
  field2: 'something else'
});
```

- **`not`** for checking the opposite of a matcher. Here's an example:

```
expect(someResult).not.toBe('something');
```

- **`toMatch`** for checking strings against **regular expressions** (**regexes**). Here's an example:

```
expect(someResult).toMatch(/error/);
```

- **`toContain`** for checking whether an element is in an array. Here's an example:

```
expect(someResult).toContain(99);
```

A complete list of all the standard matchers can be found in the Vitest documentation at https://vitest.dev/api/expect.html#expect.

Now that we understand the basics of a Vitest test, we will create our first Vitest test.

Testing isChecked

The first function we will test is **`isChecked`**. This function has two parameters:

- **`checkedIds`**: This is an array of IDs that are currently checked
- **`idValue`**: This is the ID to determine whether it is checked

We will write a test for when the list item is checked and another for when it isn't checked:

1. Start by installing Vitest by entering the following command in a terminal:

    ```
    npm i -D vitest
    ```

2. We will configure Vitest so that we can use its functions such as **`test`** and **`expect`** globally, without having to import them. Open the Vite configuration file, **`vite.config.ts`**, which is in the project root. We need to create a Vitest configuration and merge it into the Vite configuration. So, import a **`mergeConfig`** function from Vite and a **`defineConfig`** function from Vitest as follows:

    ```
    import { defineConfig, mergeConfig } from
    "vite";
    import {
      defineConfig as defineVitestConfig
    } from "vitest/config";
    ```

    We have aliased the Vitest **`defineConfig`** function so that it doesn't collide with the one from Vite.

3. Adjust the Vite configuration to be assigned to a variable and create the Vitest configuration:

```
const viteConfig = defineConfig({
  plugins: [react()],
});
const vitestConfig = defineVitestConfig({
  test: {
    globals: true,
  },
});
```

   We have configured the Vitest APIs to be available globally by setting `test.globals` to `true`.

4. The last step in the `vite.config.ts` file is to merge the configurations and export them:

```
export default mergeConfig(viteConfig,
vitestConfig);
```

5. To make TypeScript aware of the global APIs, we need to update the TypeScript configuration file. Open `tsconfig.app.json` from the project root and add the Vitest global types as follows:

```
{
  "compilerOptions": {
    ...,
    «types»: [«vitest/globals»]
  },
  ...
}
```

6. On to writing a test now. Create a file called **`isChecked.test.ts`** in the **`src/Checklist`** folder that will contain the tests.

> **NOTE**
>
> *It is best practice to place test files adjacent to the source file being tested. This allows the developers to navigate to the test for a function quickly.*

7. Open **`isChecked.test.ts`** and import the **`isChecked`** function:

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

8. Start to create the first test as follows:

```
test('', () => {
});
```

9. Add the test name as follows:

```
test('should return true when in checkedIds',
() => {
);
```

   Forming a naming convention for test names is good practice so that they are consistent and easy to understand. Here, we have used the following naming structure:

   `should {expected output / behaviour} when {input / state condition}`

10. Now, let's start to implement the logic inside the test. The first step in the test is to call the function being tested with the arguments we want

to test:

```
test('should return true when in checkedIds',
() => {
  const result = isChecked([1, 2, 3], 2);
});
```

11. The second (and last) step in the test is to check that the result is what we expect, which is **`true`** for this test:

```
test('should return true when in checkedIds',
() => {
  const result = isChecked([1, 2, 3], 2);
  expect(result).toBe(true);
});
```

Since the result is a primitive value (a Boolean), we use the **`toBe`** matcher to verify the result.

12. Add a second test to cover the case when the ID isn't in the checked IDs:

```
test('should return false when not in
checkedIds', () => {
  const result = isChecked([1, 2, 3], 4);
  expect(result).toBe(false);
});
```

That completes the tests on the **`isChecked`** function. Next, we will learn how to test exceptions that are raised. We will check that our tests work after that.

Testing exceptions

We are going to test the **`assertValueCanBeRendered`** type assertion function. This is a little different from the last function we tested because we want to test whether an exception is raised, rather than the returned value.

Vitest has a **`toThrow`** matcher that can be used to check whether an exception has been raised. For this to catch exceptions, the function being tested has to be executed inside the expectation, as follows:

```
test('some test', () => {
  expect(() => {
    someAssertionFunction(someValue);
  }).toThrow('some error message');
});
```

We will use this approach to add three tests on the **`assertValueCanBeRendered`** type assertion function. Carry out the following steps:

1. Create a file called **`assertValueCanBeRendered.test.ts`** in the **`src/Checklist`** folder for the tests. Import the **`assertValueCanBeRendered`** type assertion function:

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

2. The first test we will add is to check whether an exception is raised when the value isn't a string or number:

```
test('should raise exception when not a string
or number', () => {
  expect(() => {
    assertValueCanBeRendered(
      true
    );
  }).toThrow(
    'value is not a string or a number'
  );
});
```

   We pass the **`true`** Boolean value, which should cause an error.

3. Next, we will test whether an exception isn't raised when the value is a string:

```
test('should not raise exception when string',
() => {
  expect(() => {
    assertValueCanBeRendered(
      'something'
    );
  }).not.toThrow();
});
```

   We use the **`not`** matcher with **`toThrow`** to check that an exception is not raised.

4. The final test will verify that no exception is raised when the value is a number:

```
test('should not raise exception when number',
() => {
  expect(() => {
    assertValueCanBeRendered(
      99
    );
  }).not.toThrow();
});
```

That completes the tests for the **`assertValueCanBeRendered`** type assertion function.

Now that we have implemented some tests, we will learn how to run them next.

Running tests

In order to run the tests, we simply run Vitest. After the tests are run, a watcher will rerun the tests when the source code or test code changes.

Carry out the following steps to run all the tests:

1. First, add the following **`test`** script to the **`package.json`** file in the **`scripts`** section. This **`npm`** command will execute Vitest, which will run the tests:

```
{
  ...,
  "scripts": {
    ...
```

```
        «test»: «vitest»
      },
      ...
    }
```

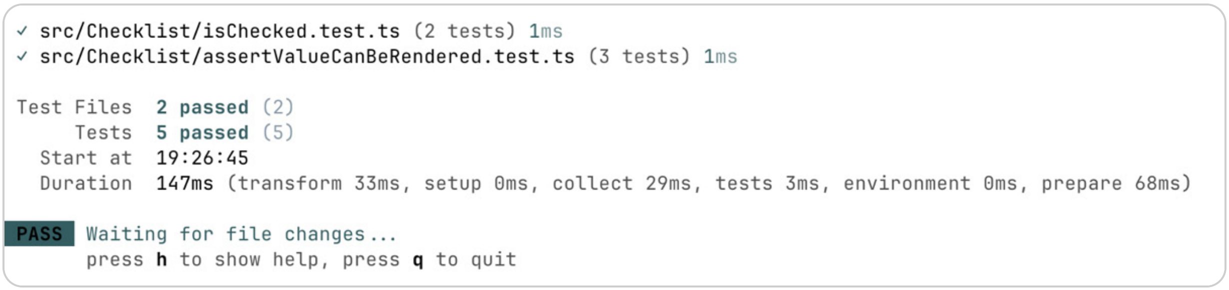
2. Open the terminal and execute the following command:

   ```
   npm run test
   ```

   `test` is a very common **`npm`** script, so the **`run`** keyword can be omitted. In addition, **`test`** can be shortened to **`t`**. So, a shortened version of the previous command is as follows:

   ```
   npm t
   ```

   The tests will be run, and the following summary will appear in the terminal:



```
✓ src/Checklist/isChecked.test.ts (2 tests) 1ms
✓ src/Checklist/assertValueCanBeRendered.test.ts (3 tests) 1ms

Test Files  2 passed (2)
     Tests  5 passed (5)
  Start at  19:26:45
  Duration  147ms (transform 33ms, setup 0ms, collect 29ms, tests 3ms, environment 0ms, prepare 68ms)

 PASS  Waiting for file changes...
       press h to show help, press q to quit
```

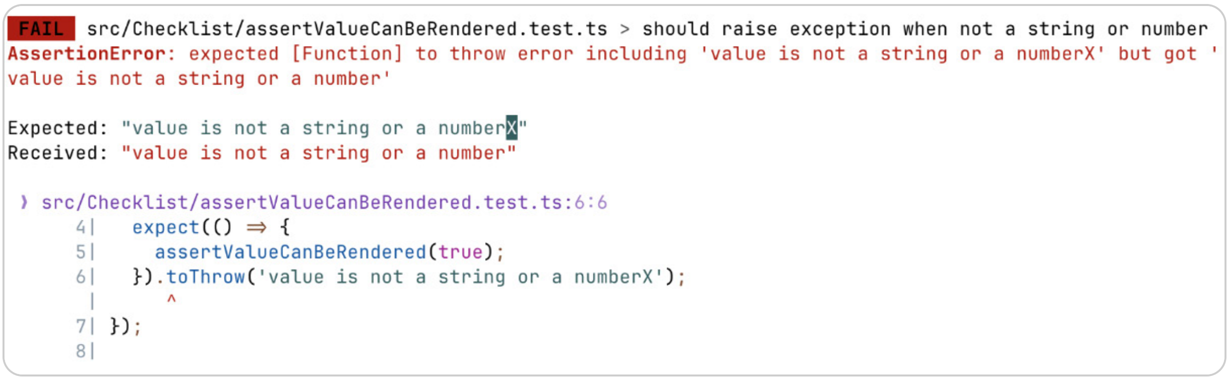
Figure 12.1 – First test run

Notice that there is no Command Prompt in the terminal like there usually is after a command has finished executing. This is because the command hasn't fully completed as the test watcher is running —this is called **watch mode**. The command won't complete until watch mode is exited using the *Q* key. Leave the terminal in watch mode and carry on to the next step.

3. All the tests pass at the moment. Now, we will deliberately make a test fail so that we can see the information Vitest provides us. So, open **`assertValueCanBeRendered.ts`** and change the expected error message on the first test as follows:

```
test('should raise exception when not a string
or number', () => {
  expect(() => {
    assertValueCanBeRendered(true);
  }).toThrow('value is not a string or a
numberX');
});
```

As soon as the test file is saved, the tests are rerun, and a failing test is reported as follows:



```
 FAIL  src/Checklist/assertValueCanBeRendered.test.ts > should raise exception when not a string or number
AssertionError: expected [Function] to throw error including 'value is not a string or a numberX' but got '
value is not a string or a number'

Expected: "value is not a string or a numberX"
Received: "value is not a string or a number"

 ❯ src/Checklist/assertValueCanBeRendered.test.ts:6:6
      4|   expect(() ⇒ {
      5|     assertValueCanBeRendered(true);
      6|   }).toThrow('value is not a string or a numberX');
       |      ^
      7| });
      8|
```

Figure 12.2 – Failing test

Vitest provides valuable information about the failure that helps us quickly resolve test failures. It tells us this:

- Which test failed
- What the expected result was, in comparison to the actual result

- The line in our code where the failure occurred

4. Resolve the test failure by reverting the test to check for the correct error message. The test should be as follows now:

```
test('should raise exception when not a string
or number', () => {
  expect(() => {
    assertValueCanBeRendered(true);
  }).toThrow('value is not a string or a
number');
});
```

That completes our exploration of running Vitest tests and this section on testing pure functions. Here's a quick recap of the key points:

- Tests are defined using Vitest's **`test`** function
- Expectations within the test are defined using Vitest's **`expect`** function in combination with one or more matchers
- The **`expect`** function argument can be a function that executes the function being tested. This is useful for testing exceptions with the **`toThrow`** matcher

Next, we will learn how to test React components.

Testing components

Testing components is important because this is what the user interacts with. Having automated tests on components gives us confidence that the app is working correctly and helps prevent regressions when we change code.

In this section, we will learn how to test components with Vitest and the React Testing Library. Then, we will create some tests on the checklist component we developed in the last chapter.

Understanding the React Testing Library

The React Testing Library is a popular library for testing React components. It provides functions to render components and then select internal elements. Those internal elements can then be checked using special matchers provided by another companion library, called **`jest-dom`**.

A basic component test

Here's an example of a component test:

```
test('should render heading when content
specified', () => {
  render(<Heading>Some heading</Heading>);
  const heading = screen.getByText('Some
heading');
  expect(heading).toBeInTheDocument();
});
```

Let's explain the test:

- React Testing Library's **`render`** function renders the component we want to test. We pass in all the appropriate attributes and content so that the component is in the required state for the checks. In this test, we have specified some text in the content.
- The next line selects an internal element of the component. There are lots of methods on React Testing Library's **`screen`** object that allow the selection of elements. These methods are referred to as **queries**. **`getByText`** selects an element by matching the text content specified. In this test, an element with **`Some heading`** text content will be selected and assigned to the **`heading`** variable.
- The last line in the test is the expectation. The **`toBeInTheDocument`** matcher is a special matcher from **`jest-dom`** that checks whether the element in the expectation is in the DOM.

Understanding queries

A React Testing Library query is a method that selects a DOM element within the component being rendered. There are many different queries that find the element in different ways:

- **`ByRole`**: Queries elements by their role.

> ***NOTE***
>
> *DOM elements have a **`role`** attribute that allows assistive technologies such as screen readers to understand what they are. Many DOM elements have*

> *this attribute preset—for example, the* ***`button`*** *element automatically has the role of* ***`'button'`****. For more information on roles, see* *https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles.*

- **`ByLabelText`**: Queries elements by their associated label. See this page in the React Testing Library documentation for the different ways elements can be associated with a label: https://testing-library.com/docs/queries/bylabeltext.
- **`ByPlaceholderText`**: Queries elements by their placeholder text.
- **`ByText`**: Queries elements by their text content.
- **`ByDisplayValue`**: Queries **`input`**, **`textarea`**, and **`select`** elements by their value.
- **`ByAltText`**: Queries **`img`** elements by their **`alt`** attribute.
- **`ByTitle`**: Queries elements by their **`title`** attribute.
- **`ByTestId`**: Queries elements by their test ID (the **`data-testid`** attribute).

There are also different types of queries that behave slightly differently on the found element. Each query type has a particular prefix on the query method name:

- **`getBy`**: Throws an error if a single element is not found. This is ideal for synchronously getting a single element.
- **`getAllBy`**: Throws an error if at least one element is not found. This is ideal for synchronously getting multiple elements.

- **`findBy`**: Throws an error if a single element is not found. The check for an element is repeated for a certain amount of time (one second by default). So, this is ideal for asynchronously getting a single element that might not be immediately in the DOM.
- **`findAllBy`**: Throws an error if at least one element is not found within a certain time (one second by default). This is ideal for asynchronously getting multiple elements that might not be immediately in the DOM.
- **`queryBy`**: This returns **`null`** if an element is not found. This is ideal for checking that an element does *not* exist.
- **`queryAllBy`**: This is the same as **`queryBy`**, but returns an array of elements. This is ideal for checking multiple elements do *not* exist.

So, the **`getByText`** query we used in the preceding test finds the element by the text content specified and raises an error if no elements are found.

For more information on queries, see the following page in the React Testing Library documentation: https://testing-library.com/docs/queries/about/.

Notice that none of these queries references implementation details such as an element name, ID, or CSS class. If those implementation details change due to code refactoring, the tests shouldn't break, which is precisely what we want.

Now that we understand what the React Testing Library is, we will use it to write our first component test.

Implementing checklist component tests

The first component test we will write is to check that list items are rendered correctly. The second component test will check that list items are rendered correctly when custom rendered. Carry out the following steps:

1. Start by installing the React Testing Library and **`jest-dom`** by executing the following command in a terminal:

   ```
   npm i -D @testing-library/react @testing-
   library/dom @testing-library/jest-dom jsdom
   ```

2. We now need to configure Vitest to use a DOM environment for the tests rather than the default node environment. Open the Vite configuration file, **`vite.config.ts`**, and configure Vitest to use the **`jsdom`** library that simulates the DOM:

   ```
   const vitestConfig = defineVitestConfig({
     test: {
       ...,
       environment: "jsdom",
     },
   });
   ```

3. In the configuration, we will also automatically import the **`jest-dom`** library so that we don't need to do this in every test file. Add a setup file

to run before the tests in the Vite configuration file as follows:

```
const vitestConfig = defineVitestConfig({
  test: {
    ...,
    setupFiles: './vitest.setup.ts',
  },
});
```

4. Create the **`vitest.setup.ts`** file in the project root with the **`jest-dom`** import statement:

```
import '@testing-library/jest-dom/vitest';
```

5. Open **`tsconfig.app.json`** and make TypeScript aware of the **`jest-dom`** types as follows:

```
{
  "compilerOptions": {
     ...
    "types": [
      "vitest/globals",
      "@testing-library/jest-dom"
    ]
  },
  ...
}
```

6. Create a new file in the **`src/Checklist`** folder called **`Checklist.test.tsx`** and add the following import statements:

```
import { expect, test } from "vitest";
import {
  render,
    screen
  } from '@testing-library/react';
  import { Checklist } from './Checklist';
```

7. Start to create the test as follows:

```
  test('should render correct list items when
  data
        specified', () => {
  });
```

8. In the test, render **Checklist** with some data:

```
  test('should render correct list items when
  data specified', () => {
    render(
      <Checklist
        data={[{ id: 1, name: 'Lucy', role:
  'Manager' }]}
        id="id"
        primary="name"
        secondary="role"
      />
    );
  });
```

We've rendered a single list item that should have the primary text **Lucy** and the secondary text **Manager**.

9. Let's check that **Lucy** has been rendered:

```
  test('should render correct list items when
  data specified', () => {
    render( ... );
    expect(
      screen.getByText('Lucy')
    ).toBeInTheDocument();
  });
```

We have selected the element using the **`getByText`** query and fed that directly into the expectation. We use the **`toBeInTheDocument`** matcher to check that the found element is in the DOM.

10. Complete the test by adding a similar expectation for checking for **`Manager`**:

```
  test('should render correct list items when
  data specified', () => {
    render( ... );
    expect( ... ).toBeInTheDocument();
    expect(
      screen.getByText('Manager')
    ).toBeInTheDocument();
  });
```

That completes our first component test.

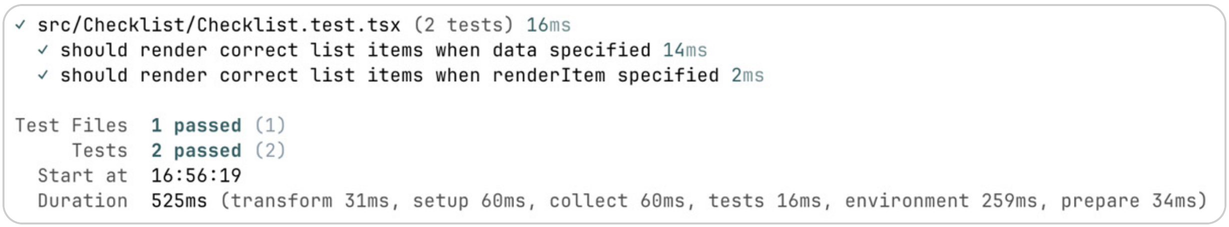
11. We will add the second test in one go, as follows:

```
  test('should render correct list items when
  renderItem specified', () => {
    render(
      <Checklist
        data={[{ id: 1, name: 'Lucy', role:
  'Manager' }]}
        id="id"
        primary="name"
        secondary="role"
```

```
          renderItem={(item) => (
            <li key={item.id}>
              {item.name}-{item.role}
            </li>
          )}
        />
      );
      expect(
        screen.getByText('Lucy-Manager')
      ).toBeInTheDocument();
    });
```

We render a single list item with the same data as the previous test. However, this test custom renders the list items with a hyphen between the name and role. We use the same **`getByText`** query to check that the list item with the correct text is found in the DOM.

12. If the tests aren't automatically running, run them by running **`npm test`** in the terminal. The two new tests should both pass:



```
✓ src/Checklist/Checklist.test.tsx (2 tests) 16ms
  ✓ should render correct list items when data specified 14ms
  ✓ should render correct list items when renderItem specified 2ms

Test Files  1 passed (1)
     Tests  2 passed (2)
  Start at  16:56:19
  Duration  525ms (transform 31ms, setup 60ms, collect 60ms, tests 16ms, environment 259ms, prepare 34ms)
```

Figure 12.3 – Component tests passing

That completes our first two component tests. See how easy React Testing Library makes this!

Using test IDs

The next test we will implement is to check that a list item is checked when specified. This test will be slightly trickier and requires a test ID on the checkboxes. Carry out the following steps:

1. Start by opening **`Checklist.tsx`** and notice the following test ID on the **`input`** element:

   ```
   <input
     ...
     data-testid={`Checklist__input__${
       idValue.toString()}`}
   />
   ```

   Test IDs are added to elements using a **`data-testid`** attribute. The list item ID is included in the ID so that it is unique for each list item.

2. Now, return to the **`Checklist.test.tsx`** file and begin to write the following new test:

   ```
   test('should render correct checked items when
   specified', () => {
     render(
       <Checklist
         data={[{ id: 1, name: 'Lucy', role:
   'Manager'
         }]}
         id="id"
         primary="name"
         secondary="role"
         checkedIds={[1]}
       />
   ```

```
    );
  });
```

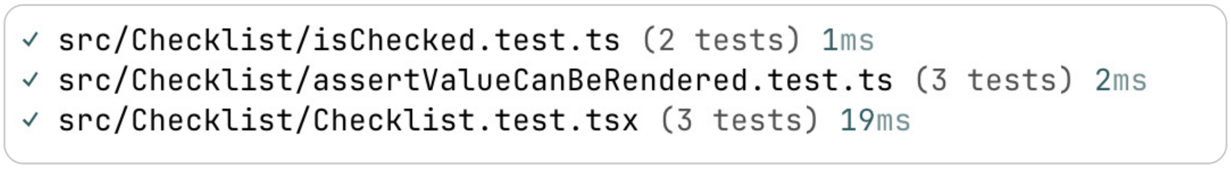
We have rendered the checklist with the same data as the previous tests. However, we have specified that the list item is checked using the **`checkedIds`** prop.

3. Now, on to the expectation for the test:

```
test('should render correct checked items when
specified', () => {
  render(
    <Checklist
      data={[{ id: 1, name: 'Lucy', role:
'Manager'
      }]}
      id="id"
      primary="name"
      secondary="role"
      checkedIds={[1]}
    />
  );
  expect(
    screen.getByTestId('Checklist__input__1')
  ).toBeChecked();
});
```

We select the checkbox by its test ID using the **`getByTestId`** query. We then use the **`toBeChecked`** matcher to verify that the checkbox is checked. **`toBeChecked`** is another special matcher from the **`Vitest-dom`** package.

This new test should pass, leaving us with three passing tests on **`Checklist`**:



```
✓ src/Checklist/isChecked.test.ts (2 tests) 1ms
✓ src/Checklist/assertValueCanBeRendered.test.ts (3 tests) 2ms
✓ src/Checklist/Checklist.test.tsx (3 tests) 19ms
```

Figure 12.4 – All three component tests passing

4. Stop the test runner by pressing the *Q* key.

That completes this section on testing components. Here's a quick recap:

- The React Testing Library contains lots of useful queries for selecting DOM elements. Different query types will find single or many elements and will or won't error if an element isn't found. There is even a query type for repeatedly searching for elements rendered asynchronously.
- **`jest-dom`** contains lots of useful matchers for checking DOM elements. A common matcher is **`toBeInTheDocument`**, which verifies an element is in the DOM. However, **`jest-dom`** contains many other useful matchers, such as **`toBeChecked`** for checking whether an element is checked or not.

Next, we will learn how to simulate user interactions in tests.

Simulating user interactions

So far, our tests have simply rendered the checklist component with various props set. Users can interact with the checklist component by checking and unchecking items. In this section, we will first learn how to simulate user interactions in tests. We will then use this knowledge to test whether list items are checked when clicked and that **`onCheckedIdsChange`** is raised.

Understanding fireEvent and user-event

The React Testing Library has a **`fireEvent`** function that can raise events on DOM elements. The following example raises a **`click`** event on a **Save** button:

```
render(<button>Save</button>);
fireEvent.click(screen.getByText('Save'));
```

This is okay, but what if logic was implemented using a **`mousedown`** event rather than **`click`**? The test would then need to be as follows:

```
render(<button>Save</button>);
fireEvent.mouseDown(screen.getByText('Save'));
```

Fortunately, there is an alternative approach to performing user interactions in tests. The alternative approach is to use the **`user-event`** package, which is a React Testing Library companion package that simulates user interactions rather than specific low-level DOM events. The same test using **`user-event`** looks like this:

```
const user = userEvent.setup();
render(<button>Save</button>);
await user.click(screen.getByText('Save'));
```

The `user-event` package will dispatch the appropriate low-level DOM events, in the appropriate order. So, the test would cover logic implemented using a `click` event or `mousedown` event. So, it is less coupled to implementation details, which is good. For this reason, we'll use the `user-event` package to write interactive tests on our checklist component.

The `user-event` package can simulate interactions other than clicks. See the documentation at the following link for more information: https://testing-library.com/docs/user-event/intro.

Implementing checklist tests for checking items

We will now write two interactive tests on the checklist component. The first test will check items are checked when clicked. The second test will check `onCheckedIdsChange` is called when items are clicked. Carry out the following steps:

1. Install the `user-event` package by executing the following command in a terminal:

   ```
   npm i -D @testing-library/user-event
   ```

2. We will add the interactive tests in the same test file as the other component tests. So, open **`Checklist.test.tsx`** and add an import statement for **`user-event`**:

   ```
   import userEvent from '@testing-library/user-
   event';
   ```

3. The first test will test that items are checked when clicked. Start to implement this as follows at the bottom of the file:

   ```
   test('should check items when clicked', async
   () => {
   });
   ```

   We have marked the test as asynchronous because the simulated user interactions in **`user-event`** are asynchronous.

4. Next, initialize the user simulation as follows:

   ```
   test('should check items when clicked', async
   () => {
     const user = userEvent.setup();
   });
   ```

5. We can now render a list item as we have done in previous tests. We will also get a reference to the checkbox in the rendered list item and check that it isn't checked:

   ```
   test('should check items when clicked', async
   () => {
     const user = userEvent.setup();
     render(
       <Checklist
   ```

```
      data={[{ id: 1, name: 'Lucy', role:
'Manager' }]}
      id="id"
      primary="name"
      secondary="role"
    />
  );
  const lucyCheckbox = screen.getByTestId(
    'Checklist__input__1'
  );
  expect(lucyCheckbox).not.toBeChecked();
});
```

6. Now, on to the user interaction. Simulate the user clicking the list item by calling the **click** method on the **user** object; the checkbox to be clicked needs to be passed into the **click** argument:

```
test('should check items when clicked', async
() => {
  const user = userEvent.setup();
  render( ... );
  const lucyCheckbox = screen.getByTestId(
    'Checklist__input__1'
  );
  expect(lucyCheckbox).not.toBeChecked();
  await user.click(lucyCheckbox);
});
```

7. The last step in the test is to check that the checkbox is now checked:

```
test('should check items when clicked', async
() => {
  const user = userEvent.setup();
  render( ... );
  const lucyCheckbox = screen.getByTestId(
      'Checklist__input__1'
    );
    expect(lucyCheckbox).not.toBeChecked();
    await user.click(lucyCheckbox);
    expect(lucyCheckbox).toBeChecked();
  });
```

8. The next test will test that the function assigned to the **`onCheckedIdsChange`** prop is called when a list item is clicked. Here is the test:

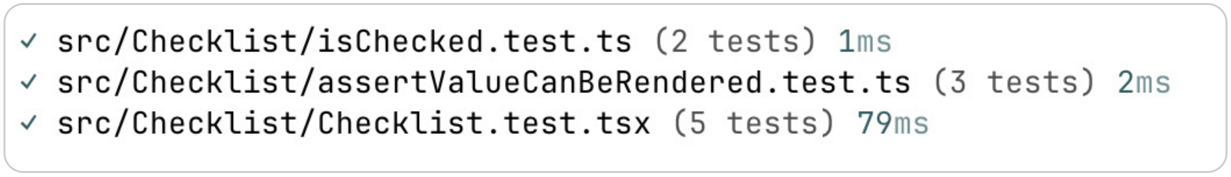
```
  test('should call onCheckedIdsChange when
  clicked', async () => {
    const user = userEvent.setup();
    let calledWith: IdValue[] | undefined =
  undefined;
    render(
      <Checklist
        data={[{ id: 1, name: 'Lucy', role:
  'Manager' }]}
        id="id"
        primary="name"
        secondary="role"
        onCheckedIdsChange={(checkedIds) =>
          (calledWith = checkedIds)
        }
      />
    );
    await user.click(screen.getByTestId(
      'Checklist__input__1'));
    expect(calledWith).toStrictEqual([1]);
  });
```

We set a **`calledWith`** variable to the value of the **`onCheckedIdsChange`** parameter. After the list item is clicked, we check the value of the **`calledWith`** variable using the **`toStrictEqual`** matcher. The **`toStrictEqual`** matcher is a standard matcher that is ideal for checking arrays and objects.

9. The second test references the **`IdValue`** type, so add an import statement for this:

```
import { IdValue } from './types';
```

10. Run the tests by running **`npm test`** in the terminal. We should now have five passing component tests:



```
✓ src/Checklist/isChecked.test.ts (2 tests) 1ms
✓ src/Checklist/assertValueCanBeRendered.test.ts (3 tests) 2ms
✓ src/Checklist/Checklist.test.tsx (5 tests) 79ms
```

Figure 12.5 – Five passing component tests

11. Stop the test runner by pressing the *Q* key.

That completes the tests for clicking items and this section on simulating user interactions. We learned that the React Testing Library's **`fireAction`** function raises a particular event that couples tests to implementation details. A better approach is to use the **`user-event`** package to simulate user interactions, potentially raising several events in the process.

Next, we will learn how to quickly determine any code that isn't covered by tests.

Getting code coverage

Code coverage is how we refer to how much of our app code is covered by unit tests. As we write our unit tests, we'll have a fair idea of what code is covered and not covered, but as the app grows and time passes, we'll lose track of this.

In this section, we'll learn how to use Vitest's code coverage option so that we don't have to keep what is covered in our heads. We will use the code coverage option to determine the code coverage on the checklist component and understand all the different statistics in the report. We will use the code coverage report to find some uncovered code in our checklist component. We will then extend the tests on the checklist component to achieve full code coverage.

Installing the code coverage tool

The default code coverage tool for Vitest is **`coverage-v8`**. To install this in our project, execute the following command in a terminal:

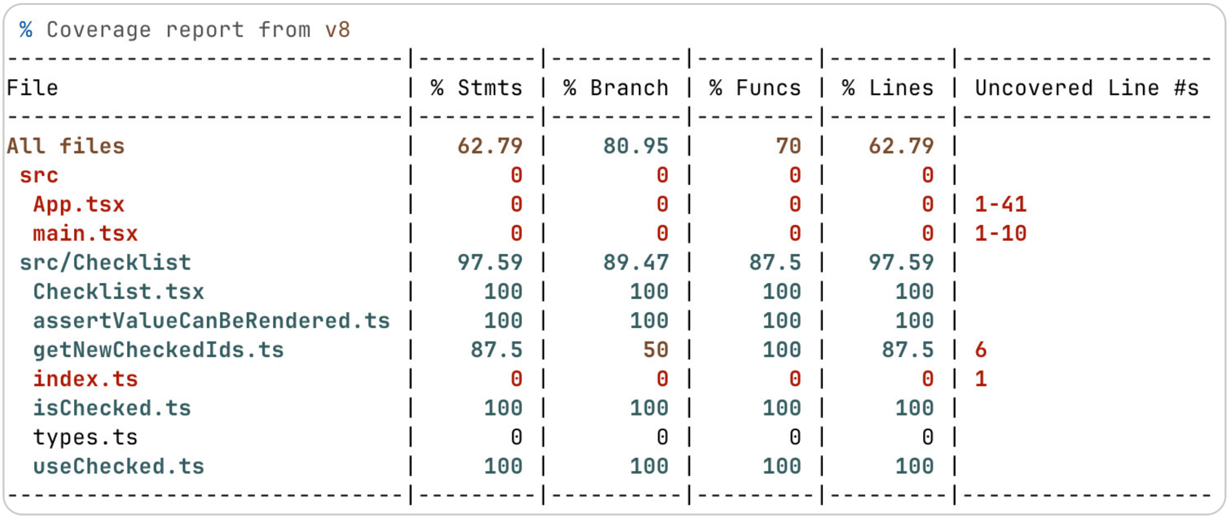
```
npm i -D @vitest/coverage-v8
```

Running code coverage

To get code coverage, we run the **`test`** command with a **`--coverage`** option. We also include a **`--watch=false`** option that tells Vitest not to run in **`watch`** mode. So, run the following command in a terminal to determine code coverage on our app:

```
npm test -- --coverage --watch=false
```

The tests take a little longer to run because of the code coverage calculations. When the tests have finished, a code coverage report is output in the terminal with the test results:



```
 % Coverage report from v8
------------------------------|---------|----------|---------|---------|-------------------
File                          | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
------------------------------|---------|----------|---------|---------|-------------------
All files                     |   62.79 |    80.95 |      70 |   62.79 |
 src                          |       0 |        0 |       0 |       0 |
  App.tsx                     |       0 |        0 |       0 |       0 | 1-41
  main.tsx                    |       0 |        0 |       0 |       0 | 1-10
 src/Checklist                |   97.59 |    89.47 |    87.5 |   97.59 |
  Checklist.tsx               |     100 |      100 |     100 |     100 |
  assertValueCanBeRendered.ts |     100 |      100 |     100 |     100 |
  getNewCheckedIds.ts         |    87.5 |       50 |     100 |    87.5 | 6
  index.ts                    |       0 |        0 |       0 |       0 | 1
  isChecked.ts                |     100 |      100 |     100 |     100 |
  types.ts                    |       0 |        0 |       0 |       0 |
  useChecked.ts               |     100 |      100 |     100 |     100 |
------------------------------|---------|----------|---------|---------|-------------------
```

Figure 12.6 – Terminal code coverage report

Next, we will take some time to understand this code coverage report.

Understanding the code coverage report

The coverage report lists the coverage for each file and aggregates coverage in a folder for all the files in the project. So, the whole app has between 62.79% and 80.95% code coverage, depending on which statistic we take.

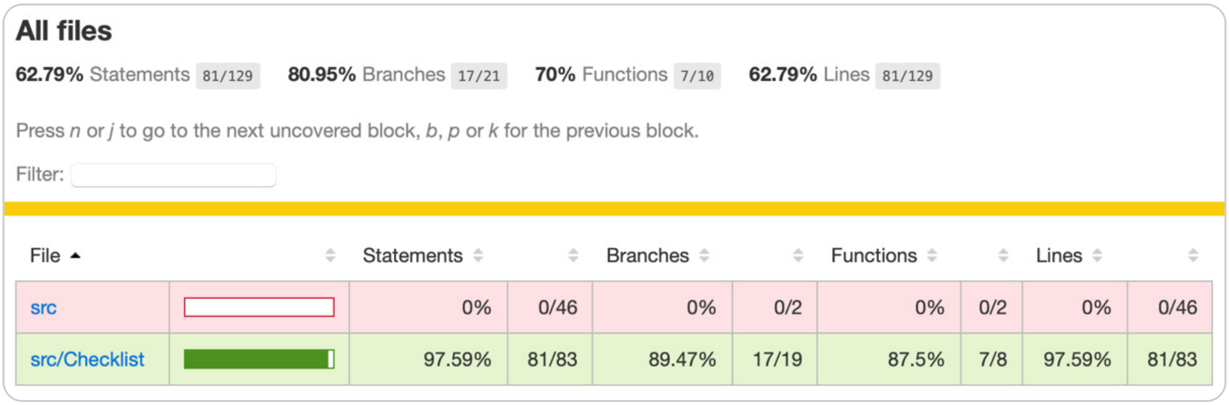
Here's an explanation of all the statistic columns:

- **% Stmts**: This is **statement coverage**, which is how many source code statements have been executed during test execution
- **% Branch**: This is **branch coverage**, which is how many of the branches of conditional logic have been executed during test execution
- **% Funcs**: This is **function coverage**, which is how many functions have been called during test execution
- **% Lines**: This is **line coverage**, which is how many lines of source code have been executed during test execution

The rightmost column in the report is very useful. It gives the lines of source code that aren't covered by tests. For example, the **`getNewCheckedIds.ts`** file in the checklist component has line 6, which is uncovered.

There is another version of the report that is generated in HTML format. This file is automatically generated every time a test is run with the **`--coverage`** option. So, this report has already been generated because we have just run the tests with the **`--coverage`** option. Carry out the following steps to explore the HTML report:

1. The report can be found in an **index.html** file in a **coverage** folder. Double-click on the file so that it opens in a browser:



All files

62.79% Statements 81/129 80.95% Branches 17/21 70% Functions 7/10 62.79% Lines 81/129

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

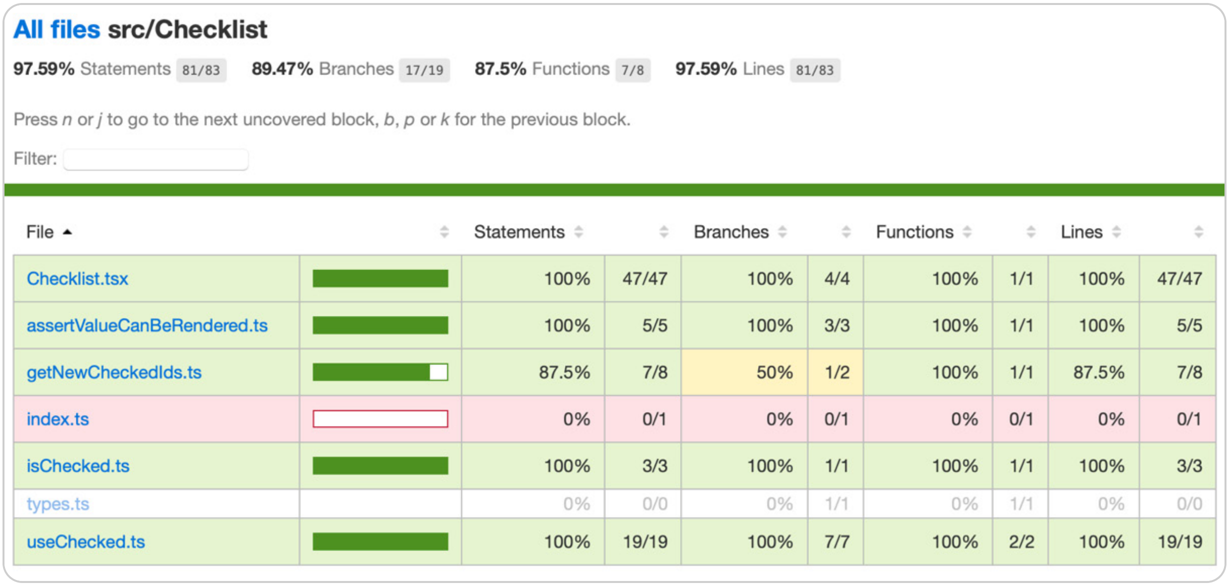
Filter:

File		Statements		Branches		Functions		Lines	
src		0%	0/46	0%	0/2	0%	0/2	0%	0/46
src/Checklist		97.59%	81/83	89.47%	17/19	87.5%	7/8	97.59%	81/83

Figure 12.7 – HTML coverage report

The report contains the same data as the terminal report, but this one is interactive.

2. Click on the **src/Checklist** link in the second row of the report. The page now shows the coverage for the files in the checklist component:



All files src/Checklist

97.59% Statements 81/83 89.47% Branches 17/19 87.5% Functions 7/8 97.59% Lines 81/83

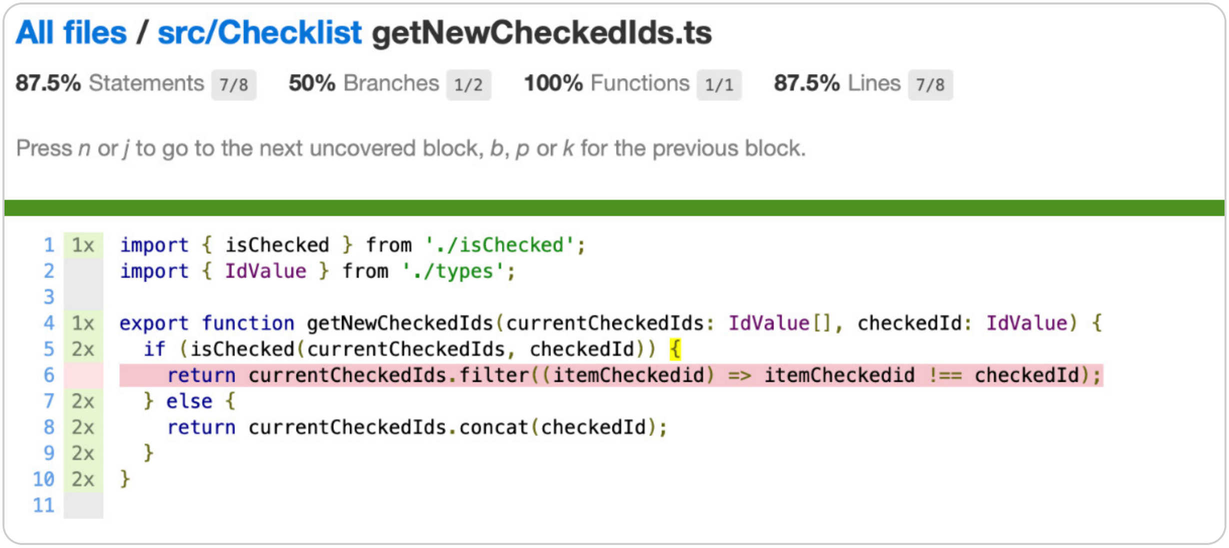
Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File		Statements		Branches		Functions		Lines	
Checklist.tsx		100%	47/47	100%	4/4	100%	1/1	100%	47/47
assertValueCanBeRendered.ts		100%	5/5	100%	3/3	100%	1/1	100%	5/5
getNewCheckedIds.ts		87.5%	7/8	50%	1/2	100%	1/1	87.5%	7/8
index.ts		0%	0/1	0%	0/1	0%	0/1	0%	0/1
isChecked.ts		100%	3/3	100%	1/1	100%	1/1	100%	3/3
types.ts		0%	0/0	0%	1/1	0%	1/1	0%	0/0
useChecked.ts		100%	19/19	100%	7/7	100%	2/2	100%	19/19

Figure 12.8 – Coverage report for checklist component files

3. Click on the **`getNewCheckedIds.ts`** link to drill into the coverage for that file:



All files / src/Checklist getNewCheckedIds.ts

87.5% Statements 7/8   50% Branches 1/2   100% Functions 1/1   87.5% Lines 7/8

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
 1 1x  import { isChecked } from './isChecked';
 2     import { IdValue } from './types';
 3
 4 1x  export function getNewCheckedIds(currentCheckedIds: IdValue[], checkedId: IdValue) {
 5 2x    if (isChecked(currentCheckedIds, checkedId)) {
 6         return currentCheckedIds.filter((itemCheckedid) => itemCheckedid !== checkedId);
 7 2x    } else {
 8 2x      return currentCheckedIds.concat(checkedId);
 9 2x    }
10 2x  }
11
```

Figure 12.9 – Coverage report for getNewCheckedIds.ts

We can see that the uncovered line 6 is clearly highlighted in the **`getNewCheckedIds.ts`** file.

So, the HTML coverage report is useful in a large code base because it starts with high-level coverage and allows you to drill into coverage on specific folders and files. When viewing a file in the report, we can quickly determine where the uncovered code is because it is clearly highlighted.

Next, we will update our tests so that line 6 in **`getNewCheckedIds.ts`** is covered.

Gaining full coverage on the checklist component

The logic not currently being checked by tests is the logic used when a list item is clicked but has already been checked. We will extend the **`'should check items when clicked'`** test to cover this logic. Carry out the following steps:

1. Open **`Checklist.test.tsx`** and rename the **`'should check items when clicked'`** test as follows:

   ```
   test('should check and uncheck items when
   clicked', async () => {
     ...
   });
   ```

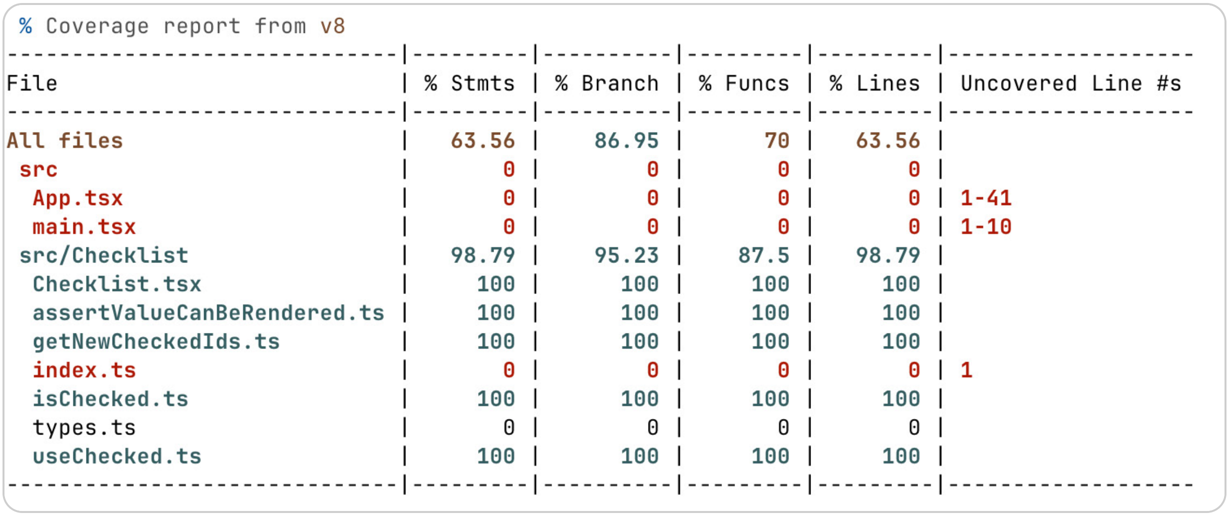
2. Add the following highlighted lines at the end of the test to click the checkbox for a second time and check it is unchecked:

   ```
   test('should check and uncheck items when
   clicked', async () => {
     const user = userEvent.setup();
     render( ... );
     const lucyCheckbox = screen.getByTestId(
        'Checklist__input__1'
     );
     expect(lucyCheckbox).not.toBeChecked();
     await user.click(lucyCheckbox);
     expect(lucyCheckbox).toBeChecked();
     await user.click(lucyCheckbox);
     expect(lucyCheckbox).not.toBeChecked();
   });
   ```

3. In the terminal, rerun the tests with coverage:

   ```
   npm run test -- --coverage --watch=false
   ```

All the tests still pass, and the coverage on the checklist component is now reported as 100% on all the statistics:



```
% Coverage report from v8
------------------------------|---------|----------|---------|---------|-------------------
File                          | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
------------------------------|---------|----------|---------|---------|-------------------
All files                     |   63.56 |    86.95 |      70 |   63.56 |
 src                          |       0 |        0 |       0 |       0 |
  App.tsx                     |       0 |        0 |       0 |       0 | 1-41
  main.tsx                    |       0 |        0 |       0 |       0 | 1-10
 src/Checklist                |   98.79 |    95.23 |    87.5 |   98.79 |
  Checklist.tsx               |     100 |      100 |     100 |     100 |
  assertValueCanBeRendered.ts |     100 |      100 |     100 |     100 |
  getNewCheckedIds.ts         |     100 |      100 |     100 |     100 |
  index.ts                    |       0 |        0 |       0 |       0 | 1
  isChecked.ts                |     100 |      100 |     100 |     100 |
  types.ts                    |       0 |        0 |       0 |       0 |
  useChecked.ts               |     100 |      100 |     100 |     100 |
------------------------------|---------|----------|---------|---------|-------------------
```

Figure 12.10 – 100% coverage on the checklist component

The checklist component is now well covered. However, it is a little annoying that **`index.ts`** and **`types.ts`** appear in the report with zero coverage. We'll resolve this next.

Ignoring files in the coverage report

We will remove **`index.ts`** and **`types.ts`** from the coverage report because they don't contain any logic and create unnecessary noise. Carry out the following steps:

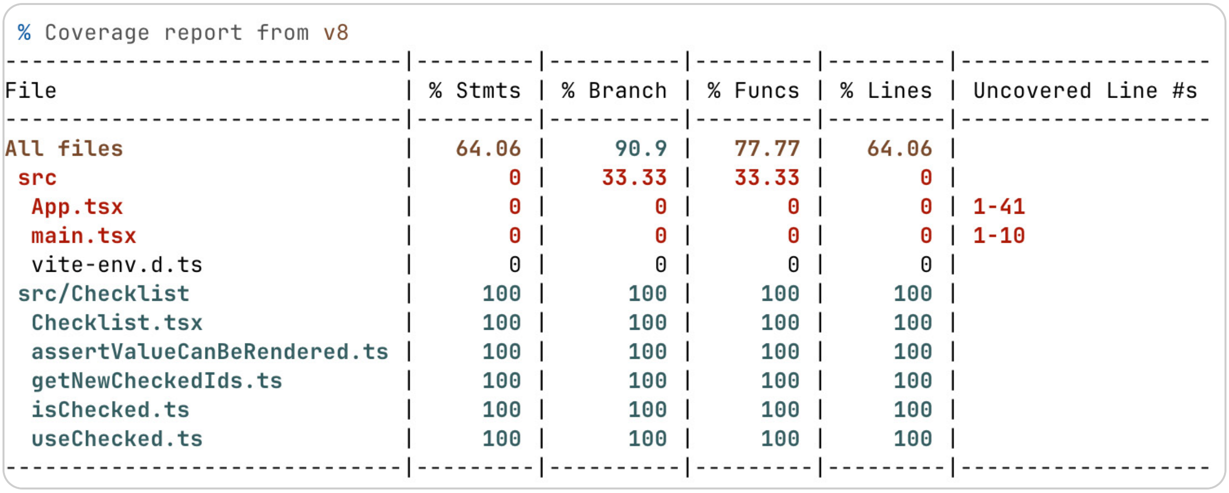
1. Open the **`vite.config.ts`** file. We can configure coverage in a **`test.coverage`** field, and there is an **`exclude`** option for removing files from the coverage report. Add the following highlighted lines to ignore the files we don't want to include in the coverage report:

```
const vitestConfig = defineVitestConfig({
  test: {
    ...,
    coverage: {
      exclude: [
        "**/types.ts",
        "**/index.ts",
        "vite.config.ts",
        "eslint.config.js",
      ],
    },
  },
});
```

2. In the terminal, rerun the tests with coverage:

```
npm run test -- --coverage --watch=false
```

The **types.ts** and **index.ts** files are removed from the coverage report:



```
% Coverage report from v8
------------------------------|---------|----------|---------|---------|-------------------
File                          | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
------------------------------|---------|----------|---------|---------|-------------------
All files                     |   64.06 |     90.9 |   77.77 |   64.06 |
 src                          |       0 |    33.33 |   33.33 |       0 |
  App.tsx                     |       0 |        0 |       0 |       0 | 1-41
  main.tsx                    |       0 |        0 |       0 |       0 | 1-10
  vite-env.d.ts               |       0 |        0 |       0 |       0 |
 src/Checklist                |     100 |      100 |     100 |     100 |
  Checklist.tsx               |     100 |      100 |     100 |     100 |
  assertValueCanBeRendered.ts |     100 |      100 |     100 |     100 |
  getNewCheckedIds.ts         |     100 |      100 |     100 |     100 |
  isChecked.ts                |     100 |      100 |     100 |     100 |
  useChecked.ts               |     100 |      100 |     100 |     100 |
------------------------------|---------|----------|---------|---------|-------------------
```

Figure 12.11 – types.ts and index.ts files removed from the coverage report

That completes this section on code coverage. Here's a quick recap:

- The `--coverage` option outputs a code coverage report after the tests have run.
- An interactive HTML code coverage report is generated in addition to the one in the terminal. This is useful on a large test suite to drill into uncovered code.
- Both report formats highlight uncovered code, giving us valuable information to improve our test suite.

It's time to summarize the chapter.

Summary

In this chapter, we created tests on a checklist component using Vitest and the React Testing Library. In addition, we learned about common matchers in Vitest's core package and useful matchers for component testing in a companion package called `Vitest-dom`.

We learned about the wide variety of queries available in the React Testing Library to select elements in different ways. We used the `getByText` query extensively in the checklist tests. We also created a test ID on list item checkboxes so that the `getByTestId` query could be used to select them uniquely.

We learned that the `user-event` package is an excellent way of simulating user interactions that are decoupled from the implementation. We used this to simulate a user clicking a list item checkbox.

Finally, we learned how to produce code coverage reports and understood all the statistics in the report. The report included information about uncovered code, which we used to gain 100% coverage on the checklist component.

So, we have reached the end of this book. You will now be comfortable with both React and TypeScript and have excellent knowledge in areas outside React core, such as styling and the popular Next.js framework. You will be able to develop components that are reusable across different pages and even different apps. On top of that, you will now be able to write a robust test suite so that you can ship new features with confidence.

In summary, the knowledge from this book will allow you to efficiently build the frontend of large and complex apps with React and TypeScript. I hope you have enjoyed reading this book as much as I did writing it!

Questions

Answer the following questions to check what you have learned in this chapter:

1. Why doesn't the following expectation pass? How could this be resolved?

   ```
   expect({ name: 'Bob' }).toBe({ name: 'Bob' });
   ```

2. Which matcher can be used to check that a variable isn't **`null`**?
3. Here's an expectation that checks whether a **Save** button is disabled:

   ```
   expect(
     screen.getByText('Save').hasAttribute('disab
   led')
   ).toBe(true);
   ```

   The expectation passes as expected, but is there a different matcher that can be used to simplify this?

4. We have a **`form`** element containing a **Save** button only when data has been loaded into fields from a server API. We have used the **`findBy`** query type so that the query retries until the data has been fetched:

   ```
   expect(screen.findByText('Save')).toBeInTheDoc
   ument();
   ```

   However, the expectation doesn't work—can you spot the problem?

5. The following expectation attempts to check that a **Save** button isn't in the DOM:

   ```
   expect(screen.getByText('Save')).toBe(null);
   ```

This doesn't work as expected, though. Instead, an error is raised because the **Save** button can't be found. How can this be resolved?

Answers

1. The **`toBe`** matcher should only be used for checking primitive values such as numbers and strings—this is an object. The **`toStrictEqual`** matcher should be used to check objects because it checks the values of all its properties instead of the object reference:

   ```
   expect({ name: 'Bob' }).toStrictEqual({
     name: 'Bob'
   });
   ```

2. The **`not`** and **`toBeNull`** matchers can be combined to check that a variable isn't **`null`**:

   ```
   expect(something).not.toBeNull();
   ```

3. The **`toBeDisabled`** matcher can be used from **`jest-dom`**:

   ```
   expect(screen.getByText('Save')).toBeDisabled(
   );
   ```

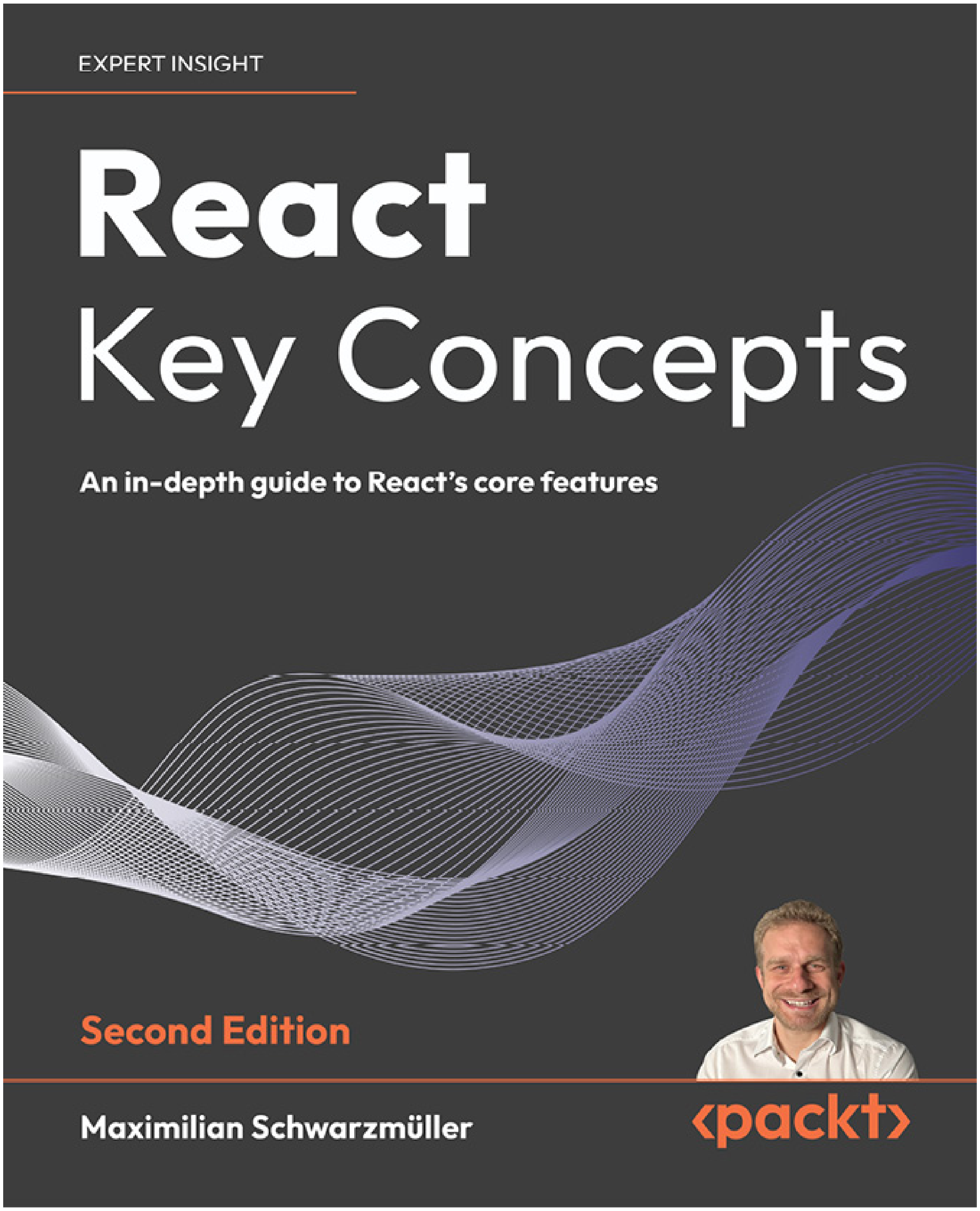
4. The **`findBy`** query type requires awaiting because it is asynchronous:

   ```
   expect(
     await screen.findByText('Save')
   ).toBeInTheDocument();
   ```

5. The **queryBy** query type can be used because it doesn't throw an exception when an element isn't found. In addition, the **not** and **toBeInTheDocument** matchers can be used to check that the element isn't in the DOM:

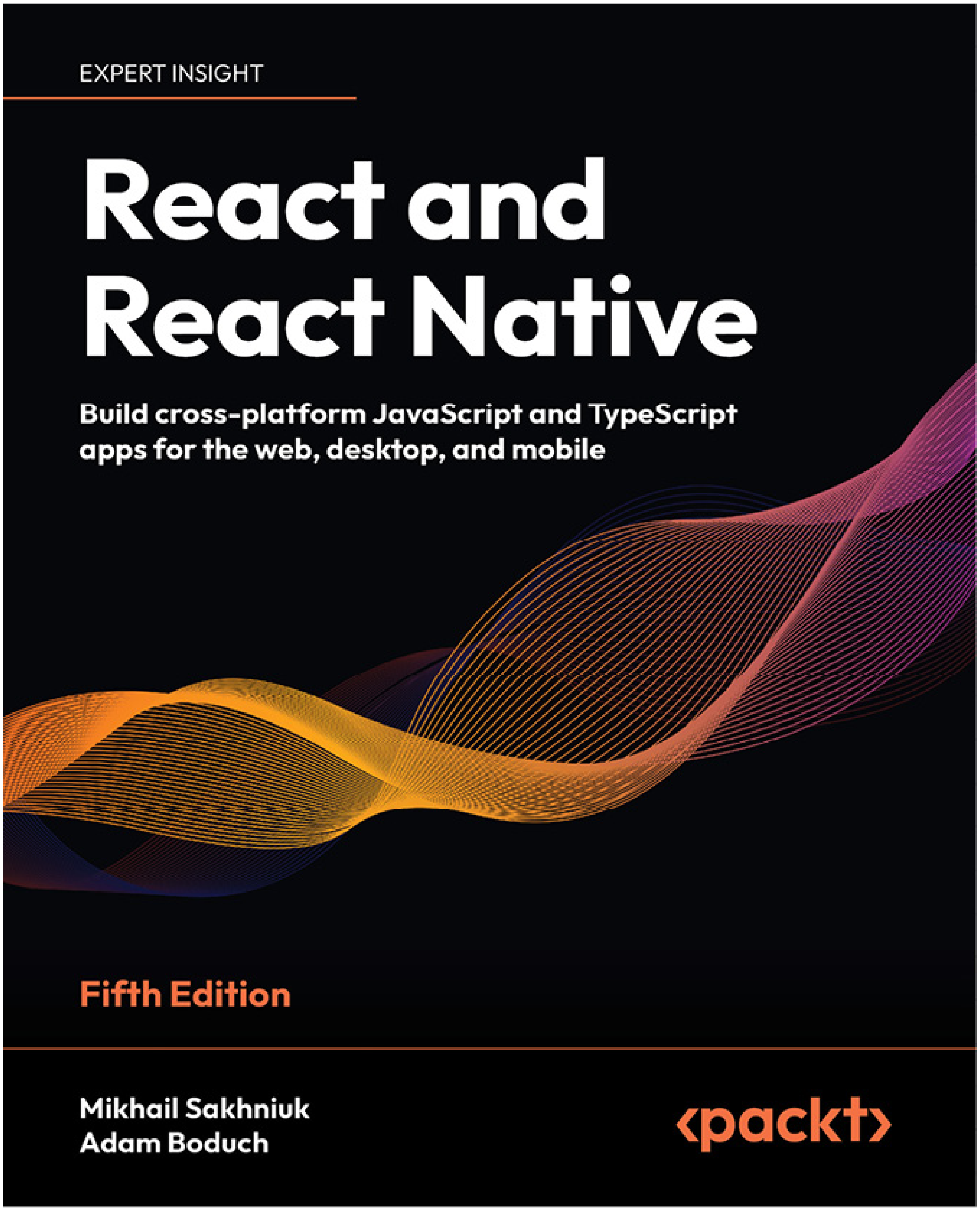
```
expect(
  screen.queryByText('Save')
).not.toBeInTheDocument();
```

Learn more on Discord



React Key Concepts, Second Edition

Maximilian Schwarzmüller


EXPERT INSIGHT
React and React Native
Build cross-platform JavaScript and TypeScript apps for the web, desktop, and mobile
Fifth Edition
Mikhail Sakhniuk
Adam Boduch
<packt>

React and React Nativ, Fifth Edition

Mikhail Sakhniuk and Adam Boduch

Share Your Thoughts

Now you've finished *Learn React with TypeScript, Third Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

As this ebook edition doesn't have fixed pagination, the page numbers below are for reference only, based on the printed edition of this book.