

React Projects

Second Edition

Build advanced cross-platform projects with React and React Native to become a professional developer

Roy Derks





BIRMINGHAM—MUMBAI

React Projects *Second Edition*

Copyright © 2022 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Associate Group Product Manager: Pavan Ramchandani

Publishing Product Manager: Ashitosh Gupta

Senior Editor: Mark Dsouza

Content Development Editor: Divya Vijayan

Technical Editor: Saurabh Kadave

Copy Editor: Safis Editing

Project Coordinator: Ajesh Devavaram

Proofreader: Safis Editing

Indexer: Hemangini Bari

Production Designer: Prashant Ghare

Marketing Coordinator: Anamika Singh

First published: December 2019

Second edition: April 2022

Production reference: 1290422

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80107-063-8

www.packt.com

To all developers who are starting their journey with React. It will be hard work, but you will succeed.

– *Roy Derks*

Contributors

About the author

Roy Derks is a serial start-up CTO, conference speaker, and developer from Amsterdam. He has been actively programming since he was a teenager, starting as a self-taught programmer using online tutorials and books. At the age of 14, he founded his first start-up, a peer-to-peer platform where users could trade DVDs with other users for free. This marked the start of his career in web development, which back then primarily consisted of creating web applications using an MVC architecture with the LAMP stack.

In 2015, he was introduced to React and GraphQL at a hackathon in Berlin, and after winning a prize for his project, he started to use these technologies professionally. Over the next few years, he helped multiple start-ups create cross-platform applications using React and React Native, including a start-up he co-founded. He also started giving workshops and talks at conferences around the globe. Over the last years he gave over 100 conference talks about React, React Native, and GraphQL, inspiring ten thousands of developers worldwide.

You can follow Roy on Twitter to get more information on the latest developments in the world of React and React Native: <https://twitter.com/gethackteam>.

About the reviewers

Kirill Ezhemenskii is an experienced software engineer, a frontend and mobile developer, a solution architect, and the CTO at a health-care company. He's a functional programming advocate and an expert in the React stack, GraphQL, and TypeScript. He's also a React Native mentor.

Emmanuel Demey works with the JavaScript ecosystem on a daily basis. He spends his time sharing his knowledge with anyone and everyone. His first goal at work is to help the people he works with. He has spoken at French conferences (such as Devfest Nantes, Devfest Toulouse, Sunny Tech, and Devox France) about topics related to the web platform, such as JavaScript frameworks (Angular, React.js, and Vue.js), accessibility, and Nest.js. He has been a trainer for 10 years at Worldline and Zenika (two French consulting companies). He is also the co-leader of the Google Developer of Lille group and the co-organizer of the Devfest Lille conference.

Table of Contents

Preface

Chapter 1: Creating a Single-Page Application in React

Project overview

Getting started

Creating a single-page application

Setting up a project

Structuring a project

Creating new components

Summary

Further reading

Chapter 2: Creating a Portfolio in React with Reusable Components and Routing.

Project overview

Getting started

Creating a portfolio in React

Creating a portfolio with Create React App

Installing Create React App

Building reusable React components

Structuring our application

Reusing components in React

Routing with react-router

Summary

Further reading

Chapter 3: Building a Dynamic Project Management Board

Project overview

Getting started

Creating a project management board application

Handling the data flow

Loading and displaying the data

Working with custom Hooks

Creating custom Hooks

Reusing a custom Hook

Making the board dynamic

Styling in React with styled-components

Summary

Further reading.

Chapter 4: Building a Server-Side-Rendered Community Feed Using Next.js

Project overview

Getting started

Community feed application

Setting up Next.js

Installing Next.js

Adding styled-components

Routing with Next.js

Handling query strings

Enabling SSR

Fetching data server side with Next.js

Adding head tags for SEO

Summary

Further reading.

Chapter 5: Building a Personal Shopping List Application Using Context and Hooks

Project overview

Getting started

Personal shopping list

Using the Context API for state management

Creating Context

Nesting Context

Mutating Context with Hooks

Using life cycles in functional components

Using advanced state with useReducer

Mutating data in the Provider

Creating an application Context

Code splitting with React Suspense

Summary

Further reading

Chapter 6: Building an Application **Exploring TDD Using the React Testing** **Library and Cypress**

Project overview

Getting started

The hotel review application

Unit testing components

Testing React state and Hooks

End-to-end testing with Cypress

Summary

Further reading

Chapter 7: Building a Full-Stack E-Commerce Application with Next.js and GraphQL

Project overview

Getting started

Getting started with the initial React application

Building a full stack e-commerce application with React, Apollo, and GraphQL

Creating a GraphQL server with Next.js

Consuming GraphQL with Apollo Client

Summary

Further reading

Chapter 8: Building an Animated Game Using React Native and Expo

Project overview

Getting started

Creating an animated game application with React Native and Expo

Setting up React Native with Expo

Adding gestures and animations in React Native

Advanced animations with Lottie

Summary

Further reading

Chapter 9: Building a Full-Stack Social Media Application with React Native and Expo

Project overview

Getting started

Checking out the initial project

Building a full-stack social media application with React Native and Expo

Advanced routing with authentication

Using the camera with React Native and Expo

Differences in styling for iOS and Android

Summary

Further reading

Chapter 10: Creating a Virtual Reality Application with React and Three.js

Project overview

Getting started

Creating a VR application with React and Three.js

Getting started with Three.js

Creating 3D objects with Three.js

Rendering 360-degree panorama images

Animating 3D objects

Summary

Further reading

Other Books You May Enjoy

Preface

This book will help you take your React knowledge to the next level by showing how to apply both basic and advanced React patterns to create cross-platform applications. The concepts of React are described in a way that's understandable to both new and experienced developers; no prior experience of React is required, although it would help.

In each of the 10 chapters of this book, you'll create a project with React or React Native. The projects created in these chapters implement popular React features such as Hooks for re-using logic, the context API for state-management, and Suspense. Popular libraries, such as React Router and React Navigation, are used for routing, while the JavaScript testing framework React Testing Library and Cypress are used to write unit and integration tests for the applications. Also, some more advanced chapters involve a GraphQL server, and Expo is used to help you create React Native applications.

Who this book is for

The book is for JavaScript developers who want to explore React tooling and frameworks for building cross-platform applications. Basic knowledge of web development, ECMAScript, and React will assist in understanding key concepts covered in this book.

The supported React versions for this book are:

- React - v18.0
- React Native - v0.64

What this book covers

[Chapter 1](#), *Creating a Single-Page Application in React*, will explore the foundation of building React projects that can scale. Best practices of how to structure your files, packages to use, and tools will be discussed and practiced. You'll learn about React architecture while building a Single-page application. Also, webpack and Babel are used to compile code.

[Chapter 2](#), *Creating a Portfolio in React with Reusable Components and Routing*, will explain how to set up and re-use styling in React components throughout your entire application. We will build a GitHub Card application to see how to use CSS in JavaScript and re-use components and styling in your application. Next to this, you'll learn about implementing navigation with React Router v6.

[Chapter 3](#), *Building a Dynamic Project Management Board*, will cover how to reuse application state-logic from components by using Hooks. You'll learn how to build custom Hooks and interact with Web APIs to make draggable components. Styled Components are introduced to make it easier to style React components in a scalable way.

[Chapter 4](#), *Build a Server-Side-Rendered Community Feed Using Next.js*, will discuss routing, ranging from setting up basic routes, dynamic route handling, and how to set up routes for server-side ren-

dering. Therefore the React web framework Next.js will be used as you'll learn while building an application based on Stack Overflow.

[Chapter 5](#), *Build a Personal Shopping List Application Using Context and Hooks*, will show you how to use the React context API with Hooks to handle the data flow throughout the application. We will create a personal shopping list to see how data can be accessed and changed from parent to child components and vice versa with Hooks and the context API.

[Chapter 6](#), *Build an Application Exploring TDD Using React Testing Library and Cypress*, will focus on unit testing with assertions and snapshots. You'll learn how to manage test coverage, and implement visual integrations tests using the Cypress framework. We will build a hotel review application to see how to test components and data flows.

[Chapter 7](#), *Build a Full-Stack E-Commerce Application with Next.js and GraphQL*, will use GraphQL to supply a backend to the application. This chapter will show you how to set up a Full Stack React application with Next.js including a basic GraphQL server. We will build an e-commerce application to see how to create a server, send requests to it, and handle authentication.

[Chapter 8](#), *Build an Animated Game Using React Native and Expo*, will discuss animations and gestures, which are what truly distinguishes a mobile application from a web application. This chapter will explain how to implement them. Also, the differences in gestures between iOS and Android will be shown by building a card game application that has animations and that responds to gestures.

[Chapter 9](#), *Build a Full-Stack Social Media Application with React Native and Expo*, will cover scaling and structuring React Native applications, which are slightly different from web applications created with React. This chapter will outline how to use native APIs of the mobile device, such as using the Camera, while building a Full Stack social media application to examine the best practices for React Native.

[Chapter 10](#), *Creating a Virtual Reality Application with React and Three.js*, will discuss how to get started with React and Three.js by creating a panorama viewer that gives the user the ability to look around in the virtual world and create components inside it. The application you'll build will look like a game you can play in Virtual Reality (VR).

To get the most out of this book

All the projects in this book are created with React or React Native. Prior knowledge of JavaScript is required for most chapters in this

book. Although all the concepts of React and related technologies are described in this book, we advise you to refer to React docs if you want to find out more about a feature. In the following section, you can find some information about setting up your machine for this book and how to download the code for each chapter.

Software/hardware covered in the book	Operating system requirements
React 18	Windows, macOS, or Linux
React Native 0.64	
Expo SDK 44	

For the applications that are created in this book, you'll need to have at least Node.js v14.19.1 installed on your machine so that you can run npm commands. If you haven't installed Node.js on your machine, please go to <https://nodejs.org/en/download/>, where you can find the download instructions for macOS, Windows, and Linux.

After installing Node.js, run the following commands in your command line to check the installed versions:

- For Node.js (should be v14.19.1 or higher):

```
node -v
```

- For npm (should be v6.14.14 or higher):

```
npm -v
```

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

The contents of this book use the latest available version when this book is completed in April 2022. Any updates after this date, might not work with the functionalities described in this book. It's advised to follow the official React and React Native documentation to get more information on the features that got released after this book was published.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/React-Projects-Second-Edition>. If there's an update to the code, it will be updated in the GitHub repository.

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

out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781801070638_ColorImages.pdf.

Conventions used

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

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "If we look at the source code for this component in **App.js**, we'll see that there's already a CSS **header** element in the **return** function."

A block of code is set as follows:

```
.App-logo {  
  height: 40vmin;  
  pointer-events: none;  
}
```

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

```
import './Header.css';  
- function Header() {  
+ function Header({ logo }) {  
    return (  
        <header className='App-header'>
```

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

```
npx create-react-app chapter-2
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Select **System info** from the **Administration** panel."

TIPS OR IMPORTANT NOTES

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customer@packtpub.com and mention the

book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *React Projects*, we'd love to hear your thoughts! Please [click here to go straight to the Amazon review page](#) for this book and share your feedback.

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

Chapter 1: Creating a Single-Page Application in React

When you bought this book, you'd probably heard of React before and maybe even tried out some of the code examples that can be found online. This book is constructed in such a way that the code examples in each chapter gradually increase in complexity, so even if you feel your experience with React is limited, each chapter should be understandable if you've read the previous one. By the end of this book, you will know how to work with React and its stable features, up until version 18, and you will also have experience with GraphQL and React Native.

This first chapter kicks off with us learning how to build a single-page application based on the popular TV show *Rick and Morty*; the application will provide us with information about its characters that we'll fetch from an external source. The core concepts for getting started with React will be applied to this project, which should be understandable if you've got some prior experience in building applications with React. If you haven't worked with React before, that's no problem either; this book describes the React features that are used in the code examples along the way.

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

- Setting up a new React project
- Structuring a project

Let's dive in!

Project overview

In this chapter, we will create a single-page application in React that retrieves data from an API and runs in the browser with Webpack and Babel. Styling will be done using Bootstrap. The application that you'll build will show information about the popular TV show *Rick and Morty*, along with images.

The build time is 1 hour.

Getting started

The complete code for this chapter can be found on GitHub:

<https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter01>.

For the applications created in this book, you'll need to have at least Node.js v14.17.0 installed on your machine so that you can run npm commands. If you haven't installed Node.js on your machine, please

go to <https://nodejs.org/en/download/>, where you can find the download instructions for macOS, Windows, and Linux.

After installing Node.js, run the following commands in your command line to check the installed versions:

- For Node.js (which should be v14.17.0 or higher), use this:

```
node -v
```

- For npm (which should be v6.14.3 or higher), use this:

```
npm -v
```

Also, you should have installed the React Developer Tools plugin (for Chrome and Firefox) and added it to your browser. This plugin can be installed from the Chrome Web Store

(<https://chrome.google.com/webstore/>) or Firefox Add-ons (<https://addons.mozilla.org>).

Creating a single-page application

In this section, we will create a new single-page React application from scratch, starting with setting up a new project with Webpack and Babel. Setting up a React project from scratch will help you under-

stand the basic needs of a project, which is crucial for any project you create.

Setting up a project

Every time you create a new React project, the first step is to create a new directory on your local machine. Since this is the first chapter for which you're going to build a single-page application, name this directory **chapter-1**.

Inside this new directory, execute the following from the command line:

```
npm init -y
```

Running this command will create a fresh **package.json** file with the bare minimum of information needed to run a JavaScript/React project. By adding the **-y** flag to the command, we can automatically skip the steps where we set information such as the name, version, and description.

After running this command, the following **package.json** file will be created for the project:

```
{
  "name": "chapter-1",
  "version": "1.0.0",
```

```
"description": "",
"main": "index.js",
"scripts": {
  "test": "echo \"Error: no test
specified\" && exit 1"
},
"keywords": [],
"author": "",
"license": "ISC"
}
```

NOTE

To learn more about the workings of `package.json`, make sure to read the documentation from npm: <https://docs.npmjs.com/cli/v6/configuring-npm/package-json>.

After creating `package.json` in this section, we're ready to add Webpack, which we will do in the next section.

Setting up Webpack

To run the React application, we need to install Webpack 5 (at the time of writing, the current stable version of Webpack is version 5) and the Webpack CLI as **devDependencies**. Webpack is a library that

lets us create a bundle out of JavaScript/React code that can be used in a browser. The following steps will help you set up Webpack:

1. Install the required packages from npm using the following command:

```
npm install --save-dev webpack webpack-cli
```

2. After installation, these packages are included inside the **package.json** file where we can have them run in our **start** and **build** scripts. But first, we need to add some files to the project:

```
chapter-1
  |- node_modules
  |- package.json
+  |- src
+    |- index.js
```

This will add the **index.js** file to a new directory called **src**. Later on, we'll configure Webpack so that this file is the starting point for our application.

3. First, the following code block must be added to this file:

```
console.log('Rick and Morty');
```

4. To run the preceding code, we will add the **start** and **build** scripts to our application using Webpack. The test script is not needed in this chapter, so this can be deleted. Also, the **main** field can be

changed to **private** with the **true** value, as the code we're building is a local project:

```
{
  "name": "chapter-1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
-    "test": "echo \"Error: no test
specified\" &&
                exit 1"
+    "start": "webpack --mode
development",
+    "build": "webpack --mode production"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

The **npm start** command will run Webpack in development mode, while **npm run build** will create a production bundle using Webpack. The biggest difference is that running Webpack in production mode will minimize our code and decrease the size of the project bundle.

5. We now run the **start** or **build** command from the command line; Webpack will start up and create a new directory called **dist**:

```
chapter-1
  |- node_modules
  |- package.json
+  |- dist
+    |- main.js
    |- src
      |- index.js
```

6. Inside this directory, there will be a file called **main.js** that includes our project code and is also known as our bundle. If successful, the following output will be visible:

```
asset main.js 794 bytes [compared for emit]
(name: main)
./src/index.js 31 bytes [built] [code
generated]
webpack compiled successfully in 67 ms
```

Depending on whether we've run Webpack in development or production mode, the code will be minimized in this file.

7. You can check whether your code is working by running the **main.js** file in your bundle from the command line:

```
node dist/main.js
```

This command runs the bundled version of our application and should return the following output:

```
> node dist/main.js
```

```
Rick and Morty
```

Now, we're able to run JavaScript code from the command line. In the next part of this section, we will learn how to configure Webpack so that it works with React.

Configuring Webpack to work with React

Now that we've set up a basic development environment with Webpack for a JavaScript application, we can start installing the packages we need in order to run any React application.

These packages are **react** and **react-dom**, where the former is the generic core package for React and the latter provides an entry point to the browser's DOM and renders React. Install these packages by executing the following command in the command line:

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

Installing only the dependencies for React is not sufficient to run it, since, by default, not every browser can read the format (such as ES-2015+ or React) that your JavaScript code is written in. Therefore, we need to compile the JavaScript code into a readable format for every browser.

For this, we'll use Babel and its related packages to create a tool-chain to use React in the browser with Webpack. These packages can be installed as **devDependencies** by running the following command:

```
npm install --save-dev @babel/core babel-  
loader @babel/preset-env @babel/preset-react
```

Next to the Babel core package, we'll also install **babel-loader**, which is a helper so that Babel can run with Webpack and two preset packages. These preset packages help determine which plugins will be used to compile our JavaScript code into a readable format for the browser (**@babel/preset-env**) and to compile React-specific code (**@babel/preset-react**). With the packages for React and the correct compilers installed, the next step is to make them work with Webpack so that they are used when we run our application.

To do this, configuration files for both Webpack and Babel need to be created in the **src** directory of the project:

```
chapter-1  
  |- node_modules  
  |- package.json  
+ |- babel.config.json  
+ |- webpack.config.js  
  |- dist
```

```
    |- main.js
  |- src
    |- index.js
```

The configuration for Webpack is added to the **webpack.config.js** file to use **babel-loader**:

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader'
        },
      },
    ],
  },
};
```

The configuration in this file tells Webpack to use **babel-loader** for every file that has the **.js** extension and excludes files in the **node_modules** directory for the Babel compiler.

To use the Babel presets, the following configuration must be added to **babel.config.json**:

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "targets": {
          "esmodules": true
        }
      }
    ],
    [
      "@babel/preset-react",
      {
        "runtime": "automatic"
      }
    ]
  ]
}
```

`@babel/preset-env` must be set to target `esmodules` in order to use the latest Node modules. Also, defining the JSX runtime to `automatic` is needed, since React 18 has adopted the new JSX Transform functionality: <https://reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html>.

NOTE

*The configuration for **babel-loader** can also be placed in the configuration inside **webpack.config.json**. But by creating a separate Babel configuration file for this, these settings can also be used by other tools in the JavaScript/React ecosystem.*

Now that we've set up Webpack and Babel, we can run JavaScript and React from the command line. In the next part of this section, we'll create our first React code and make it run in the browser.

Rendering a React project

With the packages we've installed and configured in the previous sections to set up Babel and Webpack, we need to create an actual React component that can be compiled and run. Creating a new React project involves adding some new files to the project and making changes to the setup for Webpack:

1. Let's edit the **index.js** file that already exists in our **src** directory so that we can use **react** and **react-dom**. The contents of this file can be replaced with the following:

```
import ReactDOM from 'react-dom/client';  
function App() {  
  return <h1>Rick and Morty</h1>;  
}
```



```
const container =
document.getElementById( 'app' );
const root =
ReactDOM.createRoot(container);
root.render(<App />);
```

As you can see, this file imports the **react** and **react-dom** packages, defines a simple component that returns an **h1** element containing the name of your application, and has this component rendered in the browser with **react-dom**. The last line of code mounts the **App** component to an element with the **root** ID selector in your document, which is the entry point of the application.

2. We can create a file that has this element in a new directory called **public** and name that file **index.html**:

```
chapter-1
  |- node_modules
  |- package.json
  |- babel.config.json
  |- webpack.config.js
  |- dist
    |- main.js
+  |- public
+    |- index.html
    |- src
```

```
| - index.js
```

3. After adding a new file called **index.html** to this directory, we add the following code inside it:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport"
content="width=device-width,
    initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible"
    content="ie=edge" />
    <title>Rick and Morty</title>
  </head>
  <body>
    <section id="root"></section>
  </body>
</html>
```

This adds an HTML heading and body. Within the **head** tag is the title of our application, and inside the **body** tag is a section with the "root" ID selector. This matches with the element we've mounted the **App** component to in the **src/index.js** file.

4. The final step in rendering our React component is extending Webpack so that it adds the minified bundle code to the body tags as scripts when running. Therefore, we should install the **html-webpack-plugin** package into our **devDependencies**:

```
npm install --save-dev html-webpack-plugin
```

To use this new package to render our files with React, the Webpack configuration in the **webpack.config.js** file must be extended:

```
+ const HtmlWebpackPlugin =  
    require('html-webpack-plugin');  
module.exports = {  
  module: {  
    rules: [  
      {  
        test: /\.js$/,  
        exclude: /node_modules/,  
        use: {  
          loader: 'babel-loader',  
        },  
      },  
    ],  
  },  
+  plugins: [  
+    new HtmlWebpackPlugin({
```

```
+      template: './public/index.html',  
+      filename: './index.html',  
+    }),  
+  ],  
+};
```

Now, if we run `npm start` again, Webpack will start in development mode and add the `index.html` file to the `dist` directory. Inside this file, we'll see that, inside our `body` tag, a new `scripts` tag has been inserted that directs us to our application bundle – that is, the `dist/main.js` file. If we open this file in the browser or run `open dist/index.html` from the command line, it will return the result directly inside the browser. We can do the same when running the `npm run build` command to start Webpack in production mode; the only difference is that our code will be minified:

Rick and Morty

Figure 1.1 – Rendering React in the browser

This process can be sped up by setting up a development server with Webpack. We'll do this in the final part of this section.

Creating a development server

While working in development mode, every time we make changes to the files in our application, we need to rerun the `npm start` command.

Since this is a bit tedious, we will install another package called **webpack-dev-server**. This package adds the option to force Webpack to restart every time we make changes to our project files and manages our application files in memory instead of by building the **dist** directory.

The **webpack-dev-server** package can be installed with **npm**:

```
npm install --save-dev webpack-dev-server
```

Also, we need to edit the **dev** script in the **package.json** file so that it uses **webpack-dev-server** instead of Webpack. This way, you don't have to recompile and reopen the bundle in the browser after every code change:

```
{
  "name": "chapter-1",
  "version": "1.0.0",
  "description": "",
  "private": true,
  "scripts": {
-    "start": "webpack --mode development",
+    "start": "webpack serve --mode
development",
    "build": "webpack --mode production"
  },
}
```

```
    "keywords": [],  
    "author": "",  
    "license": "ISC"  
    ...  
  }
```

The preceding configuration replaces Webpack in the **start** scripts with **webpack-dev-server**, which runs Webpack in development mode. This will create a local development server that runs the application, which makes sure that Webpack is restarted every time an update is made to any of your project files.

Run the following command from the command line:

```
npm start
```

This will cause the local development server to become active at **http://localhost:8080/**, and it will refresh every time we make an update to any file in our project.

Now, we've created the basic development environment for our React application, which we'll develop and structure further in the next section of this chapter.

Structuring a project

With the development environment set up, it's time to start creating the single-page application. In the preceding sections, we've already added new directories to the project. But let's recap the current structure of the project, where two of the directories within our project's root directory are important:

- The first directory is called **dist** and is where the output from Webpack's bundled version of our application can be found.
- The second one is called **src** and includes the source code of our application.

NOTE

*Another directory that can be found in the root directory of our project is called **node_modules**. This is where the source files for every package that we install using **npm** are placed. It is recommended you don't make any manual changes to files inside this directory.*

In the following subsections, we will learn how to structure our React projects. This structure will be used in the rest of the chapters in this book as well.

Creating new components

The official documentation for React doesn't state any preferred approach regarding how to structure our React project, although two

common approaches are popular within the community: either structuring your files by feature/page or structuring them by file type.

The single-page application in this chapter will use a hybrid approach, where files are structured by file type first and by feature second. In practice, this means that there will be two types of components: top-level components, which are sometimes called containers, and low-level components, which relate to these top-level components. Creating these components requires that we add the following files and code changes:

1. The first step to achieving this structure is by creating a new subdirectory of `src` called **components**. Inside this directory, create a file called **List.js**:

```
chapter-1
├── node_modules
├── package.json
├── babel.config.json
├── webpack.config.js
├── dist
│   ├── main.js
│   └── index.html
├── public
│   └── index.html
└── src
```



```
+   |- components
+       |- List.js
+       |- index.js
```

This file will return the component that lists all the information about *Rick and Morty*:

```
function List() {
  return <h2>Characters</h2>;
}
export default List;
```

2. This component should be included in the entry point of our application so that it's visible. Therefore, we need to include it in the **index.js** file, inside the **src** directory, and refer to it:

```
import ReactDOM fr'm 'react-dom/client';
+ import List from './components/List';
function App() {
-   return <h1>Rick and Morty</h1>;
+   return (
+     <div>
+       <h1>Rick and Morty</h1>
+       <List />
+     </div>
+   );
```

```
};  
// ...
```

If we still have the development server running (if not, execute the `npm start` command again), we'll see that our application now returns the **Characters** heading below the title.

3. The next step is to add a component to the **List** component, making it a so-called composed component, which is a component that consists of multiple components. This component will be called **Character** and should also be located in the `src` subdirectory called `components`. Inside this directory, create a file called `Character.js` and add the following code block to it:

```
function Character() {  
  return <h3>Character</h3>;  
};  
export default Character;
```

As you have probably guessed from the name of this component, it will be used to return information about a character from *Rick and Morty* later on.

4. Now, import this **Character** component into the **List** component and return this component after the `h2` element by replacing the

return function with the following code:

```
+ import Character from './Character';
  function List() {
-   return <h2>Characters</h2>;
+   return (
+     <div>
+       <h2>Characters</h2>
+       <Character />
+       <Character />
+     </div>
+   );
  }
export default List;
```

If we visit our application in the browser again at <http://localhost:8080/>, the words **Character** will be displayed below the title and heading of the page:

Rick and Morty

Characters

Character

Character

Figure 1.2 – Adding components to React

From this, we cannot see which components are being rendered in the browser. But luckily, we can open the React Developer Tools plugin in our browser; we'll notice that the application currently consists of multiple stacked components:

```
<App>
  <List>
    <Character>
```

In the next part of this section, we will use our knowledge of structuring a React project and create new components to fetch data about *Rick and Morty* that we want to display in this single-page application.

Retrieving data

With both the development server and the structure for our project set up, it's time to finally add some data to it. For this, we'll be using the *Rick and Morty* REST API (<https://rickandmortyapi.com/documentation/#rest>), which provides information about this popular TV show.

Information from APIs can be retrieved in JavaScript using, for example, the `fetch` method, which is already supported by our browser. This data will be retrieved in the top-level components only, meaning that we should add a `fetch` function in the `List` container to retrieve and store that information.

To store the information, we'll be using the built-in state management (<https://reactjs.org/docs/state-and-lifecycle.html>) in React. Anything stored in the state can be passed down to the low-level components, after which they are called props. A simple example of using state in React is by using the `useState` Hook, which can be used to store and update variables. Every time these variables change using the `update` method that is returned by the `useState` Hook, our component will re-render.

NOTE

Since the release of version 16.8.0, React has used the concept of Hooks, which are methods supplied by React that let you use its core features without using class components. More information about Hooks can be found in the documentation:

<https://reactjs.org/docs/hooks-intro.html>.

Before adding the logic to retrieve data from the *Rick and Morty* REST API, let's inspect that API to see what fields will be returned. The base URL for the API is <https://rickandmortyapi.com/api>.

This URL returns a JSON output with all the possible endpoints for this API, which are all **GET** requests, meaning read-only, and work over **https**. From this base URL, we'll be using the `/character` endpoint to get information about the characters from *Rick and Morty*.

Not all information returned by this endpoint will be used; the following are the fields that we'll actually be using:

- **id** (int): The unique identifier of the character
- **name** (string): The name of the character
- **origin** (object): The object containing the name and the link to the character's origin location
- **image** (string): The link to the character's image with the dimensions 300 x 300 px

Before retrieving the data for *Rick and Morty*, the **Character** component needs to be prepared to receive this information. To display information about *Rick and Morty*, we need to add the following lines to the **Character** component:

```
- function Character() {  
-   return <h3>Character</h3>;  
+ function Character(character) {  
+   return (  
+     <div>  
+       <h3>{character.name}</h3>  
+       <img src={character.image} alt=  
+         {character.name}  
+         width='300' />  
+       <p>{'Origin: ${character.origin} &&
```

```

        character.origin.name}' }</p>
+   </div>
+ );
};
export default Character;

```

Now, the logic to retrieve the data can be implemented by importing **useState** from React and adding this Hook to the **List** component, which will contain an empty array as a placeholder for the characters:

```

+ import { useState } from 'react';
  import Character from './Character';
  function List() {
+   const [characters, setCharacters] =
  useState([]);
    return (
      // ...

```

To do the actual data fetching, another Hook should be imported, which is the **useEffect** Hook. This one can be used to handle side effects, either when the application mounts or when the state or a prop gets updated. This Hook takes two parameters, where the first one is a callback and the second one is an array containing all of the variables this Hook depends on – the so-called dependency array. When any of these dependencies change, the callback for this Hook will be called. When there are no values in this array, the Hook will be called

constantly. After the data is fetched from the source, the state will be updated with the results.

In our application, we need to add this Hook and retrieve the data from the API, and we should use an **async/await** function, since the **fetch** API returns a promise. After fetching the data, **state** should be updated by replacing the empty array for data with the character information:

```
- import { useState } from 'react';
+ import { useEffect, useState } from
  'react';
  import Character from './Character';
  function List() {
    const [characters, setCharacters] =
useState([]);
+   useEffect(() => {
+     async function fetchData() {
+       const data = await fetch(
          'https://rickandmortyapi.com/api/ch
aracter');
+       const { results } = await
data.json();
+       setCharacters(results);
+     }
  }
```



```
+    fetchData();  
+  }, [characters.length]);  
  return (  
    // ...
```

Inside the **useEffect** Hook, the new **fetchData** function will be called, as it's advised to not use an **async/await** function directly. The Hook is only calling the logic to retrieve the data from the API when the length of the **characters** state changes. You can extend this logic by also adding a **loading** state to the application so that the user will know when the data is still being fetched:

```
function List() {  
+  const [loading, setLoading] =  
  useState(true);  
  const [characters, setCharacters] =  
  useState([]);  
  useEffect(() => {  
    async function fetchData() {  
      const data = await fetch(  
        'https://rickandmortyapi.com/api/ch  
aracter');  
      const { results } = await  
data.json();  
      setCharacters(results);  
+    setLoading(false);  
  }  
}
```

```
    }  
    fetchData();  
  }, [characters.length]);  
  return (  
    // ...
```

NOTE

The previous method that we used to retrieve information from JSON files using `fetch` doesn't take into account that the request to this file may fail. If the request fails, the `loading` state will remain `true`, meaning that the user will keep seeing the loading indicator. If you want to display an error message when the request doesn't succeed, you'll need to wrap the `fetch` method inside a `try...catch` block, which will be shown later on in this book.

To display the character information in the application, we need to pass it to the `Character` component, where it can ultimately be shown in the `Character` component that we changed in the first step.

When the data is being retrieved from the API, the `loading` state is `true`, so we cannot display the `Character` component yet. When data fetching is finished, loading will be `false`, and we can iterate over the `character` state, return the `Character` component, and pass the character information as props. This component will also get a `key` prop, which is required for every component that is rendered within an iter-

ation. Since this value needs to be unique, the id of the character is used, as follows:

```
// ...
return (
  <div>
    <h2>Characters</h2>
    -   <Character />
    -   <Character />
    +   {loading ? (
    +     <div>Loading...</div>
    +   ) : (
    +     characters.map((character) => (
    +       <Character
    +         key={character.id}
    +         name={character.name}
    +         origin={character.origin}
    +         image={character.image}
    +       />
    +     ))
    +   )}
    </div>
  );
}
export default List;
```

If we visit our application in the browser again, we'll see that it now shows a list of characters, including some basic information and an image. At this point, our application will look similar to the following screenshot:

Rick and Morty

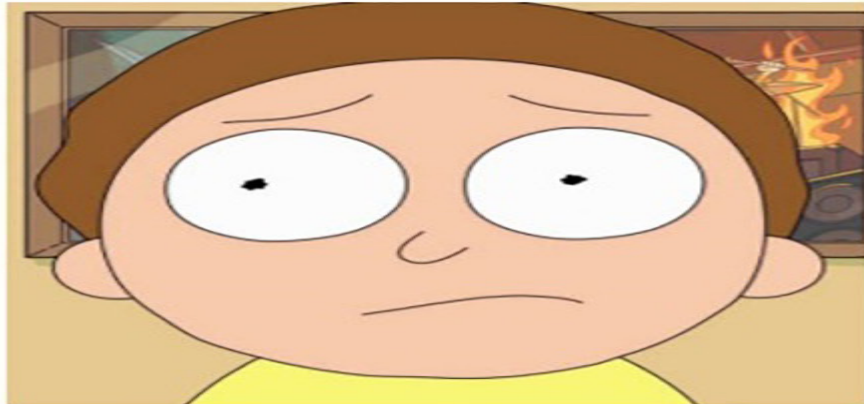
Characters

Rick Sanchez



Origin: Earth (C-137)

Morty Smith



Origin: Earth (C-137)

Summer Smith



Origin: Earth (Replacement Dimension)

Figure 1.3 – Rendering a list of components from the local state

As you can see, limited styling has been applied to the application, and it's only rendering the information that's been fetched from the API. Styling will be added in the next part of this section using a package called Bootstrap.

Adding styling

Showing just the character information isn't enough. We also need to apply some basic styling to the project. Adding styling to the project is done with the Bootstrap package, which adds styling to our components based on class names.

Bootstrap can be installed from npm using the following and added to **devDependencies**:

```
npm install --save-dev bootstrap
```

Also, import this file into the entry point of our React application, **src/index.js**, so that we can use the styling throughout the entire application:

```
import ReactDOM from 'react-dom/client';  
import List from './containers/List';  
+ import  
'bootstrap/dist/css/bootstrap.min.css';
```

```
function App() {  
  // ...
```

Webpack is unable to compile CSS files by itself; we need to add the appropriate loaders to make this happen. We can install these by running the following command:

```
npm install --save-dev css-loader style-  
loader
```

We need to add these packages as a rule to the Webpack configuration:

```
const HtmlWebpackPlugin = require('html-  
webpack-plugin');  
module.exports = {  
  module: {  
    rules: [  
      {  
        test: /\.js$/,  
        exclude: /node_modules/,  
        use: {  
          loader: 'babel-loader',  
        },  
      },  
      +    {  
      +      test: /\.css$/,
```

```

+         use: ['style-loader', 'css-
loader'],
+       },
    ],
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './public/index.html',
      filename: './index.html',
    }),
  ],
};

```

NOTE

*The order in which loaders are added is important since **css-loader** handles the compilation of the CSS file and **style-loader** adds the compiled CSS files to the React DOM. Webpack reads these settings from right to left, and the CSS needs to be compiled before it's attached to the DOM.*

The application should run in the browser correctly now and should have picked up some small styling changes from the default Bootstrap stylesheet. Let's make some changes to the **index.js** file first and style it as the container for the entire application. We need to

change the **App** component that is rendered to the DOM and wrap the **List** component with a **div** container:

```
// ...
function App() {
  return (
-    <div>
+    <div className='container'>
      <h1>Rick and Morty</h1>
      <List />
    </div>
  );
};

const root = ReactDOM.createRoot(
  document.getElementById('root')
);
root.render(<App />);
```

Inside the **List** component, we need to set the grid to display the **Characters** components, which display the character information. Wrap the **map** function in a **div** element to treat it as a row container for Bootstrap:

```
// ...
return (
  <div>
    <h2>Characters</h2>
```

```

+   <div className='row'>
      {loading ? (
        <div>Loading...</div>
      ) : (
        // ...
      )}
+   </div>
    </div>

    );
  }

  export default List;

```

The code for the **Character** component must also be altered to add styling using Bootstrap; you can replace the current contents of that file with the following:

```

function Character(character) {
  return (
    <div className='col-3'>
      <div className='card'>
        <img
          src={character.image}
          alt={character.name}
          className='card-img-top'
        />

```

```

        <div className='card-body'>
            <h3 className='card-title'>
{character.name}</h3>
            <p>{'Origin: ${character.origin &&
            character.origin.name}'}</p>
        </div>
    </div>
</div>
);
};
export default Character;

```

This lets us use the Bootstrap container layout with a column size of 3 (<https://getbootstrap.com/docs/5.0/layout/columns/>) and style the **Character** component as a Bootstrap card component (<https://getbootstrap.com/docs/5.0/components/card/>).

To add the finishing touches, open the `index.js` file and insert the following code to add a header that will be placed above our list of *Rick and Morty* characters in the application:

```

// ...
function App() {
    return (
        <div className='container'>
-         <h1>Rick and Morty</h1>

```

```

+      <nav className='navbar sticky-top
navbar-light
      bg-dark'>
+      <h1 className='navbar-brand text-
light'>
      Rick and Morty</h1>
+    </nav>
    <List />
  </div>
  );
  // ...

```

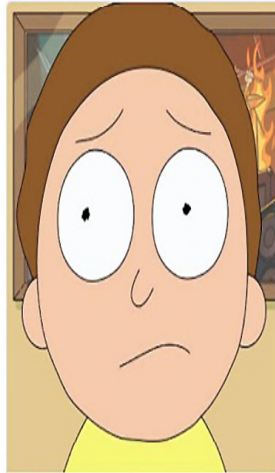
After making sure that the development server is running, we'll see that the application has had styling applied through Bootstrap, which will make it look as follows in the browser:

Characters



Rick Sanchez

Origin: Earth (C-137)



Morty Smith

Origin: Earth (C-137)



Summer Smith

Origin: Earth (Replacement Dimension)



Beth Smith

Origin: Earth (Replacement Dimension)



Jerry Smith

Origin: Earth (Replacement Dimension)



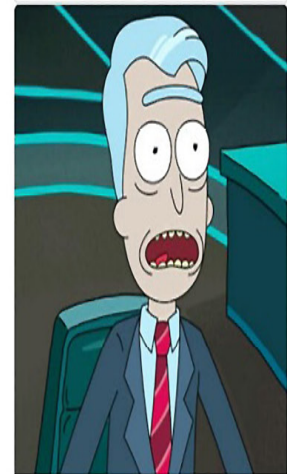
Abadango Cluster Princess

Origin: Abadango



Abradolf Lincler

Origin: Earth (Replacement Dimension)



Adjudicator Rick

Origin: unknown

Figure 1.4 – Our application styled with Bootstrap

The style rules from Bootstrap have been applied to our application, making it look far more complete than it did before. In the final part of this section, we'll add the **ESLint** package to the project, which will make maintaining our code easier by synchronizing patterns across the project.

Adding ESLint

Finally, we will add ESLint to the project to make sure our code meets certain standards – for instance, that our code follows the correct JavaScript patterns.

Install ESLint from npm by running the following command:

```
npm install --save-dev eslint eslint-webpack-plugin eslint-plugin-react
```

The first package, called **eslint**, is the core package and helps us identify any potentially problematic patterns in our JavaScript code. **eslint-webpack-plugin** is a package that is used by Webpack to run ESLint every time we update our code. Finally, **eslint-plugin-react** adds specific rules to ESLint for React applications.

To configure ESLint, we need to create a file called **.eslintrc** in the project's root directory and add the following code to it:

```
{
  "env": {
    "browser": true,
    "node": true,
    "es6": true
  },
  "parserOptions": {
    "ecmaVersion": 2020,
    "sourceType": "module"
  },
  "plugins": ["react"],
  "extends": ["eslint:recommended",
    "plugin:react/recommended"],
  "rules": {
    "react/react-in-jsx-scope": "off"
  }
}
```

The **env** field sets the actual environment our code will run in and will use **es6** functions in it, while the **parserOptions** field adds extra configuration for using **jsx** and modern JavaScript. Where things get interesting, however, is the **plugins** field, which is where we specify that our code uses **react** as a framework. The **extends** field is where the **recommended** settings for **eslint** are used, as well as framework-specific settings for React. Also, the **rules** field contains a rule to dis-

able the notification about React not being imported, as this is no longer required in React 18.

NOTE

We can run the `eslint --init` command to create custom settings, but using the preceding settings is recommended so that we ensure the stability of our React code.

If we look at our command line or browser, we will see no errors. However, we have to add the `eslint-webpack-plugin` package to the Webpack configuration. In the `webpack.config.js` file, you need to import this package and add it as a plugin to the configuration:

```
const HtmlWebpackPlugin = require('html-  
webpack-plugin');  
  
+ const ESLintPlugin = require('eslint-  
webpack-plugin');  
  
module.exports = {  
  // ...  
  plugins: [  
    new HtmlWebpackPlugin({  
      template: './public/index.html',  
      filename: './index.html',  
    }),  
  ],  
};
```



```
+    new ESLintPlugin(),  
    ],  
  };
```

By restarting the development server, Webpack will now use ESLint to check whether our JavaScript code complies with the configuration of ESLint. In our command line (or the **Console** tab in the browser), any misuse of React (or JavaScript) functionalities will be shown.

Congratulations! You have created a basic React application from scratch using React, ReactDOM, Webpack, Babel, and ESLint.

Summary

In this chapter, you've created a single-page application for React from scratch and learned about core React concepts. This chapter started with you creating a new project with Webpack and Babel. These libraries help you compile and run your JavaScript and React code in a browser with minimal setup. Then, we described how to structure a React application, and this structure will be used throughout this book. Also, you learned about state management and data fetching using React Hooks and basic styling with Bootstrap. The principles that were applied provided you with the basics from which to create React applications from nothing and structure them in a scalable way.

If you've worked with React before, then these concepts probably weren't that hard to grasp. If you haven't, then don't worry if some concepts felt strange to you. The upcoming chapters will build upon the features that you used in this chapter, giving you enough time to fully understand them.

The project you'll build in the next chapter will focus on creating reusable React components with more advanced styling. This will be available offline, since it will be set up as a **Progressive Web Application (PWA)**.

Further reading

- Thinking in React: <https://reactjs.org/docs/thinking-in-react.html>
- Bootstrap: <https://getbootstrap.com/docs/4.3/getting-started/introduction/>
- ESLint: <https://eslint.org/docs/user-guide/getting-started>

Chapter 2: Creating a Portfolio in React with Reusable Components and Routing

Do you already feel familiar with React's core concepts after completing the first chapter? Great! This chapter will be no problem for you! If not, don't worry – most of the concepts you came across in the previous chapter will be repeated. However, if you want to get more experience with Webpack and Babel, it's recommended that you try creating the project in [*Chapter 1*](#), *Creating a Single-Page Application in React*, again since this chapter won't be covering those topics.

In this chapter, you'll work with **Create React App**, a starter kit created by the React core team to get you started with React quickly. It will make the configuration of module bundlers and compilers such as Webpack and Babel unnecessary, as this will be taken care of by the Create React App package. This means you can focus on building your portfolio application, which reuses React components and has routing. Besides that, we'll be adding routing using react-router v6, which is the leading library for routing in React.

Alongside setting up Create React App, the following topics will be covered in this chapter:

- Creating a new project with Create React App
- Building reusable React components
- Routing with react-router

Can't wait? Let's go!

Project overview

In this chapter, we will create an application with React that makes use of reusable React components and styling using Create React App and **styled-components**. The application will use data that is fetched from the public GitHub API.

The build time is 1.5–2 hours.

Getting started

The project you'll create in this chapter will use the public API from GitHub, which you can find at <https://docs.github.com/en/rest>. To use this API, you need to have a GitHub account, since you'll want to retrieve information from a GitHub user account. If you don't have a GitHub account yet, you can create one on the GitHub website. The complete source code for this application can also be found on Git-

Hub: <https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter02>.

Creating a portfolio in React

In this section, we will learn how to create a new React project using Create React App and add reusable React components and routing with `react-router`.

Creating a portfolio with Create React App

Having to configure Webpack and Babel every time we create a new React project can be quite time-consuming. Also, the settings for every project can change, and it becomes hard to manage all of these configurations when we want to add new features to our project.

Therefore, the React core team introduced a starter kit known as Create React App, which is currently at version 5. By using Create React App, we no longer have to worry about managing compile and build configurations, even when newer versions of React are released, which means we can focus on coding instead of configurations.

This section will show us how to create a React application with Create React App.

Before anything else, let's see how to install Create React App.

Installing Create React App

Create React App doesn't have to be installed globally. Instead, we can use **npx**, a tool that comes preinstalled with npm (v5.2.0 or higher) and simplifies the way that we execute **npm** packages:

```
npx create-react-app chapter-2
```

This will start the installation process for Create React App, which can take several minutes, depending on your hardware. Although we're only executing one command, the installer for Create React App will install the packages we need to run our React application. Therefore, it will install **react**, **react-dom**, and **react-scripts**, where the last package includes all the configurations for compiling, running, and building React applications.

If we move into the project's root directory, which is named after our project name, we will see that it has the following structure:

```
chapter-2
|- node_modules
|- package.json
|- public
    |- index.html
|- src
```

```
| - App.css  
| - App.test.js  
| - App.js  
| - index.css  
| - index.js
```

NOTE

Not all files that were created by Create React App are listed; instead, only the ones used in this chapter are listed.

This structure looks a lot like the one we set up in the first chapter, although there are some slight differences. The **public** directory includes all the files that shouldn't be included in the compile and build process, and the files inside this directory are the only files that can be directly used inside the **index.html** file.

In the other directory, called **src**, we will find all the files that will be compiled and built when we execute any of the scripts inside the **package.json** file. There is a component called **App**, which is defined by the **App.js**, **App.test.js**, and **App.css** files, and a file called **index.js**, which is the entry point for Create React App.

If we open the **package.json** file, we'll see that four scripts have been defined: **start**, **build**, **test**, and **eject**. Since the last two aren't handled at this point, we can ignore these two scripts for now. To be able

to open the project in the browser, we can simply type the following command into the command line, which runs **package react-scripts** in development mode:

```
npm start
```

NOTE

*Instead of **npm start**, we can also run **yarn start**, as using Yarn is recommended by Create React App.*

If we visit <http://localhost:3000/>, the default Create React App page will look as follows:

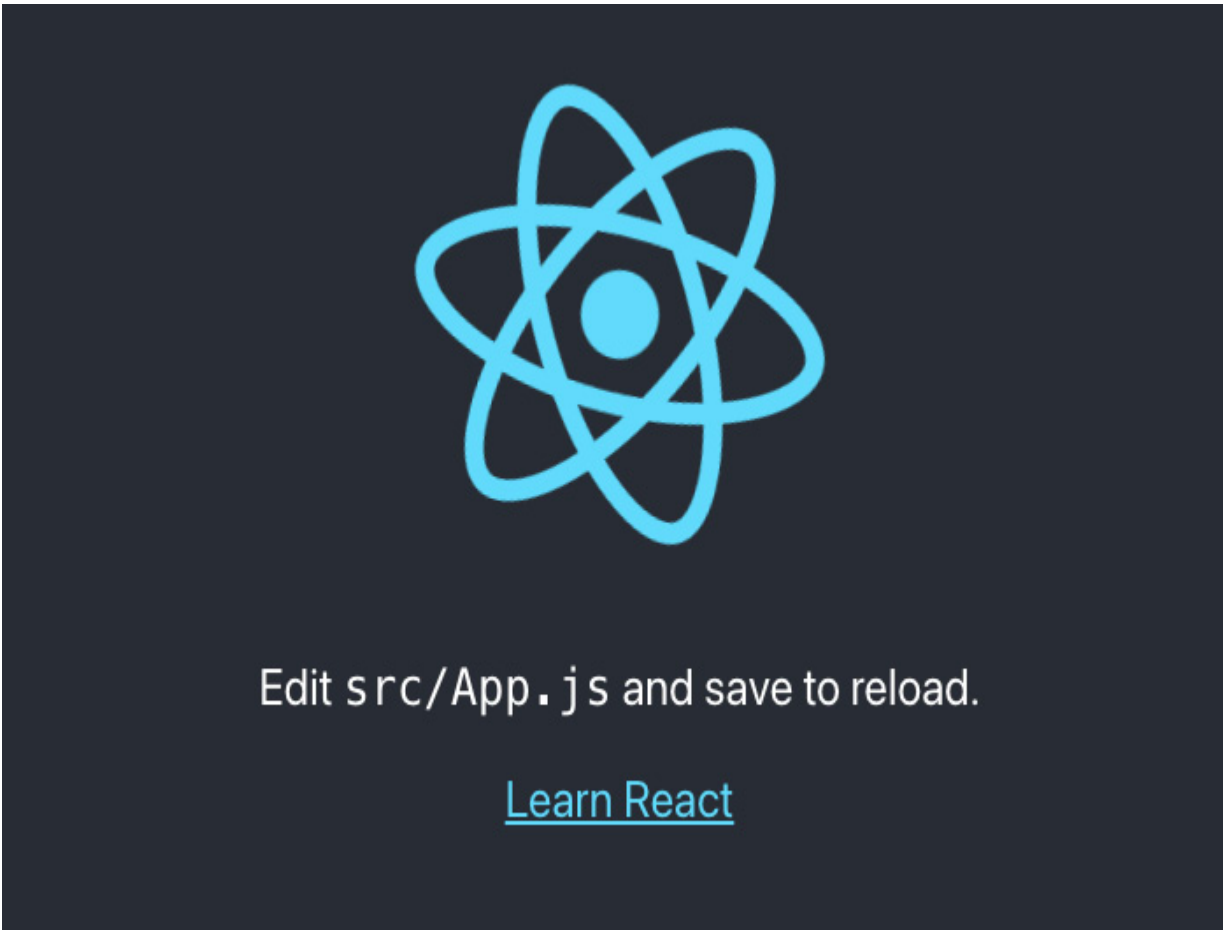


Figure 2.1 – The default Create React App boilerplate

Since **react-scripts** supports hot reloading by default, any changes we make to the code will result in a page reload. If we run the build script, a new directory called **build** will be created in the project's root directory, where the minified bundle of our application can be found.

With the basic installation of Create React App in place, we will start looking at creating the components for our project and styling them.

Building reusable React components

Creating React components with JSX was briefly discussed in the previous chapter, but in this chapter, we'll explore this topic further by creating components that we can reuse throughout our application. First, let's look at how to structure our application, which builds upon the contents of the previous chapter.

Structuring our application

Our project still consists of only one component, which doesn't make it very reusable. To begin, we'll need to structure our application in the same way that we did in the first chapter. This means that we need to split up the **App** component into multiple smaller components. If we look at the source code for this component in **App.js**, we'll see that there's already a CSS **header** element in the **return** function. Let's change that **header** element into a React component:

1. First, create a new file called **Header.css** inside a new directory called **components** within **src** and copy the styling for **classNames**, **App-header**, **App-logo**, and **App-link** into it:

```
.App-logo {  
  height: 40vmin;  
  pointer-events: none;  
}  
  
@media (prefers-reduced-motion: no-preference) {
```

```
.App-logo {
  animation: App-logo-spin infinite 20s
linear;
}

.App-header {
  background-color: #282c34;
  min-height: 100vh;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
  color: white;
}

.App-link {
  color: #61dafb;
}

@keyframes App-logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}
```

```
}  
}
```

2. Now, create a file called **Header.js** inside this directory. This file should return the same content as the **<header>** element:

```
import './Header.css';  
function Header() {  
  return (  
    <header className='App-header'>  
      <img src={logo} className='App-logo'  
alt='logo'  
      />  
      <p>Edit <code>src/App.js</code>  
        and save to reload. </p>  
      <a  
        className='App-link'  
        href='https://reactjs.org'  
        target='_blank'  
        rel='noopener noreferrer'  
      >  
        Learn React  
      </a>  
    </header>  
  );  
}
```

```
export default Header;
```

3. Import this **Header** component inside your **App** component and add it to the **return** function:

```
+ import Header from './components/Header';
import './App.css';
import logo from './logo.svg';
function App() {
  return (
    <div className="App">
-      <header className='App-header'>
-        <img src={logo} className='App-
logo'
          alt='logo' />
-        <p>Edit <code>src/App.js</code>
and save to
          reload. </p>
-        <a
-          className='App-link'
-          href='https://reactjs.org'
-          target='_blank'
-          rel='noopener noreferrer'
-        >
-          Learn React
-        <a>
```

```
-    </header>
+    <Header />
    </div>
  );
}
export default App;
```

The styles for the header need to be deleted from **App.css**. This file should only contain the following style definitions:

```
.App {
  text-align: center;
}
.App-link {
  color: #61dafb;
}
```

When we visit our project in the browser again, we'll see an error saying that the value for the logo is undefined. This is because the new **Header** component can't reach the **logo** constant that's been defined inside the **App** component. From what we learned in the first chapter, we know that this **logo** constant should be added as a prop to the **Header** component so that it can be displayed. Let's do this now:

1. Send the **logo** constant as a prop to the **Header** component in **src/App.js**:

```

// ...
function App() {
  return (
    <div className='App'>
-     <Header />
+     <Header logo={logo} />
    </div>
  );
}
}

export default App;

```

2. Get the **logo** prop so that it can be used by the **img** element as an **src** attribute in **src/components/Header.js**:

```

import './Header.css';
- function Header() {
+ function Header({ logo }) {
  return (
    <header className='App-header'>
      // ...

```

Here, we won't see any visible changes when we open the project in the browser. But if we open up the React Developer Tools, we will see that the project is now divided into an **App** component and a **Header** component. This component receives the **logo** prop in the form of a **.svg** file, as shown in the following screenshot:

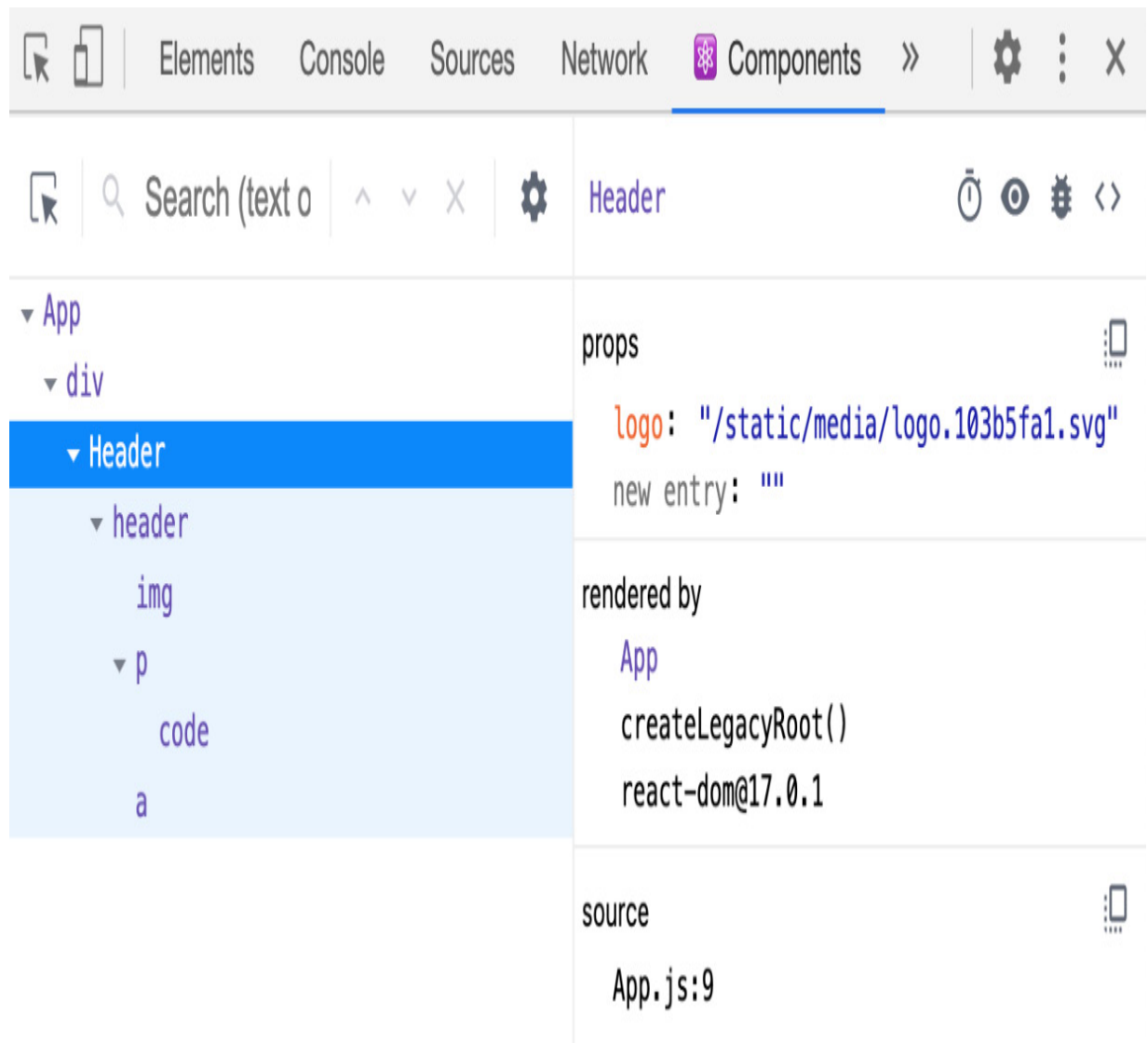


Figure 2.2 – The React Developer Tools

The **Header** component is still divided into multiple elements that can be split into separate components. Looking at the **img** and **p** elements, they look pretty simple already. However, the **a** element looks more complicated and takes attributes such as **url**, **title**, and **className**. To change this **a** element into a component we can reuse, it needs to be moved to a different location in our project.

To do this, create a new file called **Link.js** inside the **components** directory. This file should return the same **a** element that we've already got inside our **Header** component. Also, we can send both **url** and **title** to this component as a prop. Let's do this now:

1. Delete the styling for the **App-link** class from **src/components/Header.css** and place this inside a file called **Link.css**:

```
.App-link {  
  color: #61dafb;  
}
```

2. Create a new component called **Link** that takes the **url** and **title** props. This component adds these props as attributes to the **<a>** element in **src/components /Link.js**:

```
import './Link.css';  
function Link({ url, title }) {  
  return (  
    <a  
      className='App-link'  
      href={url}  
      target='_blank'  
      rel='noopener noreferrer'  
    >  
      {title}  
    </a>  
  )  
}
```

```
    );  
  };  
  export default Link;
```

3. Import this **Link** component and place it inside the **Header** component in `src/components/Header.js`:

```
+ import Link from './Link.js';  
import './Header.css';  
function Header({ logo }) {  
  return (  
    <header className='App-header'>  
      <img src={logo} className='App-  
logo'  
        alt='logo' />  
      <p>Edit <code>src/App.js</code> and  
save to  
      reload. </p>  
-      <a  
-        className='App-link'  
-        href='https://reactjs.org'  
-        target='_blank'  
-        rel='noopener noreferrer'  
-      >  
-        Learn React  
-      <a>
```

```
+      <Link
+          url='https://reactjs.org'
+          title='Learn React'
+      />
    </header>
  );
}
export default Header;
```

4. Our code should now look like the following, meaning that we've successfully split the **App** component into different files in the **components** directory. Also, the **logo.svg** file can be moved to a new directory called **assets**:

```
chapter-2
|- node_modules
|- package.json
|- public
  |- index.html
|- src
  |- assets
    |- logo.svg
  |- components
    |- Header.css
    |- Header.js
    |- Link.css
```

```
    |- Link.js
  |- App.css
  |- App.js
  |- index.css
  |- index.js
```

5. Don't forget to also change the **import** statement in the **src/App.js** file, where the **logo.svg** file is imported as a component:

```
import Header from './components/Header';
import './App.css';
- import logo from './logo.svg';
+ import logo from './assets/logo.svg';
function App() {
  return (
    // ...
```

However, if we take a look at the project in the browser, no visible changes are present. In the React Developer Tools, however, the structure of our application has already taken shape. The **App** component is shown as the parent component in the component tree, while the **Header** component is a child component that has **Link** as a child.

In the next part of this section, we'll add more components to the component tree of this application and make these reusable throughout the application.

Reusing components in React

The project we're building in this chapter is a portfolio page; it will show our public information and a list of public repositories. Therefore, we need to fetch the official GitHub REST API (v3) and pull information from two endpoints. Fetching data is something we did in the first chapter, but this time, the information won't come from a local JSON file. The method to retrieve the information is almost the same. We'll use the `fetch` API to do this.

We can retrieve our public GitHub information from GitHub by executing the following command (replace `username` at the end of the bold section of code with your own username):

```
curl 'https://api.github.com/users/username'
```

NOTE

If you don't have a GitHub profile or haven't filled out all the necessary information, you can also use the `octocat` username. This is the username of the GitHub mascot and is already filled with sample data.

This request will return the following output:

```
{  
  "login": "octocat",
```

```
"id": 583231,
"node_id": "MDQ6VXNlcjU4MzIzMQ==",
"avatar_url":
  "https://avatars.githubusercontent.com/u/
583231?v=4",
"gravatar_id": "",
"url":
"https://api.github.com/users/octocat",
"html_url": "https://github.com/octocat",
"followers_url":
  "https://api.github.com/users/octocat/fol
lowers",
"following_url":
  "https://api.github.com/users/octocat/fol
lowing{
  /other_user}",
"gists_url":
  "https://api.github.com/users/octocat/gis
ts{/gist_id}",
"starred_url":
  "https://api.github.com/users/octocat/sta
rred{/owner}{
  /repo}",
"subscriptions_url":
```

```
    "https://api.github.com/users/octocat/subscriptions",
    "organizations_url":
        "https://api.github.com/users/octocat/orgs",
    "repos_url":
        "https://api.github.com/users/octocat/repos",
    "type": "User",
    "site_admin": false,
    "name": "The Octocat",
    "company": "@github",
    "blog": "https://github.blog",
    "location": "San Francisco",
    "email": null,
    "hireable": null,
    "bio": null,
    "twitter_username": null,
    "public_repos": 8,
    "public_gists": 8,
    "followers": 3555,
    "following": 9
}
```

Multiple fields in the JSON output are highlighted, since these are the fields we'll use in the application. These are **avatar_url**, **html_url**, **repos_url**, **name**, **company**, **location**, **email**, and **bio**, where the value of the **repos_url** field is actually another API endpoint that we need to call to retrieve all the repositories of this user. This is something we'll do later in this chapter.

Since we want to display this result in the application, we need to do the following:

1. To retrieve this public information from GitHub, create a new component called **Profile** inside a new directory called **pages**. This directory will hold all the components that represent a page in our application later on. In this file, add the following code to **src/pages/Profile.js**:

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

function Profile({ userName }) {
  const [loading, setLoading] =
useState(false);
  const [profile, setProfile] =
useState({});
  useEffect(() => {
    async function fetchData() {
      const profile = await fetch(
```



```

        'https://api.github.com/users/${user
rName}')');
    const result = await profile.json();
    if (result) {
        setProfile(result);
        setLoading(false);
    }
}
fetchData();
}, [userName]);
return (
    <div>
        <h2>About me</h2>
        {loading ? (
            <span>Loading...</span>
        ) : (
            <ul></ul>
        )}
    </div>
);
}
export default Profile;

```

This new component imports two Hooks from React, which are used to handle state management and life cycles. We've already used a

`useState` Hook in the previous chapter, and it's used to create a state for `loading` and `profile`. Inside the second Hook, which is the `useEffect` Hook, we do the asynchronous data fetching from the GitHub API. No result has been rendered yet, since we still need to create new components to display the data.

2. Now, import this new component into the `App` component and pass the `userName` prop to it. If you don't have a GitHub account, you can use the username `octocat`:

```
import Header from './Header';
+ import Profile from './pages/Profile';
import './App.css';
function App() {
  return (
    <div className='App'>
      <Header logo={logo} />
+      <Profile userName="octocat" />
    </div>
  );
}
export default App;
```

3. A quick look at the browser where our project is running shows that this new `Profile` component isn't visible yet. This is because the `Header.css` file has a `height` attribute with a `view-height` value

of **100**, meaning that the component will take up the entire height of the page. To change this, open the `src/components/Header.css` file and change the following highlighted lines:

```
.App-logo {  
-   height: 40vmin;  
+   height: 60px;  
    pointer-events: none;  
}  
// ...  
.App-header {  
    background-color: #282c34;  
-   min-height: 100vh;  
+   min-height: 100%;  
    display: flex;  
    flex-direction: column;  
    align-items: center;  
    justify-content: center;  
    font-size: calc(10px + 2vmin);  
    color: white;  
}
```

4. There should be enough free space on our page to display the **Profile** component, so we can open the `src/pages/Profile.js` file once more and display the `avatar_url`, `html_url`, `repos_url`,

name, **company**, **location**, **email**, and **bio** fields that were returned by the GitHub API:

```
// ...
return (
  <div>
    <h2>About me</h2>
    {loading ? (
      <span>Loading...</span>
    ) : (
      <ul>
+      <li><span>avatar_url: </span>
        {profile.avatar_url}</li>
+      <li><span>html_url: </span>
        {profile.html_url}</li>
+      <li><span>repos_url: </span>
        {profile.repos_url}</li>
+      <li><span>name: </span>
        {profile.name}</li>
+      <li><span>company: </span>
        {profile.company}</li>
+      <li><span>location: </span>
        {profile.location}</li>
+      <li><span>email: </span>
        {profile.email}</li>
```

```

+      <li><span>bio: </span>
{profile.bio}</li>
      </ul>
    )}
  </div>
);
}
export default Profile;

```

Once we've saved this file and visited our project in the browser, we will see a bullet list of the GitHub information being displayed.

Since this doesn't look very pretty and the header doesn't match the content of the page, let's make some changes to the **styling** files for these two components:

1. Change the code for the **Header** component so that it will display a different title for the page. Also, the **Link** component can be deleted from here, as we'll be using it in a **Profile** component later on:

```

import './Header.css';
- import Link from './Link';
function Header({ logo }) {
  return (
    <header className='App-header'>

```

```

        <img src={logo} className='App-
logo'
        alt='logo' />
-      <p>
-      Edit <code>src/App.js</code> and
save to
        reload.
-    </p>
-    <Link url='https://reactjs.org'
        title='Learn React' />
+    <h1>My Portfolio</h1>
    </header>
  );
}
export default Header;

```

2. Before changing the styling of the **Profile** component, we first need to create a CSS file that will hold the styling rules for the component. To do so, create the **Profile.css** file in the **pages** directory and add the following content:

```

.Profile-container {
  width: 50%;
  margin: 10px auto;
}
.Profile-avatar {

```

```

    width: 150px;
  }
  .Profile-container > ul {
    list-style: none;
    padding: 0;
    text-align: left;
  }
  .Profile-container > ul > li {
    display: flex;
    justify-content: space-between;
  }
  .Profile-container > ul > li > span {
    font-weight: 600;
  }

```

3. In `src/pages/Profile.js`, we need to import this file to apply the styling. Remember the `Link` component we created previously? We also import this file, as it will be used to create a link to our profile and a list of repositories on the GitHub website:

```

import { useState, useEffect } from
'react';
+ import Link from '../components/Link';
+ import './Profile.css';
function Profile({ userName }) {

```

```
// ..
```

4. In the **return** statement, we'll add the **classNames** function that we defined in the styling and separate the avatar image from the bullet list. By doing that, we also need to wrap the bullet list with an extra **div**:

```
// ...
return (
-   <div>
+   <div className='Profile-container'>
      <h2>About me</h2>
      {loading ? (
        <span>Loading...</span>
      ) : (
+     <div>
+       <img
+         className='Profile-avatar'
+         src={profile.avatar_url}
+         alt={profile.name}
+       />
        <ul>
-         <li><span>avatar_url: </span>
+           {profile.avatar_url}</li>
-         <li><span>html_url: </span>
+           {profile.html_url}</li>
```



```

-      <li><span>repos_url: </span>
        {profile.repos_url}</li>
+      <li>
+        <span>html_url: </span>
+        <Link url={profile.html_url}
+          title={profile.html_url} />
+      </li>
+      <li>
+        <span>repos_url: </span>
+        <Link url={profile.repos_url}
+          title={profile.repos_url}
+      />
+      </li>
      <li><span>name: </span>
        {profile.name}</li>
      <li><span>company: </span>
        {profile.company}</li>
      <li><span>location: </span>
        {profile.location}</li>
      <li><span>email: </span>
        {profile.email}</li>
      <li><span>bio: </span>
        {profile.bio}</li>
    </ul>

```

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

Finally, we can see that the application is starting to look like a portfolio page loading your GitHub information, including your avatar and a list of the public information. This results in an application that looks similar to what's shown in the following screenshot:



My Portfolio

About me



- html_url: <https://github.com/octocat>
- repos_url: <https://api.github.com/users/octocat/repos>
- name: The Octocat
- company: @github
- location: San Francisco
- email:
- bio:

Figure 2.3 – Our styled portfolio application

If we take a look at the code in the **Profile** component, we'll see that there is a lot of duplicate code, so we need to transform the list that's displaying our public information into a separate component. Let's get started:

1. Create a new file called **List.js** inside the **components** directory, which will take a prop called **items**:

```
function List({ items }) {  
  return (  
    <ul></ul>  
  );  
}  
  
export default List;
```

2. In the **Profile** component, we can import this new **List** component. A new variable called **items** should be created, which is an array containing all the items we want to display inside this list:

```
import { useState, useEffect } from  
'react';  
  
+ import List from '../components/List';  
import Link from '../components/Link';  
import './Profile.css';  
  
function Profile({ userName }) {  
  // ...  
  
+   const items = [  
+     {
```

```

+      field: 'html_url',
+      value: <Link url={profile.html_url}
+           title={profile.html_url} />,
+    },
+    {
+      field: 'repos_url',
+      value: <Link url=
+{profile.repos_url}
+           title={profile.repos_url} />,
+    },
+    { field: 'name', value: profile.name
+  },
+    { field: 'company', value:
+profile.company },
+    { field: 'location', value:
+profile.location },
+    { field: 'email', value:
+profile.email },
+    { field: 'bio', value: profile.bio },
+  ];
+  // ...

```

3. This will be sent as a prop to the `List` component, so these items can be rendered from that component instead. This means that you can remove the `ul` element and all the `li` elements inside:

```

    // ...
    return (
      <div className='Profile-container'>
        <h2>About me</h2>
        {loading ? (
          <span>Loading...</span>
        ) : (
          <div>
            <img
              className='Profile-avatar'
              src={profile.avatar_url}
              alt={profile.name}
            />
-          <ul>
-            // ...
-          </ul>
+          <List items={items} />
          </div>
        )}
      </div>
    );
  }
}

export default Profile;

```

You can see that for the list item with the `html_url` and `repos_url` fields, we'll be sending the `Link` component as a value instead of the value that was returned from the GitHub API. In React, you can also send complete components as a prop to a different component, as props can be anything.

4. In the `List` component, we can now map over the `items` prop and return the list items:

```
// ...
function List({ items }) {
  return (
    <ul>
+      {items.map((item) => (
+        <li key={item.field}>
+          <span>{item.field}: </span>
+          {item.value}
+        </li>
+      ))}
    </ul>
  );
}
export default List;
```

The styling is inherited from the `Profile` component, as the `List` component is a child component. To structure your application better,

you can move the styling for the list of information to a separate **List.css** file and import it inside the **List** component.

Assuming we executed the preceding steps correctly, your application shouldn't have changed aesthetically. However, if we take a look at the React Developer Tools, we will see that some changes have been made to the component tree.

In the next section, we'll add routing with **react-router** and display repositories that are linked to our GitHub account.

Routing with **react-router**

react-router v6 is the most popular library in React for routing, and it supports lots of features to help you get the most out of it. With this library, you can add declarative routing to a React application, just by adding components. These components can be divided into three types: router components, route matching components, and navigation components.

Setting up routing with **react-router** consists of multiple steps:

1. To use these components, you need to install the **react-router** web package, called **react-router-dom**, by executing the following:

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


2. After installing **react-router-dom**, the next step is to import the routing and route matching components from this package into the container component of your application. In this case, that is the **App** component, which is inside the `src` directory:

```
import React from 'react';
+ import { BrowserRouter, Routes, Route }
  from 'react-router-dom';
import logo from './assets/logo.svg';
import './App.css';
import Header from './components/Header';
import Profile from './pages/Profile';
function App() {
  // ...
```

3. The actual routes must be added to the **return** statement of this component, where all of the route matching components (**Route**) must be wrapped in a routing component, called **Router**. When your URL matches a route defined in any of the iterations of **Route**, this component will render the React component that passed as a child:

```
// ...
function App() {
  return (
    <div className='App'>
+      <BrowserRouter>
```

```

        <Header logo={logo} />
-        <Profile userName='octocat' />
+        <Routes>
+            <Route
+                path='/'
+                element={<Profile
userName='octocat' />}
+            />
+        </Routes>
+    </BrowserRouter>
    </div>
    );
}
export default App;

```

If you now visit the project in the browser again at **<http://localhost:3000>**, the **Profile** component will be rendered.

Besides our GitHub profile, we also want to showcase the projects we've been working on. Let's add a new route to the application, which will render all the repositories of our GitHub account:

1. This new component will use the endpoint to get all your repositories, which you can try out by executing the following command

(replace **username** at the end of the bold section of code with your own username):

```
curl  
'https://api.github.com/users/username/repos'
```

The output of calling this endpoint will look something like this:

```
[  
  {  
    "id": 132935648,  
    "node_id":  
    "MDEwOlJlcG9zaXRvcnkxMzI5MzU2NDg=",  
    "name": "boysenberry-repo-1",  
    "full_name": "octocat/boysenberry-repo-1",  
    "private": false,  
    "html_url":  
    "https://github.com/octocat/boysenberry-repo-1",  
    "description": "Testing",  
    "fork": true,  
    "created_at": "2018-05-10T17:51:29Z",  
    "updated_at": "2021-01-13T19:56:01Z",  
    "pushed_at": "2018-05-10T17:52:17Z",
```

```
    "stargazers_count": 9,  
    "watchers_count": 9,  
    "forks": 6,  
    "open_issues": 0,  
    "watchers": 9,  
    "default_branch": "master"  
  },  
  // ...  
]
```

As you can see from the preceding sample response, the repositories data is an array with objects. We'll be using the preceding highlighted fields to display our repositories on the `/projects` route.

2. First, we need to create a new component called **Projects** in the **pages** directory. This component will have almost the same logic for state management and data fetching as the **Profile** component, but it will call a different endpoint to get the repositories instead:

```
import { useState, useEffect } from  
  'react';  
import Link from '../components/Link';  
import List from '../components/List';  
function Projects({ userName }) {  
  const [loading, setLoading] =  
    useState(true);
```

```

    const [projects, setProjects] =
useState({});
    useEffect(() => {
      async function fetchData() {
        const data = await fetch(
          'https://api.github.com/users/${
            userName}/repos',
          );
        const result = await data.json();
        if (result) {
          setProjects(result);
          setLoading(false);
        }
      }
      fetchData();
    }, [userName]);

    // ...

```

3. After putting the information from the endpoint to the local state variable `projects`, we'll use the same **List** component to render the information about the repositories:

```

    // ...
    return (
      <div className='Projects-container'>

```

```

    <h2>Projects</h2>
    {loading ? (
      <span>Loading...</span>
    ) : (
      <div>
        <List items=
{projects.map((project) => ({
      field: project.name,
      value: <Link url=
{project.html_url}
        title={project.html_url} />,
    })))} />
      </div>
    )}
  </div>
);
}
export default Projects;

```

4. To have this component render when we visit the **/profile** route, we need to add it to the **App** component using a **Route** component:

```

import React from 'react';
import { BrowserRouter, Routes, Route }
  from 'react-router-dom';
import logo from './assets/logo.svg';

```

```

import './App.css';
import Header from './components/Header';
import Profile from './pages/Profile';
+ import Projects from './pages/Projects';
function App() {
  return (
    <div className='App'>
      <Header logo={logo} />
      <BrowserRouter>
        <Routes>
          <Route path='/' element=
{ <Profile
          userName='octocat' /> } />
+      <Route path='/projects'
element=
          {<Projects
userName='octocat' />} />
        </Routes>
      </BrowserRouter>
    // ...

```

The **Profile** component will now only be rendered if you visit the `/` route, and the **Projects** component when you visit the `/projects` route. No component will be rendered besides the **Header** component if you visit any other route.

NOTE

You can set a component that will be displayed when no route can be matched by passing `` as a path to the **Route** component.*

Although we have two routes set up, the only way to visit these routes is by changing the URL in the browser. With **react-router**, we can also create dynamic links to visit these routes from any component. In our **Header** component, we can add a navigation bar that renders links to these routes:

```
import './Header.css';
+ import { Link as RouterLink } from 'react-
router-dom';
function Header({ logo }) {
  return (
    <header className='App-header'>
      <img src={logo} className='App-logo'
alt='logo' />
      <h1>My Portfolio</h1>
+      <nav>
+        <RouterLink to='/' className='App-
link'>
+          About me
+        </RouterLink>
```



```

+         <RouterLink to='/projects'
className='App-link'>
+         Projects
+         </RouterLink>
+     </nav>
    </header>
  );
}
export default Header;

```

As we already have a **Link** component defined ourselves, we're importing the **Link** component from **react-router-dom** as **RouterLink**. This will prevent confusion if you make any changes later on, or when you're using an autocomplete feature in your IDE.

Finally, we can add some styling to **Header.css** so that the links to our routes are displayed nicely:

```

.App-header {
  background-color: #282c34;
  min-height: 100%;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
}

```

```
    color: white;
  }
+ .App-header > nav {
+   margin-bottom: 10px;
+ }
+ .App-header > nav > .App-link {
+   margin-right: 10px;
+ }
```

If you now visit the application in the browser at **<http://localhost:3000/projects>**, it should look something like the following screenshot. Clicking on the links in the header will navigate you between the two different routes:



My Portfolio

[About me](#) [Projects](#)

Projects

- boysenberry-repo-1: <https://github.com/octocat/boysenberry-repo-1>
- git-consortium: <https://github.com/octocat/git-consortium>
- hello-world: <https://github.com/octocat/hello-world>
- Hello-World: <https://github.com/octocat/Hello-World>
- linguist: <https://github.com/octocat/linguist>
- octocat.github.io: <https://github.com/octocat/octocat.github.io>
- Spoon-Knife: <https://github.com/octocat/Spoon-Knife>
- test-repo1: <https://github.com/octocat/test-repo1>

Figure 2.4 – The Projects route in our application

With these routes in place, even more routes can be added to the **router** component. A logical one is having a route for individual projects, which has an extra parameter that specifies which projects should be displayed. Therefore, we have a new component called the **ProjectDetailpages** directory, which contains the logic for fetching

an individual repository from GitHub API. This component is rendered when the path matches `/projects/:name`, where **name** stands for the name of the repository that is clicked on on the projects page:

1. This route uses a new component in a file called **ProjectDetail.js**, which is similar to the **Projects** component. You can also create this file in the **pages** directory, except that it will be fetching data from the <https://api.github.com/repos/username/repo> endpoint, where **username** and **repo** should be replaced with your own username and the name of the repository that you want to display:

```
import { useState, useEffect } from
'react';
Import { useParams } from 'react-router-
dom';
function Project({ userName }) {
  const [loading, setLoading] =
useState(false);
  const [project, setProject] =
useState([]);
  const { name } = useParams();
  useEffect(() => {
    async function fetchData() {
      const data = await fetch(
        'https://api.github.com/repos/${
```

```
        userName}/${name}',  
    );  
    const result = await data.json();  
    if (result) {  
        setProject(result);  
        setLoading(false);  
    }  
}  
if (userName && name) {  
    fetchData();  
}  
}, [userName, name]);  
// ...
```

In the preceding section, you can see how the data is retrieved from the GitHub API, using both your username and the name of the repository. The name of the repository comes from the **useParams** Hook from **react-router-dom**, which gets the **name** variable from the URL for you.

2. With the repository data retrieved from GitHub, you can create the **items** variable that is used to render information about this project using the **List** component that we also used in the previous routes. The fields that are added to items are coming from GitHub and can also be seen in the response of the

<https://api.github.com/users/username/repos> endpoint that we inspected previously. Also, the name of the repository is listed previously:

```
// ...
return (
  <div className='Project-container'>
    <h2>Project: {project.name}</h2>
    {loading ? (
      <span>Loading...</span>
    ) : (
      <div></div>
    )}
  </div>
);
}
export default Project;
```

3. To render this component on the `/projects/:name` route, we need to add this component within the **Router** component inside **App.js**:

```
// ...
+ import ProjectDetail from
'./pages/ProjectDetail';
function App() {
  return (
    <div className='App'>
```

```

    <BrowserRouter>
      <Header logo={logo} />
      <Routes>
        <Route exact path="/" element=
          {<Profile userName='octocat'
/>} />

        <Route path='/projects'
elements=
          {<Projects userName='octocat'
/>} />
+      <Route path='/projects/:name'
element=
          {<ProjectDetail
userName='octocat' />}
        />
      </Routes>
    </BrowserRouter>
  );
}

```

4. You can already navigate to this route by changing the URL in the browser, but you also want to add a link to this page in the **Projects** component. Therefore, you need to make changes that will import **RouterLink** from **react-router-dom** and use it instead of your own **Link** component:


```
        )))items} />
      </div>
    )}
  </div>
);
}
export default Projects;
```

If you now visit the **`http://localhost:3000/projects`** page in the browser, you can click on the projects and move on to a new page that shows all the information for a specific project.

With these last changes, you've created a portfolio application that uses **`react-router`** for dynamic routing.

Summary

In this chapter, you used Create React App to create your starter project for a React application, which comes with an initial configuration for libraries such as Babel and Webpack. By doing this, you don't have to configure these libraries yourself and worry about how your React code will run in the browser. We've looked into building reusable components in this chapter and learned how to add dynamic routing with **`react-router`**. With this library, you can create ap-

plications that have tons of routes, and you're able to use changes in the URL to change what is displayed inside your application.

The upcoming chapters will all feature projects that are created with Create React App or other zero-config libraries, meaning that these projects don't require you to make changes to Webpack or Babel.

In the next chapter, we will build upon this chapter by creating a dynamic project management board with React that uses styled-components for styling and reuses logic with custom Hooks.

Further reading

- Using npx: <https://medium.com/@maybekatz/introducing-npx-an-npm-package-runner-55f7d4bd282b>
- Create React App: <https://create-react-app.dev/>
- React Router: <https://reactrouter.com/web/guides/quick-start>

Chapter 3: Building a Dynamic Project Management Board

In the first two chapters of this book, you created two React projects all by yourself, and you should, by now, have a solid understanding of the core concepts of React. The concepts you've used so far will also be used in this chapter to create your third project with React, including some new and more advanced concepts that will show you the strength of using React. Again, if you feel you lack some of the knowledge you'll need to finalize the contents of this chapter, you can always repeat what you have built so far.

This chapter will once again use Create React App, which you used in the previous chapter. During the development of the project management board application for this chapter, we'll create and use a custom Hook for data fetching. HTML5 web APIs will be used to dynamically drag and drop components, which are set up as reusable components with **styled-components**. Following this, you'll use more advanced React techniques to control the data flow throughout your components, such as by creating custom Hooks.

The following topics will be covered in this chapter:

- Handling the data flow

- Working with custom Hooks
- Using HTML5 web APIs
- Styling React with **styled-components**

Project overview

In this chapter, we will create a dynamic project management board that has reusable React components and styling using Create React App and **styled-components**. The application will feature a dynamic drag and drop interface that uses the HTML5 Drag and Drop API.

The build time is 1.5–2 hours.

Getting started

The project that we'll create in this chapter builds upon an initial version that you can find on GitHub:

<https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter03-initial>. The complete source code for this application can also be found on GitHub: <https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter03>.

After downloading the initial application from GitHub, we can start by moving into its root directory and running the **npm install** command.

This will install the core packages from Create React App (**react**, **react-dom**, and **react-scripts**). After the installation, we can start the application by executing the **npm start** command and visit the project in the browser by visiting **http://localhost:3000**.

As shown in the following screenshot, the application has a basic header with a title and is divided into four columns. These columns are the lanes for the project management board and will contain the individual tasks once we've connected the project to the data file.



Figure 3.1 – The initial application

If we look at the project's structure, we'll see that it's structured in the same way as the projects in the previous chapters. The entry point of the application is the `src/index.js` file, which renders a component called **App**, which holds two other components called **Header** and **Board**. The first one is the actual header of the application, while the **Board** component holds the four columns we can see in the application. These columns are represented by the **Lane** component.

In this application, you can see that we've further split up the components into separate directories. Every component in either the **components** or **pages** directory now has its own subdirectory:

```
chapter-3-initial
|- /node_modules
|- /public
|- /src
    |- /components
        |- /Header
            |- Header.css
            |- Header.js
        |- /Lane
            |- Lane.css
            |- Lane.js
```

```
|- /pages
  |- /Board
    |- Board.js
    |- Board.css
  |- App.js
  |- App.css
  |- index.js
  |- index.css
package.json
```

Creating a project management board application

In this section, we'll create a project management board PWA that uses custom Hooks for data fetching and the HTML5 Drag and Drop API to make it dynamic. We're going to use a boilerplate application that is set up with Create React App, which we can find in the GitHub repository for this chapter.

Handling the data flow

With the initial version of the application in place, the next step is to fetch the initial data for the project management board and handle its flow through the components. For this, we will create a custom Hook for data fetching that can be reused in other components.

The first part of this section will show us how to load data from a data source using React life cycle methods and display this in React components.

Loading and displaying the data

Loading and displaying data that is retrieved from a data source is something we did in the previous chapter. The data used in this chapter is coming from a mock REST API, created with My JSON Server from Typicode. Using a file called **db.json**, which is placed in the repository for this book, we can automatically create REST endpoints.

My JSON Server

Fake Online REST server for teams

Create a **JSON file** on **GitHub**

Get **instantly** a **fake server**



Figure 3.2 – Using My JSON Server

Using My JSON Server, the <https://my-json-server.typicode.com/PacktPublishing/React-Projects-Second-Edition/tasks> endpoint returns a list of tasks, which we'll load into our project management board in this section. The response is an array

consisting of objects that contain information about our tasks defined in the `id`, `title`, `body`, and `lane` fields.

This section will explore this further. Follow these steps to get started:

1. We will start by fetching the project data from the data file. To do this, we need to add the necessary functions to the **Board** component. We need these to access the React life cycles using Hooks, which we already did in the earlier chapters:

```
+ import { useState, useEffect } from
  'react';
  import Lane from
  '../components/Lane/Lane';
  import './Board.css';
  // ...

  function Board() {
+   const [loading, setLoading] =
  useState(false);
+   const [tasks, setTasks] = useState([]);
+   const [error, setError] = useState('');
+   useEffect(() => {
+     async function fetchData() {
+       try {
+         const tasks = await fetch(
```

```

        `https://my-json-
server.typicode.com/
        PacktPublishing/React-
Projects-Second-
        Edition/tasks`,
    );
+      const result = await
tasks.json();
+      if (result) {
+        setTasks(result);
+        setLoading(false);
+      }
+    } catch (e) {
+      setLoading(false);
+      setError(e.message);
+    }
+  }
+  fetchData();
+ }, []);
  return (
    // ...

```

In the `useEffect` Hook, the data is fetched inside a `try..catch` statement. This statement catches any errors that are being returned from

the data fetching process and replaces the error state with this message.

2. Now, we can distribute the tasks over the corresponding lanes:

```
// ...
return (
  <div className='Board-wrapper'>
    {lanes.map((lane) => (
      <Lane
        key={lane.id}
        title={lane.title}
+       loading={loading}
+       error={error}
+       tasks={tasks.filter((task) =>
          task.lane === lane.id)}
      />
    ))}
  </div>
);
}

export default Board;
```

In the **return** statement, you can see a function that iterates over the **lanes** constant and that these values are passed as props to the **Lane**

component. Also, something special is going on when we pass the tasks to the components since the `filter` function is being used to only return tasks from the tasks state that match the lane ID.

3. Next, we need to make some changes to the `Lane` component so that it will use the data that we fetched from the REST API to display the tasks for us:

```
+ import Task from '../Task/Task';
  import './Lane.css';
- function Lane({ title }) {
+ function Lane({ title, loading, error,
tasks }) {
  return (
    <div className='Lane-wrapper'>
      <h2>{title}</h2>
+      {loading || error ? (
+        <span>{error || 'Loading...'}
</span>
+      ) : (
+        tasks.map((task) => (
+          <Task
+            key={task.id}
+            id={task.id}
+            title={task.title}
+            body={task.body}
```

```

+           />
+         ))
+       )}
+     </div>
+   );
+ }
+ export default Lane;

```

The **Lane** component now takes three other props, which are **tasks**, **loading**, and **error**, where **tasks** contains the array of tasks from the REST API, **loading** indicates whether the loading message should be displayed, and **error** contains the error message when there is one. We can see that inside the **map** function to iterate over the tasks, the **Task** component that displays the information will be rendered.

4. To create this task, we need to create a directory called **Task** in the **components** directory and place a new file for the **Task** component inside. This new file is called **Task.js**:

```

import './Task.css';

function Task({ title, body }) {
  return (
    <div className='Task-wrapper'>
      <h3>{title}</h3>
      <p>{body}</p>
    </div>
  );
}

```

```
    );  
  }  
  export default Task;
```

5. This component takes its styling from another file that we need to create inside the **Task** directory, which is called **Task.css** and has the following contents:

```
.Task-wrapper {  
  background: darkGray;  
  padding: 20px;  
  border-radius: 20px;  
  margin: 0% 5% 5% 5%;  
}  
  
.Task-wrapper h3 {  
  width: 100%;  
  margin: 0;  
}
```

If we visit our application in a web browser at **http://localhost:3000**, we will see the following:

Project Management Board

To Do

Fix navigation bug

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque egestas dictum libero, vel tristique odio pulvinar vitae.

In Progress

Release new website

hasellus eleifend lacus vitae est ultrices placerat. Nunc at risus id risus venenatis laoreet sit amet cursus neque.

Review

Change button color

Suspendisse ac lorem a neque tempus luctus non aliquam sapien. Cras ut lacus bibendum, placerat nibh eu, tempus neque.

Deploy server on acceptance environment

Pellentesque pharetra fermentum sapien, aliquet ultrices ligula mattis porttitor.

Change layout for the content page

Cras tellus ligula, mattis at facilisis eu, ultricies vel elit. Ut aliquam volutpat lacus, a rutrum sem vulputate non.

Done

Complete the registration flow

In vel commodo ipsum. Duis id ipsum semper, condimentum ipsum sit amet, maximus massa.

Create new database instance

Curabitur nec sem lorem. Donec venenatis, arcu vitae malesuada consequat, dolor ante placerat mi, in fermentum diam ipsum id libero.

Figure 3.3 – Our application with data from the mock REST API

Fetching data from a data source is logic that can be reused throughout our application. In the next section, we will explore how this logic can be reused across multiple components by creating a custom Hook.

Working with custom Hooks

Hooks are a way to use React features for creating local state or to watch for updates in that state using life cycles. But Hooks are also a way to reuse logic that you create for your own React application. This is a pattern that is popular among a lot of libraries that create functionalities for React, such as **react-router**.

NOTE

*Before React introduced Hooks themselves, it was a popular pattern to create **Higher-Order Components (HOCs)** to reuse logic. HOCs are advanced features in React that focus on the reusability of components. The React documentation described them as follows: "A higher-order component is a function that takes a component and returns a new component."*

In the first part of this section, we'll create our first custom Hook, which uses logic to retrieve data from the data source that we created in the previous section.

Creating custom Hooks

We already saw that we can reuse components in React, but the next step is to reuse logic that you have inside these components. To explain what this means in practice, let's create an example. Our project has a **Board** component, which fetches the REST API and renders all the lanes and tasks. There is logic in this component in the form of a local state created with the `useState` Hook, data fetching inside a `useEffect` Hook, and information about how each **Lane** component is being rendered. How will we handle a situation where we just want to show a board without lanes and only tasks? Do we just send different props from the **Board** component? Sure, that's possible, but, in React, that's what custom Hooks are used for.

A **Board** component without lanes wouldn't map over all the lanes and render the corresponding lane with the tasks as a prop. Instead, it would map over all the tasks and render them directly. Although the rendered components are different, the logic to set the initial state, fetch the data, and render the component(s) can be reused. The custom Hook should be able to use the local state and execute data fetching from any component that it's used in.

To create the custom Hook, create a new file called **useDataFetching.js** inside a new directory called **hooks** in the **src** directory. Now, follow these steps:

1. Import the **useState** and **useEffect** Hooks from React and create a new function for the Hook, which becomes the default export. This function will take one parameter called **dataSource**. Since this Hook will use the life cycles for data fetching, let's call this custom Hook **useDataFetching** and have it return an empty array:

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

function useDataFetching(dataSource) {
  return [];
}

export default useDataFetching;
```

2. Inside this function, add the **useState** Hooks to create a local state for **loading**, **error**, and **data**, which has almost the same structure as our local state inside the **Board** component:

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

function useDataFetching(dataSource) {
+   const [loading, setLoading] =
useState(false);
+   const [data, setData] = useState([]);
+   const [error, setError] = useState('');
```

```
    return [];  
  }  
  export default useDataFetching;
```

3. Next, we need to use the **useEffect** hook, which is where the data fetching will be done. The **dataSource** parameter is used as the location to fetch from. Also, notice that the constant names are now more generic and no longer specify a single use:

```
import { useState, useEffect } from  
'react';  
  
function useDataFetching(dataSource) {  
  // ...  
-   return [];  
+   useEffect(() => {  
+     async function fetchData() {  
+       try {  
+         const data = await  
fetch(dataSource);  
+         const result = await data.json();  
+         if (result) {  
+           setData(result);  
+           setLoading(false);  
+         }  
+       } catch (e) {  
+         setLoading(false);
```

```

+         setError(e.message);
+     }
+ }
+     fetchData();
+ }, [dataSource]);
+     return [loading, error, data];
+ }
export default useDataFetching;

```

This adds the method to do data fetching, and in the **return** statement, we're returning the **data**, **loading**, and **error** state.

Congratulations! You've created your very first Hook! However, it still needs to be added to a component that supports data fetching.

Therefore, we need to refactor our **Board** component to use this custom Hook for data fetching instead:

1. Import the custom Hook from the **src/hooks/useDataFetching.js** file and delete the imports of the React Hooks:

```

- import { useState, useEffect } from
  'react';
+ import useDataFetching from
    '../hooks/useDataFetching';
  import Lane from
    '../components/Lane/Lane';

```

```
import './Board.css';  
// ...
```

2. Subsequently, we can delete the usage of the `useState` and `useEffect` Hooks in the `Board` component:

```
// ...  
function Board() {  
-   const [loading, setLoading] =  
useState(false);  
-   const [tasks, setTasks] = useState([]);  
-   const [error, setError] = useState('');  
-   useEffect(() => {  
-       async function fetchData() {  
-           try {  
-               const tasks = await fetch(  
                   `https://my-json-  
server.typicode.com/  
                PacktPublishing/React-Projects-  
Second-  
                Edition/tasks`,  
               );  
-               const result = await  
tasks.json();  
-               if (result) {  
-                   setTasks(result);
```

```

-         setLoading(false);
-     }
-     } catch (e) {
-         setLoading(false);
-         setError(e.message);
-     }
- }
- }
-   fetchData();
- }, []);
  return (
    // ...

```

3. Instead, use the imported custom Hook to handle our data fetching. The Hook returns the **loading**, **error**, and **tasks** state as before, but the data state is renamed **tasks** to fit the needs of our component:

```

import useDataFetching
  from '../hooks/useDataFetching';
import Lane from
'../components/Lane/Lane';
import './Board.css';
function Board() {
+   const [loading, error, tasks] =
      useDataFetching(`https://my-json-
server.

```

```
typicode.com/PacktPublishing/React-  
Projects-  
Second-Edition/tasks`);  
return (  
  // ...
```

In the next section, we'll learn how to reuse a custom Hook by importing it from a different component.

Reusing a custom Hook

With the very first custom Hook in place, it's time to think of other components that could do data fetching, such as a component that is displaying only tasks. The process to create this component consists of two steps: creating the actual component and using the custom Hook for data fetching. Let's get started:

1. Inside the directory `pages`, we need to create a new file called **Backlog.js** in a new directory called **Backlog**. In this file, we can place the following code to create the component, import the custom Hook, and import the CSS for styling:

```
import Task from  
'../../components/Task/Task';  
import useDataFetching from  
'../../hooks/useDataFetching';
```



```

import './Backlog.css';
function Backlog() {
  const [loading, error, tasks] =
useDataFetching(
  'https://my-json-server.typicode.com/
  PacktPublishing/React-Projects-Second-
Edition/
  tasks',
  );
  return (
    <div></div>
  );
}
export default Backlog;

```

2. The **return** statement is now returning an empty **div** element, so we need to add the code to render the tasks here:

```

// ...
return (
-   <div>
+   <div className='Backlog-wrapper'>
+     <h2>Backlog</h2>
+     <div className='Tasks-wrapper'>
+       {loading || error ? (

```

```

+      <span>{error || 'Loading...'}
+    </span>
+  ) : (
+    tasks.map((task) => (
+      <Task
+        key={task.id}
+        title={task.title}
+        body={task.body}
+      />
+    ))
+  )}
+ </div>
+ </div>
+ );
+ }
+ export default Backlog;

```

3. This component imports the **Backlog.css** file for styling, and we've also added classes to the elements in this file. But we also need to create and add some basic styling rules to **Backlog.css**:

```

.Backlog-wrapper {
  display: flex;
  flex-direction: column;
  margin: 5%;
}

```

```
.Backlog-wrapper h2 {  
  width: 100%;  
  padding-bottom: 10px;  
  text-align: center;  
  border-bottom: 1px solid darkGray;  
}  
.Tasks-wrapper {  
  display: flex;  
  justify-content: space-between;  
  flex-direction: row;  
  flex-wrap: wrap;  
  margin: 5%;  
}
```

4. In the **App** component, we can import this component to render it below the **Board** component:

```
import './App.css';  
import Board from './pages/Board/Board';  
import Header from  
'./components/Header/Header';  
+ import Backlog from  
'./pages/Backlog/Backlog';  
function App() {  
  return (  
    <div className='App'>
```

```
        <Header />
        <Board />
+      <Backlog />
    </div>
  );
}
export default App;
```

This will render the new **Backlog** component below our board with all the different tasks. These tasks are the same ones as in the **Board** component, as the same REST API endpoint is used. Also, you can set up **react-router** for this project to render the **Backlog** component on a different page instead.

All the tasks that are displayed in the lanes are only in one part of our application, since we want to be able to drag and drop these tasks into different lanes. We'll learn how to do this in the next section, where we'll add dynamic functionalities to the board.

Making the board dynamic

One of the things that usually give project management boards great user interaction is the ability to drag and drop tasks from one lane into another. This is something that can easily be accomplished using the

HTML5 Drag and Drop API, which is available in every modern browser, including IE11.

The HTML5 Drag and Drop API makes it possible for us to drag and drop elements across our project management board. To make this possible, it uses drag events. **onDragStart**, **onDragOver**, and **onDrop** will be used for this application. These events should be placed on both the **Lane** and the **Task** components.

In the file for the **Board** component, let's add the functions that respond to the drop events, which need to be sent to the **Lane** and **Task** components. Let's get started:

1. Start by adding the **event handler** function for the **onDragStart** event, which fires when the dragging operation is started, to the **Board** component. This function needs to be passed to the **Lane** component, where it can be passed on to the **Task** component. This function sets an ID for the task that is being dragged to the **dataTransfer** object of the element, which is used by the browser to identify the drag element:

```
// ...  
  
+ function onDragStart(e, id) {  
+   e.dataTransfer.setData('id', id);  
+ }  
  
function Board() {
```

```

    const [loading, error, tasks] =
useDataFetching(
    'https://my-json-server.typicode.com/
    PacktPublishing/React-Projects-
Second-Edition/
    tasks',
    );
    return (
    <div className='Board-wrapper'>
    {lanes.map((lane) => (
    <Lane
    key={lane.id}
    title={lane.title}
    loading={loading}
    error={error}
    tasks={tasks.filter((task) =>
    task.lane === lane.id)}
+    onDragStart={onDragStart}
    />
    )})}
    </div>
    );
  }
  export default Board;

```

2. In the **Lane** component, we need to pass this **event handler** function to the **Task** component:

```
// ...  
  
- function Lane({ title, loading, error,  
tasks }) {  
+ function Lane({ title, loading, error,  
tasks,  
  onDragStart }) {  
  return (  
    <div className='Lane-wrapper'>  
      <h2>{title}</h2>  
      {loading || error ? (  
        <span>{error || 'Loading...'}  
</span>  
      ) : (  
        tasks.map((task) => (  
          <Task  
            key={task.id}  
            title={task.title}  
            body={task.body}  
+            onDragStart={onDragStart}  
          />  
        ))  
      )}  
    </div>  
  )
```

```

        </div>
    );
}
export default Lane;

```

- Now, we can invoke this function in the **Task** component, where we also need to add the **draggable** attribute to the **div** element with the class name **Task-wrapper**. Here, we send the element and the task ID as a parameter to the event handler:

```

import './Task.css';

- function Task({ title, body }) {
+ function Task({ id, title, body,
onDragStart }) {
    return (
        <div
            className='Task-wrapper'
+         draggable
+         onDragStart={(e) => onDragStart(e,
id)}
        >
            <h3>{title}</h3>
            <p>{body}</p>
        </div>
    );
}

```



```
export default Task;
```

After making these changes, we should be able to see that each task can be dragged around. But don't drop them anywhere yet – the other drop events and event handlers that update the state should be added as well. Dragging a task from one lane to another can be done by clicking on a task without releasing the mouse and dragging it to another lane, as shown in the following screenshot:

Project Management Board

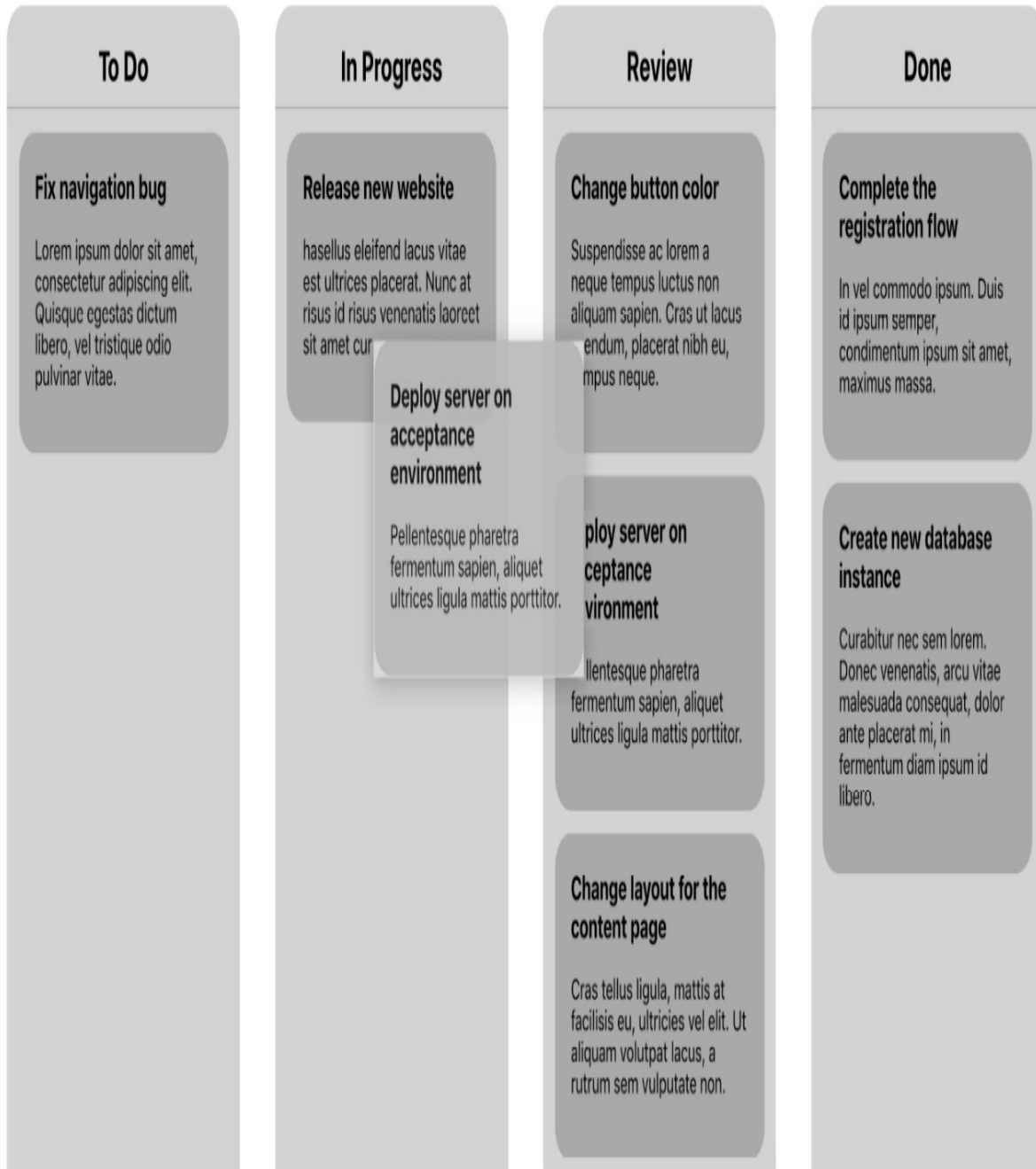


Figure 3.4 – The interactive project management board

With the **onDragStart** event implemented, the **onDragOver** and **onDrop** events can be implemented as well. Let's get started:

1. By default, it's impossible to drop elements into another element – for example, a **Task** component into a **Lane** component. This can be prevented by calling the **preventDefault** method for the **onDragOver** event:

```
// ...
function onDragStart(e, id) {
  e.dataTransfer.setData('id', id);
}
+ function onDragOver(e) {
+   e.preventDefault();
+ };
function Board() {
  const [loading, error, tasks] =
useDataFetching(
  'https://my-json-server.typicode.com/
  PacktPublishing/React-Projects-
Second-Edition/
  tasks',
  );
  return (
    <div className='Board-wrapper'>
      {lanes.map((lane) => (
```

```

        <Lane
          key={lane.id}
          title={lane.title}
          loading={loading}
          error={error}
          tasks={tasks.filter((task) =>
            task.lane === lane.id)}
          onDragStart={onDragStart}
+          onDragOver={onDragOver}
        />
      )})}
    </div>
  );
}

export default Board;

```

2. This function needs to be imported and placed as an event handler on the **div** element with the class name **Lane-wrapper** in the **Lane** component:

```

// ...
- function Lane({ title, loading, error,
  tasks,
    onDragStart }) {
+ function Lane({ title, loading, error,
  tasks,

```

```

    onDragStart, onDragOver }) {
    return (
-      <div className='Lane-wrapper'>
+      <div className='Lane-wrapper'
        onDragOver={onDragOver}>
        <h2>{title}</h2>
        // ...

```

3. The **onDrop** event is where things get interesting, since this event makes it possible for us to mutate the state after we've finished the drag operation. Before we can add this event handler, we need to create a new local state variable called **tasks** in the **Board** component. This state variable is overwritten when the data is being fetched from the **useDataFetching** Hook and is used to display the tasks from the **Lane** components:

```

+ import { useEffect, useState } from
  'react';
  import Lane from
  '../components/Lane/Lane';
  import useDataFetching from
    '../hooks/useDataFetching';
  import './Board.css';
  // ...
  function Board() {
    const [

```

```

        loading,
        error,
-       tasks
+       data] = useDataFetching(
            'https://my-json-
server.typicode.com/
            PacktPublishing/React-Projects-
Second-Edition/
            tasks',
        );
+   const [tasks, setTasks] = useState([]);
+   useEffect(() => {
+       setTasks(data);
+   }, [data]);
    // ...
    return (
        // ...

```

4. The new event handler function can now be created, and when it's invoked, we can call the **setTasks** function from the **useState** Hook for the tasks state:

```

    // ...
    function Board() {
        // ...
+       function onDrop(e, laneId) {

```

```

+     const id =
e.dataTransfer.getData('id');
+     const updatedTasks =
tasks.filter((task) => {
+       if (task.id.toString() === id) {
+         task.lane = laneId;
+       }
+       return task;
+     });
+     setTasks(updatedTasks);
+   }
  return (
    // ...

```

5. Also, this event handler function should be passed as a prop to the **Task** component:

```

    // ...
    Return (
      <div className='Board-wrapper'>
        {lanes.map((lane) => (
          <Lane
            key={lane.id}
+           laneId={lane.id}
            title={lane.title}
            loading={loading}

```

```

        error={error}
        tasks={tasks.filter((task) =>
            task.lane === lane.id)}
        onDragStart={onDragStart}
        onDragOver={onDragOver}
+       onDrop={onDrop}
        />
    ))}
</div>
);
}
export default Board;

```

This **onDrop** event handler function takes an element and the ID of the lane as a parameter because it needs the ID of the dragged element and the new lane it should be placed in. With this information, the function uses a **filter** function to find the task that needs to be moved and changes the ID of the lane. This new information will replace the current object for the task in the state with the **setState** function.

6. Since the **onDrop** event gets fired from the **Lane** component, it is passed as a prop to this component. Also, the ID of the lane is added as a prop because this needs to be passed to the **onDrop** event handler function from the **Lane** component:


```

import Task from '../Task/Task';
import './Lane.css';

function Lane({
+   laneId,
    title,
    loading,
    error,
    tasks,
    onDragStart,
    onDragOver,
+   onDrop,
}) {
    return (
        <div
            className='Lane-wrapper'
            onDragOver={onDragOver}
+         onDrop={(e) => onDrop(e, laneId)}
        >
        // ...

```

With this, we're able to drag and drop tasks onto other lanes in our board – something that you can also do for the **Backlog** component – or even make this logic reusable with another custom Hook. But instead, we'll be looking at how to make the styling for our component

more flexible and reusable by using the **styled-components** library in the next section.

Styling in React with styled-components

So far, we've been using CSS files to add styling to our React components. However, this forces us to import these files across different components, which makes our code less reusable. Therefore, we'll add the **styled-components** package to the project, which allows us to write CSS inside JavaScript (so-called CSS-in-JS) and create components.

By doing this, we'll get more flexibility out of styling our components, be able to prevent the duplication or overlapping of styles due to `classNames`, and add dynamic styling to components with ease. All of this can be done using the same syntax we used for CSS, right inside our React components.

The first step is installing **styled-components** using npm:

```
npm install styled-components
```

NOTE

*If you look at the official documentation of **styled-components**, you will notice that they strongly advise you to use the Babel plugin for this package as well. But since you're using Create React App to initialize your project, you don't need to add this plugin, as all the compilation that your application needs has already been taken care of by **react-scripts**.*

After installing **styled-components**, let's try to delete the CSS file from one of our components. A good start will be the **Task** component since this is a very small component with limited functionality:

1. Start by importing the **styled-components** package and creating a new styled component called **TaskWrapper**. This component extends a **div** element and takes the CSS rules we already have for the **Task-wrapper** class name in **Task.css**. Also, we no longer have to import this file, since all the styling is now being done inside this JavaScript file:

```
+ import styled from 'styled-components';  
- import './Task.css';  
+ const TaskWrapper = styled.div`  
+   background: darkGray;  
+   padding: 20px;  
+   border-radius: 20px;  
+   margin: 0% 5% 5% 5%;  
+   h3 {
```

```

+      width: 100%;
+      margin: 0;
+    }
+  `;

  function Task({ id, title, body,
onDragStart }) {
    return (
-      <div className="Task-wrapper"
+      <TaskWrapper
        draggable
        onDragStart={(e) => onDragStart(e,
id)}
      >
        <h3>{title}</h3>
        <p>{body}</p>
-      </div>
+      </TaskWrapper>
    );
  }

  export default Task;

```

2. In the preceding code block, we've added the styling of the **h3** element in the styled component for **TaskWrapper**, but we can also do this inside a specific styled component as well:

```
import styled from 'styled-components';
```

```

    // ...
-   h3 {
-       width: 100%;
-       margin: 0;
-   }
- `;
+ const Title = styled.h3`
+   width: 100%;
+   margin: 0;
+ `;
    function Task({ id, title, body,
onDragStart }) {
    return (
        <TaskWrapper
            draggable
            onDragStart={(e) => onDragStart(e,
id)}
        >
-       <h3>{title}</h3>
+       <Title>{title}</Title>
        <p>{body}</p>
        </TaskWrapper>
    );

```

```
}  
export default Task;
```

3. We can also do this for the other components in our project, starting with the **Lane** component, for which we first need to create the styled components that use the same styling as those in the **Lane.css** file:

```
+ import styled from 'styled-components';  
  import Task from '../Task/Task';  
- import './Lane.css';  
+ const LaneWrapper = styled.div`  
+   text-align: left;  
+   padding: 0;  
+   background: lightGray;  
+   border-radius: 20px;  
+   min-height: 50vh;  
+   width: 20vw;  
+   @media (max-width: 768px) {  
+     margin-bottom: 5%;  
+   }  
+ `;  
+ const Title = styled.h2`  
+   width: 100%;  
+   padding-bottom: 10px;  
+   text-align: center;
```

```

+   border-bottom: 1px solid darkGray;
+ `;

function Lane({
  // ...

```

4. Replace the existing **div** and **h3** elements with these new components:

```

// ...
function Lane({
  laneId,
  title,
  loading,
  error,
  tasks,
  onDragStart,
  onDragOver,
  onDrop,
}) {
  return (
-   <div className="Lane-wrapper"
+   <LaneWrapper
      onDragOver={onDragOver}
      onDrop={(e) => onDrop(e, laneId)}
    >
-   <h3>{title}</h3>

```

```

+      <Title>{title}</Title>
        {loading || error ? (
          <span>{error || 'Loading...'}
        </span>
        ) : (
          // ...
        )}
-    </div>
+    </LaneWrapper>
  );
}

export default Lane;

```

If we visit our project in the browser after running `npm start` again, we'll see that our application still looks the same after deleting the CSS files for the Ticket and **Lane** component. You can, of course, also do the same thing for the other components in the project.

Let's proceed and convert another component to use **styled-components** instead of CSS – for example, the component in the `src/App.js` file. This one is using the `src/App.css` file to style the `div` element that wraps all the components in our application:

```

- import './App.css';
+ import styled from 'styled-components';

```



```

import Board from './pages/Board/Board';
import Header from
'./components/Header/Header';
import Backlog from
'./pages/Backlog/Backlog';
+ const AppWrapper = styled.div`
+   text-align: center;
+ `;
function App() {
  return (
-   <div className='App'>
+   <AppWrapper>
      <Header />
      <Board />
      <Backlog />
-   </div>
+   </AppWrapper>
  );
}
export default App;

```

After making these changes, you can delete the `src/App.css` file as we're no longer using it to style the **App** component.

Another possibility with **styled-components** is creating a global style for our application, which is currently done in `src/index.css`. This file

is imported in `src/index.js` and therefore loaded into every page of the application, as it is the entry to our React application. But the `App` component in `src/App.js` also wraps all our components, in which we can copy the styling rules from `src/index.css` and use them to create a `GlobalStyle` component:

```
- import styled from 'styled-components';
+ import styled, { createGlobalStyle } from
  'styled-components';
  import Board from './pages/Board/Board';
  import Header from
    './components/Header/Header';
  import Backlog from
    './pages/Backlog/Backlog';
+ const GlobalStyle = createGlobalStyle`
+   body {
+     margin: 0;
+     font-family: -apple-system,
    BlinkMacSystemFont,
      'Segoe UI', 'Roboto', 'Oxygen',
    'Ubuntu',
      'Cantarell', 'Fira Sans', 'Droid
    Sans',
      'Helvetica Neue', sans-serif;
+     -webkit-font-smoothing: antialiased;
```

```

+     -moz-osx-font-smoothing: grayscale;
+   }
+ `;

const AppWrapper = styled.div`
  // ...

```

This global style that we just created must be added to the return statement for the **App** component, above the **AppWrapper** component. As we can only return a single element or component from the **return** statement, we need to wrap the contents into another element. If we wanted to apply styling to this element, we could do this with a **div** element. As we don't want that in this scenario, we'll be using a React fragment instead. With a fragment, we can wrap elements and components without rendering anything in the browser:

```

// ...
function App() {
  return (
+    <>
+      <GlobalStyle />
        <AppWrapper>
          <Header />
          <Board />
          <Backlog />
        </AppWrapper>
+    </>
  )
}

```

```
    );  
  }  
  export default App;
```

NOTE

The `<>` notation is shorthand for `<React.Fragment>`; you can use both notations in React. For the `<React.Fragment>` notation, you can also import `Fragment` from `React` to write `<Fragment>`.

Finally, you can delete the `src/index.css` file and the line in `src/index.js` that imports this file into our application:

```
import React from 'react';  
import ReactDOM from 'react-dom/client';  
- import './index.css';  
import App from './App';  
import reportWebVitals from  
'./reportWebVitals';  
const root = ReactDOM.createRoot(  
  document.getElementById('root')  
);  
// ...
```

With these final additions, we've styled large parts of our application with **styled-components** instead of CSS. By writing the styling rules directly in the components, we can reduce the number of files in the

project and also make it easier to find what styling is applied to our elements.

Summary

In this chapter, you created a project management board that lets you move, drag, and drop tasks from one lane to another using the HTML5 Drag and Drop API. The data flow of this application is handled using local state and life cycles and determines which tasks are displayed in the different lanes. This chapter also introduced the advanced React pattern of custom Hooks. With custom Hooks, you can reuse state logic in function components across your applications.

This advanced pattern will be also be used in the next chapter, which will handle routing and **Server-Side Rendering (SSR)** in React applications using Next.js. Have you ever tried using Stack Overflow to find a solution to a programming issue you once had? I have! In the next chapter, we will be building a community feed that uses Stack Overflow as a data source and React to render the application.

Further reading

- The HTML Drag and Drop API: https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API

- DataTransfer: <https://developer.mozilla.org/en-US/docs/Web/API/DataTransfer>
- React DnD: <https://github.com/react-dnd/react-dnd>

Chapter 4: Building a Server-Side-Rendered Community Feed Using Next.js

So far, you've learned how React applications are typically **Single-Page Applications (SPAs)** and can be kickstarted using Create React App. This means the application is rendered on the client side, making it load in the browser when the user visits your application. But did you know React also supports **Server-Side Rendering (SSR)**, as you might remember from back in the old days when code only rendered from a server?

In this chapter, you'll learn how to create an application that has components dynamically loaded from the server instead of the browser. To enable SSR, we'll be using Next.js instead of Create React App. Next.js is a framework for React applications and adds additional features to React. If you're interested in **Search Engine Optimization (SEO)**, SSR comes with the advantage that we can add metadata to the application so it can be better indexed by search engines.

The following topics will be covered in this chapter:

- Setting up Next.js
- Server-side rendering

- SEO in React

Project overview

In this chapter, we will create a community feed application using Next.js that supports SSR and therefore is loaded from the server rather than the browser. Also, the application is optimized for search engines.

The build time is 2 hours.

Getting started

The complete source code can also be found on GitHub:

<https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter04>. Also, this project uses the publicly available Stack Overflow API to fill the application with data. This is done by fetching questions that are posted to Stack Overflow. More information about this API can be found at: <https://api.stackexchange.com/docs/>.

Community feed application

In this section, you'll build a community feed application with Next.js that supports SSR. Next.js goes beyond the functionalities of Create React App, by delivering a framework to build React applications quickly. It has built-in features for routing, SEO, SSR, and much more, as you'll learn in this chapter. In this community feed, you can see an overview of recent questions on Stack Overflow that have the `reactjs` tag, and you can click on them to see more information and the answers.

Setting up Next.js

In previous chapters, we used Create React App to run a React application. Although Create React App is a good starting point for most React applications, it doesn't support SSR. Luckily, Next.js does offer this feature as it's considered a framework for React. In this chapter, we'll be using the latest stable version of Next.js, which is version 12.

Installing Next.js

To set up Next.js, we run the following command:

```
npx create-next-app chapter-4
```

You'll be asked to select a template, which is either the default starter app or a template that is created by the community. After selecting

the default template, the **react**, **react-dom**, and **next** libraries will be installed, among others.

After the installation is finished, we can move into the new **chapter-4** directory and run the following command:

```
npm run dev
```

This will start the new Next.js application that will become available at **http://localhost:3000**. The default starter app will look something like the following application:

Welcome to Next.js!

Get started by editing `pages/index.js`

Documentation →

Find in-depth information about Next.js features and API.

Learn →

Learn about Next.js in an interactive course with quizzes!

Examples →

Discover and deploy boilerplate example Next.js projects.

Deploy →

Instantly deploy your Next.js site to a public URL with ZEIT Now.

Powered by  ZEIT

Figure 4.1 – The initial Next.js application

In this application, you can not only see what a Next.js application looks like but also find useful links to more sources to learn about Next.js and examples of how to deploy it.

The application structure for a Next.js project is slightly different from how we structured our Create React App in the preceding chapters:

```
chapter-4
|- package.json
|- node_modules
|- public
|- pages
    |- api
        |- hello.js
    |- _app.js
    |- index.js
|- styles
    |- globals.css
    |- Home.module.css
```

In the preceding structure, you can see that there is again a directory called **pages** that will contain React components that will serve as a page. But different from Create React App, we don't need to set up **react-router** to serve the pages. Instead, Next.js automatically renders every component in that directory as a page. In this directory, we also find the **hello.js** file in the **api** directory. Next.js can also be used to create API endpoints, which we'll explore more in [*Chapter 7, Build a Full Stack E-Commerce Application with Next.js and GraphQL*](#). Also, the CSS files for our components are put in the

styles directory, where you'll find the **globals.css** file with global styling and **Home.module.css** with styling for a specific component.

Adding styled-components

Before we set up the routing, let's add **styled-components** to the project, which we've also used in the previous chapters. For this, we need to install **styled-components** by running the following commands:

```
npm install styled-components
```

This will add the packages to our project so we can use them to create and style reusable components.

NOTE

*Next.js uses the **styles** directory to store global and component-specific CSS files for styling. As we're using **styled-components** for styling instead, we don't have to add any new CSS files to this directory. If you're not using a library for styling with CSS-in-JS, you can place both global and component-level styling files in the **styles** directory instead.*

Setting up **styled-components** in Next.js is done slightly differently in comparison to Create React App:

1. In Next.js, a **Document** component wraps the **html**, **body**, and **head** tags and runs them through a so-called **renderPage** method to allow for SSR. We need to overwrite this **Document** component from a new file called **_document.js** in the **pages** folder:

```
import Document from 'next/document';
import { ServerStyleSheet } from 'styled-components';
export default class MyDocument extends Document {
  static async getInitialProps(ctx) {
    const sheet = new ServerStyleSheet();
    const originalRenderPage =
ctx.renderPage;
    try {
      ctx.renderPage = () =>
        originalRenderPage({
          enhanceApp: (App) => (props) =>
            sheet.collectStyles(<App
{...props} />),
        });
      const initialProps =
        await
Document.getInitialProps(ctx);
      return {
```

```

        ...initialProps,
        styles: (
            <>
                {initialProps.styles}
                {sheet.getStyleElement()}
            </>
        ),
    };
} finally {
    sheet.seal();
}
}
}

```

The preceding code creates an instance of **ServerStyleSheet**, which **styled-components** uses to retrieve any styles found in all the components in our application. This is needed to create a stylesheet that can be injected into our server-side-rendered application later on. The **sheets.collectStyles** method collects all of the styles from our components, while **sheets.getElement()** generates the **style** tag, which we'll need to return as a prop called **styles**.

2. To support SSR for **styled-components**, we also need to configure the **next.config.json** file in the root of the project. This file needs to hold the following configuration:

```
const nextConfig = {
  reactStrictMode: true,
+  compiler: {
+    styledComponents: true
+  }
}
module.exports = nextConfig
```

3. As with Next.js, we don't have a global entry file for the application. We need a different place to add our global styling and components that we want to display on all pages, such as a header. These styling and component must be added to **_app.js** in our **pages** directory, which returns the component for the page that we're currently visiting and any other component or styling that we return:

```
- import '../styles/globals.css';
+ import { createGlobalStyle } from
  'styled-components';
+ const GlobalStyle = createGlobalStyle`
+   body {
+     margin: 0;
+     padding: 0;
+     font-family: -apple-system,
    BlinkMacSystemFont,
```



```

        "Segoe UI", "Roboto",
    "Oxygen", "Ubuntu",
        "Cantarell", "Fira Sans", "Droid
    Sans",
        "Helvetica Neue", sans-serif;
+     -webkit-font-smoothing: antialiased;
+     -moz-osx-font-smoothing: grayscale;
+ }
+ ';

    function MyApp({ Component, pageProps })
    {
-     return <Component {...pageProps} />;
+     return (
+         <>
+             <GlobalStyle />
+             <Component {...pageProps} />
+         </>
+     );
    }

    export default MyApp;

```

In the preceding file, we've deleted the line that imports the `styles/globals.css` file and replaced it with styling in `styled-components`. This means you can also safely delete the `globals.css` file from the `styles` directory.

4. This global styling is now present on all pages that we'll create later in this Next.js application. But for starters, let's create a **Header** component by adding a file called **Header.js** in a new directory called **components**:

```
import styled from 'styled-components';
const HeaderWrapper = styled.div`
  background-color: orange;
  height: 100%;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
  color: white;
`;
const Title = styled.h1`
  height: 64px;
  pointer-events: none;
`;
function Header() {
  return (
    <HeaderWrapper>
      <Title>Community Feed</Title>
    </HeaderWrapper>
  );
}
```

```
    );  
  }  
  export default Header;
```

5. This component will return a header that will be used on every page, but we also need to add it to `_app.js` to be effective:

```
import { createGlobalStyle } from  
  'styled-components';  
+ import Header from  
'../components/Header';  
  // ...  
function MyApp({ Component, pageProps })  
{  
  return (  
    <>  
      <GlobalStyle />  
+      <Header />  
      <Component {...pageProps} />  
    </>  
  );  
}  
export default MyApp;
```

You'll see what the headers look like in the next section when we add our first route to the application.

We've added multiple new files that have global styling with **styled-components**, which supports SSR, and also a directory that holds reusable components.

With both Next.js and **styled-components** set up, we are ready to start developing with Next.js in the next section of this chapter.

Routing with Next.js

With the **react-router** package, we can add declarative routing to any React application, but you need to set up components that define which routes you want to add. With Next.js, the filesystem is being used for routing, starting at the **pages** directory. Every file and directory in **pages** can represent a route. You can check this by visiting the project at **http://localhost:3000**, where the contents of **index.js** are being rendered.

If we, for example, wanted to add the new **/questions** route to the application, we'd need to create either a new file called **questions.js** or a directory called **questions** with an **index.js** file in **pages**. Let's go with the second option and add the following code to this file:

```
import styled from 'styled-components';  
const QuestionsContainer = styled.div`  
  display: flex;
```

```
    justify-content: space-between;
    flex-direction: column;
    margin: 5%;
  ';
function Questions() {
  return (
    <QuestionsContainer>
      <h2>Questions</h2>
    </QuestionsContainer>
  );
}
export default Questions;
```

This new route has now become available at **<http://localhost:3000/questions>** where only a title is being rendered. As mentioned in the introduction of this chapter, we'll be using the Stack Overflow API to get the data for this application.

Before retrieving the data from the endpoint, we also need to create a component to render this data in. To do so, we need to create a new component that will be used to display a question. This new component can be created in a file called **Card.js** in the **components** directory with the following contents:

```
import styled from 'styled-components';
const CardWrapper = styled.div'
```

```
    text-align: left;
    padding: 1%;
    background: lightGray;
    border-radius: 5px;
    margin-bottom: 2%;
';
const Title = styled.h2`
    width: 100%;
    padding-bottom: 10px;
    text-align: center;
    border-bottom: 1px solid darkGray;
    color: black;
`;
const Count = styled.span`
    color: darkGray;
`;
function Card({ title, views, answers }) {
    return (
        <CardWrapper>
            <Title>{title}</Title>
            <Count>{
                'Views: ${views} | Answers:
                ${answers}'
            }
        </Count>
    )
}
```

```
    </CardWrapper>
  );
}
export default Card;
```

With this component in place, let's retrieve the data from the Stack Overflow API. From this API, we want to retrieve all the questions that are posted with the `reactjs` tag, using the following endpoint:

```
https://api.stackexchange.com/2.2/questions?
order=desc&sort=hot&tagged=reactjs&site=stack
overflow
```

You can find more information on this at <https://api.stackexchange.com/docs/questions#order=desc&sort=hot&tagged=reactjs&filter=default&site=stackoverflow&run=true>.

This returns an array of objects under the `items` field, and from every object, we can get information about a question, such as the title and the number of answers.

We can retrieve the data from this endpoint and display it on the `/questions` route by making some additions to the `index.js` file in the `questions` directory in `pages`:

1. First, we need to add local state variables to the `Questions` component and add a `useEffect` Hook to fetch the data from the Stack

Overflow API:

```
+ import { useState, useEffect } from
  'react';
  import styled from 'styled-components';
+ import Card from '../..//
components/Card';
  function Questions() {
+   const [loading, setLoading] =
useState(false);
+   const [questions, setQuestions] =
useState([]);
+   useEffect(() => {
+     async function fetchData() {
+       const data = await fetch(
          'https://api.stackexchange.com/2.2
/questions?
          order=desc&sort=hot&tagged=reactj
s&
          site=stackoverflow');
+       const result = await data.json();
+       if (result) {
+         setQuestions(result.items);
+         setLoading(false);
+       }
    }
```



```

+      }
+      fetchData();
+    }, []);
    return (
      // ...

```

2. After adding the data fetching logic, we need to add some more code to display the fields from the API on our page. We're passing this data to our **Card** component to render it on the page:

```

    // ...
    return (
      <QuestionsContainer>
        <h2>Questions</h2>
+      {loading ? (
+        <span>Loading...</span>
+      ) : (
+        <div>
+          {questions.map((question) =>
+            (
+              <Card
+                key=
+                {question.question_id}
+                title={question.title}
+                views=
+                {question.view_count}

```

```

+             answers=
{question.answer_count}
+             />
+         )))
+     </div>
+ })
    </QuestionsContainer>
  );
}
export default Questions;

```

If you now visit the **/questions** route on **http://localhost:3000/questions**, you can see that a list of questions is being rendered together with the Header component, as you can see in the following screenshot:

Community Feed

Questions

Render both text and FontAwesome icon in function return

Views: 19 | Answers: 1

Creating Routes using react-router-dom from map method, but it's not redirecting to the JSX component

Views: 11 | Answers: 1

Why i am facing issue while run the project

Views: 16 | Answers: 1

Figure 4.2 – Our application with Stack Overflow data

We'll be handling SSR later, but before that, we need to add routes that support parameters. To create a route that supports a parameter, we need to create a file created in the same manner as the Questions component. The new route will display a specific question, informa-

tion that we can also get from the Stack Overflow API. Again, the filesystem will be leading in creating the route:

1. To create a new route with a parameter, we need to create a file called `[id].js` inside the **questions** directory. This file has the parameter name inside square brackets, and in this file, we can use a Hook from the Next.js routing library to get this parameter value:

```
import { useRouter } from 'next/router';
import styled from 'styled-components';
const QuestionDetailContainer = styled.div`
  display: flex;
  justify-content: space-between;
  flex-direction: column;
  margin: 5%;
`;
function QuestionDetail() {
  const router = useRouter();
  const { id } = router.query;
  return (
    <QuestionDetailContainer>
      <h2>Question: {id}</h2>
    </QuestionDetailContainer>
  );
}
export default QuestionDetail;
```

By visiting <http://localhost:3000/questions/123>, you can see that the parameter that we've added has become visible on the screen.

2. In the **QuestionDetail** component, we can import the **Card** component and we can use the Stack Overflow API to fetch data for a specific question. For this, we need to add both data fetching logic and elements to render the data to the `[id].js` file in the **questions** directory that we created in the previous step:

```
+ import { useState, useEffect } from
  'react';
  import { useRouter } from 'next/router';
  import styled from 'styled-components';
+ import Card from '../components/Card';
  // ...

  function QuestionDetail() {
    const router = useRouter();
    const { id } = router.query;
+   const [loading, setLoading] =
    useState(false);
+   const [question, setQuestion] =
    useState({});
+   useEffect(() => {
+     async function fetchData() {
+       const data = await fetch(
```

```

        'https://api.stackexchange.com/2.2
/questions
        /${id}?site=stackoverflow');
+      const result = await data.json();
+      if (result) {
+        setQuestion(result.items[0]);
+        setLoading(false);
+      }
+    }
+    id && fetchData();
+  }, [id]);
  // ...

```

3. After adding the data fetching logic, the **Card** component can be returned with the question information passed to it as props:

```

  // ...
  return (
    <QuestionDetailContainer>
-      <h2>Question: {id}</h2>
+      {loading ? (
+        <span>Loading...</span>
+      ) : (
+        <Card
+          title={question.title}
+          views={question.view_count}

```

```

+           answers=
{question.answers_count}
+           />
+       )}
      </QuestionDetailContainer>
    );
}
export default QuestionDetail;

```

The API endpoint that gets a question by its identifier returns an array, as it expects multiple IDs at once. Therefore, we need to get the first item that's returned by the endpoint as we only provide one ID.

4. To get to a specific question, you need to have the ID of the question. This is best done from the **Questions** component where we can import a **Link** component from the routing library of Next.js. Every **Link** component needs to wrap a component that's able to do routing, so we'll add a styled **a** element inside it. Also, the elements used to display the questions will be replaced by the **Card** component that we created before:

```

import { useState, useEffect } from
'react';
import styled from 'styled-components';
+ import Link from 'next/link';
import Card from '../components/Card';

```

```

    // ...
+   const CardLink = styled.a`
+     text-decoration: none;
+   `;
  function Questions() {
    // ...
    return (
      <QuestionsContainer>
        // ...
        {questions.map((question) => (
+          <Link
+            key={question.question_id}
+            href=
+            {'/questions/${question.question_id}'}
+            passHref
+          >
+            <CardLink>
              <Card
-                key={question.question_id}
                title={question.title}
                views={question.view_count}
                answers=
+            {question.answer_count}
              />

```



```
+         </CardLink>
+         </Link>
      ) ) }
      // ...
```

As you might notice when visiting

`http://localhost:3000/questions`, the **Card** components are now clickable and link to a new page showing the question you've just clicked on.

5. Finally, we want the basic `/` route to also show the **Questions** component. We can do this by importing this component inside **`/pages/index.js`** and having it returned from there:

```
import Questions from './questions';
export default function Home() {
  return <Questions />;
}
```

The application will now return a list of questions on both the `/` and **`/questions`** routes, and is able to display a specific question when you click on any of the questions from these routes:

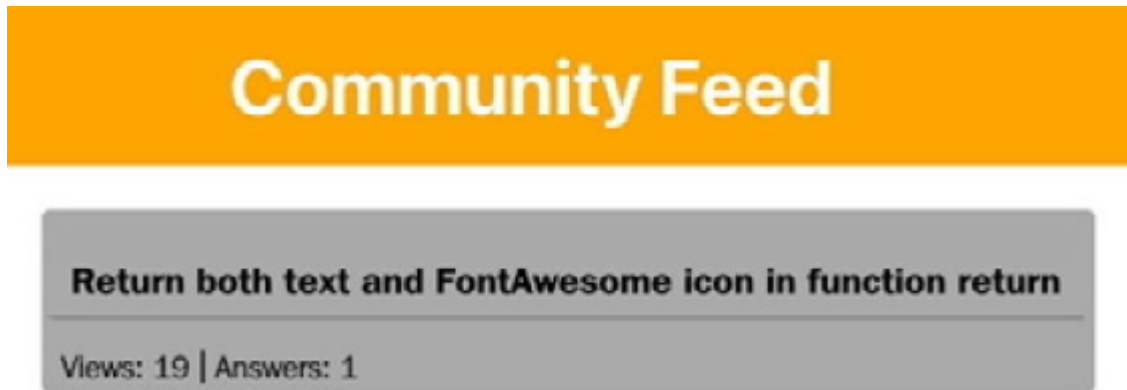


Figure 4.3 – Our application with basic styling and dynamic routes

Besides routing using parameters, we can also add routing using a query string for features such as pagination. This will be shown in the next part of this section, about routing with Next.js.

Handling query strings

Being able to navigate to individual questions is only one piece of the cake when you want to add routing to a project, and pagination could be another one.

The Stack Overflow API already has support for pagination, which you can see if you look at the API response. The object that is being returned when you call the endpoint that is described on <https://api.stackexchange.com/docs/questions#order=desc&sort=hot&tagged=reactjs&filter=default&site=stackoverflow&run=true> has a field called `has_more`. If this field has the `true` value, you can request more questions by adding the `page` query string to the API request.

Just as we got the parameters from the URL with the **useRouter** Hook from Next.js, we can also get the query strings with this Hook. To add pagination to the **Questions** component, we need to make the following changes:

1. In the **Questions** page component in **pages/questions/index.js**, we need to import the **useRouter** Hook and get the query strings from the URL:

```
import { useState, useEffect } from
'react';
import styled from 'styled-components';
import Link from 'next/link';
+ import { useRouter } from 'next/router';
import Card from '../components/Card';
// ...

function Questions() {
  const [loading, setLoading] =
useState(false);
  const [questions, setQuestions] =
useState([]);
+   const router = useRouter();
+   const { page } = router.query;
  useEffect(() => {
    // ...
```

2. The **page** constant can then be appended to the endpoint to retrieve the questions from the Stack Overflow API:

```
// ...
useEffect(() => {
  async function fetchData() {
-    const data = await fetch(
      'https://api.stackexchange.com/2.2/
questions
      ?
order=desc&sort=hot&tagged=reactjs&site=
      stackoverflow');
+    const data = await fetch(
      'https://api.stackexchange.com/2.2/
questions
      ?${page ? 'page=${page}&' :
''}order=
      desc&sort=hot&tagged=reactjs&site=
      stackoverflow');
    const result = await data.json();
    if (result) {
      setQuestions(result.items);
      setLoading(false);
    }
  }
}
```

```
        fetchData();  
-   }, []);  
+   }, [page]);  
    return (  
        // ...
```

NOTE

*In the preceding code, we've also added **page** to the dependency array of the **useEffect** Hook to do the data fetching. When the application first renders, the value for **page** is not set as the query string should still be retrieved from the API. This is causing the API to be called twice, something that we won't optimize now but will do later once we add SSR to the application.*

You can test whether this is working by changing the query string for **page** to different numbers, such as **http://localhost:3000/questions?page=1** or **http://localhost:3000/questions?page=3**. To make the application more user-friendly, let's add pagination buttons to the bottom of the page.

3. Create the **Pagination** component inside the **components** directory, which holds two **Link** components from Next.js. The component will display a link to the previous page if the current page number is above 1 and will also show a link to the next page if more pages are available:

```
import styled from 'styled-components';
```

```

import Link from 'next/link';
const PaginationContainer = styled.div`
  display: flex;
  justify-content: center;
`;
const PaginationLink = styled.a`
  padding: 2%;
  margin: 1%;
  background: orange;
  cursor: pointer;
  color: white;
  text-decoration: none;
  border-radius: 5px;
`;
function Pagination({ currentPage, hasMore
}) {
  return (
    <PaginationContainer>
      <Link
        href={'?
page=${parseInt(currentPage) - 1}'}>
        <PaginationLink>Previous</Paginatio
nLink>
      </Link>

```

```

        <Link
            href={'?
page=${parseInt(currentPage) + 1}'}>
            <PaginationLink>Next</PaginationLin
k>

        </Link>
    </PaginationContainer>
);
}
export default Pagination;

```

4. We need to import this new **Pagination** component inside the **Questions** page component, but we also need to retrieve the value for **hasMore** from the Stack Overflow API:

```

import { useState, useEffect } from
'react';
import Link from 'next/link';
import { useRouter } from 'next/router';
import styled from 'styled-components';
import Card from '../components/Card';
+ import Pagination from
    '../components/Pagination';
// ...
function Questions() {

```

```

    const [loading, setLoading] =
useState(false);
    const [questions, setQuestions] =
useState([]);
+   const [hasMore, setHasMore] =
useState(false);
    const router = useRouter();
    const { page } = router.query;
    useEffect(() => {
        async function fetchData() {
            const data = await fetch(
                'https://api.stackexchange.com/2.
2/questions
                ?${page ? 'page=${page}&' :
''}order=
                desc&sort=hot&tagged=reactjs&sit
e=
                stackoverflow');
            const result = await data.json();
            if (result) {
                setQuestions(result.items);
+               setHasMore(result.has_more);
                setLoading(false);
            }

```



```

    }
    fetchData();
  }, [page]);
  // ...

```

5. Also, the **Pagination** component must be rendered at the end of the **Questions** component. Make sure to also pass the **currentPage** and **hasMore** props to the component:

```

  // ...
  return (
    <QuestionsContainer>
      <h2>Questions</h2>
      {loading ? (
        <span>Loading...</span>
      ) : (
+      <>
        <div>
          {questions.map((question) =>
(
            // ...
          ))}
        </div>
+      <Pagination currentPage=
{parseInt(page) ||
        1} hasMore={hasMore} />

```

```

+      </>
      )}
    </QuestionsContainer>
  );
}
export default Questions;

```

6. Finally, we want the user to not be able to navigate to a page that doesn't exist. So, in the **Pagination** component, make the following changes to disable the previous or next button if there is no page available:

```

// ...
const PaginationLink = styled.a`
  padding: 2%;
  margin: 1%;
+   background: ${({props) =>
      (!props.disabled ? 'orange' :
'lightGrey')}};
+   pointer-events: ${({props) =>
      (!props.disabled ? 'all' : 'none')}};
+   cursor: ${({props) =>
      (!props.disabled ? 'pointer' : 'not-
allowed')}};
  color: white;
  text-decoration: none;

```

```

    border-radius: 5px;
  ';
  function Pagination({ currentPage,
hasMore }) {
    return (
      <PaginationContainer>
        <Link href={'?
page=${parseInt(currentPage) - 1}'}>
-      <PaginationLink>Previous</Paginatio
nLink>
+      <PaginationLink disabled=
{currentPage <= 1}>
        Previous
      </PaginationLink>
    </Link>
    <Link href={'?
page=${parseInt(currentPage) + 1}'}>
-      <PaginationLink>Next</PaginationLin
k>
+      <PaginationLink disabled=
{!hasMore}>
        Next
      </PaginationLink>
    </Link>

```

```
        </PaginationContainer>
      );
    }
    export default Pagination;
```

You have now implemented the parsing of the query string to dynamically change the route for your application. With the addition of the **Pagination** component, both the / and /questions routes will look something like this:

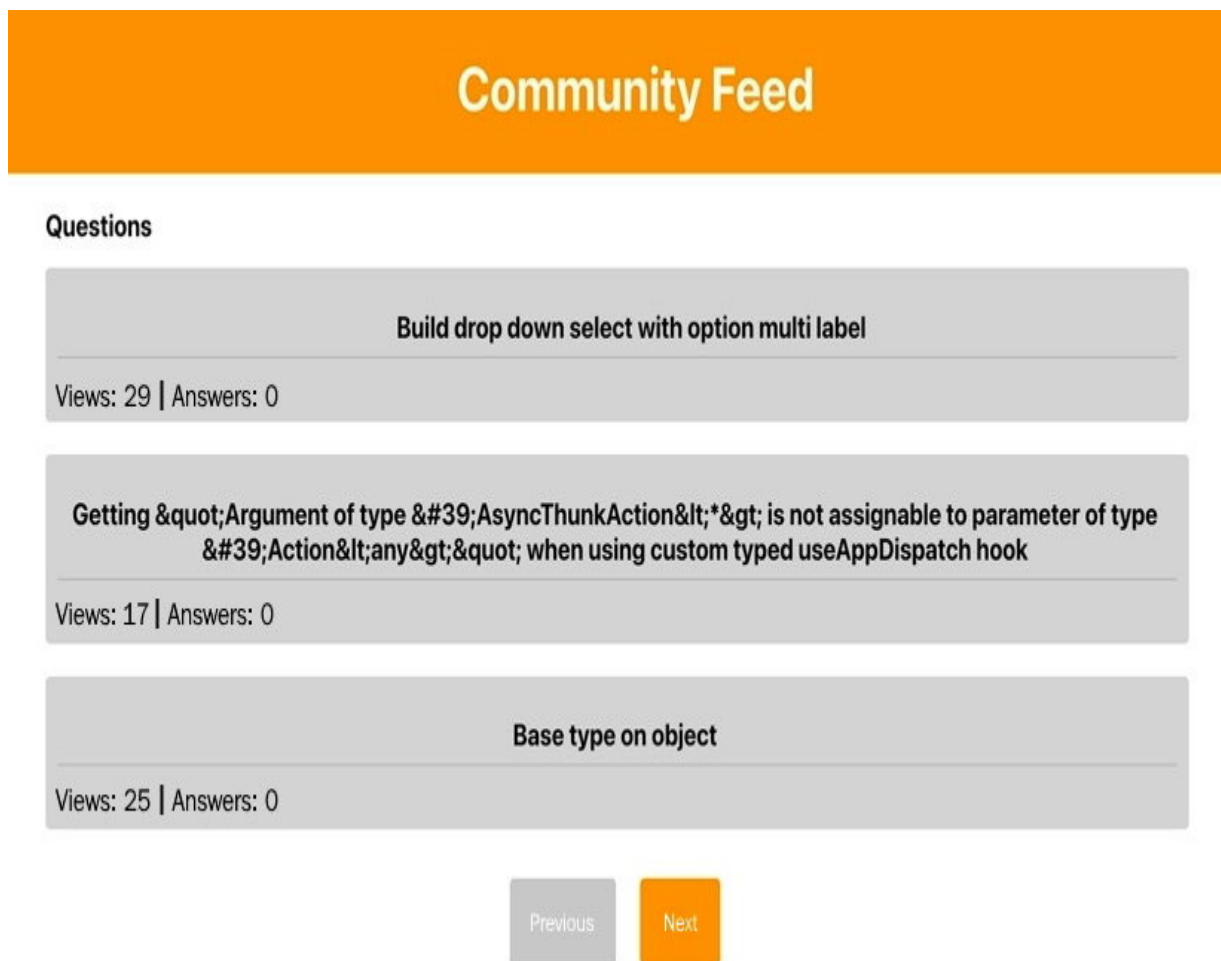


Figure 4.4 – The application after adding pagination

In the next section, you'll explore another thing you can do with React in combination with Next.js, SSR, which enables you to serve your application from the server instead of rendering it at runtime.

Enabling SSR

Using SSR can be helpful if you're building an application that needs to render very quickly or when you want certain information to be loaded before the web page is visible. Although most search engines are now able to render SPAs, this can still be an improvement, for example, if you want users to share your page on social media or when you want to enhance the performance of your application.

Fetching data server side with Next.js

There is no standard pattern to enable SSR for your React application, but luckily, Next.js supports multiple ways to do data fetching, such as dynamically from the client, server side on every request, or statically during build time. The first way is what we've done in this chapter so far and in this section, we'll be requesting our data server side on every request. For this, the Next.js `getServerSideProps` method will be used.

NOTE

*Next.js also offers the **getStaticProps** and **getStaticPaths** methods to generate the content of your application statically at build time. This is especially useful if your content doesn't change that often and you want to serve your website as fast as possible.*

At the beginning of this chapter, we already set up **styled-components** in such a way that it will support SSR, so we just have to alter how we do data fetching to enable it for the entire application. Therefore, we need to refactor our **Questions** component so that it will get the data from the Stack Overflow API on the server side instead of dynamically from the client:

1. In the **Questions** page component, we no longer have to import the **useState** and **useEffect** Hooks to set up state management and data fetching, so these can be removed. The **useRouter** Hook from Next.js can also be deleted:

```
- import { useState, useEffect } from
  'react';
  import styled from 'styled-components';
  import Link from 'next/link';
- import { useRouter } from 'next/router';
  import Card from '../components/Card';
  import Pagination from
  '../components/Pagination';
  // ...
```

```

function Questions() {
-   const [loading, setLoading] =
useState(false);
-   const [questions, setQuestions] =
useState([]);
-   const [hasMore, setHasMore] =
useState(false);
-   const router = useRouter();
-   const { page } = router.query;
-   useEffect(() => {
-       async function fetchData() {
-           const data = await fetch(
                'https://api.stackexchange.com/2.
2/questions
                ?${page ? 'page=${page}&' :
''}order=
                desc&sort=hot&tagged=reactjs&sit
e
                =stackoverflow');
-           const result = await data.json();
-           if (result) {
-               setQuestions(result.items);
-               setHasMore(result.has_more);
-               setLoading(false);

```

```

-      }
-    }
-    fetchData();
-  }, [page]);
  return (
    // ...

```

2. Instead, the **getServerSideProps** method needs to be used to do the data fetching on the server side. As the data is then not retrieved by the client, we no longer need to set local state variables or life cycles to keep track of changes in the data. The data will already be there once we load our React application in the browser:

```

// ...

+ export async function
getServerSideProps(context) {
+   const { page } = context.query;
+   const data = await fetch(
+     'https://api.stackexchange.com/2.2/qu
estions?${
+     page ? 'page=${page}&' : ''
+     }order=desc&sort=hot&tagged=reactjs
&site=
      stackoverflow',
+   );
+   const result = await data.json();

```



```

+   return {
+     props: {
+       questions: result.items,
+       hasMore: result.has_more,
+       page: page || 1,
+     }
+   };
+ }
export default Questions;

```

In this method, the value for **page** is retrieved from a constant called **context**, which is passed to **getServerSideProps** by Next.js and gets the page from the router. Using this value, we can do the data fetching in the same way as we did before in the life cycle. Instead of storing the data in local state variables, we're now passing it as props to the **Questions** component by returning it from the method we created.

3. Our **Questions** page component can use these props to render our list of questions in the browser. As the data is retrieved from the server side before passing the application to the client, we no longer have to set a loading indicator to wait for the data to be fetched:

```

// ...
- function Questions() {

```

```

+ function Questions({ questions, hasMore,
page }) {
  return (
    <QuestionsContainer>
      <h2>Questions</h2>
-     {loading ? (
-       <span>Loading...</span>
-     ) : (
-       <>
        <div>
          {questions &&
            questions.map((question) => (
              // ...
            ))}
        </div>
        <Pagination currentPage=
{parseInt(page) || 1}
          hasMore={hasMore} />
-     </>
-   )}
    </QuestionsContainer>
  );
}
// ...

```

You can check this by going to **http://localhost:3000/questions** and seeing that the questions are no longer being retrieved on the client side but on the server side. In the **Network** tab, there's no request made to the Stack Overflow API, while the questions are being rendered in the browsers. You can also verify this by inspecting the page source:

The screenshot displays a web application interface on the left and a browser's developer tools Network tab on the right.

Community Feed

Questions

- Calculate total price of products based on prices for each day (ReactJS)**
Views: 14 | Answers: 0
- I imported `<Routing>`**
Views: 13 | Answers: 0
- Is there a way to loop over array of arrays of objects?**
Views: 50 | Answers: 1
- React.cloneElement not appending className**

Network Tab:

Name	Status	Type	Initiator	Size	T..	Waterfall
webpack-hmr?page=/questi...	200	events...	questions	1.1 kB	5...	
_devPagesManifest.json	200	fetch	page-loader...	256 B	3...	

Figure 4.5 – SSR application using Next.js

However, the **Questions** page component is also imported in **pages/index.js** and returned there. But opening our main / route at **http://localhost:3000/** doesn't show any questions. This is because this file doesn't have a **getServerSideProps** to get the data. Therefore, we need to create this method in **pages/index.js** as well and have it return the method that we can import from **pages/questions/index.js** so that we don't have to duplicate the data fetching logic. The **Home** component in this file can then get the data from the props and pass it to the **Questions** component:

```
- import Questions from './questions';
+ import Questions, {
+   getServerSideProps as
  getServerSidePropsQuestions,
+ } from './questions';
+ export function getServerSideProps(context)
+ {
+   return
  getServerSidePropsQuestions(context);
+ }
- export default function Home() {
-   return <Questions />;
+ export default function Home(props) {
+   return <Questions {...props} />;
+ }
```

After making this change, both the `/` and `/questions` routes will have SSR enabled. Another advantage of SSR is that your application can be discovered by search engines more effectively. In the next part of this section, you'll add the tags that make your application discoverable by these engines.

Adding head tags for SEO

Assuming you want your application to be indexed by search engines, you need to set head tags for the crawlers to identify the content on your page. This is something you want to do dynamically for each route, as each route will have different content.

Next.js can define the head tags in any component that is rendered by your application by importing **Head** from `next/head`. If nested, the lowest definition of a **Head** component in the component tree will be used. That's why we can create a **Head** component in our **Header** component for all routes and in each of the components that is being rendered on a route:

1. Import the **Head** component in the `components/Header.js` file, and create a **Head** component that sets **title** and a meta description:

```
import styled from 'styled-components';  
+ import Head from 'next/head';  
// ...
```

```

    const Header = () => (
+   <>
+     <Head>
+       <title>Community Feed</title>
+       <meta name='description'
content='This is a
          Community Feed project build with
React' />
+     </Head>
      <HeaderWrapper>
        <Title>Community Feed</Title>
      </HeaderWrapper>
+   </>
    );
export default Header;

```

2. Also, create a **Head** component in **pages/questions/index.js** that only sets a title for this route, so it will use the meta description of the **Header** component:

```

import styled from 'styled-components';
import Link from 'next/link';
+ import Head from 'next/head';
import Card from '../components/Card';
import Pagination from
  '../components/Pagination';

```

```

    // ...
    function Questions({ questions, hasMore,
page }) {
        return (
+         <>
+         <Head>
+         <title>Questions</title>
+         </Head>
            <QuestionsContainer>
                // ...
            </QuestionsContainer>
+         </>
        );
    }
    // ...

```

3. Do the same for the `pages/questions/[id].js` file, where you can also take the title of the question to make the page title more dynamic:

```

import { useState, useEffect } from
'react';

import { useRouter } from 'next/router';
+ import Head from 'next/head';
import styled from 'styled-components';
import Card from '../components/Card';

```

```

// ...
function QuestionDetail() {
  // ...
  return (
    <QuestionDetailContainer>
      {loading ? (
        <span>Loading...</span>
      ) : (
+        <>
+          <Head>
+            <title>{question.title}
</title>
+          </Head>
            <Card
              title={question.title}
              views={question.view_count}
              answers=
{question.answer_count}
              />
+          </>
            )}
      </QuestionDetailContainer>
    );
  }
}

```



```
export default QuestionDetail;
```

These head tags will now be used when you're running your application on both the server and the client side, making your application more suitable for being indexed by search engines, which improves your SEO.

Summary

In this chapter, you've learned how to use Next.js as an alternative to Create React App. Next.js is a framework to create React applications, without having to add configuration for compiling and building your code or to handle features such as routing and data fetching. The project you created in this chapter supports SSR, as this is built in by Next.js. Also, we've added dynamic head tags to the application for SEO purposes.

After completing this chapter, you must already feel like an expert with React! The next chapter will take your skill to the next level as you'll learn how to handle state management using the context API. With the context API, you can share the state and data between multiple components in your application, no matter whether they're direct children of the parent component or not.

Further reading

For more information on Next.js, you can refer to <https://nextjs.org/docs/>.

Chapter 5: Building a Personal Shopping List Application Using Context and Hooks

State management is a very important part of modern web and mobile applications and is something that React is very good at. Handling state management in React applications can be quite confusing, as there are multiple ways you can handle the current state of your application. The projects you created in the first four chapters of this book didn't focus on state management too much, something that will be investigated much more in this chapter.

This chapter will show how you can handle state management in React by creating an application state that is accessible from every component. Before React v16.3, you needed third-party packages to handle state in React, but with the renewed version of the context API, this is no longer mandatory. Also, with the release of React Hooks, more ways to mutate this Context were introduced. Using an example application, the methods for handling state management for your application are demonstrated.

The following topics will be covered in this chapter:

- Using the context API for state management

- Mutating Context with Hooks
- Code splitting with React Suspense

Project overview

In this chapter, we will build a personal shopping list by adding state management using Context and Hooks. We will advance upon a boilerplate application that has been created with Create React App and has declarative routing using `react-router`. Also, code splitting of the bundle is added with React Suspense.

The build time is 2.5 hours.

Getting started

The project that we'll create in this chapter builds upon an initial version that you can find on GitHub:

<https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter05-initial>. The complete source code can also be found on GitHub: <https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter05>.

After downloading the initial application, make sure that you run `npm install` from the project's root directory. This project is created using

Create React App and installs the `react`, `react-dom`, `react-scripts`, `styled-components`, and `react-router-dom` packages, which you've already seen in previous chapters. After finishing the installation process, you can run `npm start` from the same tab in Terminal and view the project in your browser (<http://localhost:3000>).

The initial application for this section is created with Create React App and has routing and data fetching already implemented. When you open the application, a screen displaying a header, a navigation bar, and two lists are being rendered. If, for example, you click on the first list that is displayed here, a new page will open that displays the items of this list. On this page, you can click on the **Add Item** button in the top-right corner to open a new page, which has a form to add a new list and looks like this:

Personal Shopping List

< Go Back

Add Item

Title

Insert title

Quantity

0

Price

0.00

Add Item

Figure 5.1 – The initial application

This form is rendered by the **ListForm** component and has no functionality yet, as you'll add this later on. When you click on the left button, it redirects you to the previously visited page, using the `navigate` method from **react-router-dom**.

NOTE

When you try to submit the form to either add a new list or add a new item to a list, nothing happens yet. The functionality of these forms will be added in this section later on, for which you'll use the Context API and React Hooks.

The project is structured in the same manner as the applications you've created before. A distinction is made, however, between reusable function components in the **components** directory and components that represent a route in the **pages** directory. The page components are using the **useDataFetching** Hook, which we saw earlier in [*Chapter 3, Building a Dynamic Project Management Board*](#), which adds data fetching.

The following is an overview of the complete structure of the project:

```
chapter-5-initial
├── /node_modules
├── /public
└── /src
```

```
|- /components
  |- /Button
    |- Button.js
  |- /FormItem
    |- FormItem.js
  |- /Header
    |- Header.js
  |- /NavBar
    |- NavBar.js
  |- /ListItem
    |- ListItem.js
|- /hooks
  |- useDataFetching.js
|- /pages
  |- ListDetail.js
  |- ListForm.js
  |- Lists.js
|- App.js
|- index.js
package.json
```

The entry point of this application is the `src/index.js` file that renders the **App** component, which sets up routing using a **Router** component from `react-router-dom`. The **App** component contains a **Header** com-

ponent and a **Switch** router component that defines four routes.

These routes are as follows:

- `/`: Renders **Lists**, with an overview of all of the lists
- `/list/:listId`: Renders **ListDetail**, with an overview of all items from a specific list
- `/list/:listId/new`: Renders **ListForm**, with a form to add new items to a specific list

The data is fetched from a mock server that was created using the free service, My JSON Server, which creates a server from the `db.json` file in the root directory of your project in GitHub. This file consists of a JSON object that has two fields, `items` and `lists`, which creates multiple endpoints on a mock server. The ones you'll be using in this chapter are as follows:

- <https://my-json-server.typicode.com/PacktPublishing/React-Projects-Second-Edition/items>
- <https://my-json-server.typicode.com/PacktPublishing/React-Projects-Second-Edition/lists>

NOTE

The `db.json` file must be present in the master branch (or default branch) of your GitHub repository for My JSON Server to work.

*Otherwise, you'll receive a **404 Not Found** message when trying to request the API endpoints.*

Personal shopping list

In this section, you'll build a personal shopping list application that has state management using Context and Hooks. With this application, you can create shopping lists that you can add items to, along with their quantities and prices. The starting point of this section is an initial application that has routing and local state management already enabled.

Using the Context API for state management

State management is very important, as the current state of the application holds data that is valuable to the user. In previous chapters, you've already used local state management by using `useState` and `useEffect` Hooks. This pattern is very useful when the data in the state is only of importance to the components you're setting the state in. As passing down the state as props through several components can become confusing, you'd need a way to access props throughout your application even when you're not specifically passing them as props. For this, you can use the Context API from React, which is also used by packages you've already used in previous chapters such as `styled-components` and `react-router-dom`.

To share state across multiple components, a React feature called Context will be explored, starting in the first part of this section.

Creating Context

When you want to add Context to your React application, you can do this by creating a new Context with the `createContext` method from React. This creates a Context object that consists of two React components, called Provider and Consumer. The Provider is where the initial (and subsequently current) value of the Context is placed, which can be accessed by components that are present within the Consumer.

This is done in the **App** component in `src/App.js`, as you want the Context for the lists to be available in every component that is rendered by **Route**:

1. Let's start by creating a Context for the lists and making it exportable so that the list data can be used everywhere. For this, you can create a new file called `ListsContext.js` inside a new directory, `src/context`. In this file, you can add the following code:

```
import { createContext } from 'react';
import useDataFetching from
  '../hooks/useDataFetching';
```

```

export const ListsContext =
  createContext();
export const ListsContextProvider = ({
  children }) => {
  const [loading, error, data] =
    useDataFetching(
      'https://my-json-server.typicode.com/
      PacktPublishing/React-Projects-Second-
      Edition/
      lists',
    );

  return (
    <ListsContext.Provider value=
      {{ lists: data, loading, error }}>
      {children}
    </ListsContext.Provider>
  );
};
export default ListsContext;

```

The previous code creates a Provider based on a **Context** object that is passed as a prop and sets a value based on the return from the **useDataFetching** Hook that is fetching all of the lists. Using the **children** prop, all of the components that will be wrapped inside the

ListsContextProvider component can retrieve the data for the value from a Consumer.

2. This **ListsContextProvider** component can be imported inside your **App** component to wrap the Router component that is wrapping all the routes for our application:

```
import styled, { createGlobalStyle } from
  'styled-components';
import { Route, Routes, BrowserRouter }
from
  'react-router-dom';
+ import { ListsContextProvider } from
  './context/ListsContext';
// ...
function App() {
  return (
    <>
      <GlobalStyle />
      <AppWrapper>
        <BrowserRouter>
          <Header />
+      <ListsContextProvider>
        <Routes>
          // ...
        </Routes>
```

```

+         </ListsContextProvider>
        </BrowserRouter>
        </AppWrapper>
    </>

    );
}

export default App;

```

3. This way, you're now able to consume the value from the Provider for **ListsContext**, from all the components wrapped within **ListsContextProvider**. In the **Lists** component, this data can be retrieved using the **useContext** Hook from React by passing the **ListsContext** object to it. This data can then be used to render the lists, and the **useDataFetching** Hook can be removed from **src/pages/Lists.js**:

```

+ import { useContext } from 'react';
  import styled from 'styled-components';
  import { Link, useNavigate } from
    'react-router-dom';
- import useDataFetching from
  '../hooks/useDataFetching';
  import NavBar from
  '../components/NavBar/NavBar';
+ import ListsContext from
  '../context/ListsContext';

```

```

// ...
const Lists = () => {
  let navigate = useNavigate();
-   const [loading, error, data] =
      useDataFetching('https://my-json-
server.
      typicode.com/PacktPublishing/React-
Projects-
      Second-Edition/lists');
+   const { loading, error, lists } =
      useContext(ListsContext);
  return (
    <>
      {navigate && <NavBar title='Your
Lists' />}
      <ListWrapper>
        {loading || error ? (
          <span>{error || 'Loading...'}
</span>
        ) : (
-          data.map((list) => (
+          lists.map((list) => (
            <ListLink key={list.id}
              to={`list/${list.id}`}>

```

```
// ...
```

Now that you've removed the `useDataFetching` Hook from `Lists`, no requests to the API are sent directly from this component anymore. The data for the lists is instead fetched from `ListsContextProvider` and is passed by `ListsContext`, which is used by the `useContext` Hook in `Lists`. If you open the application in the browser by going to `http://localhost:3000/`, you can see the lists are being rendered just as before.

In the next section, you'll also add a Context object for the items, so the items are also available to all of the components within the Routes component from `react-router`.

Nesting Context

Just as for the list data, the item data could also be stored in Context and passed to the components that need this data. That way, data is no longer fetched from any of the rendered components but from the Context only:

1. Again, start by creating a new file where both a Context and Provider are created. This time, it's called `ItemsContext.js`, which can also be added to the `src/context` directory:

```
import { createContext } from 'react';  
import useDataFetching from
```



```

    '../hooks/useDataFetching';
export const ItemsContext =
createContext();
export const ItemsContextProvider = ({
children }) => {
    const [loading, error, data] =
useDataFetching(
    'https://my-json-server.typicode.com/
    PacktPublishing/React-Projects-Second-
    Edition/items',
    );
    return (
        <ItemsContext.Provider value=
            {{ items: data, loading, error }}>
            {children}
        </ItemsContext.Provider>
    );
};
export default ItemsContext;

```

2. Next, import this new Context and Provider in **src/App.js**, where you can nest this below the **ListsContextProvider** component:

```

// ...
import { ListsContextProvider } from
    '../context/ListsContext';

```

```

+ import { ItemsContextProvider } from
    './context/ItemsContext';
    // ...

function App() {
    return (
        <>
        <GlobalStyle />
        <AppWrapper>
            <BrowserRouter>
                <Header />
                <ListsContextProvider>
+                 <ItemsContextProvider>
                    <Routes>
                        // ...
                    </Routes>
+                 </ItemsContextProvider>
                </ListsContextProvider>
            </BrowserRouter>
        </AppWrapper>
        </>
    );
}
export default App;

```

3. The **ListDetail** component can now get the item from **ItemsContext**, meaning we no longer have to use the **useDataFetching** Hook in this component. To accomplish this, you need to make the following changes to **src/pages/ListDetail.js**:

```
- import { useState, useEffect } from
  'react';
+ import { useState, useEffect, useContext
} from
  'react';
import styled from 'styled-components';
import { useNavigate, useParams } from
  'react-router-dom';
- import useDataFetching from
  '../hooks/useDataFetching';
import NavBar from
  '../components/NavBar/NavBar';
import ListItem from
  '../components/ListItem/ListItem';
+ import ItemsContext from
  '../context/ItemsContext';
// ...
function ListDetail() {
  let navigate = useNavigate();
  const { listId } = useParams();
```

```
-   const [loading, error, data] =  
      useDataFetching('https://my-json-  
server.  
      typicode.com/PacktPublishing/React-  
Projects-  
      Second-Edition/items/');  
+   const { loading, error, items: data } =  
      useContext(ItemsContext);  
    // ...
```

All of the data fetching is now no longer by the **List** and **Lists** components. By nesting these Context Providers, the return values can be consumed by multiple components. But this still isn't ideal, as you're now loading all of the lists and all of the items when starting your application.

The downside of this approach is that once we open a detail page for a list, it will retrieve all items, even if they are not for this list. In the next section, you'll see how to get only the data you need by combining Context with custom Hooks.

Mutating Context with Hooks

There are multiple ways in which you can get data conditionally from the Context; one of these is placing the data from the Context in the

local state. That could be a solution for a smaller application, but can be inefficient for larger applications, as you'd still need to pass this state down your component tree. Another solution is to use React Hooks to create a function that is added to the value of your Context and can be invoked from any of the components that are nested in this Context. Also, this method of getting the data lets you efficiently load only the data that you'd need.

How this can be used together with React life cycles and state management using Hooks is demonstrated in the first part of this section.

Using life cycles in functional components

Previously, we used the `useDataFetching` Hook to do the data fetching for us, but this doesn't let us control when the data will be fetched exactly. From the components that are consuming our Context data, we want to be able to initiate the data fetching. Therefore we need to add life cycles to them, which invoke a function to do the data fetching inside our Context components. Follow these steps to implement this:

1. The first step in achieving this is by adding logic to do data fetching in the `src/context/ItemsContext.js` file. This logic will replace the

usage of the `useDataFetching` Hook, starting with adding local state variables for the data fetching state:

```
- import { createContext } from 'react';
- import useDataFetching from
  '../hooks/useDataFetching';
+ import { createContext, useCallback,
  useState } from
  'react';
  export const ItemsContext =
  createContext();
  export const ItemsContextProvider = ({
  children })
    => {
-   const [loading, error, data] =
      useDataFetching('https://my-json-
      server.
      typicode.com/PacktPublishing/React-
      Projects-
      Second-Edition/items');
+   const [loading, setLoading] =
      useState(true);
+   const [items, setItems] = useState([]);
+   const [error, setError] = useState('');
      // ...
```

2. After this, we can add a function called **fetchItems** that we pass to **ItemsContextProvider**, meaning it will be added to the Context. This function is wrapped in a **useCallback** Hook to prevent unneeded (re)renders of your component:

```
    // ...
    const [error, setError] = useState('');
+   const fetchItems = useCallback(async
    (listId) => {
+       try {
+           const data = await
    fetch(`https://my-json-
            server.typicode.com/PacktPublishi
    ng/
            React-Projects-Second-
    Edition/lists/
            ${listId}/items`);
+       const result = await data.json();
+       if (result) {
+           setItems(result);
+           setLoading(false);
+       }
+   } catch (e) {
+       setLoading(false);
+       setError(e.message);
```

```

+      }
+    }, [])
    return (
-      <ItemsContext.Provider value={{ data:
items,
          loading, error }}>
+      <ItemsContext.Provider value={{
items, loading,
          error, fetchItems }}>
      // ...

```

3. With this function in place, the next step would be to invoke it with a value for `listId` from the `ListDetail` component. This would mean that we no longer retrieve all the items once we load this component, but use the params from the URL to determine what data should be fetched and added to the Context:

```

- import { useState, useEffect, useContext
} from
  'react';
+ import { useEffect, useContext } from
  'react';
  import styled from 'styled-components';
  // ...
  function ListDetail() {
    let navigate = useNavigate();

```



```

    const { listId } = useParams();
-   const { loading, error, items: data } =
        useContext(ItemsContext);
+   const { loading, error, items,
fetchItems } =
        useContext(ItemsContext);
-   const [items, setItems] = useState([]);
-   useEffect(() => {
-       data && listId &&
setItems(data.filter((item) =>
            item.listId === parseInt(listId)));
-   }, [data, listId]);
+   useEffect(() => {
+       listId && !items.length &&
fetchItems(listId);
+   }, [fetchItems, items, listId]);
    return (
        // ...

```

The preceding `useEffect` Hooks call the `fetchItems` function when there's a `listId` present in the URL of the page, and when the value for `items` is an empty array. This prevents us from fetching the items again if they already exist in `ItemsContext`.

By creating a function to do data fetching in our Context, we can now control when the data should be fetched, so there will no longer be unnecessary requests to the API. But other Hooks can also directly pass data to the Provider without having to duplicate **useState** Hooks. This will be demonstrated in the next part of this section.

Using advanced state with **useReducer**

Another way to use actions for adding data to the Provider is by using a pattern similar to Flux, which was introduced by Facebook. The Flux pattern describes a data flow where actions are being dispatched that retrieve data from a store and return it to the view. This would mean that actions need to be described somewhere; there should be a central place where data is stored and this data can be read by the view. To accomplish this pattern with the Context API, you can use another Hook, called **useReducer**. This Hook can be used to return data not from a local state, but from any data variable:

1. Just as with the **useState** Hook, the **useReducer** Hook needs to be added to the component that is using it. **useReducer** will take an initial state and a function that determines which data should be returned. This initial value needs to be added to the **src/context/ListsContext.js** file before adding the Hook:

```
- import { createContext } from 'react';
```

```

+ import { createContext, useCallback,
useReducer }
    from 'react';
    const ListsContext = createContext();
+ const initialState = {
+   lists: [],
+   loading: true,
+   error: '',
+ };
    // ...

```

2. Next to an initial value, the `useReducer` Hook also takes a function that's called `reducer`. This `reducer` function should also be created and is a function that updates `initialState`, which was passed and returns the current value, based on the action that was sent to it. If the action that was dispatched doesn't match any of those defined in `reducer`, the reducer will just return the current value without any changes:

```

    import { createContext, useReducer } from
'react';
    const ListsContext = createContext();
    // ...
+ const reducer = (state, action) => {
+   switch (action.type) {
+     case 'GET_LISTS_SUCCESS':

```

```

+         return {
+             ...state,
+             lists: action.payload,
+             loading: false,
+         };
+     case 'GET_LISTS_ERROR':
+         return {
+             ...state,
+             lists: [],
+             loading: false,
+             error: action.payload,
+         };
+     default:
+         return state;
+   }
+ };

export const ListsContextProvider = ({
  children }) => {
  // ...

```

3. The two parameters for the **useReducer** Hook are now added to the file, so you need to add the actual Hook and pass **initialState** and **reducer** to it. The **useDataFetching** Hook can be removed, as

this will be replaced with a new function that has data fetching logic:

```
// ...  
const ListsContextProvider = ({ children  
}) => {  
-   const [loading, error, data] =  
        useDataFetching('https://my-json-  
server.  
        typicode.com/PacktPublishing/React-  
Projects-  
        Second-Edition/lists');  
+   const [state, dispatch] =  
        useReducer(reducer, initialState);  
// ...
```

4. As you can see, **reducer** changes the value it returns when the **GET_LISTS_SUCCESS** or **GET_LISTS_ERROR** action is sent to it. Before it was mentioned, you can call this **reducer** by using the **dispatch** function that was returned by the **useReducer** Hook. However, as you also have to deal with the asynchronous fetching of the data, you can't invoke this function directly. Instead, you need to create an **async/await** function that calls the **fetchData** function and dispatches the correct action afterward:

```
// ...
```

```

    export const ListsContextProvider = ({
children })
    => {
        const [state, dispatch] =
            useReducer(reducer, initialState);
+   const fetchLists = useCallback(async ()
=> {
+       try {
+           const data = await
fetch(`https://my-json-
            server.typicode.com/PacktPublishi
ng/React-
            Projects-Second-Edition/lists`);
+           const result = await data.json();
+           if (result) {
+               dispatch({ type:
'GET_LISTS_SUCCESS',
                        payload: result });
+           }
+       } catch (e) {
+           dispatch({ type: 'GET_LISTS_ERROR',
                        payload: e.message });
+       }
+   }, [])

```

```
return (  
  // ...
```

5. The preceding **fetchLists** function calls the API and if there is a result, the **GET_LISTS_SUCCESS** action will be dispatched to the reducer using the **dispatch** function from the **useReducer** Hook. If not, the **GET_LISTS_ERROR** action will be dispatched, which returns an error message.
6. The values from the state and the **fetchLists** function must be added to the Provider so that we can access them from other components through the Context:

```
  // ...  
  return (  
    -    <ListsContext.Provider value=  
        {{ loading, error, data: lists }}>  
    +    <ListsContext.Provider value=  
        {{ ...state, fetchLists }}>  
        {children}  
    </ListsContext.Provider>  
  );  
};  
export default ListsContext;
```

7. This **getLists** function can now be invoked from the **useEffect** Hook in the component where the lists are displayed, the **Lists**

component, on the first render. The lists should only be retrieved when there aren't any lists available yet:

```
- import { useContext } from 'react';
+ import { useContext, useEffect } from
  'react';

  import styled from 'styled-components';
  import { Link, useNavigate } from
    'react-router-dom';
  import NavBar from
    '../components/NavBar/NavBar';
  import ListsContext from
    '../context/ListsContext';
  // ...

  function Lists() {
    let navigate = useNavigate();
-    const { loading, error, lists } =
      useContext(ListsContext);
+    const { loading, error, lists,
      fetchLists } =
      useContext(ListsContext);
+    useEffect(() => {
+      !lists.length && fetchLists()
+    }, [fetchLists, lists])
    return (
```



```
// ...
```

If you now visit the project in the browser again, you can see the data from the lists is loaded just as before. The big difference is that the data is fetched using a Flux pattern, meaning this can be extended to fetch the data in other instances as well. The same can be done for **ItemsContext** as well, in the **src/context/ItemsContext.js** file:

1. First, import the **useReducer** Hook, and add the initial value for the items and the **reducer** function that we use with this Hook later:

```
- import { createContext, useState } from
  'react';
+ import { createContext, useReducer } from
  'react';

  export const ItemsContext =
  createContext();
+ const initialState = {
+   items: [],
+   loading: true,
+   error: '',
+ };
+ const reducer = (state, action) => {
+   switch (action.type) {
+     case 'GET_ITEMS_SUCCESS':
+       return {
```

```

+      ...state,
+      items: action.payload,
+      loading: false,
+    };
+    case 'GET_ITEMS_ERROR':
+      return {
+        ...state,
+        items: [],
+        loading: false,
+        error: action.payload,
+      };
+    default:
+      return state;
+  }
+ };

export const ItemsContextProvider =
  ({ children }) => {
    // ...

```

2. After this, you can add the initial state and reducer to the **useReducer** Hook. The **fetchItems** function that already exists in this file must be changed so that it will use the **dispatch** function from **useReducer** instead of the **update** functions from the **useState** Hooks:

```

// ...

export const ItemsContextProvider =

```

```

    ({ children }) => {
-   const [loading, setLoading] =
useState(true);
-   const [items, setItems] = useState([]);
-   const [error, setError] = useState('');
+   const [state, dispatch] =
        useReducer(reducer, initialState);
    const fetchItems = useCallback(async
(listId) => {
    try {
        const data = await fetch(`https://my-
json-
        server.typicode.com/PacktPublishing
/React-
        Projects-Second-
Edition/lists/${listId}/
        items`);
        const result = await data.json();
        if (result) {
-            setItems(result);
-            setLoading(false);
+            dispatch({ type:
'GET_ITEMS_SUCCESS',
                payload: result });

```

```

    }
  } catch (e) {
-    setLoading(false);
-    setError(e.message);
+    dispatch({ type: 'GET_ITEMS_ERROR',
                payload: e.message });
  }
}, [])
return (
  // ...

```

3. Also, add the state and the `fetchItems` function to

ListsContextProvider:

```

  // ...
  return (
-    <ItemsContext.Provider value={{
items, loading,
    error, fetchItems }}>
+    <ItemsContext.Provider value=
    {{ ...state, fetchItems }}>
    {children}
    </ItemsContext.Provider>
  );
};
export default ItemsContext;

```

If you were to open a specific list on the `/lists/:listId` route, for example, `http://localhost:3000/list/1`, you would see that nothing has changed and that the items for the list are still displayed.

You might notice that the title of the list isn't displayed here. The information for the lists is only fetched when the `Lists` component is first rendered, so you'd need to create a new function to always fetch the information for the list that you're currently displaying in the `List` component:

1. In the `src/context/ListsContextProvider.js` file, you need to extend `initialState` to also have a field called `list`:

```
import { createContext, useReducer } from
'react';

export const ListsContext =
createContext();

const initialState = {
  lists: [],
+  list: {},
  loading: true,
  error: '',
};

const reducer = (state, action) => {
  // ...
```

2. In **reducer**, you also now have to check for two new actions that either add the data about a list to the context or add an error message:

```
// ...
const reducer = (state, action) => {
  switch (action.type) {
    case 'GET_LISTS_SUCCESS':
      // ...
    case 'GET_LISTS_ERROR':
      // ...
+   case 'GET_LIST_SUCCESS':
+     return {
+       ...state,
+       list: action.payload,
+       loading: false,
+     };
+   case 'GET_LIST_ERROR':
+     return {
+       ...state,
+       list: {},
+       loading: false,
+       error: action.payload,
+     };
    default:
```

```

        return state;
    }
};
export const ListsContextProvider =
    ({ children }) => {
        // ...

```

3. These actions will be dispatched from a new **fetchList** function that takes the specific ID of a list to call the API. If successful, the **GET_LIST_SUCCESS** action will be dispatched; otherwise, the **GET_LIST_ERROR** action is dispatched. Also, pass the function to the Provider so that it can be used from other components:

```

// ...
+   const fetchList = useCallback(async
+   (listId) => {
+       try {
+           const data = await
+   fetch(`https://my-json-
+           server.typicode.com/PacktPublishi
+   ng/React-
+           Projects-Second-
+   Edition/lists/${listId}`);
+           const result = await data.json();
+           if (result) {

```

```

+         dispatch({ type:
+         'GET_LIST_SUCCESS',
+             payload: result });
+     }
+ } catch (e) {
+     dispatch({ type: 'GET_LIST_ERROR',
+         payload: e.message });
+ }
+ }, [])
    return (
-     <ListsContext.Provider value=
+         {{ ...state, fetchLists }}>
+     <ListsContext.Provider value=
+         {{ ...state, fetchLists, fetchList
+     }}>
+         {children}
+     </ListsContext.Provider>
+ );
+ };
    export default ListsContext;

```

4. And, in the **ListDetail** component, we can get the list data from **ListsContext** by calling the **fetchList** function in a **useEffect** Hook. Also, pass it as a prop to the **NavBar** component so that it will be displayed:


```

import { useEffect, useCallback,
useContext } from
  'react';
import styled from 'styled-components';
import { useNavigate, useParams } from
  'react-router-dom';
import NavBar from
'../components/NavBar/NavBar';
import ListItem from
  '../components/ListItem/ListItem';
import ItemsContext from
'../context/ItemsContext';
+ import ListsContext from
'../context/ListsContext';
// ...
function ListDetail() {
  let navigate = useNavigate();
  const { listId } = useParams();
  const { loading, error, items,
fetchItems } =
    useContext(ItemsContext);
+  const { list, fetchList } =
    useContext(ListsContext);
  useEffect(() => {

```

```

        listId && !items.length &&
fetchItems(listId);
    }, [fetchItems, items, listId]);
+   useEffect(() => {
+       listId && fetchList(listId);
+   }, [fetchList, listId]);
    return (
        <>
            {navigate && (
                <NavBar
                    goBack={() => navigate(-1)}
                    openForm={() =>
                        navigate(`/list/${listId}/new
`)}}
+               title={list && list.title}
                />
            )}
        // ...

```

In the previous code block, we're calling the **fetchList** function from a different **useEffect** Hook than the **fetchItems** function. This is a good way to separate concerns to keep our code clean and concise.

All of the data in your application is now being loaded using the Providers, which means it's now detached from the views. Also, the

`useDataFetching` Hook is completely removed, making your application structure more readable.

Not only can you use the context API with this pattern to make data available to many components, but you can also mutate the data. How to mutate this data will be shown in the next section.

Mutating data in the Provider

Not only can you retrieve data using this Flux pattern, but you can also use it to update data. The pattern remains the same: you dispatch an action that would trigger the request to the server and, based on the outcome, the reducer will mutate the data with this result. Depending on whether or not it was successful, you could display a success message or an error message.

The code already has a form for adding a new item to a list—something that is not working yet. Let's create the mechanism to add items by updating the Provider for `items`:

1. The first step is to add new actions to the reducer in `ItemsContext`, which will be dispatched once we try to add a new item:

```
// ...  
const reducer = (state, action) => {  
  switch (action.type) {
```

```

        // ...
+       case 'ADD_ITEM_SUCCESS':
+         return {
+           ...state,
+           items: [...state.items,
action.payload],
+           loading: false,
+         };
        default:
          return state;
      }
    };
    export const ItemsContextProvider =
      ({ children }) => {
        // ...

```

2. We also need to add a new function that can handle **POST** requests, as this function should also set the method and a body when handling the **fetch** request. You can create this function in the preceding file as well, and pass it to the Provider:

```

    // ...
+   const addItem = useCallback(async ({
listId, title,
      quantity, price }) => {

```

```

+   const itemId = Math.floor(Math.random()
+     * 100);
+   try {
+     const data = await fetch(`https://my-
+       json-
+         server.typicode.com/PacktPublishing
+       /React-
+         Projects-Second-Edition/items`,
+       {
+         method: 'POST',
+         body: JSON.stringify({
+           id: itemId,
+           listId,
+           title,
+           quantity,
+           price,
+         }),
+       },
+     );
+     const result = await data.json();
+     if (result) {
+       dispatch({
+         type: 'ADD_ITEM_SUCCESS',
+         payload: {

```

```

+         id: itemId,
+         listId,
+         title,
+         quantity,
+         price,
+     },
+     ));
+ }
+ } catch {}
+ }, [])
    return (
-     <ItemsContext.Provider value=
        {{ ...state, fetchItems }}>
+     <ItemsContext.Provider value=
        {{ ...state, fetchItems, addItem }}>
        // ...

```

NOTE

*The mock API from My JSON Server doesn't persist data once it is added, updated, or deleted with a request. However, you can see whether the request was successful by checking the request in the **Network** tab in the Developer Tools section of your browser.*

*That's why the input content is spread over the value for **items**, so this data is available from the Consumer.*

3. As the function to add a new item to a list is now available from the Provider, the **ListForm** component in **src/pages/ListForm.js** is now able to use the **addItem** function, which will dispatch the action that will call the API and add the item to the state. However, the values of the input fields in the form need to be determined first. Therefore, the input fields need to be controlled components, meaning their value is controlled by the local state that encapsulates the value. Therefore we need to import the **useState** Hook and also a **useContext** Hook, which we'll use later to get values from the Context:

```
+ import { useState, useContext } from
  'react';
  import styled from 'styled-components';
  import { useNavigate, useParams } from
    'react-router-dom';
  import NavBar from
    '../components/NavBar/NavBar';
  import FormItem from
    '../components/FormItem/FormItem';
  import Button from
    '../components/Button/Button';
+ import ItemsContext from
  '../context/ItemsContext';
  // ...
```

```

function ListForm() {
  let navigate = useNavigate();
  const { listId } = useParams();
+  const [title, setTitle] = useState('');
+  const [quantity, setQuantity] =
useState('');
+  const [price, setPrice] = useState('');
  return (
    // ...

```

For this, we used the **useState** Hook for every **state** value that we created.

4. The local state values and the function that triggers an update of the local state values must be set as a prop on the **FormItem** components:

```

    // ...
    return (
      <>
        {navigate && <NavBar goBack={() =>
          navigate(-1)} title={`Add Item`}
        />}

        <FormWrapper>
          <form>
            <FormItem

```



```

        id='title'
        label='Title'
        placeholder='Insert title'
+       value={title}
+       onChange={ (e) =>
            setTitle(e.currentTarget.v
value)}
    />
    <FormItem
        id='quantity'
        label='Quantity'
        type='number'
        placeholder='0'
+       value={quantity}
+       onChange={ (e) =>
            setQuantity(e.currentTarge
t.value)}
    />
    <FormItem
        id='price'
        label='Price'
        type='number'
        placeholder='0.00'
+       value={price}

```

```

+             handleChange={ (e) =>
                    setPrice(e.currentTarget.v
alue)}
            />
        <SubmitButton>Add
Item</SubmitButton>
    </form>
</FormWrapper>
</>
    );
};
export default Form;

```

5. The last thing you need to do now is to add a function that will be dispatched when the form is submitted by clicking the submit button. This function takes **value** for the local state, adds information about the item, and uses this to call the **addItem** function. After this function is called, the navigate function from **useNavigate** is called to go back to the overview for this list:

```

// ...
+ const { addItem } =
  useContext(ItemsContext);
+ function onSubmit(e) {
+   e.preventDefault();
+   if (title && quantity && price) {

```

```

+      addItem({
+        title,
+        quantity,
+        price,
+        listId,
+      });
+    }
+    navigate(`/list/${listId}`);
+  }
  return (
    <>
      {navigate && <NavBar goBack={() =>
        navigate(-1)} title={`Add Item`}
      />}

      <FormWrapper>
-      <form>
+      <form onSubmit={onSubmit}>
        // ...

```

When you now submit the form, a **POST** request to the mock server will be sent. You'll be sent back to the previous page where you can see the result. If successful, the **GET_LIST_SUCCESS** action was dispatched and the item you inserted was added to the list.

So far, the information from the Context has been used only separately by using the Providers, but this can also be combined into one application Context, as shown in the next section.

Creating an application Context

If you look at the current structure of the routes in your **App** component, you can imagine that this will get messy if you add more Providers and Consumers to your application. State management packages such as Redux tend to have an application state where all of the data for the application is stored. When using Context, it's possible to create an application Context that can be accessed using the **useContext** Hook. This Hook acts as a Consumer and can retrieve values from the Provider of the Context that was passed to it. Let's refactor the current application to have an application Context:

1. Start by creating a file called **AppContext.js** in the **src/context** directory. This file will import both **ListsContextProvider** and **ItemsContextProvider**, nest them, and have them wrap any component that will be passed to it as a **children** prop:

```
import { ListsContextProvider } from
'./ListsContext';
import { ItemsContextProvider } from
'./ItemsContext';
```

```

const AppContext = ({ children }) => {
  return (
    <ListsContextProvider>
      <ItemsContextProvider>{children}
    </ItemsContextProvider>
    </ListsContextProvider>
  );
};

export default AppContext;

```

2. In the **src/App.js** file, we can now import this **AppContext** file in favor of the Providers for the lists and items and replace **ListsContextProvider** and **ItemsContextProvider** with **AppContext**:

```

import styled, { createGlobalStyle } from
  'styled-components';

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

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

import Lists from './pages/Lists';

import ListDetail from
  './pages/ListDetail';

import ListForm from './pages/ListForm';

```

```

- import { ListsContextProvider } from
  './context/ListsContext';
- import { ItemsContextProvider } from
  './context/ItemsContext';
+ import AppContext from
  './context/AppContext';
  // ...
function App() {
  return (
    <>
      <GlobalStyle />
      <AppWrapper>
        <BrowserRouter>
          <Header />
+          <AppContext>
-          <ListsContextProvider>
-          <ItemsContextProvider>
            <Routes>
              // ...
            </Routes>
-          </ItemsContextProvider>
-          </ListsContextProvider>
+          </AppContext>
        </BrowserRouter>

```

```
        </AppWrapper>
      </>
    );
  }
  export default App;
```

The **AppContext** component can be extended with all of the Context objects that you might want to add in the future. Our application now has a much cleaner structure, while the data is still being retrieved by the Providers.

Code splitting with React Suspense

So far, we've focused mostly on adding new features, such as routing or state management, to our application. But not much focus has been devoted to making our application more performant, something that we can do with code splitting. A React feature called Suspense can be used for code splitting, which means you split the compiled code (your bundle) into smaller chunks. This will prevent the browser from downloading the entire bundle with your compiled code at once, and instead load your bundle in chunks depending on the components that are rendered by the browser.

NOTE

In the previous chapter, we used Next.js instead of Create React App to create our React application, which has code splitting enabled by default.

Suspense lets your components wait until the component you're importing is ready to be displayed. Before React 18 it could only be used for code splitting, but since the latest version of React it serves more purposes. When you fetch data from a component that is imported with Suspense, React will also wait until the data for that component is completely fetched.

Suspense must be used together with the lazy method, which involves using JavaScript dynamic imports to load the component only when requested. Both methods can be imported from React in `src/App.js`, where the lazy method is used to import the components for our pages:

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



```

import Header from
'./components/Header/Header';
- import Lists from './pages/Lists';
- import ListDetail from
'./pages/ListDetail';
- import ListForm from './pages/ListForm';
import AppContext from
'./context/AppContext';
+ const Lists = lazy(() =>
import('./pages/Lists'));
+ const ListDetail = lazy(() =>
import('./pages/ListDetail'));
+ const ListForm = lazy(() =>
import('./pages/ListForm'));
// ...

function App() {
// ...

```

In the **return** statement for the **App** component, **Suspense** must be used with a fallback that will be displayed when the dynamically imported components are being loaded:

```

// ...
function App() {
return (

```

```

    </>
    <GlobalStyle />
    <AppWrapper>
      <BrowserRouter>
        <Header />
+      <Suspense fallback=
{<div>Loading...</div>}>
        <AppContext>
          // ...
        </AppContext>
+      </Suspense>
        </BrowserRouter>
      </AppWrapper>
    </>
  );
}
export default App;

```

When you look at the application in the browser, you don't see any changes, unless you have a slow internet connection. In that case, the fallback for Suspense will be displayed while the component is being loaded. However, when you open the **Network** tab in the developer console, you do see a difference. Here, all the network requests are shown, and all the downloaded JavaScript as well. For our application, we can see that multiple files are loaded, such as **bundle.js**

and `main.chunk.js`. However, after applying code splitting, chunked components are also being loaded, for example, `src_pages_ListDetail.js.js`.

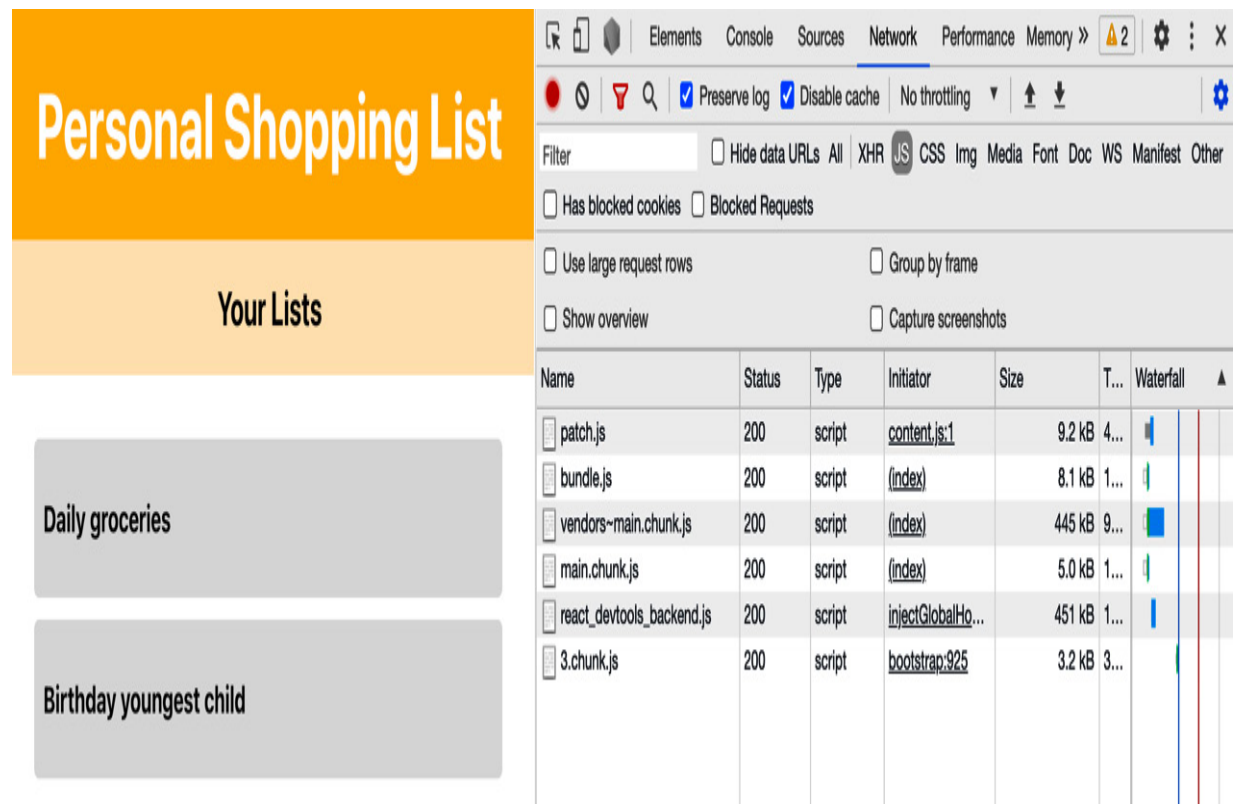


Figure 5.2 – The network requests for our application after code splitting

Looking at the main route, which is `/`, we can see that a chunk named `3.chunk.js` is loaded. This isn't a very helpful filename, something we can change with `webpackChunkName` in an inline comment. With this addition, we can instruct webpack to name the file something more user friendly:

```

    // ...

- const Lists = lazy(() =>
import('./pages/Lists'));
- const ListDetail = lazy(() =>
    import('./pages/ListDetail'));
- const ListForm = lazy(() =>
import('./pages/ListForm'));
+ const Lists = lazy(() => import(/*
webpackChunkName:
    "Lists" */ './pages/Lists'));
+ const ListDetail = lazy(() => import(/*
webpackChunkName:
    "ListDetail" */ './pages/ListDetail'));
+ const ListForm = lazy(() => import(/*
webpackChunkName:
    "ListForm" */ './pages/ListForm'));
function App() {
    // ...

```

This latest addition makes recognizing which chunks (or components) are loaded in our application much easier, as you can see by reloading the application in the browser and checking the Network tab again.

Summary

In this chapter, you've created a shopping list application that uses the Context API and Hooks to pass and retrieve data. Context is used to store data and Hooks are used to retrieve and mutate data. With the Context API, you can create more advanced scenarios for state management using the `useReducer` Hook. Context is also used to create a situation where all of the data is stored application-wide and can be accessed from any component by creating a shared Context. Finally, we've used React Suspense to apply code splitting to our bundle for improved performance.

The Context API will be used in the next chapter as well, which will show you how to build a hotel review application with automated testing using libraries such as Jest and Enzyme. It will introduce you to the multiple ways in which you can test your UI components created with React, and also show you how to test state management in your application using the Context API.

Further reading

For more information, refer to the following links:

- Consuming multiple Context objects: <https://reactjs.org/docs/context.html#consuming-multiple-contexts>
- React Suspense: <https://reactjs.org/docs/react-api.html#reactsuspense>

Chapter 6: Building an Application Exploring TDD Using the React Testing Library and Cypress

To keep your application maintainable, it is good practice to have testing set up for your project. Whereas some developers hate writing tests and therefore try to avoid writing them, other developers like to make testing the core of their development process by implementing a **Test-Driven Development (TDD)** strategy. There are many opinions about testing your applications and how to do this. Luckily, when building an application with React, many great libraries can help you with testing.

In this chapter, you'll use the **React Testing Library** tool to unit-test React applications. This library is maintained by the React community itself and ships with Create React App. It has lots of functionalities tailored to testing entire life cycles within your components and other React features. Therefore, the React Testing Library is a great fit for testing most React applications if you want to test whether functions or components behave as expected when they're given a certain in-

put. Also, we'll be exploring another tool called **Cypress** that is perfect for end-to-end testing of our React application.

The following topics will be covered in this chapter:

- Unit testing components
- Testing React state and Hooks
- End-to-end testing with Cypress

Project overview

In this chapter, we will create a hotel review application build with React that has state management with the Context API. The React Testing Library will be added to perform unit and integration testing for this application, while Cypress is used for end-to-end testing. The application has been prebuilt and uses the same patterns that we've looked at in the previous chapters.

The build time is 2 hours.

Getting started

The application for this chapter builds upon an initial version, which can be found at <https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter06-initial>. The complete code for

this chapter can be found on GitHub: <https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter06>.

Start by downloading the initial project from GitHub and move into the root directory for this project, where you must run the `npm install` command. Since this project builds upon Create React App, running this command will install `react`, `react-dom`, and `react-scripts`. Also, `styled-components` and `react-router-dom` will be installed so that they can handle styling and routing for the application. Something else that will be installed is the React Test Library, which you will recognize with the `@testing-library/*` prefix. After finishing the installation process, you can execute the `npm start` command to run the application so that you can visit the project in the browser at `http://localhost:3000`. The initial application consists of a simple header and a list of hotels. These hotels have a title and meta information, such as a thumbnail. This page will look as follows:

Hotel Reviews

Hotels



Downtown Hotel (***)



Fairytale castle (*****)



Local Bed and Breakfast (**)

Hotel Reviews

< Go
Back

Downtown Hotel (***)

+ Add
Review



Downtown Hotel (***)

Best holiday ever

Rating: 5

Donec a nisi in mi pellentesque placerat vel eu leo. Nulla facilisi. Duis dui nulla, ornare sed efficitur vitae, aliquam vel tortor. Nunc porta varius ex. Donec id porta lacus, ac facilisis nulla. Proin sed felis nec tellus dictum commodo nec quis lorem. Nam ultrices, risus ut maximus ullamcorper, elit tellus tincidunt tortor, eu euismod turpis ipsum et leo.

Very clean

Rating: 4

Duis blandit, dolor sed posuere sodales, diam lorem tempor libero, at vestibulum turpis nisl porttitor enim. Cras accumsan felis orci, a sagittis lectus porta ut.

Figure 6.1 – The initial application

If you click on any of the hotels in the list, a new page will open with a list of reviews for this hotel. By clicking the button at the top left of this page, you can move back to the previous page, and with the button at the top right, a page with a form where you can add a review will open. If you add a new review, this data will be stored in a global context and sent to a mock API server:

If you look at the project's structure, you'll see that it's using the same structure as the projects we created previously:

```
chapter-6-initial
|- node_modules
|- public
|- package.json
|- src
    |- components
        |- Button
            |- Button.js
        |- FormItem
            |- FormItem.js
        |- Header
            |- Header.js
```

```
    |- HotelItem
      |- HotelItem.js
    |- NavBar
      |- NavBar.js
    |- ReviewItem
      |- ReviewItem.js
  |- context
    |- AppContext.js
    |- HotelsContext.js
    |- ReviewsContext.js
  |- pages
    |- HotelDetail.js
    |- Hotels.js
    |- ReviewForm.js
  |- App.js
  |- index.js
  |- setupTests.js
```

Important for this chapter is the **setupTests.js** file, which is used to configure the React Testing Library for this project. The entry point of this application is a file called **src/index.js**, which renders a component called **App**. In this **App** component, all the routes are declared and wrapped within a **Router** component. These routes are as follows:

- **/**: This renders **Hotels**, with an overview of all of the hotels.

- `/hotel/:hotelId`: This renders `HotelDetail`, with an overview of all reviews for a specific hotel.
- `/hotel/:hotelId/new`: This renders `ReviewForm`, with a form to add new reviews to a specific hotel.

The data is fetched from a mock server that was created using the free **My JSON Server** service, which creates a server from the `db.json` file in the root directory of your project in GitHub. This file consists of a JSON object that has two fields, `hotels` and `reviews`, which creates multiple endpoints on a mock server. The ones you'll be using in this chapter are as follows:

- <https://my-json-server.typicode.com/PacktPublishing/React-Projects-Second-Edition/hotels>
- <https://my-json-server.typicode.com/PacktPublishing/React-Projects-Second-Edition/reviews>

The `db.json` file must be present in the master branch (or default branch) of your GitHub repository for My JSON Server to work. Otherwise, you'll receive a **404 Not Found** message when trying to request the API endpoints.

The hotel review application

In this section, we will add unit and integration testing to the hotel review application that was created in Create React App. This application lets you add reviews to a list of hotels and controls this data from an application context. The React Testing Library will be used to render React components to test assertions on these components.

Unit testing components

Unit testing is an important part of your application, since you want to know that your functions and components behave as expected, even when you make code changes. For this, we're going to use the React Testing Library, an open source testing package for React applications that was created by the React community. With the React Testing Library, you can test assertions – for example, whether the output of a function matches the value you expected.

To get started, we don't have to install anything; it's part of Create React App. If you look at the `package.json` file, you will see that a script is already there for running tests. Let's see what happens if you execute the following command from your terminal:

```
npm run test
```

This will return a message saying **No tests found related to files changed since last commit.**, which means our tests are running in watch mode and only running tests for files that have been changed.

Under the hood, the Jest test runner is used to run our tests. By pressing the **A** key, you can run all the tests, even if you haven't modified any files. If you press this key, the following message will be displayed:

```
No tests found related to files changed since
last commit.
```

Jest will automatically check all our files within the `src` directory and look for test files. In the first part of this section, we'll show how we can create tests that can be run with Jest using the React Test Library.

Creating a unit test

Since there are multiple ways that Jest can detect which file contains a test, let's choose a structure where every component has a separate test file. This test file will have the same name as the file that holds the component, with the `.test` suffix. If we choose the `NavBar` component, we can create a new file called `NavBar.test.js` in the `src/components/NavBar` directory. Add the following code to this file:

```
test('The NavBar component should render', ()
=> {

});
```

The global **test** function from Jest is used here to define a test; the test assertions can be placed within the curly brackets. Alternatively, you can also use the **describe** or **it** functions to declare a (block) of tests.

If we now run the **npm run test** command again, the Jest runner will find our first test and show the following output:

```
PASS  src/components/NavBar/NavBar.test.js
    ✓ The NavBar component should render (1 ms)
Test Suites: 1 passed, 1 total
Tests:      1 passed, 1 total
Snapshots:  0 total
Time:       3.105 s
Ran all test suites related to changed files.
```

Within the definition of a test, you can add assumptions such as **toEqual** or **toBe**, which check whether the value is exactly equal to something or whether the types just match respectively. The assumptions can be added within the callback of the **test** function:

```
test('The NavBar component should render',
() => {
+   expect(1 + 2).toBe(3);
});
```

If you still have the test script running in your terminal, you will see that Jest has detected your test. The test succeeds, since **1+2** is indeed **3**. Let's go ahead and change the assumption to the following:

```
test('The NavBar component should render',  
  () => {  
    - expect(1 + 2).toBe(3);  
    + expect(1 + 2).toBe('3');  
  });
```

Now, the test will fail, as the second assumption doesn't match. Although **1+2** still equals **3**, it's assumed that a string type with a value of **3** is returned, while in fact a number type is returned. If you're still running the **npm run test** command in the terminal, you can also see this explanation described there.

However, this assumption has no actual usage, as it doesn't test your component. To test your component, you need to render it. Rendering components so that you can test them will be handled in the next part of this section.

Rendering a React component for testing

Jest is based upon Node.js, meaning that it can't use the browser or (virtual) DOM to render your component and test its functionality.

Therefore, we'll be using the React Testing Library to help us render

these components. Create React App comes with this library by default, and the packages it uses can be found in the `package.json` file:

- `@testing-library/jest-dom`: Provides custom elements to test the DOM
- `@testing-library/react`: The core package for the React Testing Library
- `@testing-library/user-events`: Provides methods to test user interactions

The React Testing Library can render React components for us so that we can write tests for them. The preceding packages will be used to create our tests:

1. In our test file for the `NavBar` component, we can render the component with the `render` method from `@testing-library/react` and get the output of this component. With the Jest `toMatchSnapshot` assumption, we can test the structure of the component by creating a snapshot from this render and comparing it to the actual component every time this test is run:

```
+ import { render } from '@testing-  
library/react';  
+ import NavBar from './NavBar';  
  test('The NavBar component should  
render', () => {
```

```
-   expect(1 + 2).toBe('3');
+   const view = render(<NavBar />);
+   expect(view).toMatchSnapshot();
});
```

2. In the `src/components/NavBar` directory, a new directory called `__snapshots__` has now been created by Jest. Inside this directory is a file called `NavBar.test.js.snap`, which includes the snapshot. If you open this file, you will see that a rendered version of the `NavBar` component is stored here:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP
exports[`The NavBar component should render
1`] = `
Object {
  "asFragment": [Function],
  "baseElement": <body>
    <div>
      <div
        class="sc-gsDJrp PAvEv"
      >
        <h2
          class="sc-dkPtyc jFfuUr"
        />
      </div>
    </div>
  </div>
}
```

```
</body>,  
// ...
```

The components that have been created with **styled-components** will be rendered as HTML elements with a class name prefixed by **sc-***.

3. No actual values are being rendered by the React Testing Library since no props have been passed to the **NavBar** component. You can inspect how the snapshot works by passing, for instance, a **title** prop to the component:

```
import { render } from '@testing-  
library/react';  
import NavBar from './NavBar';  
// ...  
  
+ test('The NavBar component should render  
with a title',  
  () => {  
+   const view = render(<NavBar title='Test  
application'  
                           />);  
  
+   expect(view).toMatchSnapshot();  
+ });
```

4. The next time the tests are run, a new snapshot will be added to the **src/components/NavBar/__snapshots__/NavBar.test.js.snap** file. This snapshot has a value rendered for the **title** prop. If you

change the `title` prop that is displayed by the `NavBar` component in your test file, the rendered component will no longer match the snapshot. You can try this by changing the value for the `title` prop in the test scenario:

```
import { render } from '@testing-  
library/react';  
import NavBar from './NavBar';  
// ...  
test('The NavBar component should render  
with a  
  title', () => {  
-   const view =  
      render(<NavBar title='Test  
application' />);  
+   const view =  
      render(<NavBar title='Test  
application #2' />);  
    expect(view).toMatchSnapshot();  
  });
```

Jest will return the following message in the terminal, where it specifies which lines have changed in comparison to the snapshot. In this case, the title that's being displayed is no longer **Test Application** but **Test Application #2**, which doesn't match the title in the snapshot:

```

FAIL  src/components/NavBar/NavBar.test.js
  ✓ The NavBar component should render (29
ms)
  × The NavBar component should render with a
title
    (10 ms)
  ● The NavBar component should render with a
title
    expect(received).toMatchSnapshot()
    Snapshot name: `The NavBar component
should render
with a title 1`
    - Snapshot   - 3
    + Received   + 3
    @@ -6,23 +6,23 @@
        class="sc-gsDJrp PAvEv"
      >
      <h2
        class="sc-dkPtyc jFfuUr"
      >
    -      Test application
    +      Test application #2
    // ...

```

By pressing the *U* key, you can update the snapshot to handle this new test scenario. This is an easy way to test the structure of your component and see whether the title has been rendered. With the preceding test, the initially created snapshot still matches the rendered component for the first test. Also, another snapshot was created for the second test, where a **title** prop was added to the **NavBar** component.

NOTE

*You can do the same for the other props that are passed to the **NavBar** component, which renders differently if you do or don't pass certain props to it. Next to **title**, this component takes **goBack** and **openForm** as props, which can also be tested.*

We've now created two tests for our **NavBar** component, which is a good start. But something else that Jest does is show you how many lines of code have been covered by your tests. The higher your testing coverage, the more reason to assume your code is stable. You can check the test coverage of your code by executing the **test** script command with the **--coverage** flag and an extra **--** in between, or use the following command in your terminal:

```
npm run test -- --coverage
```

This command will run your tests and generate a report with all the test coverage information about your code per file. After adding the tests for **NavBar**, this report will look as follows:

----- ----- -----			
--- ----- -----			
File			% Stmts %
Branch	% Funcs	% Lines	Uncovered Line
#s			
----- ----- -----			
--- ----- -----			
All files			5
	4.68	3.12	5
src			0
	100	0	0
App.js			0
	100	0	0
index.js			0
	100	100	0
src/components/Button			100
	100	100	100

```

    Button.js | 100
| 100 | 100 | 100
|
src/components/FormItem | 0
| 0 | 0 | 0
|
    FormItem.js | 0
| 0 | 0 | 0 |
src/components/Header | 0
| 100 | 0 | 0
|
    Header.js | 0
| 100 | 0 | 0 |
src/components/HotelItem | 0
| 100 | 0 | 0
|
    HotelItem.js | 0
| 100 | 0 | 0 |
src/components/NavBar | 100
| 60 | 100 | 100
|
    NavBar.js | 100
| 60 | 100 | 100 |
// ...

```


NOTE

Testing coverage only tells us something about the lines and the functions of your code that have been tested and not their actual implementation. Having a test coverage of 100% doesn't mean that there aren't any bugs in your code, as there will always be edge cases. Also, reaching a testing coverage of 100% means that you may end up spending more time on writing tests than on actual code. Usually, a testing coverage above 80% is considered good practice.

As you can see, the test coverage for the component is 60%, meaning that most of the lines are covered in your test. To go to a 100% coverage, we will also need to add tests for the other props that are used in the **NavBar** component to render the buttons to return to the previous stage or the **form** component. Also, the coverage for the **Button** component is 100% due to no actual elements being rendered there.

However, this method of testing with snapshots will create a lot of new files and lines of code. We'll look at other ways we can test our components in the next part of this section.

Testing components with assertions

In theory, snapshot testing is not necessarily a bad practice; however, your files can get quite big over time. Also, since you're not explicitly

telling Jest what part of the component you want to test, you might need to update your code regularly.

Luckily, using snapshots isn't the only method we can use to test whether our components are rendering the correct props. Instead, we can also directly compare which props are being rendered by checking the value of the component and making assertions. The big advantage of testing with assertions is that you can test a lot without having to dig deeper into the logic of the component you're testing. For instance, you can see what the children that are being rendered look like.

Let's change our second snapshot test for the **NavBar** component to compare the impact on the test coverage. We'll need to import the **screen** method from the React Testing Library, which is used to scan the rendered components. Instead of making a snapshot of the whole component and finding the title in there, we will look for any heading components (such as **h2**) and check whether their value is equal to the prop that we set on **NavBar**:

```
- import { render } from '@testing-  
library/react';  
+ import { render, screen } from '@testing-  
library/react';  
    import NavBar from './NavBar';
```

```

    // ...
+   test('The NavBar component should render
with a title',
    () => {
-     const view =
        render(<NavBar title='Test application
#2' />);
-     expect(view).toMatchSnapshot();
+     const title = 'Test application';
+     render(<NavBar title={title} />);
+     expect(screen.getByRole('heading')).
        toHaveTextContent(title);
    });

```

We've used the `getByRole` React Testing Library method to find the `Title` component in the `NavBar` component, and the `toHaveTextContent` method to check whether the text inside `Title` is equal to our prop. The test still passes and also allows us to delete the snapshot, as we're now using an assumption to test this part of the component:

```

PASS   src/components/NavBar/NavBar.test.js
  ✓ The NavBar component should render (13
ms)
  ✓ The NavBar component should render with a
title (54 ms)
  › 1 snapshot obsolete.

```

- The NavBar component should render with a title 1

Snapshot Summary

› 1 snapshot obsolete from 1 test suite. To remove it, press ``u``.

↳ src/components/NavBar/NavBar.test.js

- The NavBar component should render with a title 1

By pressing *U* or running `npm run test` with the `-u` flag, the snapshot for the `NavBar` component is removed by Jest:

Snapshot Summary

› 1 snapshot removed from 1 test suite.

↳ src/components/NavBar/NavBar.test.js

- The NavBar component should render with a title 1

The test coverage of the `NavBar` component should still be 60%, as we continued testing whether the `title` prop was presented and rendered, which you can check by running again:

```

-----|-----|-----
---|-----|-----|
File                                     | % Stmts | %
Branch | % Funcs | % Lines | Uncovered Line
#s

```

```

-----|-----|-----
---|-----|-----|
All files | 5
| 4.84 | 3.33 | 5
|
src | 0
| 100 | 0 | 0
|
App.js | 0
| 100 | 0 | 0 |
index.js | 0
| 100 | 100 | 0
|
src/components/NavBar | 100
| 60 | 100 | 100
|
NavBar.js | 100
| 60 | 100 | 100 |
// ...

```

However, the **NavBar** component doesn't just take the **title** prop – it also takes the **goBack** and **openForm** functions as props. You also want to test whether these functions are triggered when you click on any of the buttons.

To test these props, we need to create a mock function that we can pass as a prop to **NavBar** and mock the user click events to test whether this function is being called. The **fireEvent** method from the React Testing Library can be used to mock user events, and with Jest, we can mock a function and check whether that function is called:

```
- import { render, screen } from '@testing-  
library/react';  
+ import { render, screen, fireEvent } from  
  '@testing-library/react';  
  import NavBar from './NavBar';  
  
  // ...  
+ test('The NavBar component should respond  
to button  
  clicks', () => {  
+   const mockFunction = jest.fn();  
+   render(<NavBar goBack={mockFunction} />);  
+   fireEvent.click(screen.getByText('< Go  
Back'));  
+   expect(mockFunction).toHaveBeenCalled();  
+ });
```

By running the preceding test, a click on the *back* button in **NavBar** will be simulated, and Jest will check whether the mocked function is be-

ing called. The same can be done for the **openForm** prop:

```
// ...
test('The NavBar component should respond
to button
  clicks', () => {
  const mockFunction = jest.fn();
-   render(<NavBar goBack={mockFunction} />);
+   render(<NavBar goBack={mockFunction}
openForm=
  {mockFunction} />);
  fireEvent.click(screen.getByText(' < Go
Back '));
  expect(mockFunction).toHaveBeenCalled();
+   fireEvent.click(screen.getByText(' + Add
Review '));
+   expect(mockFunction).toHaveBeenCalledTimes
s(2);
  });
```

The mocked function for both the **goBack** and **openForm** props are the same, so we need to check whether this function is called twice after clicking the open form button. By testing the user events on these two buttons, we've tested 100% of the **NavBar** component, as you can also see in the coverage report:

PASS src/components/NavBar/NavBar.test.js

✓ The NavBar component should render (27 ms)

✓ The NavBar component should render with a title (45 ms)

✓ The NavBar component should respond to button clicks (13 ms)

-----	-----	-----

File	% Stmts	%
------	---------	---

Branch	% Funcs	% Lines	Uncovered Line
--------	---------	---------	----------------

#s

-----	-----	-----

All files			5
-----------	--	--	---

	8.06		3.33		5
--	------	--	------	--	---

|

src			0
-----	--	--	---

	100		0		0
--	-----	--	---	--	---

|

App.js			0
--------	--	--	---

	100		0		0
--	-----	--	---	--	---

index.js			0
----------	--	--	---

	100		100		0
--	-----	--	-----	--	---


```

|
src/components/NavBar | 100
| 100 | 100 | 100
|
NavBar.js | 100
| 100 | 100 | 100 |
// ...

```

In this section, we've created unit tests that will test a specific part of our code. However, it can be interesting to test how different parts of our code work together. For this, we'll add integration tests to test our state management and Hooks.

Testing React state and Hooks

The tests that we've created all render components without state management, but with the React Testing Library, we also have the option to test state and Hooks. In our setup, the pages that are rendered by our router are wrapped in an application context component. If we want to test the page components, we need to make sure that the data for these pages is being mocked or stubbed, so the integration of this component with the state can be tested.

A good example of where we can test this is the **Hotels** component, which renders the list of hotels that were returned by the context:

1. As always, the starting point is to create a new file with the `.test` suffix in the same directory where the component we want to test is located. Here, we need to create the `Hotels.test.js` file in the `src/pages` directory. In this file, we need to add the following code:

```
import { render, screen } from
  '@testing-library/react';
import Hotels from './Hotels';
import HotelsContext from
  '../context/HotelsContext';
test('The Hotels component should render',
  async () => {
    const wrapper = ({ children }) => (
      <HotelsContext.Provider
        value={{
          loading: true,
          error: '',
          hotels: [],
          fetchHotels: jest.fn(),
        }}
      >
        {children}
      </HotelsContext.Provider>
    );
    render(<Hotels />, { wrapper });
```

```
expect(await screen.findByText(
  'Loading...' )).toBeVisible();
});
```

The preceding test imports the context object that the **Hotels** component uses to render the page and creates a wrapper function that creates a provider on the **HotelsContext**. To this **Provider**, we've added the mock values for the context that is used by the **Hotels** component. Our test assertion tries to find an element with the **Loading...** text value and checks whether it exists. As the value for **loading** in our context is **true**, that element can indeed be found.

NOTE

*To run just a selection of tests, you can press **P** after running the **npm run test** command; you can now type a string in the terminal that will be used to pattern-match the test files.*

2. To test whether the hotels are being rendered when data is present in the context, we need to mock this data in a new test in the **Hotels.test.js** file:

```
import { render, screen } from
  '@testing-library/react';
import Hotels from './Hotels';
import HotelsContext from
```

```

    '../context/HotelsContext';
+ import { BrowserRouter } from 'react-
router-dom';
// ...
+ test('The Hotels component should render
a list of
    hotels', async () => {
+   const wrapper = ({ children }) => (
+     <BrowserRouter>
+       <HotelsContext.Provider
+         value={{
+           loading: false,
+           error: '',
+           hotels: [
+             { id: 1, title: 'Test hotel
1',
+               thumbnail: '' },
+             { id: 2, title: 'Test hotel
2',
+               thumbnail: '' },
+           ],
+           fetchHotels: jest.fn(),
+         }}
+       >

```

```

+         {children}
+       </HotelsContext.Provider>
+     </BrowserRouter>
+   );
+ });

```

In the preceding mocked context value, the value for **loading** is set to **false**, and mocked hotels are also added. Note that we also wrapped the Provider with **BrowserRouter** from React Router, as the **Hotels** component uses a **Link** component to make the hotels clickable.

3. To test whether the hotels are being rendered, we need to add a test assertion to check whether the **loading** message is gone and whether the correct number of links to hotels are rendered:

```

// ...
test('The Hotels component should render
a list of
hotels', async () => {
  const wrapper = ({ children }) => (
    // ...
  );
+  render(<Hotels />, { wrapper });
+  expect(screen.queryByText('Loading...'))
    .toBeNull();
+  expect(screen.getAllByRole('link'

```

```
    ).length).toBe(2);  
  });
```

The **getBy** methods we used before will throw an error when an element cannot be found; to test whether something is not rendered, we need to use the **queryBy** methods instead. Also, we need to check whether two **Link** components are present by looking for the **link** role and counting them.

4. The **useEffect** Hook in the **Hotels** component can also be tested to check whether the **fetchHotels** function is being called if there are no hotels in the context. Therefore, we can edit the first test by importing the **waitFor** method from the React Testing Library and altering the context value slightly:

```
- import { render, screen } from  
  '@testing-library/react';  
+ import { render, screen, waitFor } from  
  '@testing-library/react';  
import Hotels from './Hotels';  
import HotelsContext from  
  '../context/HotelsContext';  
import { BrowserRouter } from 'react-  
router-dom';  
test('The Hotels component should  
render', async ()
```

```

=> {
+   const mockFunction = jest.fn()
  const wrapper = ({ children }) => (
    <HotelsContext.Provider
      value={{
        loading: true,
        error: '',
        hotels: [],
-       fetchHotels: jest.fn(),
+       fetchHotels: mockFunction,
      }}
    >
      {children}
    </HotelsContext.Provider>
  );
  // ...

```

5. We also add the test assertion to wait for the mock function to be called. Here, we specifically state that the mocked function, which is **fetchHotels** from **HotelsContext**, is called only once. In our **Hotels** component, **useEffect** has a check to only fetch the hotels if there is no data:

```

// ...
render(<Hotels />, { wrapper });

```

```

    expect(await
screen.findByText('Loading...')
    ).toBeVisible();
+   await waitFor(() =>
        expect(mockFunction).toHaveBeenCalledTimes(1));
    });
    // ...

```

With this test, we've tested both the context in the **Hotels** component and the **useEffect** Hook to fetch hotel data in that function.

After running the tests again with the **--coverage** flag, we will be able to see what impact writing this integration test has on our coverage. Since an integration test not only tests one specific component but multiple components at once, the testing coverage for **Hotels** will be updated. This test also covers the **HotelItem** component, which we will be able to see in the coverage report after running **npm run test --coverage**:

```

-----|-----|-----
---|-----|-----|
File                                     | % Stmts | %
Branch | % Funcs | % Lines | Uncovered Line
#s

```



```

-----|-----|-----
---|-----|-----|
All files | 19.83
| 29.03 | 16.67 | 19.83
|
src | 0
| 100 | 0 | 0
|
App.js | 0
| 100 | 0 | 0 |
index.js | 0
| 100 | 100 | 0 |
src/components/HotelItem | 100
| 100 | 100 | 100
|
HotelItem.js | 100
| 100 | 100 | 100
|
src/components/NavBar | 100
| 100 | 100 | 100
|
NavBar.js | 100
| 100 | 100 | 100
|

```

```

src/context                                |      8.16
|          0 |          0 |      8.16
|
  AppContext.js                            |          0
|      100 |          0 |          0 |
  HotelsContext.js                        |    16.67
|          0 |          0 |    16.67 |
  ReviewsContext.js                       |          0
|          0 |          0 |          0 |
src/pages                                |    21.95
|    34.21 |    20 |    21.95
|
  Hotels.js                              |    100
|    100 |    100 |    100 |
  // ...

```

According to Jest, the coverage for **Hotels** is 100%. The test coverage for **HotelItems** has also reached 100%. This means that we can skip writing unit tests for **HotelItem**, assuming that we only use this component within the **Hotels** component. The **HotelsContext** component with our context already has a small amount of coverage through testing it from the **Hotels** component. To get a higher coverage, we can test even more, such as testing how the context itself or the **useReducer** Hook in the context is behaving.

The only downside of having integration tests over unit tests is that they're harder to write, as they usually contain more complex logic. Also, these integration tests will run slower than unit tests because of them having more logic and bringing together multiple components. In the final section of this chapter, we'll be adding an end-to-end test that will test the entire application from a user perspective using Cypress.

End-to-end testing with Cypress

So far, we've covered both unit and integration testing with the React Testing Library that tests our code in an isolated setting. But in order to make sure that our application is tested as a whole, we can also write an end-to-end test to cover certain aspects of our application from start to finish. End-to-end tests are considered more time-consuming to write and run than unit or integration tests. It's recommended to have more unit and integrations tests for your project than end-to-end tests, and think about what aspects of your application you really want to have tested.

To add end-to-end tests, we'll install the open source test tool Cypress, which can be used to write and run such tests. Adding Cypress to the project requires several steps:

1. Install the library from npm in our project by running the following command from the terminal:

```
npm install cypress --save-dev
```

2. After completing the installation, the script to run Cypress needs to be added to the `package.json` file so that it can be started with a single command:

```
// ...  
"scripts": {  
  "start": "react-scripts start",  
  "build": "react-scripts build",  
  "test": "react-scripts test",  
+  "cypress": "cypress open"  
  "eject": "react-scripts eject"  
},  
// ...
```

3. You can now run the `npm run cypress` command to start Cypress. Make sure to do this in a new tab in the terminal, as you need to have both the application and Cypress running simultaneously. If this is the first time that you're running Cypress, it will validate whether it's able to run on your system. When all goes well, Cypress will open and create a new directory called `cypress` in our project, as shown in the following screenshot:

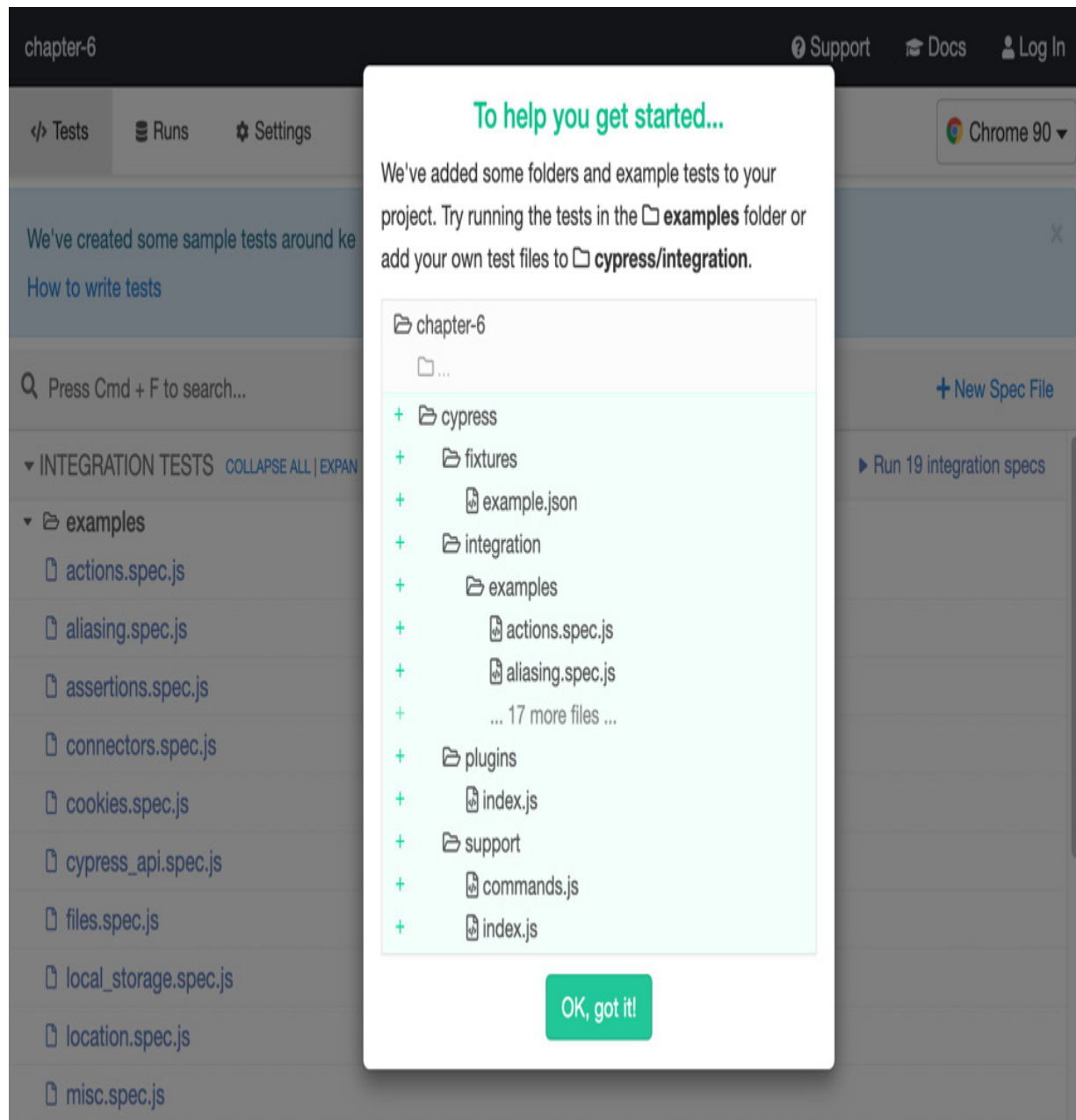


Figure 6.2 – Cypress running for the first time

4. Cypress has created example tests in the **cypress/integration/examples** directory, which you can use to learn more about how the library is working. Otherwise, you can delete these, as they will clutter the Cypress runner when we add

new tests. In the **cypress/integration** directory ,we can add a new end-to-end test for our project called **hotels.spec.js** with the following contents:

```
describe('Cypress', () => {  
  it('opens the app', () => {  
    cy.visit('http://localhost:3000')  
  })  
})
```

5. You can run this test by pressing on it in the Cypress runner, after which the test will run in a browser. Which browser it uses depends on what you've selected at the top right in Cypress. This test will open the application within a browser and test it with Cypress, giving the following output:

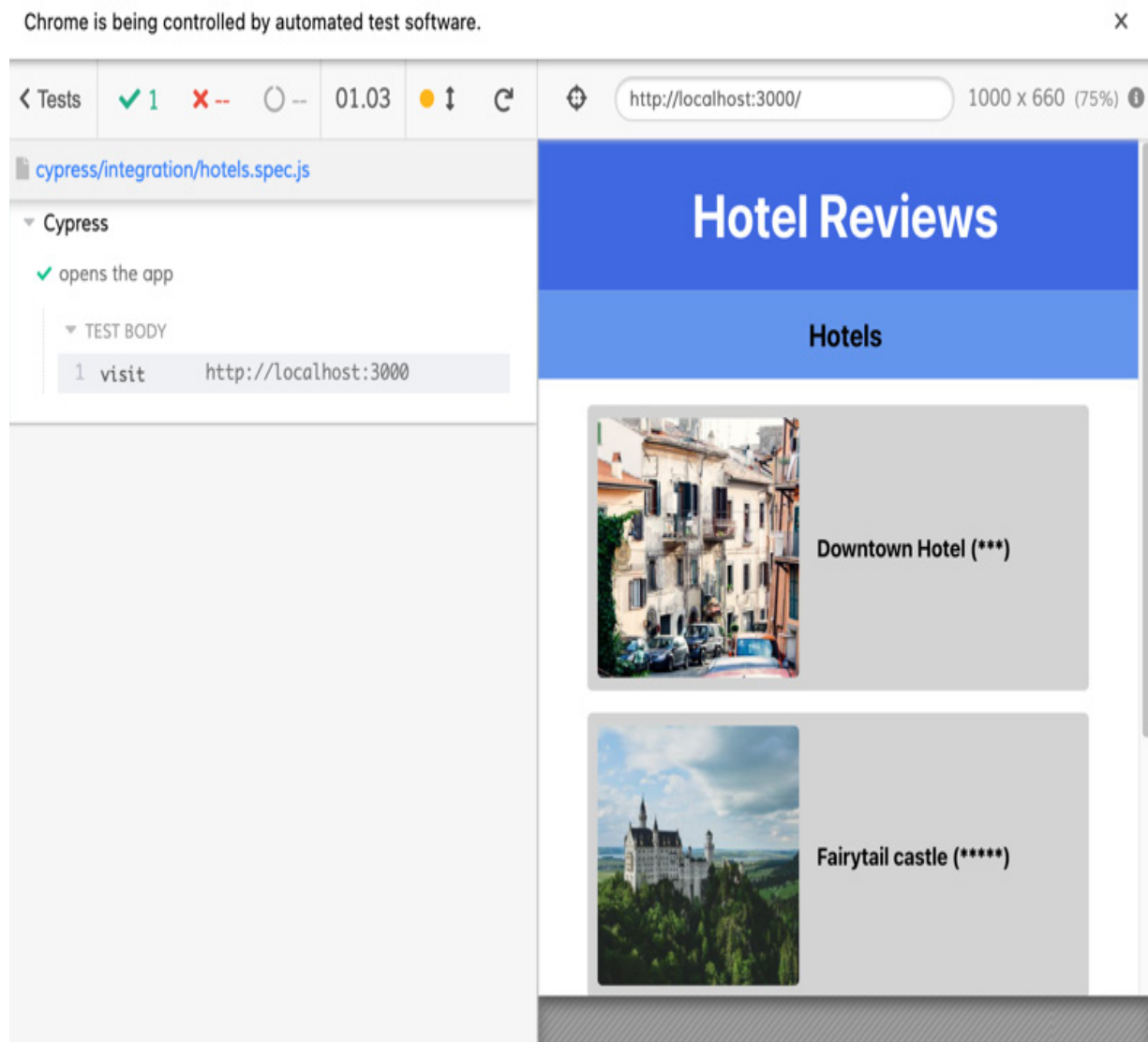


Figure 6.3 – Cypress rendering our application in a test

NOTE

You need to make sure that you have both Cypress and the application running in your terminal. This means that one terminal tab must have `npm run cypress` running and the other `npm start`. If you don't

have the application running, you'll get an error that the web page is not available.

The preceding test will just render our application, without making any assertions. To test the application using Cypress, we need to use any of the assertions that are provided by the library. Using these assertions, we'll write an end-to-end test that checks the entire flow, from visiting the application to adding a review for a hotel. This way, we have tested the most critical process of our application, namely reviewing hotels.

To start, we need to change the **hotels.spec.js** file so that it will open the application and navigate to a hotel page, and on that page, click on the button to open the form. After filling in this form, we want our test to submit the form and check whether our review has been added. Follow these steps to make these changes:

1. All the tests in the **hotels.spec.js** file will start by visiting the application in the browser, so it can click on any of the hotels listed on this page. After clicking on a hotel, we need to verify whether the location in the browser has changed by using the **cy.location** method:

```
describe('Cypress', () => {  
  -   it('opens the app', () => {
```



```
+   it('opens the app and clicks on a
hotel', () => {
      cy.visit('http://localhost:3000');
+     cy.get('a').first().click();
+     cy.location('pathname').should('includ
de',
      'hotel');
    });
  });
```

Running this test will validate that you can click on a hotel and navigate to the correct page, which you can check in the Cypress runner.

NOTE

When you need to visit many URLs over different development environments, you can also define `baseUrl` in a `cypress.json` file:

```
{
  "baseUrl": "http://localhost:3000",
}
```

2. In the second test, we tell Cypress to find the button with the **+ Add Review** text and click on it, which should change the browser's lo-

cation to the page to add the review. This page is located at the `/hotel/:hotelId/new` route and includes the `new` string. Note that we don't have to navigate to the application anymore, as this test builds upon the previous test and is therefore already at the correct page:

```
describe('Cypress', () => {
  // ...
+  it('navigates to the form to add a
review', () =>
    {
+    cy.get('button').contains('+ Add
      Review').click();
+    cy.location('pathname').should('inclu
de',
      'new' );
+  });
});
```

3. In the preceding test, Cypress will look for a button that contains a specific word, something that is not future-proof if someone changes the content of the button. Deciding which selector to use (`id`, `class`, or `content`) is important when writing tests. To prevent failing tests, you can also add the `data-cy`, `data-test`, or `data-testid` attributes to your elements. Therefore, we need to change how the `Button` component is rendered in `src/components/NavBar`:

```

// ...
function NavBar({ goBack, title, openForm
= false })
{
  return (
    <NavBarWrapper>
      {goBack && <NavBarButton onClick=
{goBack}>{`<
        Go Back`} </NavBarButton>}
      <Title>{title}</Title>
      {openForm &&
        <NavBarButton
          onClick={openForm}
+          data-cy='addReview'
        >
          {`+ Add Review`}
        </NavBarButton>
      }
    </NavBarWrapper>
  );
}
export default NavBar;

```

4. In the `cypress/integrations/hotels.spec.js` Cypress test file, we can look for the `data-cy` attribute instead of using the content

of the button as a selector:

```
describe('Cypress', () => {
  // ...
  it('navigates to the form to add a
review', () =>
    {
-      cy.get('button').contains('+ Add
      Review').click();
+      cy.get('[data-
cy=addReview]').click();
      cy.location('pathname').should('inclu
de',
        'new' );
    });
});
```

5. A third test to fill in the form to add the review and submit it can also be added to this file. Using the **cy.get** command, Cypress can find the **form** element on this page, and the **within** method is used to find the **input** elements inside the form. It will search for the name of the **input** field, add a value to it, and finally submit **button** inside the form:

```
describe('Cypress', () => {
  // ...
```

```

+   it('fills in and submits the form', ()
=> {
+     cy.get('form').within(() => {
+       cy.get('input[name=title]').type('Test
review');
+       cy.get('input[name=description]').type('Is a
test review by Cypress');
+       cy.get('input[name=rating]').type(4
);
+       cy.get('button').click();
+     });
+   });
});

```

6. Finally, we need to write a test that checks the hotel details page again and tries to find the new review that we've added. To find this review, we need to search the page for the contents of the review that was just added by Cypress; also, we need to add a `wait` command to make sure that the review has been processed and displayed on the screen:

```

describe('Cypress', () => {
  // ...

```

```
+   it('and verifies if the review is
added', () => {
+     cy.wait(600);
+     cy.get('h3').contains('Test review');
+     cy.get('div').contains('Is a test
review by
      Cypress');
+   });
  });
```

By adding this final test to Cypress, we've tested the most important scenario of our application, which you can expand even more by adding tests for edge cases such as error messages.

TIP

*We didn't add a **data-cy** attribute to the elements that display the review, which is something that you could add yourself. As we're aware of the content that we just added, it's safe to assume that we don't need a complicated selector for this.*

You can add more functionalities, such as mocking the API requests and responses, as the preceding test is using the same API as the application itself. In this scenario, there's nothing wrong with that, as the API we're using is already a mock API. If you're working in a pro-

duction environment, you will want to replace that with a mocked response that can be generated by Cypress.

For this, we need to add a **beforeEach** Hook to our test file that intercepts the API calls and replaces the response with a mocked value. The format of that mocked value should be equal to the format of the actual API. Luckily, our API is being populated from the **db.json** file that you can find in the repository for this book. From the contents of that file, you can take the data for the **hotels** field and paste it into two separate files inside the **cypress/fixtures** directory. Let's look at the steps:

1. The first one can be called **hotels.json** and needs to have an array of objects with hotel details:

```
[
  {
    "id": 1,
    "title": "Downtown Hotel (***)",
    "thumbnail":
      "https://picsum.photos/id/369/400/400"
  }
]
```

2. The second fixture needs to have a single object that replaces the API request for a single hotel, in a file called **hotel.json**:

```
{
  "id": 1,
  "title": "Downtown Hotel (***)",
  "thumbnail":
    "https://picsum.photos/id/369/400/400"
}
```

3. Intercepting the calls to the actual API can be done from the test file in `cypress/integrations/hotels.spec.js` by adding a **beforeEach** Hook and the `cy.intercept` method. For the `hotels` and `hotels/*` endpoints, it can return the fixture, and the `reviews` endpoint can return an empty array, as Cypress will add a review itself:

```
describe('Cypress', () => {
+   beforeEach(() => {
+     cy.intercept('GET', 'PacktPublishing/
      React-Projects-Second-
      Edition/hotels',
        { fixture: 'hotels.json' });
+     cy.intercept('GET', 'PacktPublishing/
      React-Projects-Second-
      Edition/hotels/*',
        { fixture: 'hotel.json' });
+     cy.intercept('GET', 'PacktPublishing/
      React-Projects-Second-
      Edition/hotels/*/reviews',
```



```
        []);  
+    })  
        it('opens the app and clicks on a  
hotel', () => {  
        // ...
```

By opening the Cypress runner, you can see that our tests are now being executed with the data from the fixtures, as the API calls are being intercepted.

The tests we've created in this section gave you a good start on writing end-to-end tests for React applications with Cypress. Also, Cypress can be used to do visual regression testing for your application or to test API responses.

Summary

In this chapter, we covered testing for React applications using the React Testing Library in combination with Jest. Both packages are a great resource for any developer that wants to add test scripts to their application, and they work very well with React. The advantages of having tests for your application were discussed in this chapter, and hopefully, you now know how to add test scripts to any project. The differences between unit tests and integration tests were shown, and you've also learned how to write end-to-end tests with Cypress.

Since the application that was tested in this chapter has the same structure as the applications from previous chapters, the same testing principles can be applied to any of the applications we've built in this book.

The next chapter will combine a lot of the patterns and libraries we've already used in this book, as we'll be creating a full-stack e-commerce store with React, GraphQL, and Apollo.

Further reading

For more information, refer to the following links:

- The React Testing Library: <https://testing-library.com/docs/react-testing-library/intro/>
- Cypress: <https://docs.cypress.io/>

Chapter 7: Building a Full-Stack E-Commerce Application with Next.js and GraphQL

If you're reading this, this means you've reached the last chapter of this book that is focused on building web applications with React. In the preceding chapters, you've already used the core features of React, such as rendering components, state management with Context, and Hooks. You've learned how to add routing to your React application, or SSR with Next.js. Also, you know how to add testing to a React application with Jest and Enzyme. Let's make this experience full stack by adding GraphQL to the list of things you've learned about so far.

In this chapter, you will not only build the frontend of an application, but also the backend. For this, GraphQL will be used, which can best be defined as a query language for APIs. Using mock data, you'll create a GraphQL server in Next.js that exposes a single endpoint for your React application. On the frontend side, this endpoint will be consumed using Apollo Client, which helps you handle sending requests to the server and state management for this data.

In this chapter, the following topics will be covered:

- Creating a GraphQL server with Next.js
- Consuming GraphQL with Apollo Client
- Handling authentication in GraphQL

Project overview

In this chapter, we will create a full stack e-commerce application in Next.js that has a GraphQL server as a backend and consumes this server in React using Apollo Client. For the frontend, an initial application is available to get you started quickly.

The build time is 3 hours.

Getting started

The project that we'll create in this chapter builds upon an initial version that you can find on GitHub:

<https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter07-initial>. The complete source code can also be found on GitHub: <https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter07>.

The initial project consists of a boilerplate application based on Next.js to get you started quickly. This application requires the installation

of several dependencies, which you can do by running the following commands:

```
npm install && npm run dev
```

This command will install all the dependencies that are needed to run the React application with Next.js, such as **react**, **next**, and **styled-components**. Once the installation process has finished, both the GraphQL server and the React application will be started.

Getting started with the initial React application

Since the React application is created with Next.js, it can be started with **npm run dev** and will be available at **http://localhost:3000/**. This initial application doesn't show any data as it still needs to be connected to the GraphQL server, which you'll do later in this chapter. At this point, the application will, therefore, just render a header with the title **E-Commerce Store** and a subheader as well, which looks something like this:

:

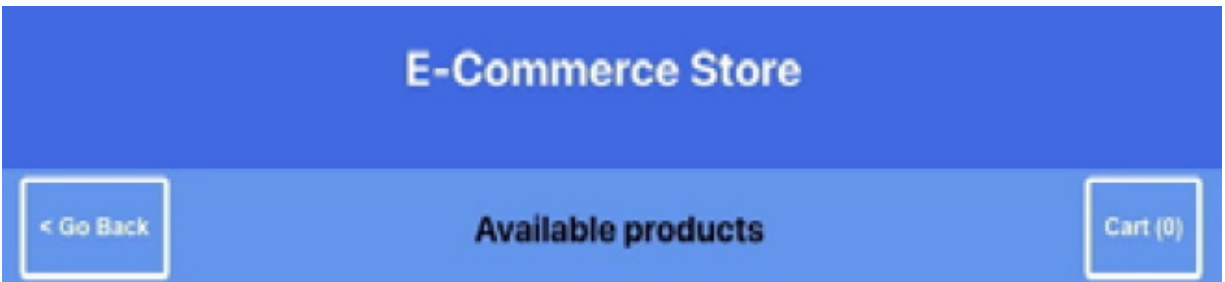


Figure 7.1 – The initial application

The structure of this initial React application built with Next.js is as follows:

```
chapter-7-initial
|- /node_modules
|- /public
|- /pages
    |- /api
        |- /hello.js
    |- /products
        |- /index.js
    |- /cart
        |- /index.js
    |- /login
        |- /index.js
    |- _app.js
    |- index.js
|- /utils
    |- hooks.js
```

```
|– authentication.js  
package.json
```

In the **pages** directory, you'll find all the routes for this application. The route `/` is rendered by **pages/index.js**, and the routes `/cart`, `/login`, and `/products` are rendered by the `.js` files in those respective directories. All routes will be wrapped within **pages/_app.js**. In this file, the header for all pages is constructed for example. All routes will also contain a **SubHeader** component, with a **Button** to go back to the previous page or a **Button** to the **Cart** component. The **utils** directory contains two files with methods that you'll be needing later in this chapter. Also, this application will have a REST endpoint available under `http://localhost:3000/api/hello` coming from the **pages/api/hello.js** file.

Building a full stack e-commerce application with React, Apollo, and GraphQL

In this section, you'll connect the React web application to the GraphQL server. A GraphQL Server on a Next.js API Route is used to create a single GraphQL endpoint that uses dynamic mock data as a source. Apollo Client is used by React to consume this endpoint and handle state management for your application.

Creating a GraphQL server with Next.js

In [Chapter 3](#), *Building a Dynamic Project Management Board*, we already created a React application with Next.js, in which it was already mentioned that you can also use it to create API endpoints. By looking at the files in our directory for this chapter, you can see that the **pages** directory contains a directory called **api** with a file called **hello.js**. All the directories and files that you create in the **pages** directory become available as a route in the browser, but if you create them under the **api** directory in **pages**, they are called API routes. The **hello.js** file is such an API route, which is available under **http://localhost:3000/api/hello**. This endpoint returns a JSON blob with the following contents:

```
{ "name": "John Doe" }
```

This is a REST endpoint, which we've also explored in the previous chapters of this book. In this chapter, we'll be using a GraphQL endpoint instead, as GraphQL is a popular format for APIs that are used by web and mobile applications.

GraphQL is best described as a query language for APIs and is defined as a convention for retrieving data from an API. Often, GraphQL APIs are compared to RESTful APIs, which is a well-known convention for sending HTTP requests that are dependent on multiple endpoints that will all return a separate data collection. As opposed to the

well-known RESTful APIs, a GraphQL API will provide a single endpoint that lets you query and/or mutate data sources such as a database. You can query or mutate data by sending a document containing either a query or mutation operation to the GraphQL server.

Whatever data is available can be found in the schema of the GraphQL server, which consists of types that define what data can be queried or mutated.

Before creating the GraphQL endpoint, we need to set up the server in Next.js. Therefore, we need to install the following dependencies that are needed to do so:

```
npm install graphql @graphql-tools/schema  
@graphql-tools/mock express-graphql
```

The **graphql** library is needed to use GraphQL in our application, while **express-graphql** is a tiny implementation of GraphQL Server for Node.js. Both **@graphql-tools/schema** and **@graphql-tools/mock** are open source libraries that helps you create GraphQL servers. We can also delete the **pages/api/hello.js** file as we won't be using this API route.

To set up the GraphQL server, we must create a new file, **pages/api/graphql/index.js**, that will contain the single GraphQL endpoint for our application. We need to import **graphqlHTTP** to create

the server. The schema for the GraphQL server is written under a variable called **typeDefs**:

```
import { graphqlHTTP } from 'express-graphql';
import { makeExecutableSchema } from '@graphql-tools/schema';
import { addMocksToSchema } from '@graphql-tools/mock';

const typeDefs = /* GraphQL */ `
  type Product {
    id: Int!
    title: String!
    thumbnail: String!
    price: Float
  }
  type Query {
    product: Product
    products(limit: Int): [Product]
  }
`;
```

Below the schema, we can initiate the GraphQL server using the **graphqlHTTP** instance and pass the schema to it. We also configure the server to create mocks for all the values in our schema. At the bottom of the file, we return the **handler** that is used by Next.js to

make the GraphQL server available at the route

http://localhost:3000/api/graphql:

```
// ...
const executableSchema = addMocksToSchema({
  schema: makeExecutableSchema({ typeDefs,
}),
});
function runMiddleware(req, res, fn) {
  return new Promise((resolve, reject) => {
    fn(req, res, (result) => {
      if (result instanceof Error) {
        return reject(result);
      }
      return resolve(result);
    });
  });
}
async function handler(req, res) {
  const result = await runMiddleware(
    req,
    res,
    graphqlHTTP({
      schema: executableSchema,
      graphiql: true,
```

```
    }),  
  );  
  res.json(result);  
}  
export default handler;
```

After making sure you've run the application again, the GraphQL API becomes available on **`http://localhost:3000/api/graphql`**. On this page in the browser, the GraphiQL playground will be displayed, and here is where you can use and explore the GraphQL server.

With this playground, you can send queries and mutations to the GraphQL server, which you can type on the left-hand side of this page. The queries and mutations that you're able to send can be found in **DOCS** for this GraphQL server, which you can find by clicking on the green button labeled **DOCS**. This button will open an overview with all the possible return values of the GraphQL server.

GraphiQL



Prettify

Merge

Copy

History

< Query

Product



1

🔍 Search Product...

No Description

FIELDS

id: Int!

title: String!

thumbnail: String!

price: Float

QUERY VARIABLES

Figure 7.2 – Using the GraphiQL playground

Whenever you describe a query or mutation on the left-hand side of this page, the output that is returned by the server will be displayed on the right-hand side of the playground. The way a GraphQL query is constructed will determine the structure of the returned data since GraphQL follows the principle of *ask for what you need, get exactly that*. Since GraphQL queries always return predictable results, we can have a query that looks like this:

```
query {  
  products {  
    id  
    title  
    price  
  }  
}
```

This will return an output that will follow the same structure of the query that's defined in the document that you sent to the GraphQL server. Sending this document with a query to the GraphQL server will return an array consisting of objects with product information, which has a limit of 10 products by default. The result will be returned in JSON format and will consist of different products every time you send the requests since the data is mocked by the GraphQL server. The response has the following format:

```
{
  "data": {
    "products": [
      {
        "id": 85,
        "title": "Hello World",
        "price": 35.610056991945214
      },
      {
        "id": 24,
        "title": "Hello World",
        "price": 89.47561381959673
      }
    ]
  }
}
```

Applications using GraphQL are often fast and stable because they control the data they get, not the server. With GraphQL we can also create relations between certain fields in our data, for example, by adding a category field to our products. This is done by adding the following to the GraphQL schema in **pages/api/graphql/index.js**:

```
// ...
const typeDefs = `
  type Product {
```

```

        id: Int!
        title: String!
        thumbnail: String!
        price: Float
+       category: Category
    }
+   type Category {
+       id: Int!
+       title: String!
+   }
    type Query {
        product: Product
        products(limit: Int): [Product]
    }
`;
// ...

```

And we can also add a query for **type Category** by adding it to the schema:

```

// ...
const typeDefs = `
    // ...
    type Category {
        id: Int!
        title: String!
    }
`

```



```

    }
    type Query {
      product: Product
      products(limit: Int): [Product]
+     categories: [Category]
    }
  `;
  // ...

```

The products will now have a new field called **category**, but you can also query a list of categories on its own. As all the data for the GraphQL server is currently mocked, you don't need to connect a data source that makes the category information available. But we can specify how certain fields should be mocked, for example, by adding a thumbnail to our products. Therefore, we need to create a variable called **mocks** that sets the field **thumbnail** on the **Product** type to be a URL to <https://picsum.photos>. This is a free server for generating mock images on the fly:

```

  // ...
+  const mocks = {
+    Product: () => ({
+      thumbnail: () =>
+    'https://picsum.photos/400/400'
+    }),
+  };

```

```

    const executableSchema =
addMocksToSchema({
    schema: makeExecutableSchema({ typeDefs,
}),
+   mocks,
    });
    // ...

```

In addition to mocking the **thumbnail** field on the **Product** type, we also want to mock all the values of fields with the **Int** or **Float** type everywhere. Both fields are now often negative values, which is incorrect for its usage as an identifier or price. The **Int** type is used to define identifiers, while the **Float** type is used for prices. We can also mock these by adding the following:

```

// ...
const mocks = {
+   Int: () => Math.floor(Math.random() * 99)
+ 1,
+   Float: () => (Math.random() * 99.0 +
1.0).toFixed(2),
    Product: () => ({
        thumbnail: () =>
'https://picsum.photos/400/400'
    }),
};

```

```
// ...
```

You can check this by trying the following query that also requests a category and the thumbnail for the products:

```
query {  
  products {  
    id  
    title  
    price  
    thumbnail  
    category {  
      id  
      title  
    }  
  }  
}
```

You can insert the preceding query in the GraphQL playground to get the response, which will look something like the following screenshot:



```
1 query {  
2   products {  
3     id  
4     title  
5     price  
6     thumbnail  
7     category {  
8       id  
9       title  
10    }  
11  }  
12 }  
13
```

QUERY VARIABLES

```
{  
  "data": {  
    "products": [  
      {  
        "id": 76,  
        "title": "Hello World",  
        "price": 90.74,  
        "thumbnail": "https://picsum.photos/400/400",  
        "category": {  
          "id": 70,  
          "title": "Hello World"  
        }  
      },  
      {  
        "id": 35,  
        "title": "Hello World",  
        "price": 27.72,  
        "thumbnail": "https://picsum.photos/400/400",  
        "category": {  
          "id": 3,  
          "title": "Hello World"  
        }  
      }  
    ]  
  }  
}
```

Figure 7.3 – Sending a query to the GraphQL server

As the data is mocked by the GraphQL Server, the values will change every time you send a new request with this query. But you can get the same response by sending the query in the body of an HTTP request, from either the command line or from a React application with **fetch**.

You can also use a library such as Apollo Client to make this more intuitive. This will be explained in the next section of this chapter, where you'll connect the GraphQL server to the React web application using Apollo and send documents to the server from your application.

Consuming GraphQL with Apollo Client

With the GraphQL server in place, let's move on to the part where you make requests to this server from a React application. For this, you'll use Apollo packages that help you add an abstraction layer between your application and the server. That way, you don't have to worry about sending documents to the GraphQL endpoint yourself by using, for example, **fetch**, and can send documents directly from a component.

Setting up Apollo Client

As we mentioned previously, you can use Apollo to connect to the GraphQL server; for this, Apollo Client will be used. With Apollo Client, you can set up the connection with the server, handle queries and mutations, and enable caching for data that's been retrieved from the GraphQL server, among other things. You can add Apollo Client to your application by following these steps:

1. To install Apollo Client and its related packages, you need to run the following command from the `client` directory where the React application is initialized:

```
npm install @apollo/client
```

This will install Apollo Client as well as the other dependencies you need to use Apollo Client and GraphQL in your React application.

NOTE

*Normally, we also need to install **graphql** when installing Apollo Client, but this library is already present in our application.*

2. These packages should be imported into the `pages/_app.js` file, where you want to create the Apollo Provider that wraps our application with the connection to the GraphQL server:

```
import { createGlobalStyle } from
  'styled-components';
+ import {
```

```

+   ApolloClient,
+   InMemoryCache,
+   ApolloProvider,
+ } from "@apollo/client";
import Header from
'../components/Header';
const GlobalStyle = createGlobalStyle`
  // ...

```

3. Now you can define the `client` constant using the **ApolloClient** class, and pass the location of the local GraphQL server to it:

```

  // ...
+ const client = new ApolloClient({
+   uri:
+ 'http://localhost:3000/api/graphql/',
+   cache: new InMemoryCache()
+ });
function MyApp({ Component, pageProps })
{
  return (
    // ...

```

4. Within the `return` function for the **MyApp** component, you need to add **ApolloProvider** and pass the `client` you've just created as a prop:

```

  // ...

```

```

function MyApp({ Component, pageProps })
{
  return (
-    <>
+    <ApolloProvider client={client}>
      <GlobalStyle />
      <Header />
      <Component {...pageProps} />
+    </ApolloProvider>
-    </>
    );
  }
  export default MyApp;

```

After these steps, all the components that are nested within **ApolloProvider** can access this **client** and send documents with queries and/or mutations to the GraphQL server. In Next.js, all the page components are rendered under **Component** based on the route. The method for getting data from **ApolloProvider** is similar to the context API that we've used before.

Sending GraphQL queries with React

Apollo Client doesn't only export a Provider, but also methods to consume the value from this Provider. That way, you can easily get any

value using the client that was added to the Provider. One of those methods is **Query**, which helps you to send a document containing a query to the GraphQL server without having to use a **fetch** function, for example.

Since a **Query** component should always be nested inside an **ApolloProvider** component, they can be placed in any component that's been rendered within **App**. One of those is the **Products** component in **pages/product/index.js**. This component is being rendered for the **/** route and should display products that are available in the e-commerce store.

To send a document from the **Products** component, follow these steps, which will guide you in the process of sending documents using **react-apollo**:

1. In the **Products** page component, you can import the **useQuery** Hook from **@apollo/client** and define a constant for the named **getProducts** query. Also, you need to import **gql** to use the GraphQL query language inside your React file as follows:

```
import styled from 'styled-components';
+ import { useQuery, gql } from
  '@apollo/client';
import SubHeader from
  '../components/SubHeader';
```

```

import ProductItem from
  '../..components/ProductItem';
// ...

+ const GET_PRODUCTS = gql`
+   query getProducts {
+     products {
+       id
+       title
+       price
+       thumbnail
+     }
+   }
+ `;

function Products() {
  // ...

```

2. The imported **useQuery** Hook can be called from **Products** and handle the data fetching process based on the query that you pass to it. In the same way as the context API, **useQuery** can consume the data from the Provider by returning a **data** variable. You can iterate over the **products** field from this object and return a list of **ProductItem** components already imported into this file. Also, a **loading** variable is returned that will be **true** when the GraphQL server hasn't returned the data yet:

```

// ...

```

```

    function Products() {
+   const { loading, data } =
useQuery(GET_PRODUCTS);
    return (
        <>
            <SubHeader title='Available
products' goToCart />
+       {loading ? (
+           <span>Loading...</span>
+       ) : (
            <ProductItemsWrapper>
+           {data && data.products &&
                data.products.map((product) =>
                    (
+                       <ProductItem key=
{product.id}
                                data={product} />
+                   )}}
            </ProductItemsWrapper>
+       )}
        </>
    );
};
export default Products;

```

This will send a document with the **GET_PRODUCTS** query to the GraphQL server when your application mounts and subsequently display the product information in the list of **ProductItem** components. After adding the logic to retrieve the product information from the GraphQL server, your application will look similar to the following:

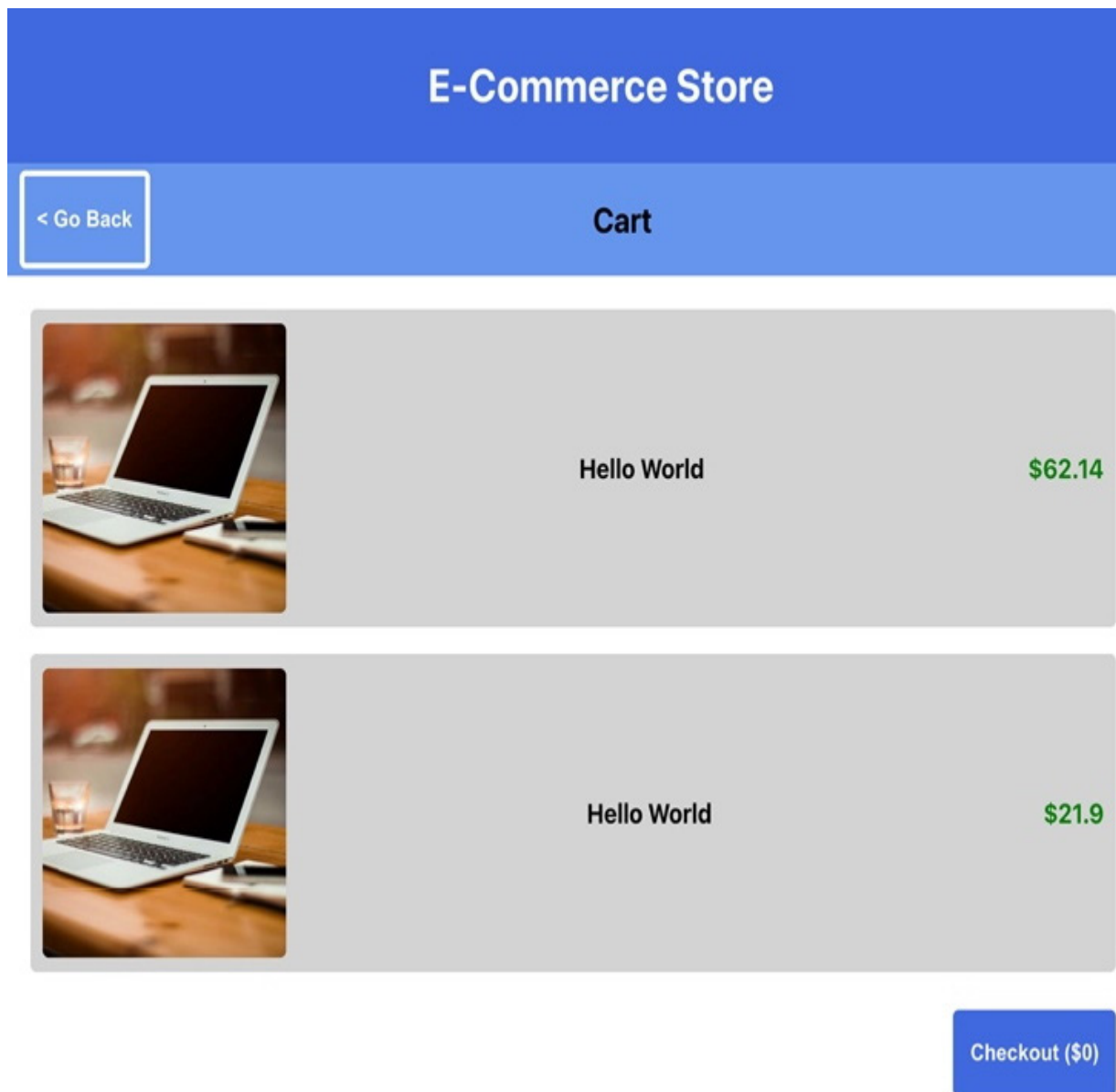


Figure 7.4 – Rendering products from GraphQL

By clicking on the button in the top-right corner of this page, you'll navigate to the `/cart` route, which also needs to query data from the GraphQL server. As we don't have a query to retrieve the cart yet, we need to add it to the GraphQL server in `pages/api/graphql/index.js`.

3. We can create a mutable variable using `let` because there is no connected data source for the GraphQL server. This is an object that we want to update later, for example, when we add products to the cart:

```
import { graphqlHTTP } from 'express-graphql';
import { makeExecutableSchema }
  from '@graphql-tools/schema';
import { addMocksToSchema } from
  '@graphql-tools/mock';
+ let cart = {
+   count: 0,
+   products: [],
+   complete: false,
+ };
const typeDefs = `
  // ...
```

4. In the schema, we need to define a type for `Cart` and add this type to the list of queries for our GraphQL server:

```

    // ...
    const typeDefs = `
      // ...
+   type Cart {
+     count: Int
+     products: [Product]
+     complete: Boolean
+   }
      type Query {
        product: Product
        products(limit: Int): [Product]
        categories: [Category]
+     cart: Cart
      }
    `;
    const mocks = {
      // ...

```

5. In the `pages/cart/index.js` file, the components to render the products in the cart are already imported. We do have to import the `useQuery` Hook and `gql` from `@apollo/client` and create the query constant:

```

import styled from 'styled-components';
+ import { useQuery, gql } from
  '@apollo/client';

```

```
import { usePrice } from
'../../utils/hooks';

import SubHeader from
'../../components/SubHeader';

import ProductItem from
  '../../components/ProductItem';

import Button from
'../../components/Button';

// ...

+ const GET_CART = gql`
+   query getCart {
+     cart {
+       products {
+         id
+         title
+         price
+         thumbnail
+       }
+     }
+   }
+ `;

function Cart() {
  // ...
}
```

6. In the **Cart** component, we need to get the data that we want to display using the **useQuery** Hook. After getting the data, we can return a list of products that are added to the cart together with the button to check out:

```
// ...
function Cart() {
+   const { loading, data } =
useQuery(GET_CART);
    return (
        <>
            <SubHeader title='Cart' />
+           {loading ? (
+               <span>Loading...</span>
+           ) : (
                <CartWrapper>
                    <CartItemsWrapper>
+                       {data && data.cart.products
&&
                        data.cart.products.map((prod
uct) => (
+                           <ProductItem key=
{product.id}
                                data={product} />
+                           )}}

```



```

        </CartItemWrapper>
+      {data &&
data.cart.products.length >
        0 && (
+      <Button
backgroundColor='royalBlue'>
        Checkout
      </Button>
+    )}
    </CartWrapper>
+  )}
  </>
  );
};
export default Cart;

```

7. This won't show any products yet as the cart is empty; the cart will be filled with products in the next section. However, let's proceed by adding a **useQuery** Hook to the button that navigates to the cart in **SubHeader**, which is rendered on routes other than **/cart** itself. A new file called **CartButton.js** can be created in the **components** directory. In this file, a **useQuery** Hook will return data from a query that requests the total number of products in the cart. Also, we can add a value to the **Button** component by adding the following code to this file:

```

import { useQuery, gql } from
 '@apollo/client';
import Button from './Button';
export const GET_CART_TOTAL = gql`
  query getCart {
    cart {
      count
    }
  }
`;
function CartButton({ ...props }) {
  const { loading, data } =
useQuery(GET_CART_TOTAL);
  return (
    <Button {...props}>
      {loading ? 'Cart' : `Cart
(${data.cart.count})`}
    </Button>
  );
}
export default CartButton;

```

8. This **CartButton** component replaces **Button**, which is now being displayed with a placeholder count for the number of products in the cart, in the **components/SubHeader.js** file:

```

import styled from 'styled-components';
import { useRouter } from 'next/router';
- import Button from './Button';
+ import CartButton from './CartButton';
// ...

function SubHeader({ title, goToCart =
false }) {
  const router = useRouter();
  return (
    <SubHeaderWrapper>
      // ...
      {goToCart && (
-        <Button onClick={() =>
          router.push('/cart')}>
-          Cart (0)
-        </Button>
+        <CartButton onClick={() =>
          router.push('/cart')} />
      )}
    </SubHeaderWrapper>
  );
}

export default SubHeader;

```

With all the components that show either a product or cart information connected to the GraphQL server, you can proceed by adding mutations that add products to the cart. How to add mutations to the application and send document container mutations to the GraphQL server will be shown in the next part of this section.

Handling mutations in GraphQL

Mutating data makes using GraphQL more interesting because when data is mutated, some side effects should be executed. For example, when a user adds a product to their cart, the data for the cart should be updated throughout the component as well. This is quite easy when you're using Apollo Client since the Provider handles this in the same way as the context API.

The GraphQL server now only has queries, but no operations as yet. Adding mutations is quite like how we've added queries to the schema before, but for the mutation, we also need to add resolvers. Resolvers are where the magic happens in GraphQL and where the schema is linked to logic to get the data, possibly from a data source. The addition of mutations is done in the `pages/api/graphql/index.js` file:

1. The first step is to add the mutation to add a product to the cart to the schema. This mutation takes `productId` as an argument. Also,

we need to mock a list of types later:

```
// ...
const typeDefs = `
// ...
const typeDefs = gql`
// ...
  type Cart {
    total: Float
    count: Int
    products: [Product]
    complete: Boolean
  }
  type Query {
    product: Product
    products(limit: Int): [Product]
    categories: [Category]
    cart: Cart
  }
+   type Mutation {
+     addToCart(productId: Int!): Cart
+   }
  `;
const mocks = {
  // ...
```

2. So far, all the values for our schema are mocked the GraphQLServer, but normally you would add resolvers for every type in the schema. These resolvers will contain the logic to get something from a data source. As we want to store the values for the **Cart** type in the **cart** object that is created at the top of this file, we need to add a resolver for the **addToCart** mutation:

```
// ...  
  
+ const resolvers = {  
+   Mutation: {  
+     addToCart: (_, { productId }) => {  
+       cart = {  
+         ...cart,  
+         count: cart.count + 1,  
+         products: [  
+           ...cart.products,  
+           {  
+             productId,  
+             title: 'My product',  
+             thumbnail:  
+               'https://picsum.photos/400/400',  
+             price: (Math.random() * 99.0  
+ 1.0).  
+               toFixed(2),  
+             category: null,
```

```

+         },
+       ],
+     };
+     return cart;
+   },
+ },
+ };

  const executableSchema =
  addMocksToSchema({
    // ...

```

3. When creating the **graphqlHTTP** instance, we need to pass the resolver that we created to it in order for our changes to become effective:

```

// ...

  const executableSchema =
  addMocksToSchema({
    schema: makeExecutableSchema({
typeDefs, }),
    mocks,
+   resolvers,
  });
  // ...

  export default handler;

```

You can already test this mutation by trying it out on the GraphQL playground that's available at <http://localhost:3000/api/graphql>. Here, you'd need to add the mutation in the upper-left box of this page. The variable that you want to include in this mutation for **productId** must be placed in the bottom-left box of this page, called **QUERY VARIABLES**. This would result in the following output:



```

1 mutation addToCart($productId: Int!) {
2   addToCart(productId: $productId) {
3     count
4     products {
5       title
6     }
7   }
8 }

```

QUERY VARIABLES

```

1 {
2   "productId": 1
3 }

```

```

{
  "data": {
    "addToCart": {
      "count": 4,
      "products": [
        {
          "title": "My product"
        },
        {
          "title": "My product"
        },
        {
          "title": "My product"
        },
        {
          "title": "My product"
        }
      ]
    }
  }
}

```

Figure 7.5 – Using mutations in the GraphQL playground

Every time you send a document to the GraphQL server with this mutation, a new product will be added to the list. Also, the **count** field will be incremented by 1. But, when you want to retrieve this information using the query for the **Cart** type, the values will still be mocked by the GraphQL Server. To return the **cart** object instead, we also need to add a resolver for the query to get the cart information:

```
// ...
const resolvers = {
+   Query: {
+     cart: () => cart,
+   },
  Mutation: {
    // ...
  },
};

const executableSchema =
addMocksToSchema({
  // ...
```

The response that will now be returned after using the **addToCart** mutation will reflect what you can retrieve with the cart query.

To be able to use this mutation from our React application, we will need to make the following changes:

1. Currently, there's no button to add a product to the cart yet, so you can create a new file in the **components** directory and call this **AddToCartButton.js**. In this file, you can add the following code:

```
import { useMutation, gql } from
  '@apollo/client';
import Button from './Button';
const ADD_TO_CART = gql`
  mutation addToCart($productId: Int!) {
    addToCart(productId: $productId) {
      count
      products {
        id
        title
        price
      }
    }
  }
`;
function AddToCartButton({ productId }) {
  const [addToCart, { data }] =
    useMutation(ADD_TO_CART);
  return (
```

```

    <Button
      onClick={() =>
        !data && addToCart({ variables: {
productId } })
      }
    >
      {data ? 'Added to cart!' : 'Add to
cart'}
    </Button>
  );
}
export default AddToCartButton;

```

This new **AddToCartButton** takes **productId** as a prop and has a **useMutation** Hook from **@apollo/client**, which uses the mutation we've created earlier. The output of **Mutation** is the actual function to call this mutation, which takes an object containing the inputs as an argument. Clicking on the **Button** component will execute the mutation and pass the **productId** to it.

2. This button should be displayed next to the products in the list on the **/** or **/products** routes, where each product is displayed in a **ProductItem** component. This means that you will need to import **AddCartButton** in **components/ProductItem.js** and pass a **productId** prop to it by using the following code:

```

import styled from 'styled-components';
import { usePrice } from
'../utils/hooks';
+ import AddToCartButton from
'./AddToCartButton';
// ...
function ProductItem({ data }) {
  const price = usePrice(data.price);
  return (
    <ProductItemWrapper>
      {data.thumbnail && <Thumbnail
        src={data.thumbnail} width={200}
      />}
      <Title>{data.title}</Title>
      <Price>{price}</Price>
+      <AddToCartButton productId=
{data.id} />
    </ProductItemWrapper>
  );
}
export default ProductItem;

```

Now, when you open the React application in the browser, a button will be displayed next to the product titles. If you click this button, the mutation will be sent to the GraphQL server, and the product will be

added to the cart. However, you won't see any changes to the button that displays **Cart (0)** in the **SubHeader** component.

3. Executing this query after sending the mutation can be done by setting a value for the **refetchQueries** option of the **useMutation** Hook in **components/AddToCartButton.js**. This option takes an array of objects with information about the queries that should be re-requested. In this case, it's only the **GET_CART_TOTAL** query, which is executed by **CartButton**. To do this, make the following changes:

```
import { useMutation, gql } from
 '@apollo/client';
import Button from './Button';
+ import { GET_CART_TOTAL } from
 './CartButton';
// ...
function AddToCartButton({ productId }) {
  const [addToCart, { data }] =
    useMutation(ADD_TO_CART);
  return (
    <Button
      onClick={() =>
        !data && addToCart({
          variables: { productId },
+          refetchQueries:
            [{ query: GET_CART_TOTAL }],
```

```

        })
      }
    >
      {data ? 'Added to cart!' : 'Add to
cart'}
    </Button>
  );
}
export default AddToCartButton;

```

4. When you click on **CartButton**, we'll navigate to the `/cart` route, where the products that we have in the cart are displayed. On here, **AddToCartButton** is also rendered, as this is defined in the **ProductItem** component. Let's change this by going to the **components/ProductItem.js** file and add the following lines of code, which will render this button conditionally:

```

// ...
- function ProductItem({ data }) {
+ function ProductItem({ data, addToCart =
false }) {
  const price = usePrice(data.price);
  return (
    <ProductItemWrapper>
      {data.thumbnail && <Thumbnail

```

```

        src={data.thumbnail} width={200}
      />}

      <Title>{data.title}</Title>
      <Price>{price}</Price>
      -      <AddToCartButton productId=
{data.id} />
      +      {addToCart && <AddToCartButton
        productId={data.id} />}
    </ProductItemWrapper>
  );
}

export default ProductItem;

```

5. From the **Products** page component, we need to pass the **addToCart** prop to render the button on this page:

```

// ...
return (
  <>
    <SubHeader title='Available
products' goToCart
    />
    {loading ? (
      <span>Loading...</span>
    ) : (
      <ProductItemsWrapper>

```



```

        {data && data.products &&
          data.products.map((product)
=> (
            <ProductItem
              key={product.id}
              data={product}
+          addToCart
            />
          ) )}
        </ProductItemsWrapper>
      )}
    </>
  );
};
export default Products;

```

Now, every time you send a mutation in a document to the GraphQL server from this component, the **GET_CART_TOTAL** query will be sent as well. If the results have changed, the **CartButton** and **Cart** components will be rendered with this new output. Therefore, the **CartButton** component will be updated to display **Cart (1)** if you click on the **AddToCartButton** component:

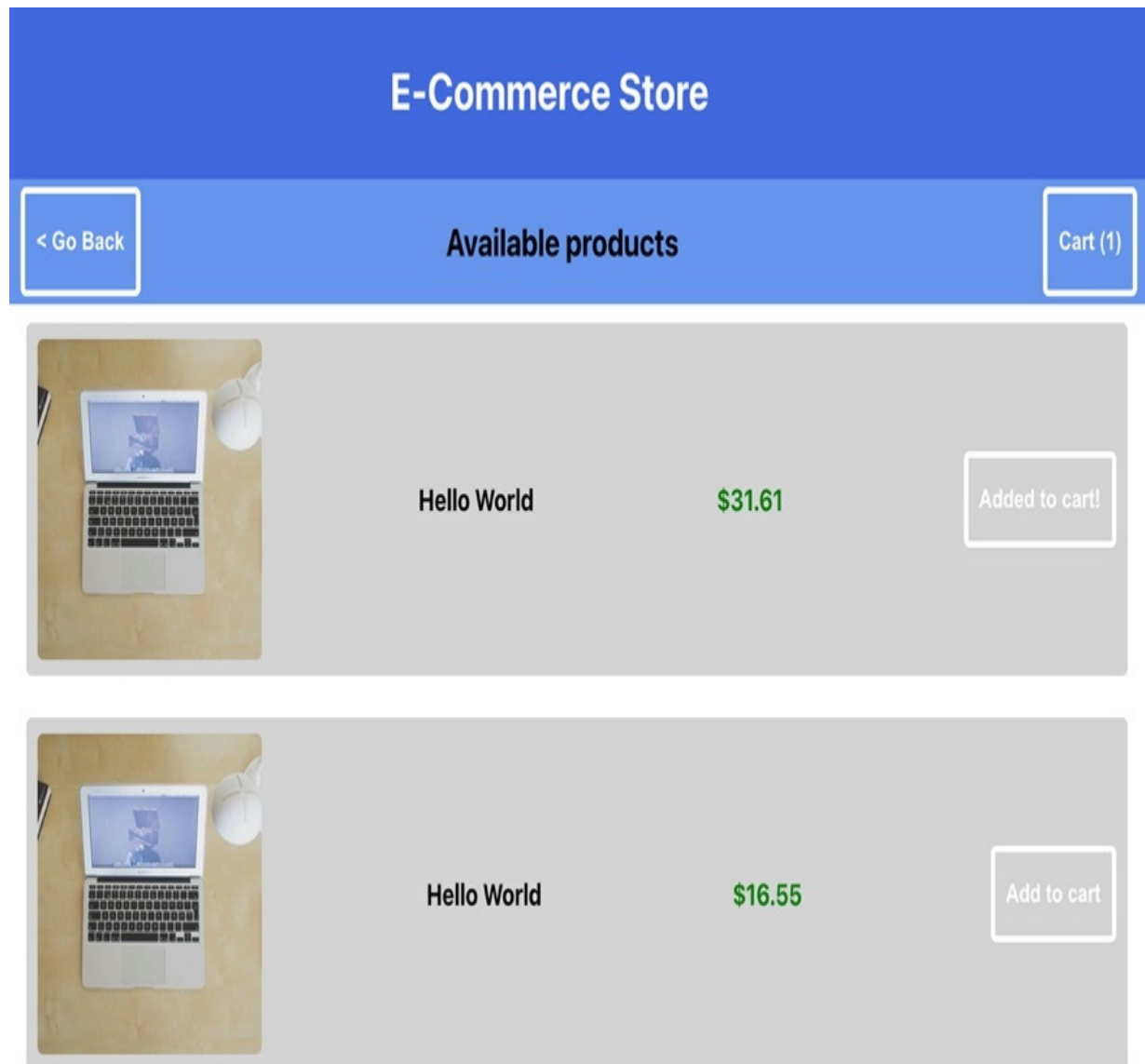


Figure 7.6 – Updating the products in the cart

In this section, we've learned how to set up Apollo Client and use it to send documents to the GraphQL server. In the next section of this chapter, we'll expand on this by handling authentication.

Handling authentication in GraphQL

Until now, we've created a GraphQL server that can be consumed by an application built with Next.js and React. Using queries and mutation, we can view a list of products and add them to a shopping cart. But we haven't added logic to check out that cart yet, which we'll do in this section.

When users have added products to the cart, you want them to be able to check out; but before that, the users should be authenticated as you want to know who's buying the product.

For authentication in frontend applications, most of the time, **JSON Web Tokens (JWTs)** are used, which are encrypted tokens that can easily be used to share user information with a backend. The JWT will be returned by the backend when the user is successfully authenticated and often, this token will have an expiration date. With every request that the user should be authenticated for, the token should be sent so that the backend server can determine whether the user is authenticated and allowed to take this action. Although JWTs can be used for authentication since they're encrypted, no private information should be added to them since the tokens should only be used to authenticate the user. Private information can only be sent from the server when a document with the correct JWT has been sent.

Before we can add the checkout process to the React application, we need to make it possible for customers to authenticate. This consists

of multiple steps:

1. We need to create a new type in the schema that defines a user and a mutation to log in a user, which we can do in

pages/api/graphql/index.js:

```
// ...
const typeDefs = `
  // ...
+   type User {
+     username: String!
+     token: String!
+   }
  type Query {
    product: Product
    products(limit: Int!): [Product]
    categories: [Category]
    cart: Cart
  }
  type Mutation {
    addToCart(productId: Int!): Cart
+   loginUser(username: String!,
password: String!):
      User
  }
`;
```

```
// ...
```

2. With the mutation defined in the schema, it can be added to the resolvers. In the `utils/authentication.js` file, a method to check the `username` and `password` combination is already present. This method will return a valid token together with the username if that combination is correct. From this file, we also import a method to check whether a token is valid:

```
import { graphqlHTTP } from 'express-graphql';
import { makeExecutableSchema }
  from '@graphql-tools/schema';
import { addMocksToSchema } from
  '@graphql-tools/mock';
+ import { loginUser, isValidToken }
  from '../utils/authentication';
// ...
const resolvers = {
  Query: {
    cart: () => cart,
  },
  Mutation: {
+   loginUser: async (_, { username,
password }) =>
    {
```

```
+      const user = loginUser(username,  
password);  
+      if (user) {  
+          return user;  
+      }  
+      },  
      // ...
```

From the GraphQL playground, we can now check whether this mutation is working by entering the username **test** and the password **test**:

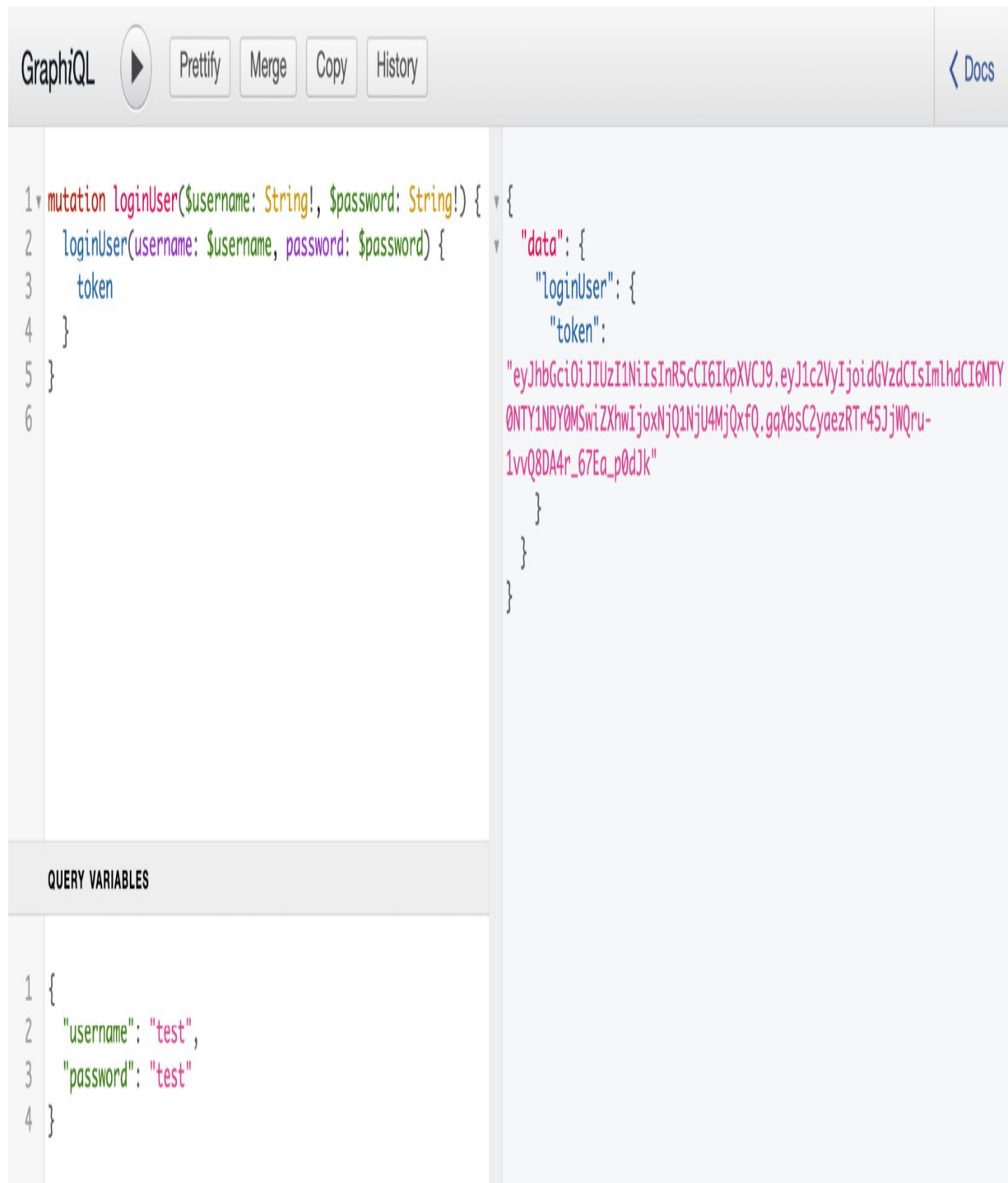


Figure 7.7 – Creating a JWT with GraphQL

3. In the `pages/login/index.js` file, we can add the logic to use the input from a form to send a document with the `loginUser` mutation

to the GraphQL server. The **Login** page component is already using **useState** Hooks to control the value of the input fields for **username** and **password**. The **useMutation** Hook can be imported from **@apollo/client**:

```
import { useState } from 'react';
+ import { useMutation, gql } from
  '@apollo/client';
  // ...

+ const LOGIN_USER = gql`
+   mutation loginUser($username: String!,
+     $password:
+       String!) {
+     loginUser(username: $username,
+       password: $password) {
+       username
+       token
+     }
+   }
+ `;

function Login() {
  const [username, setUsername] =
useState('');
  const [password, setPassword] =
useState('');
```



```

+   const [loginUser, { data }] =
      useMutation(LOGIN_USER);
    return (
      // ...

```

4. After creating the `loginUser` function, this can be added to the `onSubmit` event from the `form` element, and the values for `username` and `password` should be passed to this function as variables:

```

// ...
function Login() {
  // ...
  return (
    <>
      <SubHeader title='Login' />
      <FormWrapper>
        <form
+          onSubmit={(e) => {
+            e.preventDefault();
+            loginUser({ variables: {
username,
+              password } });
+          }}
        >
          // ...

```

5. Clicking the **Button** will send the document containing the **username** and **password** values to the GraphQL server and, if successful, it returns the JWT for this user. This token should also be stored in the session storage so that it can be used later. Also, we want to redirect the user back to the home page when logged in. To do this, we need to import a **useEffect** Hook from React that watches for changes in the data. When the token is present, we can use a **router** object obtained from a **useRouter** Hook that we need to import from Next.js:

```
- import { useState } from 'react';
+ import { useState, useEffect } from
  'react';
  import { useMutation, gql } from
  '@apollo/client';
+ import { useRouter } from 'next/router';
  // ...
  function Login() {
    const [username, setUsername] =
useState('');
    const [password, setPassword] =
useState('');
    const [loginUser, { data }] =
      useMutation(LOGIN_USER);
+   const router = useRouter();
```

```

+   useEffect(() => {
+     if (data && data.loginUser &&
        data.loginUser.token) {
+       sessionStorage.setItem('token',
        data.loginUser.token);
+       router.push('/');
+     }
+   }, [data]);
  return (
    // ...

```

6. Every time a customer logs in via the `/login` route, the token is stored in the session storage in the browser. You can delete the token from the session storage by going to the **Application** tab in the **Developer tools** section of your browser; there, you'll find another tab called **Session Storage**. The customer's authentication details in the form of the JWT are now stored in the session storage. But for the customer to check out, this token should also be sent to the GraphQL server, along with every document for the server, to validate whether the user is authenticated or whether the token has expired. Therefore, you need to extend the setup of Apollo Client to also send the token when you make a request to the server and prefix it with **Bearer**, since this is how a JWT is recognized. This requires us to make multiple changes to `pages/_app.js`:

```

import { createGlobalStyle } from
  'styled-components';
import {
  ApolloClient,
  InMemoryCache,
  ApolloProvider,
+   createHttpLink,
  } from '@apollo/client';
+ import { setContext } from
  '@apollo/client/link/context';
import Header from
'../components/Header';
  // ...

+ const httpLink = createHttpLink({
+   uri:
  'http://localhost:3000/api/graphql/',
+ });
+ const authLink = setContext((_, { headers
  }) => {
+   const token =
  sessionStorage.getItem('token');
+   return {
+     headers: {
+       ...headers,

```

```

+      authorization: token ? `Bearer
${token}` : '',
+    },
+  };
+ });
  const client = new ApolloClient({
-    uri:
    'http://localhost:3000/api/graphql/',
+    link: authLink.concat(httpLink),
    cache: new InMemoryCache(),
  });
  function MyApp({ Component, pageProps })
  {
    // ...

```

On every request to the GraphQL server, the token will now be added to the headers of the HTTP request.

7. The GraphQL Server can now get the token from the HTTP request headers and store them in the context. The context is an object that you use to store data that you want to use in your resolvers, such as a JWT. This can be done in

pages/api/graphql/index.js:

```

// ...

```

```

    const executableSchema =
addMocksToSchema({
    schema: makeExecutableSchema({
typeDefs, }),
    mocks,
    resolvers,
+   context: ({ req }) => {
+       const token =
req.headers.authorization || '';
+       return { token }
+   },
    });
// ...

```

Finally, we can also create a mutation to check out the items. This mutation should empty the card and, in a production environment, redirect the customer to a payment provider. In this scenario, we'll just empty the card and display a message that the order has been created successfully. To aid the checkout process, we need to make the following changes:

1. We require a new mutation in the schema of our GraphQL server in `pages/api/graphql/index.js`:

```

// ...
type Mutation {

```

```

        addToCart(productId: Int!): Cart
        loginUser(username: String!,
password: String!):
            User
+       completeCart: Cart
    }
    `;
    const mocks = {
        // ...

```

2. With the mutation defined in the schema, it can be added to the resolvers. The mutation needs to clear the products in the cart, set the **count** field to **0**, and the **complete** field to **true**. Also, it should check whether the user has a token stored in the context and whether this is a valid token. To check the token, we can use the previously imported **isTokenValid** method:

```

    // ...
    const resolvers = {
        Query: {
            cart: () => cart,
        },
        Mutation: {
            // ...
+       completeCart: (_, {}, { token }) => {
+           if (token && isTokenValid(token)) {

```

```

+         cart = {
+             count: 0,
+             products: [],
+             complete: true,
+         };

+         return cart;
+     }
+ },
+ },
+ };
// ...

```

3. In the `pages/cart/index.js` file, we need to import this Hook from `@apollo/client` and import `useRouter` from `Next.js` to redirect the user to the `/login` page if they are not authenticated. Also, the mutation to complete the cart can be added here:

```

import styled from 'styled-components';
import {
    useQuery,
+   useMutation,
    gql
} from '@apollo/client';
+ import { useRouter } from 'next/router';
// ...

```



```

+ const COMPLETE_CART = gql`
+   mutation completeCart {
+     completeCart {
+       complete
+     }
+   }
+ `;

function Cart() {
  // ...

```

In the return statement of the **Cart** component, there is a button to check out. This button will need to call a function created by a **useMutation** Hook that takes this new mutation. This mutation completes the cart and clears its content. If the user isn't authenticated, it should redirect the user to the **/login** page:

```

// ...

function Cart() {
  const { loading, data } =
useQuery(GET_CART);
+  const [completeCard] =
useMutation(COMPLETE_CART);
  return (
    <>
      <SubHeader title='Cart' />

```

```

    {loading ? (
      <span>Loading...</span>
    ) : (
      <CartWrapper>
        // ...
        {data &&
          data.cart.products.length > 0
        &&
+         sessionStorage.getItem('token')
        && (
          <Button
            backgroundColor='royalBlue'
+           onClick={() => {
+             const isAuthenticated =
              sessionStorage.getItem(
                'token');
+             if (isAuthenticated) {
+               completeCard();
+             }
+           }}
          >
            Checkout
          </Button>
        )}
      </CartWrapper>
    )}
  )}

```

```
        </CartWrapper>
      )}
    </>
  );
}
export default Cart;
```

This concludes the checkout process for the application and thereby this chapter, where you've used React and GraphQL to create an e-commerce application.

Summary

In this chapter, you've created a full stack React application that uses GraphQL as its backend. Using a GraphQL server and mock data, the GraphQL server was created within Next.js using API routes. This GraphQL server takes queries and mutations to provide you with data and lets you mutate that data. This GraphQL server is used by a React application that uses Apollo Client to send and receive data from the server.

That's it! You've completed the seventh chapter of this book and have already created seven web applications with React. By now, you should feel comfortable with React and its features and be ready to learn some more. In the next chapter, you'll be introduced to React

Native and learn how you can use your React skills to build a mobile application by creating an animated game with React Native and Expo.

Further reading

- Next.js API routes: <https://nextjs.org/docs/api-routes/introduction>
- GraphQL: <https://graphql.org/learn/>
- Apollo Client: <https://www.apollographql.com/docs/react/>

Chapter 8: Building an Animated Game Using React Native and Expo

One of the taglines for development with React is *"learn once, write anywhere,"* which is due to the existence of React Native. With React Native, you can write native mobile applications using JavaScript and React, and easily run and deploy these applications using a toolchain called **Expo**. The previous applications created in this book were all web applications, meaning that they will run in a browser. A downside of running applications in a browser is the lack of interaction when you click on a button or navigate to a different page. When building a mobile application that runs directly on a mobile phone, your users expect animations and gestures that make using the application easy and familiar. This is something that you'll focus on in this chapter.

In this chapter, you'll create a React Native application with add animations and gestures using the Animated API from React Native, a package called **Lottie**, and Expo's **GestureHandler**. Together, they make it possible for us to create applications that make the best use of a mobile's interaction methods, which is perfect for a game such as the *Higher/Lower* game.

To create this game, the following topics will be covered:

- Setting up React Native with Expo
- Adding gestures and animations to React Native
- Advanced animations with Lottie

Project overview

In this chapter, we will be creating an animated *Higher/Lower* game build with React Native and Expo, which uses the Animated API to add basic animations, Lottie for advanced animations, and `GestureHandler` from Expo to handle native gestures.

The build time is 1.5 hours.

NOTE

This chapter is using React Native version 0.64.3 and Expo SDK version 44. As React Native and Expo are updated frequently, make sure that you're working with this version to ensure the patterns described in this chapter are behaving as expected.

Getting started

The complete source code for the project we build in this chapter can be found on GitHub: <https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter08>. Also, the `winner.json` file that is needed in the final section of this chapter can be found at <https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter08-assets>.

You need to have the Expo Go application installed on a mobile iOS or Android device to run the project on a physical device. Once you've downloaded the application, you need to create an Expo account to make the development process smoother. Make sure to store your account details somewhere safe, as you will need these later on in this chapter.

Alternatively, you can install either Xcode or Android Studio on your computer to run the application on a virtual device:

- **For iOS:** Information on how to set up your local machine to run the iOS simulator can be found here: <https://docs.expo.io/workflow/ios-simulator/>.
- **For Android:** Information on how to set up your local machine to run the emulator from Android Studio can be found here: <https://docs.expo.io/workflow/android-studio-emulator/>.

NOTE

It's highly recommended to use the Expo Client application to run the project from this chapter on a physical device. Receiving notifications is currently only supported on physical devices, and running the project on either the iOS simulator or Android Studio emulator will result in error messages.

Creating an animated game application with React Native and Expo

In this section, you'll build an animated game with React Native and Expo that runs directly on a mobile device. React Native allows you to use the same syntax and patterns you already know from React, as it's using the core React library. Also, Expo makes it possible to prevent having to install and configure Xcode (for iOS) or Android Studio to start creating native applications on your machine. Therefore, you can write applications for both the iOS and Android platforms from any machine.

Expo combines both React APIs and JavaScript APIs to the React Native development process, such as JSX components, Hooks, and native features such as camera access. Briefly, the Expo toolchain consists of multiple tools that help you with React Native, such as the

Expo CLI, which allows you to create React Native projects from your terminal, with all the dependencies that you need to run React Native. With the Expo client, you can open these projects from iOS and Android mobile devices that are connected to your local network, and Expo SDK is a package that contains all the libraries that make it possible to run your application on multiple devices and platforms.

Setting up React Native with Expo

Applications that we previously created in this book used Create React App or Next.js to set up a starter application. For React Native, a similar boilerplate is available, which is part of the Expo CLI and can be set up just as easily.

You need to globally install the Expo CLI with the following command, using Yarn:

```
yarn global add expo-cli
```

Alternatively, you can use npm:

```
npm install -g expo-cli
```

NOTE

Expo is using Yarn as its default package manager, but you can still use it with npm instead as weve done in the previous React chapters.

This will start the installation process, which can take some time, as it will install the Expo CLI with all its dependencies to help you develop mobile applications. After that, you will be able to create a new project using the `init` command from the Expo CLI:

```
expo init chapter-8
```

Expo will now create the project for you, but before that, it will ask you whether you want to create just a blank template, a blank template with TypeScript configuration, or a sample template with some example screens set up. For this chapter, you'll need to choose the first option. Expo automatically detects whether you have Yarn installed on your machine; if so, it will use Yarn to install the other dependencies that are needed to set up your computer.

Your application will now be created, using the setting you've previously selected. This application can now be started by moving into the directory that was just created by Expo, using the following commands:

```
cd chapter-8  
yarn start
```

This will start Expo and give you the ability to start your project from both the terminal and your browser. In the terminal, you will now see a QR code, which you can scan with the Expo application from your mobile device, or you can start either the iOS or Android emulator if

you have Xcode or Android studio installed. Also, *Expo DevTools* will be opened in your browser after running the **start** command:

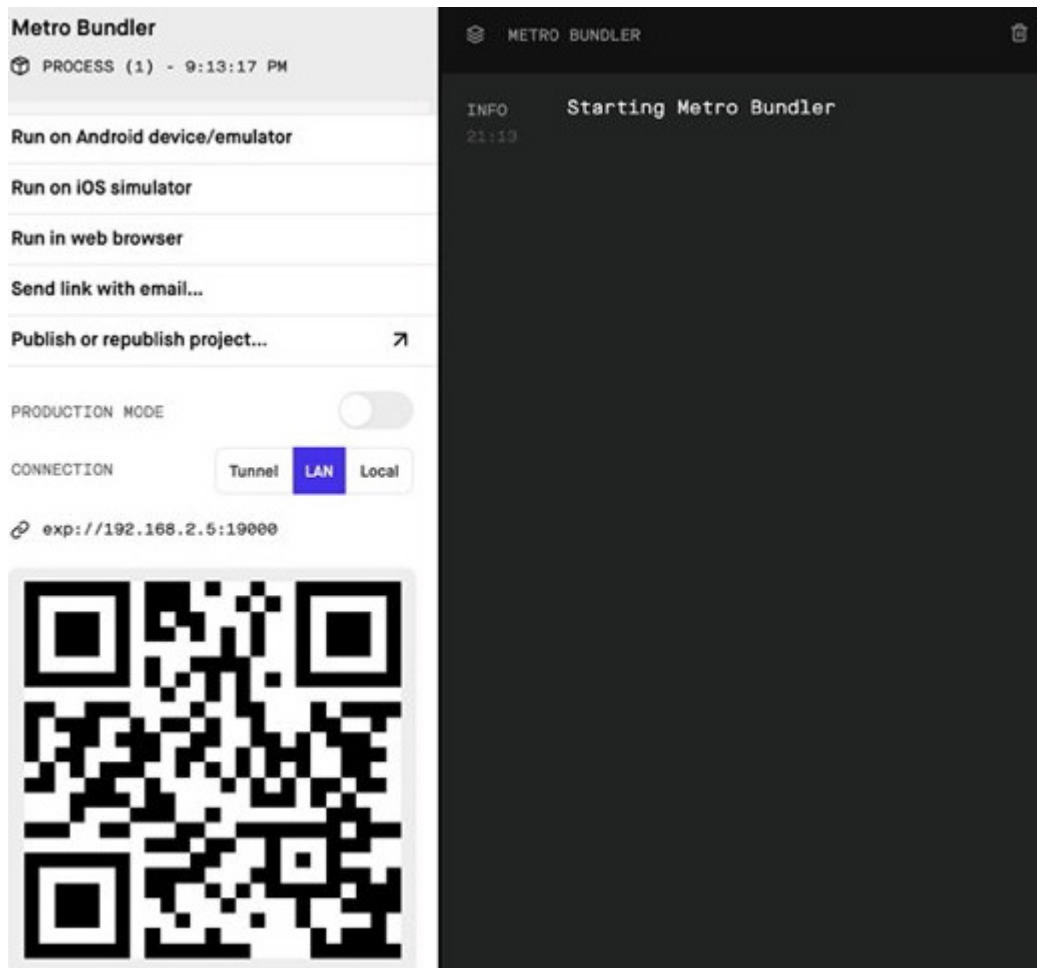


Figure 8.1 – Expo DevTools when running Expo

On this page, you will see a sidebar on the left and the logs from your React Native application on the right. If you're using an Android device, you can scan the QR code directly from the Expo Go application. On iOS, you need to use your camera to scan the code, which will ask you to open the Expo client. Alternatively, the sidebar in Expo DevTools has buttons to start the iOS or Android emulator, for which

you need to have either Xcode installed or Android Studio installed. Otherwise, you can also find a button to send a link to the application by email.

It doesn't matter whether you've opened the application using the emulator for iOS or Android, or from an iOS or Android device; the application at this point should be a white screen displaying **Open up App.js to start working on your app!**.

NOTE

If you don't see the application, but a red screen displaying an error, you should make sure that you're running the correct version of React Native and Expo on your local machine and mobile device. These versions should be React Native version 0.64.3 and Expo version 44. Using any other version can lead to errors, as the versions for React Native and Expo should be in sync.

The project structure from this React Native application created with Expo is quite similar to the React projects you've created in the previous chapters:

```
chapter-8
|- node_modules
|- assets
|- package.json
```

```
| - App.js  
| - app.json  
| - babel.config.js
```

In the **assets** directory, you can find the images that are used as the application icon on the home screen once you've installed this application on your mobile device and the image that will serve as the splash screen, which is displayed when you start the application. The **App.js** file is the actual entry point of your application, where you'll put code that will be rendered when the application mounts. Configurations for your application – for example, the App Store – are placed in **app.json**, while **babel.config.js** holds specific Babel configurations.

Adding basic routing

For web applications created with React, we've used React Router for navigation, while with Next.js, the routing was already built in using the filesystem. For React Native, we'll need a different routing library that supports both iOS and Android. The most popular library for this is **react-navigation**, which we can install from Yarn:

```
yarn add @react-navigation/native
```

This will install the core library, but we need to extend our current Expo installation with dependencies that are needed for **react-navigation** by running the following:

```
expo install react-native-screens react-native-safe-area-context
```

To add routing to your React Native application, you will need to understand the difference between routing in a browser and a mobile application. History in React Native doesn't behave the same way as in a browser, where users can navigate to different pages by changing the URL in the browser and previously visited URLs are added to the browser history. Instead, you will need to keep track of transitions between pages yourself and store local history in your application.

With React Navigation, you can use multiple different navigators to help you do this, including a stack navigator and a tab navigator. The stack navigator behaves in a way that is very similar to a browser, as it stacks pages after transition on top of each other and lets you navigate using native gestures and animations for iOS and Android. Let's get started:

1. First, we need to install the library to use stack navigation and an additional library with navigation elements from **react-navigation**:

```
yarn add @react-navigation/native-  
stack@react-navigation/elements
```

2. From this library and the core library from **react-navigation**, we need to import the following to create a stack navigator in **App.js**:

```
import { StatusBar } from 'expo-status-  
bar';
```

```

import React from 'react';
import { StyleSheet, Text, View } from
  'react-native';
+ import { NavigationContainer }
    from '@react-navigation/native';
+ import { createNativeStackNavigator }
    from '@react-navigation/native-stack';
+ const Stack =
  createNativeStackNavigator();
  export default function App() {
    // ...

```

3. From the **App** component, we need to return this stack navigator, which also needs a component to return to the home screen. Therefore, we need to create a **Home** component in a new directory called **screens**. This component can be created in a file called **Home.js** with the following content:

```

import React from 'react';
import { StyleSheet, Text, View } from
  'react-native';
export default function Home() {
  return (
    <View style={styles.container}>
      <Text>Home screen</Text>
    </View>

```

```

    );
  }
  const styles = StyleSheet.create({
    container: {
      flex: 1,
      backgroundColor: '#fff',
      alignItems: 'center',
      justifyContent: 'center',
    },
  });

```

4. In **App.js**, we need to import this **Home** component and set up the stack navigator by returning a **NavigationContainer** component from the **App** component. Inside this component, the stack navigator is created by the **Navigator** component from the **Stack** component, and the home screen is described in a **Stack.Screen** component. Also, the status bar for the mobile device is defined here:

```

import { StatusBar } from 'expo-status-
bar';
import React from 'react';
- import { StyleSheet, Text, View } from
  'react-native';
+ import { StyleSheet } from 'react-
native';
import { NavigationContainer }

```



```

    from '@react-navigation/native';
    import { createNativeStackNavigator }
    from '@react-navigation/native-stack';
+   import Home from './screens/Home';
const Stack = createNativeStackNavigator();
export default function App() {
  export default function App() {
    return (
-     <View style={styles.container}>
-     <Text>Open up App.js to start
working on your
      app!</Text>
+     <NavigationContainer>
      <StatusBar style='auto' />
+     <Stack.Navigator>
+     <Stack.Screen name='Home'
component={Home}
      />
+     </Stack.Navigator>
+     </NavigationContainer>
-     </View>
    );
  }
  // ...

```

Make sure that you still have Expo running from your terminal; otherwise, start it again with the `yarn start` command. The application on your mobile device or emulator should now look like this:

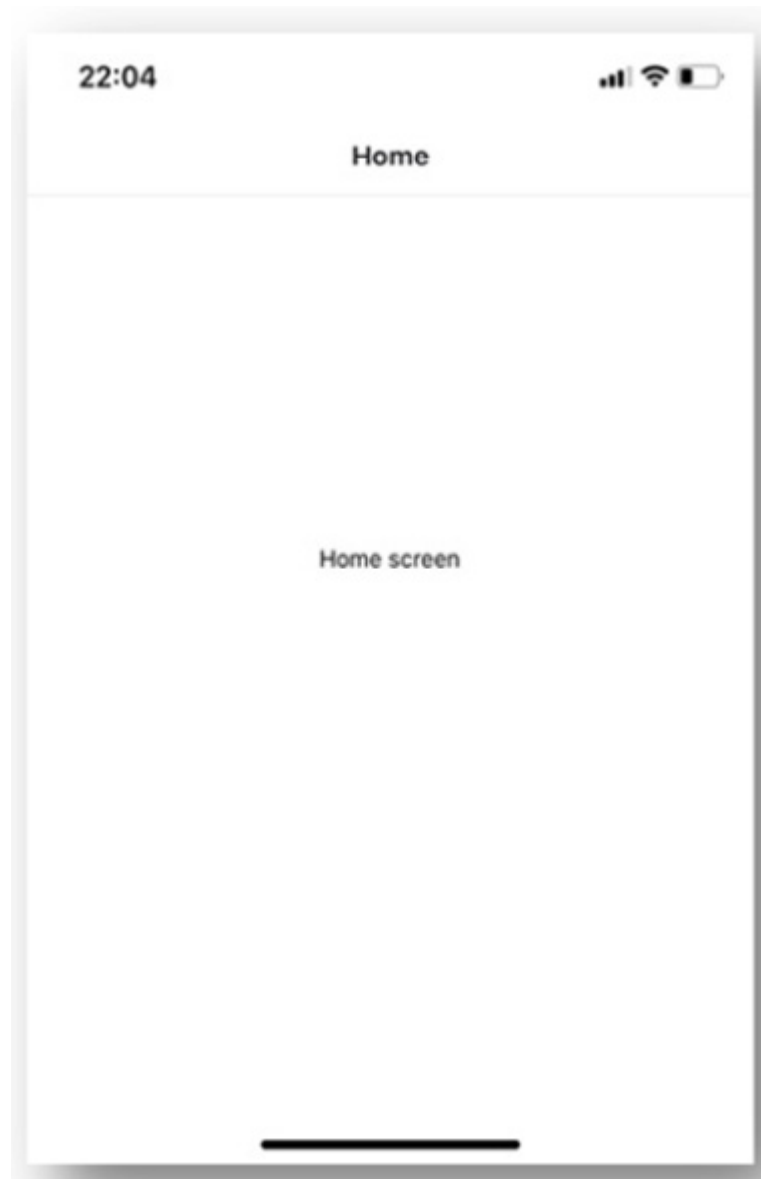


Figure 8.2 – The application with a stack navigator

NOTE

To reload the application in Expo Go, you can shake the device when you're using an iOS or Android phone. By shaking the device, a menu with an option to reload the application will appear. In this menu, you must also select to enable a fast refresh to refresh the application automatically when you make changes to the code.

We've got our stack navigator with the first page set up, so let's add more pages and create buttons to navigate between them in the next part of this section.

Navigate between screens

Navigating between screens in React Native also works a bit differently than in the browser, as again there are no URLs. Instead, you need to use the navigation object that is available as a prop from components that are rendered by the stack navigator, or by calling the **useNavigation** Hook from **react-navigation**.

Before learning how to navigate between screens, we need to add another screen to navigate to:

1. This screen can be added by creating a new component in a file called **Game.js** in the **screens** directory with the following code:

```
import React from 'react';  
import { StyleSheet, Text, View } from  
'react-native';
```

```

export default function Game() {
  return (
    <View style={styles.container}>
      <Text>Game screen</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});

```

2. This component must be imported in **App.js** and added as a new screen to the stack navigator. Also, on the navigator, we need to set the default screen that must be displayed by setting the **initialRouteName** prop:

```

import { StatusBar } from 'expo-status-bar';

import React from 'react';

import { StyleSheet } from 'react-native';

```

```

import { NavigationContainer }
  from '@react-navigation/native';
import { createNativeStackNavigator }
  from '@react-navigation/native-stack';
import Home from './screens/Home';
+ import Game from './screens/Game';
const Stack =
createNativeStackNavigator();
export default function App() {
  return (
    <NavigationContainer>
      <StatusBar style='auto' />
-      <Stack.Navigator>
+      <Stack.Navigator
initialRouteName='Home'>
        <Stack.Screen name='Home'
component={Home}
        />
+      <Stack.Screen name='Game'
component={Game}
        />
      </Stack.Navigator>
    </NavigationContainer>
  );

```

```
}  
// ...
```

3. From the **Home** component in `screens/Home.js`, we can get the navigation object from the **useNavigation** Hook and create a button that will navigate to the **Game** screen when pressed. This is done by using the **navigate** method from the **navigation** object and passing it to the **onPress** prop of the **Button** component from React Native:

```
import React from 'react';  
- import { StyleSheet, Text, View } from  
  'react-native';  
+ import { StyleSheet, View, Button } from  
  'react-native';  
+ import { useNavigation } from  
  '@react-navigation/native';  
export default function Home() {  
+   const navigation = useNavigation();  
  return (  
    <View style={styles.container}>  
-      <Text>Home screen</Text>  
+      <Button onPress={() =>  
navigation.navigate(  
          'Game')} title='Start game!' />  
    </View>
```

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

From the application, you can now move between the **Home** and **Game** screen by using the button that we just created or by using the button in the header. This header is automatically generated by react-navigation, but you can also customize this, which we'll do in [*Chapter 9, Building a Full-Stack Social Media Application with React Native and Expo*](#):

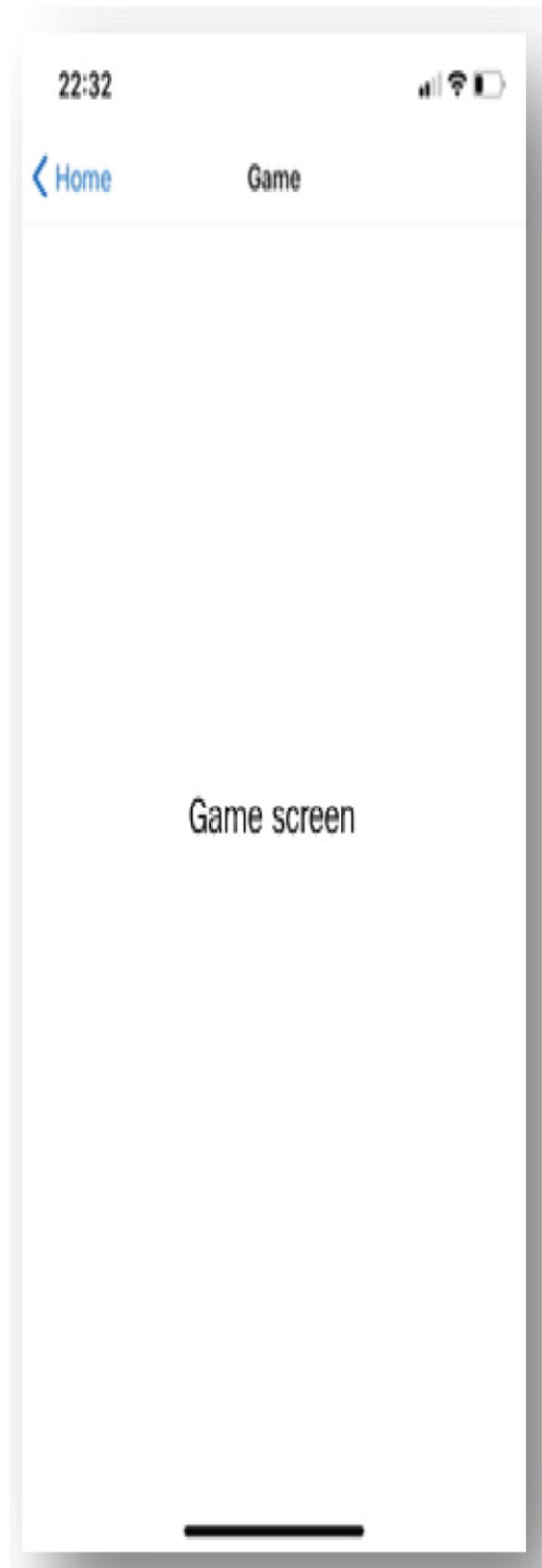
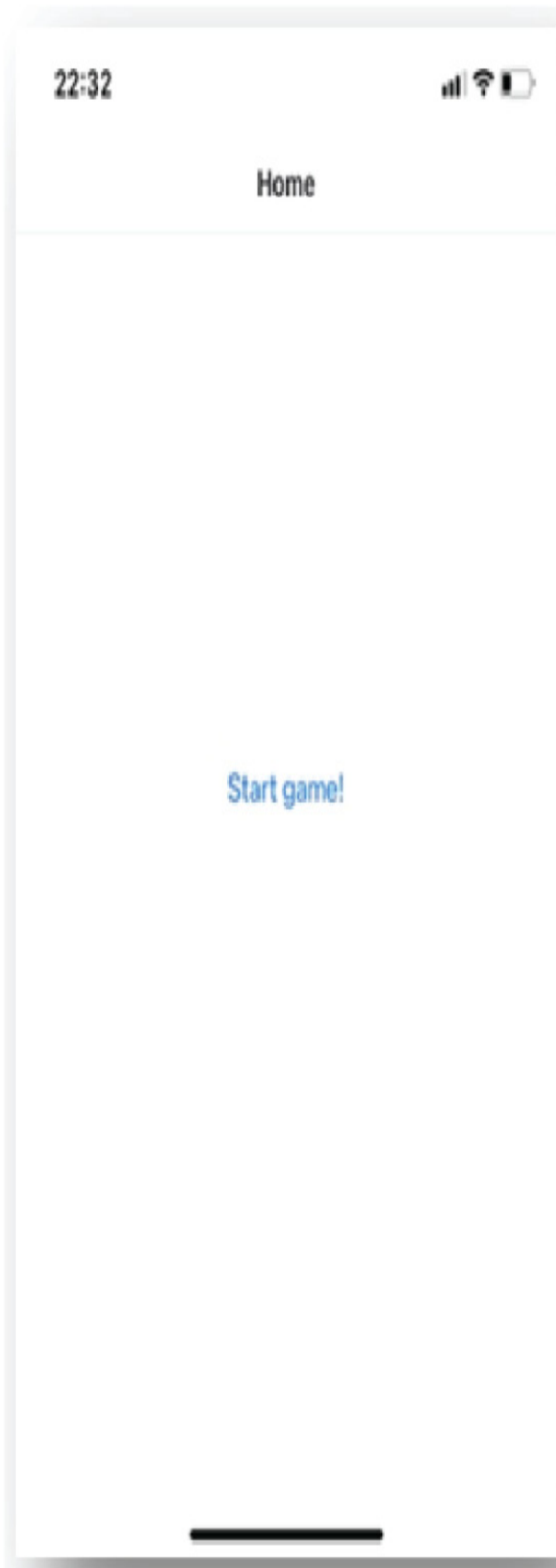


Figure 8.3 – Our application with basic routing

At this point, we've added basic routing to our application, but we don't have a game yet. In the `screens/Game.js` file, the logic for the *Higher/Lower* game can be added by using local state management, using the `useState` and `useEffect` Hooks. These Hooks work the same in React Native as they do in a React web application. Let's add the game logic:

1. Import these Hooks from React in the Game component, next to the `Button` and `Alert` components from React Native. After importing them, we need to create a local state variable to store the user's choice and create the randomized number and score for the game. Also, import the `useNavigation` Hook from `react-navigation`:

```
- import React from 'react';
- import { StyleSheet, Text, View } from
  'react-native';
+ import React, { useEffect, useState }
  from 'react';
+ import { Button, StyleSheet, Text, View,
  Alert }
  from 'react-native';
+ import { useNavigation } from
  '@react-navigation/native';
```

```

    export default function Game() {
+   const baseNumber =
Math.floor(Math.random() *
        100);
+   const score = Math.floor(Math.random()
* 100);
+   const [choice, setChoice] =
useState('');

    return (
        <View style={styles.container}>
        // ...

```

The **baseNumber** value is the number that starts the game with an initial random value between 1 and 100, created with a **Math** method from JavaScript. The score value also has a random number as a value, and this value is used to compare with **baseNumber**. The **choice** local state variable is used to store the choice of the user if a score is either higher or lower than **baseNumber**.

2. To be able to make a choice, we need to add two **Button** components that set the value for a choice to be higher or lower, depending on which button you've pressed:

```

// ...
return (

```

```

        <View style={styles.container}>
-      <Text>Game screen</Text>
+      <Text>Starting: {baseNumber}</Text>
+      <Button onPress={() =>
setChoice('higher')}
          title='Higher' />
+      <Button onPress={() =>
setChoice('lower')}
          title='Lower' />
        </View>
      );
    }
    const styles = StyleSheet.create({
      // ...

```

3. From an `useEffect` Hook, we can compare the values for `baseNumber` and `score` and, based on the value choice, show an alert. Depending on the choice, the user sees an `Alert` component displayed with a message saying whether they've won or not, and the score. Next to displaying the alert, the values for `baseNumber`, `score`, and `choice` the navigation object will be used to navigate back to the previous page. This will reset the `Game` component as well:

```

      // ...
+    const navigation = useNavigation();

```

```
+ useEffect(() => {
+   if (choice) {
+     const winner =
+       (choice === 'higher' && score >
baseNumber) ||
+       (choice === 'lower' && baseNumber >
score);
+     Alert.alert(`You've ${winner ? 'won'
: 'lost'}`,
+       `You scored: ${score}`);
+     navigation.goBack();
+   }
+ }, [baseNumber, score, choice]);
  return (
    <View style={styles.container}>
      // ...
```

You're now able to play the game and choose whether you think the score will be higher or lower than the displayed **baseNumber**. But we haven't added any styling yet, which we'll do in the next part of this section.

Styling in React Native

You might have seen in the previous components that we changed or added to the project that we used a variable called **StyleSheet**. Using this variable from React Native, we can create an object of styles, which we can attach to React Native components by passing it as a prop called **style**. We've already used this to style the components with a style called **container**, but let's make some changes to also add styling to the other components:

1. In `screens/Home.js`, we need to replace the **Button** component with a **TouchableHighlight** component, as **Button** components in React Native are hard to style. This **TouchableHighlight** component is an element that can be pressed, and it gives the user feedback by getting highlighted when pressed. Inside this component, a **Text** component must be added to display the label for the button:

```
import React from 'react';
- import { StyleSheet, View, Button } from
  'react-native';
+ import { StyleSheet, Text, View,
  TouchableHighlight
    } from 'react-native';
import { useNavigation } from
  '@react-navigation/native';
export default function Home() {
  const navigation = useNavigation();
```

```

    return (
      <View style={styles.container}>
-      <Button onPress={() =>
navigation.navigate(
      'Game')} title='Start game!' />
+      <TouchableHighlight
+      onPress={() =>
navigation.navigate('Game')}
+      style={styles.button}
+      >
+      <Text style={styles.buttonText}>
      Start game!</Text>
+      </TouchableHighlight>
      </View>
    );
  }
  // ...

```

2. The **TouchableHighlight** and **Text** components use the **button** and **buttonText** styles from the **styles** object, which we need to add to the **create** method of **StyleSheet** at the bottom of the file:

```

// ...
const styles = StyleSheet.create({
  container: {

```

```
        flex: 1,
        backgroundColor: '#fff',
        alignItems: 'center',
        justifyContent: 'center',
      },
+   button: {
+     width: 300,
+     height: 300,
+     display: 'flex',
+     alignItems: 'center',
+     justifyContent: 'space-around',
+     borderRadius: 150,
+     backgroundColor: 'purple',
+   },
+   buttonText: {
+     color: 'white',
+     fontSize: 48,
+   },
  });
```

Creating styles with React Native means you need to use *camelCase* notation instead of *kebab-case* as we're used to with CSS – for example, `background-color` becomes `backgroundColor`.

3. We also need to make styling additions to the buttons on the **Game** screen by opening the `screens/Game.js` file. In this file, we again need to replace the **Button** components from React Native with a **TouchableHighlight** component with an inner **Text**:

```
import React, { useEffect, useState }
from 'react';
import {
-   Button,
    StyleSheet,
    Text,
    View,
    Alert,
+   TouchableHighlight,
} from 'react-native';
import { useNavigation } from
    '@react-navigation/native';
export default function Game() {
    // ...
    return (
        <View style={styles.container}>
-           <Text>Starting: {baseNumber}</Text>
-           <Button onPress={() =>
setChoice('higher')}
                title='Higher' />
```



```

-      <Button onPress={() =>
setChoice('lower')}
      title='Lower' />
+      <Text style={styles.baseNumber}>
      Starting: {baseNumber}</Text>
+      <TouchableHighlight onPress={() =>
      setChoice('higher')} style=
{styles.button}>
+      <Text style=
{styles.buttonText}>Higher
      </Text>
+      </TouchableHighlight>
+      <TouchableHighlight onPress={() =>
      setChoice('lower')} style=
{styles.button}>
+      <Text style=
{styles.buttonText}>Lower</Text>
+      </TouchableHighlight>
      </View>
    );
  }
  // ...

```

4. The **styles** object must have the new **baseNumber**, **button**, and **buttonText** styles, which we can add at the bottom of the file:


```
});
```

5. However, both buttons will now have the same white background.

We can change this by adding additional styling to them. The **style** prop on React Native components can also take an array of styling objects instead of just a single object:

```
// ...
return (
  <View style={styles.container}>
    <Text style={styles.baseNumber}>
      Starting: {baseNumber}</Text>
    <TouchableHighlight
      onPress={() => setChoice('higher')}
-      style={styles.button}
+      style={[styles.button,
styles.buttonGreen]}
    >
      <Text style=
{styles.buttonText}>Higher</Text>
    </TouchableHighlight>
    <TouchableHighlight
      onPress={() => setChoice('lower')}
-      style={styles.button}
+      style={[styles.button,
styles.buttonRed]}
  </View>
);
```

```

        >
        <Text style=
{styles.buttonText}>Lower</Text>
    </TouchableHighlight>
</View>
);
// ...

```

6. These **buttonGreen** and **buttonRed** objects must also be added to the styling object:

```

// ...
const styles = StyleSheet.create({
    // ...
+   buttonRed: {
+       backgroundColor: 'red',
+   },
+   buttonGreen: {
+       backgroundColor: 'green',
+   },
    buttonText: {
        color: 'white',
        fontSize: 24,
    },
});

```

With these additions, the application is now styled, which makes it more appealing to play. We've used the **StyleSheet** object from React Native to apply this styling, making your application look like this:

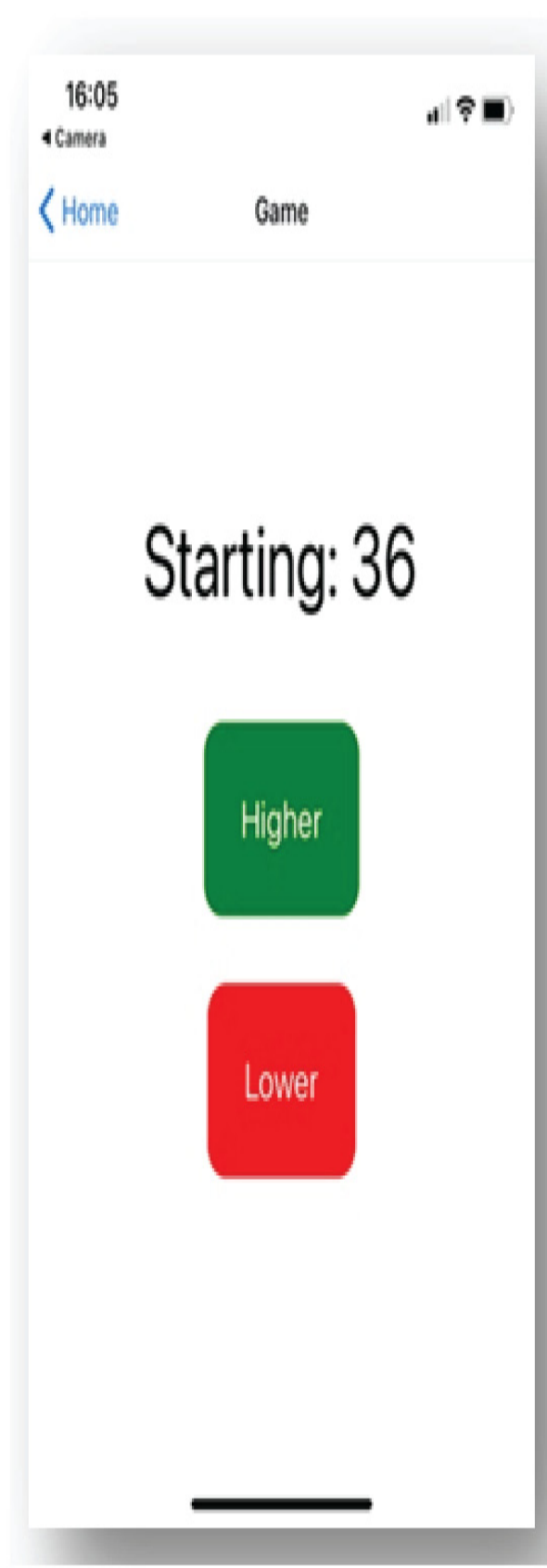


Figure 8.4 – The styled React Native application

Mobile games often have flashy animations that make the user want to keep playing and make the game more interactive. The *Higher/Lower* game that is already functioning uses no animations so far and just has some transitions that have been built in with React Navigation. In the next section, you'll be adding animations and gestures to the application, which will improve the game interface and make the user feel more comfortable while playing the game.

Adding gestures and animations in React Native

There are multiple ways to use animations in React Native, and one of those is to use the Animated API, which can be found at the core of React Native. With the Animated API, you can create animations for the **View**, **Text**, **Image**, and **ScrollView** components from React Native by default. Alternatively, you can use the `createAnimatedComponent` method to create your own.

Creating a basic animation

One of the simplest animations you can add is fading an element in or out by changing the value for the opacity of that element. In the *Higher/Lower* game you created previously, the buttons were styled.

These colors already show a small transition, since you're using the **TouchableHighlight** element to create the button. However, it's possible to add a custom transition to this by using the Animated API. To add an animation, the following code blocks must be changed:

1. Start by creating a new directory called **components**, which will hold all our reusable **components**. In this directory, create a file called **AnimatedButton.js**, which will contain the following code to construct the new component:

```
import React from 'react';
import { StyleSheet, Text,
TouchableHighlight }
  from 'react-native';
export default function AnimatedButton({
action,
  onPress }) {
  return (
    <TouchableHighlight
      onPress={onPress}
      style={[
        styles.button,
        action === 'higher' ?
styles.buttonGreen :
        styles.buttonRed,
      ]}
```



```
        >
        <Text style={styles.buttonText}>
{action}</Text>
    </TouchableHighlight>
    );
}
```

2. Add the following styling to the bottom of this file:

```
// ...
const styles = StyleSheet.create({
  button: {
    display: 'flex',
    alignItems: 'center',
    justifyContent: 'space-around',
    borderRadius: 15,
    padding: 30,
    marginVertical: 15,
  },
  buttonRed: {
    backgroundColor: 'red',
  },
  buttonGreen: {
    backgroundColor: 'green',
  },
  buttonText: {
```

```
        color: 'white',
        fontSize: 24,
        textTransform: 'capitalize',
      },
    });
```

3. As you can see, this component is comparable to the buttons we have in `screens/Game.js`. Therefore, we can remove the **TouchableHighlight** buttons in that file and replace them with the **AnimatedButton** component. Make sure to pass the correct values for **action** and **onPress** as a prop to this component:

```
import React, { useEffect, useState }
from 'react';
import {
  StyleSheet,
  Text,
  View,
  Alert,
  - TouchableHighlight,
  } from 'react-native';
import { useNavigation } from
  '@react-navigation/native';
+ import AnimatedButton from
  '../components/AnimatedButton';
export default function Game() {
```

```

    // ...
    return (
      <View style={styles.container}>
        <Text style={styles.baseNumber}>
          Starting: {baseNumber}</Text>
-      <TouchableHighlight onPress={() =>
          setChoice('higher')} style=
{[styles.button,
          styles.buttonGreen]}>
-      <Text style=
{styles.buttonText}>Higher
          </Text>
-      </TouchableHighlight>
-      <TouchableHighlight onPress={() =>
          setChoice('lower')} style=
{[styles.button,
          styles.buttonRed]}>
-      <Text style=
{styles.buttonText}>Lower</Text>
-      </TouchableHighlight>
+      <AnimatedButton action='higher'
onPress={() =>
          setChoice('higher')} />

```

```

+      <AnimatedButton action='lower'
onPress={() =>
      setChoice('lower')} />
    </View>
  );
}
// ...

```

4. No visible changes are present if you look at the application on your mobile device or the emulator on your computer, since we need to change the clickable element from a **TouchableHighlight** element to a **TouchableWithoutFeedback** element first. That way, the default transition with the highlight will be gone, and we can replace this with our own effect. The **TouchableWithoutFeedback** element can be imported from React Native in **components/Animated-Button.js** and should be placed around a **View** element, which will hold the default styling for the button:

```

import React from 'react';
import {
  StyleSheet,
  Text,
  - TouchableHighlight,
  + TouchableWithoutFeedback,
  + View
} from 'react-native';

```

```

    export default function
AnimatedButton({ action,
    onPress }) {
    return (
-      <TouchableHighlight onPress={onPress}
style={[
    styles.button, action === 'higher'
?
    styles.buttonGreen :
styles.buttonRed ]}>
+      <TouchableWithoutFeedback onPress=
{onPress}>
+      <View style={[ styles.button,
action === 'higher'
? styles.buttonGreen :
styles.buttonRed ]}>
        <Text style={styles.buttonText}>
{action}</Text>
-      </TouchableHighlight>
+      </View>
+      </TouchableWithoutFeedback>
    );
  }
  // ...

```

5. To create a transition when we click on the button, we can use the **Animated** API. We'll use this to change the opacity of the **AnimatedButton** component from the moment it's pressed. A new instance of the **Animated** API starts by specifying a value that should be changed during the animation that we created with the **Animated** API. This value should be changeable by the **Animated** API in your entire component, so you can add this value to the top of the component. This value should be created with a **useRef** Hook, since you want this value to be changeable later on. Also, we need to import **Animated** from **React Native**:

```
- import React from 'react';
+ import React, { useRef } from 'react';
import {
  StyleSheet,
  Text,
  TouchableWithoutFeedback,
-   View,
+   Animated,
  } from 'react-native';
export default function AnimatedButton({
  action,
  onPress }) {
+   const opacity = useRef(new
    Animated.Value(1));
```

```
return (  
  // ...
```

6. This value can now be changed by the Animated API using any of the three animations types that are built in. These are **decay**, **spring**, and **timing**, where you'll be using the **timing** method from the Animated API to change the animated value within a specified time frame. The Animated API can be triggered from the **onPress** event on **TouchableWithoutFeedback** and calls the **onPress** prop after finishing the animation:

```
// ...  
export default function AnimatedButton({  
  action,  
  onPress }) {  
  const opacity = useRef(new  
    Animated.Value(1));  
  
  return (  
    <TouchableWithoutFeedback  
-      onPress={onPress}  
+      onPress={() => {  
+        Animated.timing(opacity.current,  
+        {  
+          toValue: 0.2,
```

```
+         duration: 800,  
+         useNativeDriver: true,  
+     }).start(() => onPress());  
+     }}  
  
    >  
  
    // ...
```

The **timing** method takes the **opacity** that you've specified at the top of your component and an object with the configuration for the Animated API. We need to take the current value of the opacity, as this is a **ref** value. One of the fields is **toValue**, which will become the value for **opacity** when the animation has ended. The other field is for the field's duration, which specifies how long the animation should last.

NOTE

*The other built-in animation types next to **timing** are **decay** and **spring**. Whereas the **timing** method changes gradually over time, the **decay** type has animations that change fast in the beginning and gradually slow down until the end of the animation. With **spring**, you can create animations that move a little outside of their edges at the end of the animation.*

7. The **View** component can be replaced by an **Animated.View** component. This component uses the **opacity** variable created by the

useRef Hook to set its opacity:

```
        // ...  
-      <View  
+      <Animated.View  
        style={ [  
          styles.button,  
          action === 'higher' ?  
styles.buttonGreen :  
          styles.buttonRed,  
+      { opacity: opacity.current },  
        ]}  
      >  
        <Text style={styles.buttonText}>  
{action}  
        </Text>  
-    </View>  
+    </Animated.View>  
    </TouchableWithoutFeedback>  
  );  
}  
// ...
```

Now, when you press any of the buttons on the **Game** screen, they will fade out, since the opacity transitions from **1** to **0.2** in 400 milliseconds.

Something else you can do to make the animation appear smoother is to add an **easing** field to the **Animated** object. The value for this field comes from the **Easing** module, which can be imported from React Native. The **Easing** module has three standard functions: **linear**, **quad**, and **cubic**. Here, the **linear** function can be used for smoother timing animations:

```
import React, { useRef } from 'react';
import {
  StyleSheet,
  Text,
  TouchableWithoutFeedback,
  Animated,
+  Easing,
  } from 'react-native';
export default function AnimatedButton({
  action, onPress }) {
  const opacity = useRef(new
  Animated.Value(1));
  return (
    <TouchableWithoutFeedback
      onPress={() => {
        Animated.timing(opacity.current, {
          toValue: 0.2,
          duration: 400,
```

```
        useNativeDriver: true,  
+        easing: Easing.linear(),  
        }).start(() => onPress());  
    }}  
    >  
    // ...
```

With this last change, the animation is complete, and the game interface already feels smoother, since the buttons are being highlighted using our own custom animation. In the next part of this section, we will combine some of these animations to make the user experience for this game even more advanced.

NOTE

*You can also combine animations – for example, with the **parallel** method – from the Animated API. This method will start the animations that are specified within the same moment and take an array of animations as its value. Next to the **parallel** function, three other functions help you with animation composition. These functions are **delay**, **sequence**, and **stagger**, which can also be used in combination with each other. The **delay** function starts any animation after a pre-defined delay, the **sequence** function starts animations in the order you've specified and waits until an animation is resolved before starting another one, and the **stagger** function can start animations both in order and parallel with specified delays in between.*

Handling gestures with Expo

Gestures are an important feature of mobile applications, as they make the difference between a mediocre and a good mobile application. In the *Higher/Lower* game you've created, several gestures can be added to make the game more appealing.

Previously, you used the **TouchableHighlight** element, which gives the user feedback after they press it by changing it. Another element that you could have used for this was the **TouchableOpacity** element. These gestures give the user an impression of what happens when they make decisions within your application, leading to improved user experience. These gestures can be customized and added to other elements as well, making it possible to have custom touchable elements as well.

For this, you can use a package called **react-native-gesture-handler**, which helps you access native gestures on every platform. All of these gestures will be run in the native thread, which means you can add complex gesture logic without having to deal with the performance limitations of React Native's gesture responder system. Some of the gestures it supports include *tap*, *rotate*, *drag*, and *pan*, and a *long press*. In the previous section, we installed this package, as it's a requirement for **react-navigation**.

NOTE

*You can also use gestures directly from React Native, without having to use an additional package. However, the gesture responder system that React Native currently uses doesn't run in the native thread. Not only does this limit the possibilities of creating and customizing gestures, but you can also run into cross-platform or performance problems. Therefore, it's advised that you use the **react-native-gesture-handler** package, but this isn't necessary for using gestures in React Native.*

The gesture we will implement is a *long press* gesture, which will be added to the start button in our **Home** screen, located at **screens/Home.js**. Here, we'll use the **TapGestureHandler** element from **react-native-gesture-handler**, which runs in the native thread, instead of the **TouchableWithoutFeedback** element from React Native, which uses the gesture responder system. To implement this, we need to do the following this becomes number 2 please make sure the rest of the numbers are updated accordingly:

1. Install using Expo:

```
expo install react-native-gesture-handler
```

2. Import **TapGestureHandler** and **State** from **react-native-gesture-handler**, next to **View** and **Alert** from React Native. The

TouchableHighlight import can be removed, as this will be replaced:

```
import React from 'react';
import {
  StyleSheet,
  Text, View,
+  Alert,
-  TouchableHighlight,
} from 'react-native';
import { useNavigation } from
  '@react-navigation/native';
+ import { TapGestureHandler, State } from
  'react-native-gesture-handler';
export default function Home() {
  // ...
```

3. We can replace the **TouchableHighlight** component with **TapGestureHandler**, and we need to put a **View** component inside it, to which we apply the styling. **TapGestureHandler** doesn't take an **onPress** prop but an **onHandlerStateChange** prop instead, to which we pass the new **on Tap** function. In this function, we need to check whether the state of the tap event is active. For this, you need to know that the tap event goes through different states: **UNDETERMINED**, **FAILED**, **BEGAN**, **CANCELLED**, **ACTIVE**, and **END**. The naming of these states is pretty straightforward, and usually, the

handler will have the following flow: **UNDETERMINED > BEGAN > ACTIVE > END > UNDETERMINED**:

```
// ...
export default function Home() {
  const navigation = useNavigation();
+   function onTap(e) {
+     if (e.nativeEvent.state ===
State.ACTIVE) {
+       Alert.alert('Long press to start
the game');
+     }
+   }
  return (
    <View style={styles.container}>
-     <TouchableHighlight
-       onPress={() =>
navigation.navigate('Game')}
-       style={styles.button}
-     >
+     <TapGestureHandler
onHandlerStateChange={onTap}>
+       <View style={styles.button}> <Text
style={styles.buttonText}>Start
game!</Text>
```

```

+      </View>
-      </TouchableHighlight>
+      </TapGestureHandler>
      </View>
    );
  }
  // ...

```

4. If you now press the start button on the **Home** screen, you will receive the message that you need to long press the button to start the game. To add this long press gesture, we need to add a **LongPressGestureHandler** component inside the **TapGestureHandler** component. Also, we need to create a function that can be called by the **LongPressGestureHandler** component, which navigates us to the **Game** screen:

```

import React from 'react';
import { StyleSheet, Text, View, Alert }
  from 'react-native';
import { useNavigation }
  from '@react-navigation/native';
import {
+  LongPressGestureHandler,
    TapGestureHandler,
    State,
  } from 'react-native-gesture-handler';

```



```

    export default function Home() {
      const navigation = useNavigation();
+   function onLongPress(e) {
+     if (e.nativeEvent.state ===
State.ACTIVE) {
+       navigation.navigate('Game');
+     }
+   }
    // ...

```

5. Inside the **TapGestureHandler** the newly imported **LongPressGestureHandler** component should be placed. This component takes the function to navigate to the game, and a prop to set the minimal duration of the long press. If you don't set this prop, the minimal duration will be 500ms by default::

```

    // ...
    export default function Home() {
      // ...
      return (
        <View style={styles.container}>
          <TapGestureHandler
            onHandlerStateChange=
{onSingleTap}
          >
+       <LongPressGestureHandler+

```

```

        onHandlerStateChange=
        {onLongPress}
+        minDurationMs={600}
+        >
        <View style={styles.button}>
            <Text style=
            {styles.buttonText}>
                Start game!</Text>
            </View>
+        </LongPressGestureHandler>
        </TapGestureHandler>
    </View>
    );
}
// ...

```

With this latest change, you can only start the game by long pressing the **start** button on the **Home** screen. These gestures can be customized even more, since you can use composition to have multiple tap events that respond to each other. By creating so-called **cross-handler interactions**, you can create a touchable element that supports a *double-tap* gesture and a *long-press* gesture.

The next section will show you how to handle even more advanced animations, such as displaying animated graphics when any of two

players win. For this, we'll use the Lottie package, since it supports more functionalities than the built-in Animated API.

Advanced animations with Lottie

The React Native Animated API is great for building simple animations, but building more advanced animations can be harder. Luckily, Lottie offers a solution for creating advanced animations in React Native by making it possible for us to render After Effects animations in real time for iOS, Android, and React Native.

NOTE

*When using Lottie, you don't have to create these After Effects animations yourself; there's a whole library full of resources that you can customize and use in your project. This library is called **LottieFiles** and is available at <https://lottiefiles.com/>.*

Since we've already added animations to the buttons of our game, a nice place to add more advanced animations would be the message that is displayed when you win or lose the game. This message can be displayed on a screen instead of an alert, where a trophy can be displayed if the user won. Let's do this now:

1. To get started with Lottie, run the following command, which will install Lottie to our project:

```
yarn add lottie-react-native
```

2. After the installation is completed, we can create a new screen component called `screens/Result.js` with the following content:

```
import React from 'react';
import { StyleSheet, Text, View } from
'react-native';
export default function Result() {
  return (
    <View style={styles.container}>
      <Text></Text>
    </View>
  );
}
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

3. Add this screen to the stack navigator so that it can be used in the navigation for this mobile application by importing it in `App.js`. Also, the navigation element `HeaderBackButton` should be imported:

```

import { StatusBar } from 'expo-status-
bar';
import React from 'react';
import { StyleSheet } from 'react-
native';
import { NavigationContainer }
  from '@react-navigation/native';
import { createNativeStackNavigator }
  from '@react-navigation/native-stack';
+ import { HeaderBackButton }
  from '@react-navigation/elements';
import Home from './screens/Home';
import Game from './screens/Game';
+ import Result from './screens/Result';
// ...

```

4. We also imported the **HeaderBackButton** component from React Navigation when adding the **Result** screen, as we also want to change the **go back** button in the header for this screen. This way, it will navigate back to the **Home** screen instead of the **Game** screen so that the user can start a new game after finishing it:

```

// ...
export default function App() {
  return (
    <NavigationContainer>

```

```

        <StatusBar style='auto' />
        <Stack.Navigator
initialRouteName='Home'>
            <Stack.Screen name='Home'
component={Home} />
            <Stack.Screen name='Game'
component={Game} />
+         <Stack.Screen
+             name='Result'
+             component={Result}
+             options={({ navigation }) => ({
+                 headerLeft: (props) => (
+                     <HeaderBackButton
+                         {...props}
+                         label='Home'
+                         onPress={() =>
+                             navigation.navigate('Ho
me')} )}
+                 />
+             )},
+             )})
+         />
    </Stack.Navigator>
</NavigationContainer>

```

```
);  
// ...
```

5. From the **Game** screen in `screens/Game.js`, we can navigate the user to the **Result** screen after playing the game and also pass a param to this screen. Using this param, a message can be displayed with the result of the game:

```
// ...  
export default function Game() {  
  // ...  
  useEffect(() => {  
    if (choice.length) {  
      const winner = (choice === 'higher'  
&& score >  
        baseNumber) || (choice ===  
'lower' &&  
        baseNumber > score);  
      -      Alert.alert(`You've ${winner ?  
'won' :  
        'lost'}`, `You scored:  
        ${score}`);  
      -      navigation.goBack();  
      +      navigation.navigate('Result', {  
winner })  
    }  
  }  
}
```

```
    }, [baseNumber, score, choice]);  
    return (  
      // ...
```

6. From the **Result** screen in the `screens/Result.js` file, we can import **LottieView** from **lottie-react-native** and get the param from the `route` object using the `useRoute` Hook from React Navigation. Using this param, we can return a message if the user has won or lost:

```
import React from 'react';  
import { StyleSheet, Text, View } from  
  'react-native';  
+ import LottieView from 'lottie-react-  
  native';  
+ import { useRoute } from '@react-  
  navigation/native';  
export default function Result() {  
+   const route = useRoute();  
+   const { winner } = route.params;  
  return (  
    <View style={styles.container}>  
+    <Text>You've {winner ? 'won' :  
'lost'}</Text>  
    // ...
```


7. The imported **Lottie** component can render any Lottie file that you either create yourself or that is downloaded from the **LottieFiles** library. In the GitHub repository for this chapter, you will find a Lottie file that can be used in this project called **winner.json**. This file must be placed in the **assets** directory and can be rendered by the **LottieView** component when you add it to the source, and the **width** and **height** values of the animation can be set by passing a **style** object. Also, you should add the **autoPlay** prop to start the animation once the component renders:

```
// ...
export default function Result() {
  const route = useRoute();
  const { winner } = route.params;
  return (
    <View style={styles.container}>
      <Text>You've {winner ? 'won' :
+       {winner && (
+         <LottieView
+           autoPlay
+           style={{
+             width: 300,
+             height: 300,
+             }}
+       )}
      </Text>
```

```

+         source=
{require('../assets/winner.json')}
+         />
+     )}
    </View>
  );
}
// ...

```

8. As a finishing touch, we can add some styling to the message that is displayed on this screen and make it bigger:

```

// ...
return (
  <View style={styles.container}>
-    <Text>You've {winner ? 'won' :
'lost'}</Text>
+    <Text style={styles.message}>
      You've {winner ? 'won' : 'lost'}
</Text>
    // ...
    const styles = StyleSheet.create({
      // ...
+    message: {
+      fontSize: 48,
+    },

```

```
} );
```

When the **Result** screen component receives the **winner** param with the **true** value, instead of the board, the user will see the trophy animation being rendered. An example of how this will look when you're running the application with the iOS simulator or on an iOS device can be seen here:



Figure 8.5 – The Lottie animation after winning a game

NOTE

*If you find the speed of this animation too fast, you can reduce it by combining the Animated API with Lottie. The **LottieView** component can take a **progress** prop that determines the speed of the animation. When passing a value that is created by the Animated API, you can tweak the speed of the animation as per your preference.*

By adding this animation using Lottie, we've created a mobile application with an animated game that you can play for hours.

Summary

In this chapter, we've created a React Native application with Expo. React Native uses the same principles as React and can be used to create mobile applications. We've added basic routing with React Navigation, based on stack navigation. We've also added basic and more complex gestures to the game, which run in the native thread thanks to the **react-native-gesture-handler** package. Finally, animations were created using the React Native Animated API and Lottie, which is available from the Expo CLI.

The project that we'll create in the next chapter will explore handling data in React Native. We'll also learn about the differences in styling between iOS and Android.

Further reading

- Expo: <https://docs.expo.io/>
- Various Lottie files: <https://lottiefiles.com/>
- More on the Animated API: <https://facebook.github.io/react-native/docs/animated>
- Gesture Handler: <https://docs.swmansion.com/react-native-gesture-handler/>

Chapter 9: Building a Full-Stack Social Media Application with React Native and Expo

Most of the projects that you've created in this book focused on displaying data and making it possible to navigate between pages.

When we created our first mobile application with React Native, animations were one of the focus points, which is a must-have when creating a mobile application. In this chapter, we'll be exploring a big advantage of mobile applications, namely the ability to use the camera (or camera roll) from the phone.

The application we'll be creating in this chapter will follow the same patterns for data-heavy applications as in previous chapters. React techniques such as Context and Hooks are used to get data from a local API that also supports authentication, while React Navigation is used again to create a more advanced routing setup. Also, Expo is used to post images to a social feed by using the camera of the mobile device the application is running on.

The following topics will be covered in this chapter:

- Advanced routing with authentication
- Using the camera with React Native and Expo

- Differences in styling for iOS and Android

Project overview

In this chapter, we will build a mobile social media application that is using a local API to request and add posts to the social feed, including using the camera on the mobile device. Advanced routing with authentication is added using the local API and React Navigation, while Expo is used for access to the camera (roll).

The build time is 2 hours.

NOTE

This chapter is using React Native version 0.64.3 and Expo SDK version 44. As React Native and Expo are updated frequently, make sure that you're working with these versions to ensure the patterns described in this chapter are behaving as expected.

Getting started

The project that we'll create in this chapter builds upon an initial version that you can find on GitHub:

<https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter09-initial>. The complete source code can

also be found on GitHub: <https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter09>.

You need to have the Expo Go application installed on a mobile iOS or Android device to run the project on a physical device. Once you've downloaded the application, you need to create an Expo account to make the development process smoother. Make sure to store your account details somewhere safe, as you need these later on in this chapter.

Alternatively, you can install either Xcode or Android Studio on your computer to run the application on a virtual device:

- **For iOS:** Information on how to set up your local machine to run the iOS simulator can be found here: <https://docs.expo.io/workflow/ios-simulator/>.
- **For Android:** Information on how to set up your local machine to run the emulator from Android Studio can be found here: <https://docs.expo.io/workflow/android-studio-emulator/>.

NOTE

*It's **highly recommended** to use the Expo client application to run the project from this chapter on a physical device. Receiving notifications is currently only supported on physical devices, and running the project on either the iOS simulator or Android Studio emulator will result in error messages.*

Checking out the initial project

For this chapter, an initial application has been created with Expo using their CLI, as you learned in the previous chapter. To get started, you'll need to run the following command in this chapter's directory to install all of the dependencies and start both the server and application:

```
yarn && yarn start
```

This command will start Expo after installing the dependencies, and it gives you the ability to start your project from either the terminal or your browser. In the terminal, you can now either use the QR code to open the application on your mobile device or open the application in a simulator. In the browser, the Expo DevTools will be opened, which also lets you scan the QR code with your phone using the camera or the Expo Go application.

The local API from which to get the data for our application was created using JSON Server. We've already used this library before, as we used the **My JSON Server** endpoint based on the `db.json` file in this repository. For this project, we have a separate `db.json` file in the directory for this chapter, which is loaded by the `server.js` file to create a local API. The local API can be started by running the following command in a separate terminal tab or window:

```
yarn start-server
```

This spins up a server at `http://localhost:3000/api/` with, for example, the `http://localhost:3000/api/posts` endpoint, which returns an array of posts. However, when building mobile applications, you cannot use a `localhost` address (or any other address without HTTPS) for security reasons. To be able to use this endpoint in the React Native application, you need to find the local IP address of your machine.

To find your local IP address, you'll need to do the following depending on your operating system:

- **For Windows:** Open the terminal (or Command Prompt) and run this command:

```
Ipconfig
```

This will return a list like the one you see in the following screenshot with data from your local machine. In this list, you need to look for the **IPv4 Address** field:

```
Windows IP Configuration

Ethernet adapter Ethernet0:

    Connection-specific DNS Suffix  . : 
    IPv6 Address. . . . . : 2a02:2f01:5060:9cb:3499:3b63:e8ab:5967
    Temporary IPv6 Address. . . . . : 2a02:2f01:5060:9cb:e9d9:4dbb:4ddc:a934
    Link-local IPv6 Address . . . . . : fe80::3499:3b63:e8ab:5967%4
    IPv4 Address. . . . . : 192.168.1.107
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : fe80::1eb7:2cff:fe74:fef8%4
                                192.168.1.1

Tunnel adapter Teredo Tunneling Pseudo-Interface:

    Connection-specific DNS Suffix  . : 
    IPv6 Address. . . . . : 2001:0:9d38:90d7:285a:3873:3f57:fe94
    Link-local IPv6 Address . . . . . : fe80::285a:3873:3f57:fe94%13
    Default Gateway . . . . . :
```

Figure 9.1 – Finding a local IP address in Windows

- **For macOS:** Open the Terminal and run this command:

```
ipconfig getifaddr en0
```

After running this command, the local IPv4 address of your machine gets returned, which looks like this:

```
192.168.1.107
```

The local IP address can be used as an alternative for `localhost`, which you can try by visiting the following page:

`http://192.168.1.107/api/posts`. Make sure to replace the IP address with your own.

Our application for this chapter has already been set up and needs to know what URL to use for the local API. Configuration in Expo can be

stored in **app.json**, but also in **app.config.js** if you want to store specific configuration environment variables. In this file, you can add the following configuration:

```
export default {
  extra: {
    apiUrl: 'http://LOCAL_IP_ADDRESS:3000',
  },
};
```

In the preceding **app.config.js** file, you need to replace **LOCAL_IP_ADDRESS** with your own IP address that you acquired from your machine.

To use this environment variable in our code, we use the **expo-constants** library. This has already been installed in the initial application for this chapter, and an example of how to get **apiUrl** from **app.config.js** can be seen in the **context/PostsContext.js** file:

```
import React from 'react';
import { createContext, useReducer } from
'react';
import Constants from 'expo-constants';
const { apiUrl } = Constants.manifest.extra;
export const PostsContext = createContext();
// ...
```

The `apiUrl` constant is now used to fetch the following local API. No matter whether you've opened the application from a virtual or physical device, the initial application at this point should look something like this:

14:58



Posts



First post



14:59



[← Posts](#)

PostDetail



First post

Figure 9.2 – The initial application

The **screens** directory for the initial application consists of five screens, which are **Posts**, **PostDetail**, **PostForm**, **Profile**, and **Login**. The **Posts** screen will be the initial screen that is loaded and shows a list of posts on which you can tap to continue to the **PostDetail** screen. For now, the **PostForm**, **Profile**, and **Login** screens aren't visible yet, as we'll add advanced routing and authentication later on in this chapter.

The project structure from this React Native app is as follows, where the structure is similar to the projects you've created before in this book:

```
chapter-9-initial
|- /.expo
|- /.expo-shared
|- /node_modules
|- /assets
|- /components
    |- Button.js
    |- FormItem.js
    |- PostItem.js
|- /context
    |- AppContext.js
```



```
    |- PostsContext.js
    |- UserContext.js
|- /screens
    |- Login.js
    |- PostDetail.js
    |- PostForm.js
    |- Posts.js
    |- Profile.js
app.config.js
app.json
App.js
babel.config.js
db.json
server.js
```

In the **assets** directory, you can find the images that are used as the application icon on the home screen once you've installed this application on your mobile device, and the image that will serve as the splash screen, which is displayed when you start the application. The **App.js** file is the actual entry point of your application and all of the components for this application are located in the **screens** and **components** directories. You can also find a directory called **context**. This directory has all the state management components for this application.

NOTE

*If you get an error when loading the application on your local device or emulator stating **Network request failed**, make sure that you've added your local IP address in `app.config.js`. Also, the server must be running in a separate terminal tab.*

Configurations for your application, for example, the App Store, are placed in `app.json`, while `babel.config.js` holds specific Babel configurations. As mentioned earlier, the `app.config.js` file holds the configuration for the URL to the local API. There are also two files that are needed to create the local API. These are `db.json` and `server.js`, as described previously in this section.

Building a full-stack social media application with React Native and Expo

The application that you're going to build in this chapter will use a local API to retrieve and mutate data that is available in the application. This application will display data from a social media feed, lets you add new posts containing images, and allows you to respond to these social media posts.

Advanced routing with authentication

We've already learned how to add routing to a React Native application using React Navigation. The routing we added was using a stack navigator, which doesn't have a way to display some sort of menu or navigation bar with all the routes. In this section, we'll be adding a tab navigator using React Navigation to display a tab bar at the bottom of the application. Later on, we'll also be adding an authentication flow.

Adding bottom tabs

Bottom tabs are common on applications for iOS, but less popular on Android applications. In the final section of this chapter, we'll learn more about the styling differences between iOS and Android. But first, we'll focus on adding bottom tabs to our application.

To add a tab navigator, we need to complete the following actions:

1. React Navigation has a separate library to create a tab navigator, which we need to install from npm:

```
yarn add @react-navigation/bottom-tabs
```

When the installation of `@react-navigation/bottom-tabs` is complete, make sure to restart Expo using the `npm start` command.

2. In the `App.js` file, all the routes for this application are listed, and we need to import the method to create a tab:

```

import { StatusBar } from 'expo-status-
bar';
import React from 'react';
import { NavigationContainer } from
  '@react-navigation/native';
import { createStackNavigator } from
  '@react-navigation/stack';
+ import { createBottomTabNavigator } from
  '@react-navigation/bottom-tabs';
// ...

```

3. The tab navigator can be created using the **createBottomTabNavigator** method. These screens for the navigator must be created in a separate component inside the **App.js** file, where the **Posts**, **PostForm**, and **Profile** screens will be added to it. These screens will later become available in the bottom tabs. It's important to pass the option to not show the header, as the title of the screen will be rendered by the parent navigator:

```

// ...
+ const Tab = createBottomTabNavigator();
+ function Home() {
+   return (
+     <Tab.Navigator>
+       <Stack.Screen
+         name= 'Posts '

```

```

+         component={Posts}
+         options={{ headerShown: false }}
+     />
+     <Stack.Screen
+         name='Profile'
+         component={Profile}
+         options={{ headerShown: false }}
+     />
+     <Stack.Screen
+         name='PostForm'
+         component={PostForm}
+         options={{ headerShown: false }}
+     />
+ </Tab.Navigator>
+ );
+ }
export default function App() {
    // ...

```

4. To render the navigator in the application, we need to add it to the **return** statement inside the **App** component:

```

export default function App() {
    return (
        <AppContext>
            <NavigationContainer>

```

```

        <StatusBar style='auto' />
-        <Stack.Navigator
initialRouteName='Posts'>
-            <Stack.Screen name='Posts'
                component={Posts} />
-            <Stack.Screen name='Profile'
                component={Profile} />
-            <Stack.Screen name='PostForm'
                component={PostForm} />
+        <Stack.Navigator
initialRouteName='Home'>
+            <Stack.Screen name='Home'
component={Home} />
                <Stack.Screen name='PostDetail'
                    component={PostDetail} />
                <Stack.Screen name='Login'
                    component={Login} />
            </Stack.Navigator>
        </NavigationContainer>
    </AppContext>
  );
}

```

5. When you now navigate to any of the screens using the tab navigator, you see that the title in the header is always **Home**. Be-

cause the nested **Home** component is rendered, that on its own end renders the different screens. We can force the header title to be that of the tab that is active by using **getFocusedRouteNameFromRoute** from React Navigation in the **options** prop for the home screen:

```
import { StatusBar } from 'expo-status-
bar';
import React from 'react';
- import { NavigationContainer } from
  '@react-navigation/native';
+ import { NavigationContainer,
    getFocusedRouteNameFromRoute }
    from '@react-navigation/native';

// ...

export default function App() {
  return (
    <AppContext>
      <NavigationContainer>
        <StatusBar style='auto' />
        <Stack.Navigator>
          <Stack.Screen
            name='Home'
```

```

        component={Home}
+      options={({ route }) => ({
+        headerTitle:
            getFocusedRouteNameFromRoute(route),
+      })}
    />
    <Stack.Screen name='PostDetail'
      component={PostDetail} />
    <Stack.Screen name='Login'
      component={Login} />
  </Stack.Navigator>
</NavigationContainer>
</AppContext>
  );
}

```

6. The bottom tabs can also have an icon and a custom color when active. For this, we can alter the **screenOptions** and of the tab navigator. The icons for the tab can be imported from **@expo/vector-icons**, which is already included in Expo:

```

import { StatusBar } from 'expo-status-bar';
+ import { FontAwesome } from
  '@expo/vector-icons';

```



```

import React from 'react';

// ...

function Home() {
  return (
    <Tab.Navigator
+      screenOptions={({ route }) => ({
+        tabBarActiveTintColor: 'blue',
+        tabBarInactiveTintColor: 'gray',
+        tabBarIcon: ({ color, size }) =>
+        {
+          const iconName =
+            (route.name === 'Posts' &&
+ 'feed') ||
+            (route.name === 'PostForm' &&
+ 'plus-square') ||
+            (route.name === 'Profile' &&
+ 'user');
+          return <FontAwesome name=
+ {iconName}
+           size={size} color={color}
+ />;
+        },
+      )})}

```

```

        >
        // ...
    </Tab.Navigator>
  );
}
// ...

```

7. Finally, we can also change the labels of the tabs, for example, for the **PostForm** screen that displays the form to add a new post:

```

// ...
function Home() {
  return (
    <Tab.Navigator
      // ...
    >
      <Stack.Screen
        name='PostForm'
        component={PostForm}
        options={{
          headerShown: false,
+         tabBarLabel: 'Add post',
          }}
      />
      <Stack.Screen name='Profile'
        component={Profile}

```

```
        />  
      </Tab.Navigator>  
    );  
  }  
  // ...
```

With these changes, the application now has routing with both a stack navigator and a tab navigator, and should look something like this:



Figure 9.3 – The application with bottom tabs

We're now able to reach almost all the screens, with only the **Login** screen still hidden. This screen is added to the stack navigator and should be displayed when the user isn't authenticated. In the next part of this section, we'll add the authentication flow to handle this.

Authentication flow

For authentication in frontend applications, most of the time, **JSON Web Tokens (JWTs)** are used, which are encrypted tokens that can easily be used to share user information with a backend. The JWT will be returned by the backend when the user is successfully authenticated and often, this token will have an expiration date. With every request that the user should be authenticated for, the token should be sent so that the backend server can determine whether the user is authenticated and allowed to take this action. Although JWTs can be used for authentication since they're encrypted, no private information should be added to them since the tokens should only be used to authenticate the user. Private information can only be sent from the server when a document with the correct JWT has been sent.

The mobile application we're building in this chapter is only using **GET** requests to retrieve posts, but the local API also supports **POST** requests. But to send **POST** requests, we need to be authenticated, meaning we need to retrieve a token that we can send along with our

request to the API. For this, we can use the **api/login** endpoint of the API:

1. The **Login** component can be used to log in but isn't displayed at the moment. To display this component, we need to change the logic in the stack navigator in **App.js**. Instead of having the **App** component return the stack navigator, we need to create a new component in this file called **Navigator**:

```
// ...  
+ function Navigator() {  
+   return (  
+     <NavigationContainer>  
+       <StatusBar style='auto' />  
+       <Stack.Navigator>  
+         <Stack.Screen name='Login'  
component={Login}  
          />  
+         <Stack.Screen  
+           name='Home '  
+           component={Home}  
+           options={{({ route }) => ({  
+             headerTitle:  
getFocusedRouteNameFromRoute(route),  
+           })}}  
+       </Stack.Navigator>  
+     </NavigationContainer>  
+   );  
+ }
```

```

+         />
+         <Stack.Screen name='PostDetail'
+             component={PostDetail} />
+     </Stack.Navigator>
+ </NavigationContainer>
+ );
+ }
  export default function App() {
    // ...

```

2. The preceding code block can be deleted from **App** and replaced by this new **Navigator** component:

```

  // ...
  export default function App() {
    return (
      <AppContext>
-      // ...
+      <Navigator />
      </AppContext>
    );
  }

```

3. We also need to check the value for the token in the **Navigator** component, as we don't want to include the home screen when there is no token provided. The logic to log in is already present in the **UserContext** in the `context/UserContext.js` file and from the

Navigator component, you can get the **user** object from this context:

```
import { StatusBar } from 'expo-status-bar';

import { FontAwesome } from
 '@expo/vector-icons';
- import React from 'react';
+ import React, { useContext } from
 'react';
  // ...
  import AppContext from
 './context/AppContext';
+ import UserContext from
 './context/UserContext';
  const Stack = createStackNavigator();
  const Tab = createBottomTabNavigator();
  function Home() {
    // ...
```

4. Now, we can get the **user** object from the context and add the logic to return the **Login** screen only when no token is present:

```
  // ...
  function Navigator() {
+   const { user } =
  useContext(UserContext);
```



```

    return (
      <NavigationContainer>
        <StatusBar style='auto' />
-      <Stack.Navigator>
+      <Stack.Navigator initialRouteName=
        {user.token.length ? 'Home' :
'Login'}>
        <Stack.Screen
          name='Home '
          // ...
        />
        <Stack.Screen
name='PostDetail'
          component={PostDetail} />
        <Stack.Screen name='Login'
          component={Login} />
      )}
    </Stack.Navigator>
  </NavigationContainer>
);
}
export default function App() {
  // ...

```

5. If you now refresh the application, you can see the **Login** component being displayed. You can log in with a username and password combination, which is **test** for both values. After logging in, we want to navigate to the home screen, for which we need to make a change in `screens/Login.js`:

```
+ import { useNavigation } from
    '@react-navigation/core';
+ import React, { useContext, useState }
from 'react';
- import React, { useContext, useEffect,
useState }
    from 'react';
// ...
export default function Login() {
    const [username, setUsername] =
useState('');
    const [password, setPassword] =
useState('');
-    const { error, loginUser } =
        useContext(UserContext);
+    const { user, error, loginUser } =
        useContext(UserContext);
+    const navigation = useNavigation();
+    useEffect(() => {
```

```
+     if (user.token) {  
+         navigation.navigate( 'Home' );  
+     }  
+ }, [user.token]);  
    return (  
        // ...
```

When the value for **token** in the **user** object in the context changes, the user will now be navigated to the home screen. This can be shown by logging in with a username and password combination, which is **test** for both values. If you put in an incorrect value, you'll see an error message, as visible here:

16:10

Login

Your username

Your password

Login

16:10

Login

Something went wrong

test

wrong

Login

Figure 9.4 – Handling authentication

The token, however, isn't persisted as the context gets restored when you reload the application. For web applications, we could have used `localStorage` or `sessionStorage`. But for mobile applications, you'd need to use the `AsyncStorage` library from React Native to have persistent storage on both iOS and Android. On iOS, it will use native code blocks to give you the global persistent storage that `AsyncStorage` offers, while on devices running Android, either RocksDB- or SQLite-based storage will be used.

NOTE

For more complex usages, it's recommended to use an abstraction layer on top of `AsyncStorage` as encryption isn't supported out of the box. Also, the use of a key-value system can give you performance issues if you want to store a lot of information for your application using `AsyncStorage`. Both iOS and Android will have set limitations on the amount of storage each application can use.

To add the persistence of the user token, we need to install the correct library from Expo and make changes to the context:

1. We can install `AsyncStorage` from Expo by running the following command:

```
expo install @react-native-async-storage/async-storage
```

2. To persist, the **AsyncStorage** token can be imported in the **UserContext** in the **context/UserContext.js** file:

```
import React, { createContext, useReducer
}
    from 'react';
+ import AsyncStorage from
    '@react-native-community/async-storage';
import Constants from 'expo-constants';
// ...
```

3. In the same file, it can be used to store the token in **AsyncStorage** after adding it to the context:

```
// ...
export const UserContextProvider = ({
children }) => {
    const [state, dispatch] =
useReducer(reducer,
    initialState);
    async function loginUser(username,
password) {
        try {
            // ...
            if (result) {
```

```

        dispatch({ type:
'SET_USER_TOKEN',

                    payload: result.token

});
+      AsyncStorage.setItem('token',
result.token);
    }
    } catch (e) {
      dispatch({ type: 'SET_USER_ERROR',
                    payload: e.message });
    }
  }
  // ...

```

4. Now that the token is persisted after it's retrieved from the local API, it can also be retrieved from **AsyncStorage**. Therefore, we need to create a new function that retrieves the token and adds it to the context:

```

  // ...

+  async function getToken() {
+    try {
+      const token =
        await
      AsyncStorage.getItem('token');
+      if (token !== null) {

```

```

+      dispatch({ type:
+        'SET_USER_TOKEN',
+          payload: token });
+    }
+  } catch (e) {}
+ }
+   return (
-     <UserContext.Provider value={{
+     ...state,
+       loginUser, logoutUser }}>
+     <UserContext.Provider value={{
+       ...state,
+       loginUser, logoutUser, getToken }}>
+       {children}
+     </UserContext.Provider>
+   );
+ };
+   export default UserContext;

```

5. Finally, this function needs to be called from **App.js** when the application first renders. That way, you'll get the token once the application starts or is refreshed and the authentication is persisted:

```

import { StatusBar } from 'expo-status-
bar';

```



```

import { FontAwesome } from
 '@expo/vector-icons';
- import React, { useContext } from
 'react';
+ import React, { useContext, useEffect }
 from 'react';
  // ...
  function Navigator() {
-   const { user } =
useContext(UserContext);
+   const { user, getToken } =
      useContext(UserContext);
+   useEffect(() => {
+     getToken();
+   }, []);
    return (
      // ...

```

6. The token is now persisted after logging in once, the application will skip the **Login** screen when it's loaded, and there is a token present in **AsyncStorage**. However, as the token is persisted, we also need a way to log out and remove the token. In the **context/UserContext.js** file, the **logoutUser** function must be altered:

```

// ...

```

```
    async function logoutUser() {  
+   try {  
+     await  
    AsyncStorage.removeItem('token');  
      dispatch({ type: 'REMOVE_USER_TOKEN'  
    });  
+   } catch (e) { }  
    }  
    async function getToken() {  
      // ...
```

When you now go to the **Profile** screen and click the **Logout** button, nothing happens. Unless you reload the application, you can still visit all the different screens. As the token is already removed from **AsyncStorage** and the application state, we need to navigate the user back to the **Login** screen. Navigating between different nested navigators is demonstrated in the next part of this section.

NOTE

*To reload the application in Expo Go, you can shake the device when you're using an iOS or Android phone. By shaking the device, a menu with an option to reload the application will appear. In this menu, you must also select to enable **Fast refresh** to refresh the application automatically when you make changes to the code.*

Navigating between nested routes

In React Navigation, we can nest different navigators, such as the stack navigator that renders when the application starts and shows either the **Login** screen or the tab navigator. From a nested navigator, it isn't possible to navigate to the parent navigator directly, as the **navigation** object for parent navigators cannot be accessed. But luckily, we can use a **ref** to create a reference to the "highest" possible navigator. From this reference, we could then access the **navigation** object, which we otherwise would have accessed using the **useNavigation** Hook. To do this for our application, we need to change the following:

1. Create a new file called **routing.js** with the following content:

```
import React, { createRef } from 'react';  
export const navigationRef = createRef();
```

2. This **navigationRef** can be imported in **App.js** and attached to the **NavigationContainer** in the **App** component:

```
// ...  
import AppContext from  
'./context/AppContext';  
import UserContext from  
'./context/UserContext';  
+ import { navigationRef } from  
'./routing';
```

```

    // ...
function Navigator() {
    const { user, getToken } =
        useContext(UserContext);
    // ...
    return (
-      <NavigationContainer>
+      <NavigationContainer ref=
{navigationRef}>
        // ...

```

3. The **navigation** object for the stack navigator that contains the **Login** screen can now be accessed using this **ref** from the **Profile** screen in **screens/Profile.js**. Using the **reset** method, we can reset the entire **navigation** object and navigate to the **Login** screen:

```

    // ...
+ import { navigationRef } from
'../routing';
export default function Profile() {
    const { logoutUser } =
useContext(UserContext);
    return (
        <View style={styles.container}>

```

```

        <Button
          onPress={() => {
            logoutUser();
+           navigationRef.current.reset({
+             index: 0,
+             routes: [{ name: 'Login' }],
+             });
          }}
          label='Logout'
        />
      </View>
    );
  }
  // ...

```

With the authentication of the user handled, we can continue to add the functionalities to create a new post with an image in the next section.

Using the camera with React Native and Expo

Next to displaying the posts that were already added to the local API, you can also add a post yourself using a **POST** request and send text and an image as variables. Uploading images to your React Native

application can be done by either using the camera to take an image or selecting an image from your camera roll. For both use cases, there are APIs available from React Native and Expo, or numerous packages that are installable from npm. For this project, you'll use the **ImagePicker** API from Expo, which combines these functionalities into just one component.

To add the feature to create new posts to your social media application, the following changes need to be made to create the new screen to add the post:

1. We need to install a library from Expo that allows us to access the camera roll on any device:

```
expo install expo-image-picker
```

2. To use the camera roll, we need to request the **CAMERA_ROLL** permissions from the device, using the **ImagePicker** library we import in the **screens/PostForm.js** file:

```
import React, { useContext, useState }
from 'react';

import { StyleSheet, TouchableOpacity,
View, Text,
      KeyboardAvoidingView, Platform, Alert,
Image } from
      'react-native';
```

```

+ import * as ImagePicker from 'expo-image-
picker';
  // ...
  export default function PostForm() {
    // ...
+   async function uploadImage() {
+     const { status } = await ImagePicker
      .requestMediaLibraryPermissionsAsyn
c();
+     if (status !== 'granted') {
+       Alert.alert('Sorry', 'We need
camera roll
      permissions to make this work!');
+     }
+   }
    return (
      // ...

```

3. This **uploadImage** function must then be added to the **TouchableOpacity** component in this same file:

```

  // ...
  return (
    <KeyboardAvoidingView
      behavior={Platform.OS === 'ios' ?
'padding' :

```

```

        'height'}
        style={styles.container}
    >
    <View style={styles.form}>
        <TouchableOpacity
+           onPress={() => uploadImage()}
            style={styles.imageButton}
        >
            <Text style=
{styles.imageButtonText}>+
            </Text>
        </TouchableOpacity>
        // ...

```

4. When you now press the button to add the post on this screen, a popup asking to give Expo Go permission to access your camera roll will be displayed. Also, note that on this page, we're not using a **View** component to wrap the screen but a **KeyboardAvoidingView** component. This makes sure that the components on this screen won't be hidden behind the keyboard when you're typing something.

NOTE

You can't ask the user for permission a second time; instead, you'd need to manually grant the permission to the camera roll. To set this permission again, you should go to the settings screen on iOS

and select the Expo application. On the next screen, you're able to add permission to access the camera.

5. When the user has granted permission to access the camera roll, you can call the **ImagePicker** API from Expo to open the camera roll. This is again an asynchronous function that takes some configuration fields, such as the aspect ratio. If the user has selected an image, the **ImagePicker** API will return an object containing the field **URI**, which is the URL to the image on the user's device:

```
// ...
async function uploadImage() {
  const { status } = await ImagePicker
    .requestMediaLibraryPermissionsAsync(
);
  if (status !== 'granted') {
    Alert.alert(
      'Sorry',
      'We need camera roll permissions to
make this
      work!',
    );
+   } else {
+     const result =
        await
ImagePicker.launchImageLibraryAsync({
```

```

+      mediaTypes:
+      ImagePicker.MediaTypeOptions.All,
+      allowsEditing: true,
+      aspect: [4, 3],
+      quality: 1,
+    });
+    if (!result.cancelled) {
+      setImageUri(result.uri);
+    }
  }
}
return (
  // ...

```

6. As the URL to the image is now stored in the local state to the **imageUri** constant, you can display this URL in an **Image** component. This **Image** component takes **imageUri** as the value for the source and has been set to use a 100% **width** and **height**:

```

// ...
return (
  <KeyboardAvoidingView
    behavior={Platform.OS == 'ios' ?
'padding' :
    'height'}
    style={styles.container}

```

```

    >
      <View style={styles.form}>
        <TouchableOpacity onPress={() =>
          uploadImage()} style=
{styles.imageButton}>
+          {imageUrl.length ? (
+            <Image
+              source={{ uri: imageUrl }}
+              style={{ width: '100%',
height: '100%'
+                }}
+            />
+          ) : (
            <Text style=
{styles.imageButtonText}>+</Text>
+          )}
        </TouchableOpacity>
      // ...

```

With these changes, the **AddPost** screen should look something like the following screenshots, which were taken from a device running iOS. There might be slight differences in the appearance of this screen if you're using the Android Studio emulator or a device that runs Android:

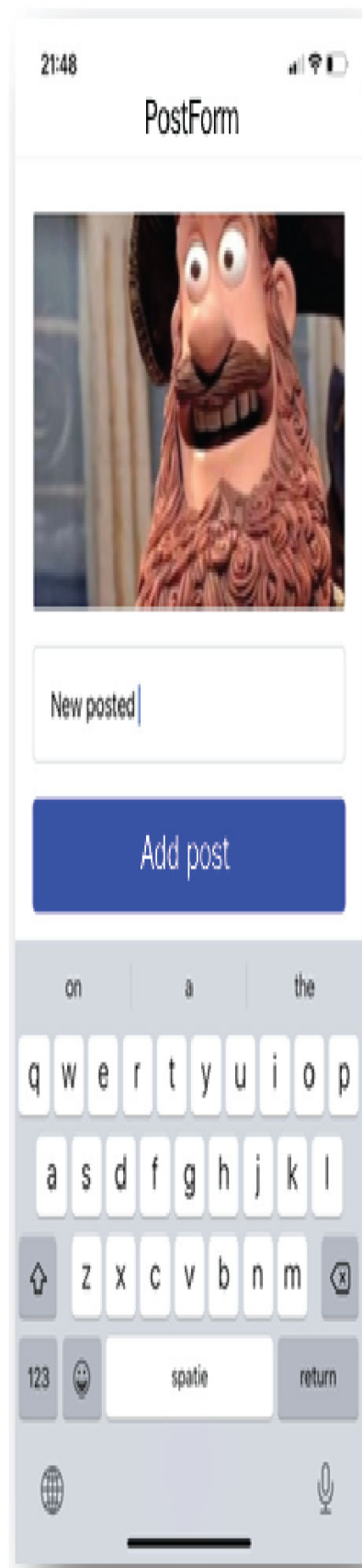
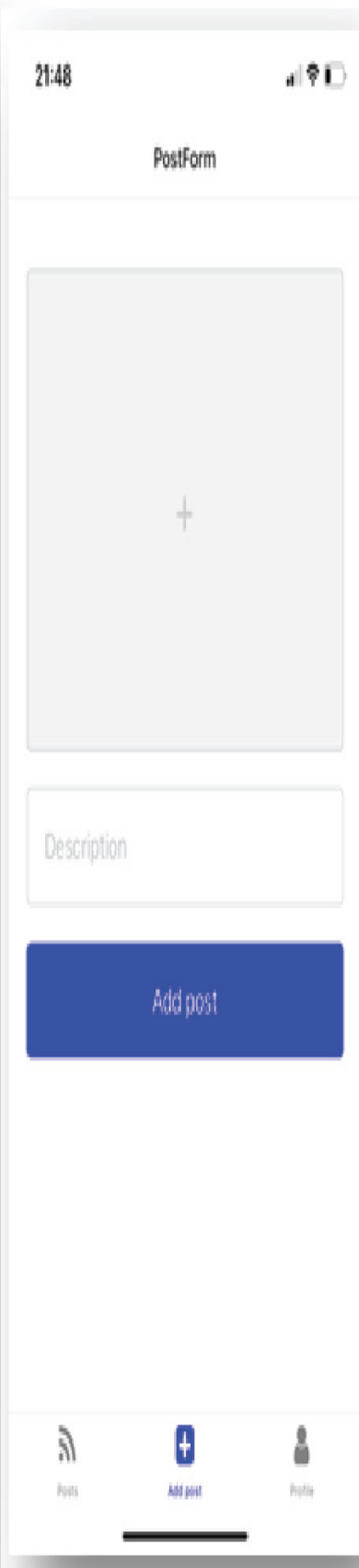
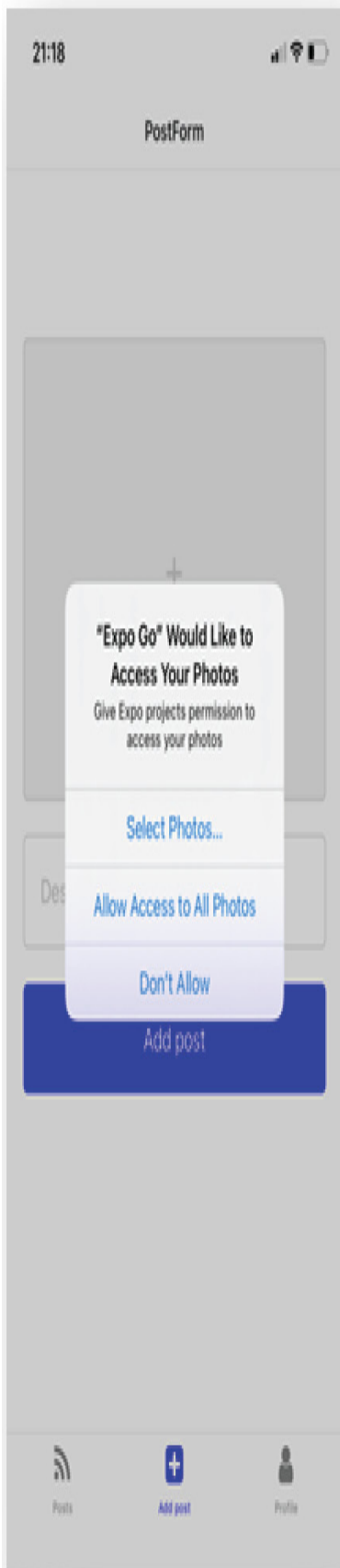


Figure 9.5 – Using the camera roll

These changes will make it possible to select a photo from your camera roll, but your users should also be able to upload an entirely new photo by using their camera. With the **ImagePicker** API from Expo, you can handle both scenarios, as this component also has a **launchCameraAsync** method. This asynchronous function will launch the camera and return it the same way as it returns a URL to the image from the camera roll.

To add the functionality to directly use the camera on the user's device to upload an image, you can make the following changes:

1. When the user clicks on the image placeholder, the image roll will be opened by default. But you also want to give the user the option to use their camera. Therefore, a selection must be made between using the camera or the camera roll for uploading the image, which is a perfect use case for implementing an **ActionSheet** component. React Native and Expo both have an **ActionSheet** component; it's advisable to use the one from Expo as it will use the native **UIActionSheet** component on iOS and a JavaScript implementation for Android:

```
yarn add @expo/react-native-action-sheet
```

2. After this, we need to import **ActionSheetProvider** from **@expo/react-native-action-sheet** in our **App.js** file:

```

import { StatusBar } from 'expo-status-
bar';

import { FontAwesome } from
 '@expo/vector-icons';

import React, { useContext, useEffect }
from 'react';

import { NavigationContainer,
  getFocusedRouteNameFromRoute }
  from '@react-navigation/native';
import { createNativeStackNavigator }
  from '@react-navigation/native-stack';
import { createBottomTabNavigator }
  from '@react-navigation/bottom-tabs';
+ import { ActionSheetProvider }
  from '@expo/react-native-action-sheet';
// ...

```

3. We wrap the navigator that contains the **PostForm** screen in this same file so that we can use the Hook to create the action sheet in that screen component:

```

function Home() {
  return (
+    <ActionSheetProvider>
      // ...
+    </ActionSheetProvider>
  )
}

```

```
    );  
  }  
  function Navigator() {  
    // ...
```

4. In the `screens/PostForm.js` file, we can now import the Hook to create the action sheet from `@expo/react-native-action-sheet`:

```
    // ...  
    import * as ImagePicker from 'expo-image-picker';  
+ import { useActionSheet } from  
    '@expo/react-native-action-sheet';  
    import { useNavigation } from '@react-navigation/core';  
    import Button from  
    '../components/Button';  
    import FormInput from  
    '../components/FormInput';  
    import PostsContext from  
    '../context/PostsContext';  
    export default function PostForm() {  
      // ...
```

5. To add the action sheet, a function to open this **ActionSheet** must be added, and by using the **showActionSheetWithOptions** prop and the options, **ActionSheet** should be constructed. The options

are **Camera**, **Camera roll**, and **Cancel**, and based on the index of the button that gets pressed, a different function should be called:

```
// ...
export default function PostForm() {
  // ...
  const { addPost } =
useContext(PostsContext);
  const navigation = useNavigation();
+  const { showActionSheetWithOptions } =
    useActionSheet();
  // ...
+  function openActionSheet() {
+    const options = ['Camera roll',
'Camera',
    'Cancel'];
+    const cancelButtonIndex = 2;
+    showActionSheetWithOptions(
+      { options, cancelButtonIndex },
+      (buttonIndex) => {
+        if (buttonIndex === 0) {
+          uploadImage()
+        }
+      },
+    );
```



```

+   }
    return (
      // ...

```

6. When the **buttonIndex** is 0, the function to ask for permission to access the camera roll and select an image from it is called, but we also need a function to ask for camera permission and use the camera:

```

    // ...

+   async function takePicture() {
+     const { status } = await
        UIImagePickerController.requestCameraPermissionsAsA
sync();
+     if (status !== 'granted') {
+       Alert.alert('Sorry', 'We need camera
permissions
        to make this work!');
+     } else {
+       const result =
        await UIImagePickerController.
launchCameraAsync ({
+         mediaTypes:
UIImagePickerController.MediaTypeOptions.All,
+         aspect: [4, 3],
+         quality: 1,

```

```

+     });
+     if (!result.cancelled) {
+         setImageUri(result.uri);
+     }
+ }
+ }

function openActionSheet() {
    // ...

```

7. Finally, the **openActionSheet** function to open the action sheet must be attached to the **TouchableOpacity** component:

```

// ...
return (
    <KeyboardAvoidingView
        behavior={Platform.OS == 'ios' ?
'padding' :
        'height'}
        style={styles.container}
    >
        <View style={styles.form}>
            <TouchableOpacity
-                onPress={() => uploadImage()}
+                onPress={() => openActionSheet()}
                style={styles.imageButton}
            >

```

```
// ...
```

Pressing the image placeholder will now open up the action sheet to select whether you want to use the camera roll or the camera for the image:

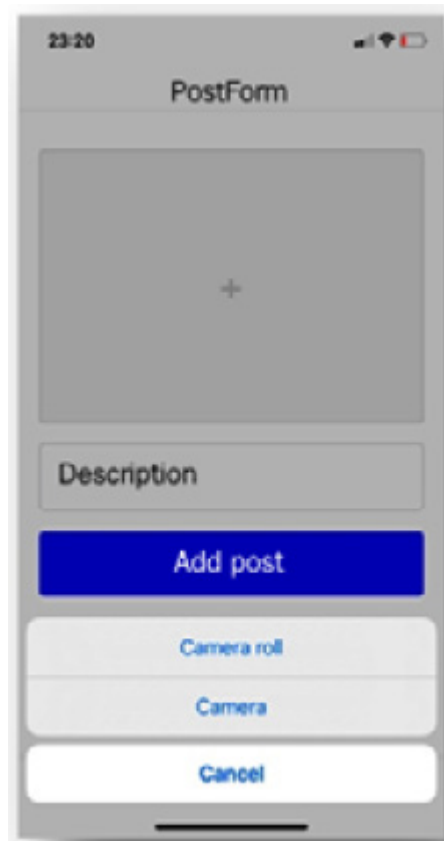


Figure 9.6 – The action sheet on iOS

Your post and image will now be displayed at the top of the **Posts** screen, meaning you've added the post successfully. In the final section of this chapter, we'll be exploring differences in styling between iOS and Android for this application.

Differences in styling for iOS and Android

When styling your application, you might want to have different styling rules for iOS and Android, for example, to match the styling of the Android operating system better. There are multiple ways to apply different styling rules to different platforms; one of them is by using the **Platform** module, which can be imported from React Native.

This module has already been used in parts of this application, but let's have a closer look at its workings by adding different icons to the tabs in the navigator tab depending on the operating system of the device:

1. In **App.js**, we've already imported the **FontAwesome** icons from Expo, but for Android, we want to import **MaterialIcons** so they can be displayed instead. Also, we need to import **Platform** from React Native:

```
import { StatusBar } from 'expo-status-bar';  
import {  
  FontAwesome,  
+  MaterialIcons,  
} from '@expo/vector-icons';  
import React, { useContext, useEffect }  
  from 'react';
```

```
+ import { Platform } from 'react-native';  
  // ...
```

2. With the **Platform** module, you can check whether your mobile device is running iOS or Android by checking whether the value of **Platform.OS** is **ios** or **android**. The module must be used in the tab navigator, where we can make the distinction between the two platforms:

```
  // ...  
  function Home() {  
    return (  
      <ActionSheetProvider>  
        <Tab.Navigator  
          // ...  
          screenOptions={({ route }) => ({  
            tabBarIcon: ({ color, size })  
=> {  
              // ...  
-              return <FontAwesome name=  
{iconName}  
              size={size} color={color}  
/>;  
+              return Platform.OS === 'ios'  
? (  

```

```

+           <FontAwesome name=
{iconName}
           size={size} color={color}
/>
+           ) : (
+           <MaterialIcons name=
{iconName}
           size={size} color={color}
/>
+           );
        },
    )}}
  >
  // ...

```

3. This will replace the **FontAwesome** icons on Android with **MaterialIcons**. This icon library uses different names for the icons, so we also need to make the following change:

```

// ...
function Home() {
  return (
    <ActionSheetProvider>
      <Tab.Navigator
        // ...
        screenOptions={({ route }) => ({

```

```

        tabBarIcon: ({ color, size })
=> {
        const iconName =
-          (route.name === 'Posts' &&
'feed') ||
-          (route.name === 'PostForm'
&&
          'plus-square') ||
-          (route.name === 'Profile'
&& 'user');
+          (route.name === 'Posts' &&
+          (Platform.OS === 'ios' ?
'feed' :
          'rss-feed')) ||
+          (route.name === 'PostForm'
&&
+          (Platform.OS === 'ios' ?
          'plus-square' : 'add-
box')) ||
+          (route.name === 'Profile'
&&
          (Platform.OS === 'ios' ?
'user' :
          'person'));

```

```
        return Platform.OS === 'ios'

? (

        // ...
```

When you're running the application on a mobile device with Android, the navigator tab will display the icons based on Material Design. If you're using an Apple device, it will display different icons; you can change the **Platform.OS === 'ios'** condition to **Platform.OS === 'android'** to add the Material Design icons to iOS instead. If you don't see any changes yet, try reloading the application on your device.

4. We can also use the **Platform** module directly inside a **StyleSheet**, for example, to change the color of the **Button** component in our application. By default, our **Button** component has a blue background color, but let's change it to purple on Android. In **components/Button.js**, we need to import the **Platform** module:

```
import React from 'react';
import {
  StyleSheet,
  TouchableOpacity,
  View,
  Text,
+   Platform,
} from 'react-native';
```



```
export default function Button({ onPress,  
  label }) {  
  // ...
```

5. We use the **select** method inside the creation of **StyleSheet**:

```
  // ...  
  const styles = StyleSheet.create({  
    button: {  
      width: '100%',  
      padding: 20,  
      borderRadius: 5,  
-     backgroundColor: 'blue',  
+     ...Platform.select({  
+       ios: {  
+         backgroundColor: 'blue',  
+       },  
+       android: {  
+         backgroundColor: 'purple',  
+       },  
      })),  
    },  
    // ...
```

Another component that can be styled differently between iOS and Android is the **PostItem** component. As mentioned before, there are multiple ways to do this; besides using the **Platform** module, you can

also use platform-specific file extensions. Any file that has the `*.ios.js` or `*.android.js` extension will only be rendered on the platform specified in the extension. You can not only apply different styling rules but also have changes in functionality on different platforms:

1. Rename the current `components/PostItem.js` file `components/PostItem.android.js`, and create a new file called `components/PostItem.ios.js` with the following contents:

```
import React from 'react';
import { StyleSheet, Text, Dimensions,
Image, View }
  from 'react-native';
const PostItem = ({ data }) => (
  <View style={styles.container}>
    <View style={styles.details}>
      <Text>{data.description}</Text>
    </View>
    <Image source={{ uri: data.imageUrl }}
      style={styles.thumbnail} />
  </View>
);
```

2. This will change the order of the title and the image of a post on iOS, showing the title above the image. Also, we need to add the following styling to the end of this file:

```
// ...
const styles = StyleSheet.create({
  container: {
    display: 'flex',
    alignItems: 'center',
    justifyContent: 'flex-start',
    backgroundColor: 'white',
    borderWidth: 1,
    borderColor: '#ccc',
    marginBottom: '2%',
  },
  thumbnail: {
    width: Dimensions.get('window').width *
0.98,
    height: Dimensions.get('window').width
* 0.98,
    margin: Dimensions.get('window').width
* 0.01,
  },
  details: {
    width: '95%',
    margin: '2%',
  },
});
```

```
export default PostItem;
```

3. Instead of a border around this component on iOS, we want to display a shadow. To add this shadow, we need to alter the styles for the component:

```
// ...
const styles = StyleSheet.create({
  container: {
    display: 'flex',
    alignItems: 'center',
    justifyContent: 'flex-start',
    backgroundColor: 'white',
    - borderWidth: 1,
    - borderColor: '#ccc',
    - marginBottom: '2%',
    + margin: '2%',
    + shadowColor: 'black',
    + shadowOffset: {
    +   width: 0,
    +   height: 2,
    + },
    + shadowOpacity: 0.25,
    + shadowRadius: 4,
    + elevation: 4,
  },
},
```

```
// ...
```

NOTE

*To have a depth of the shadow, iOS will look at the **shadowRadius** rule, while Android uses the **elevation** rule.*

4. Finally, we also need to change the dimensions of the image as we added a margin to the **container** style:

```
// ...
const styles = StyleSheet.create({
  // ...
  thumbnail: {
    - width: Dimensions.get('window').width *
    0.98,
    - height: Dimensions.get('window').width
    * 0.98,
    + width: Dimensions.get('window').width *
    0.94,
    + height: Dimensions.get('window').width
    * 0.94,
    margin: Dimensions.get('window').width
    * 0.01,
  },
});
```

This will have the following result on iOS and Android, where the border has been replaced by a shadow:

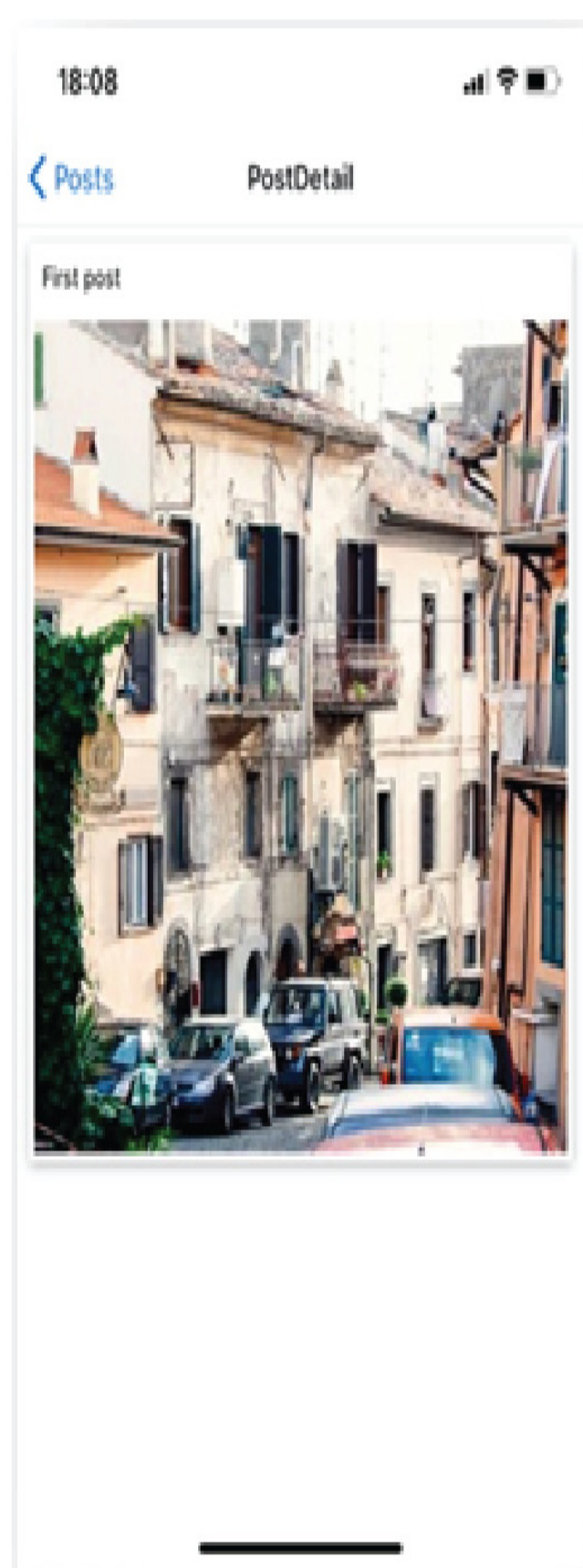
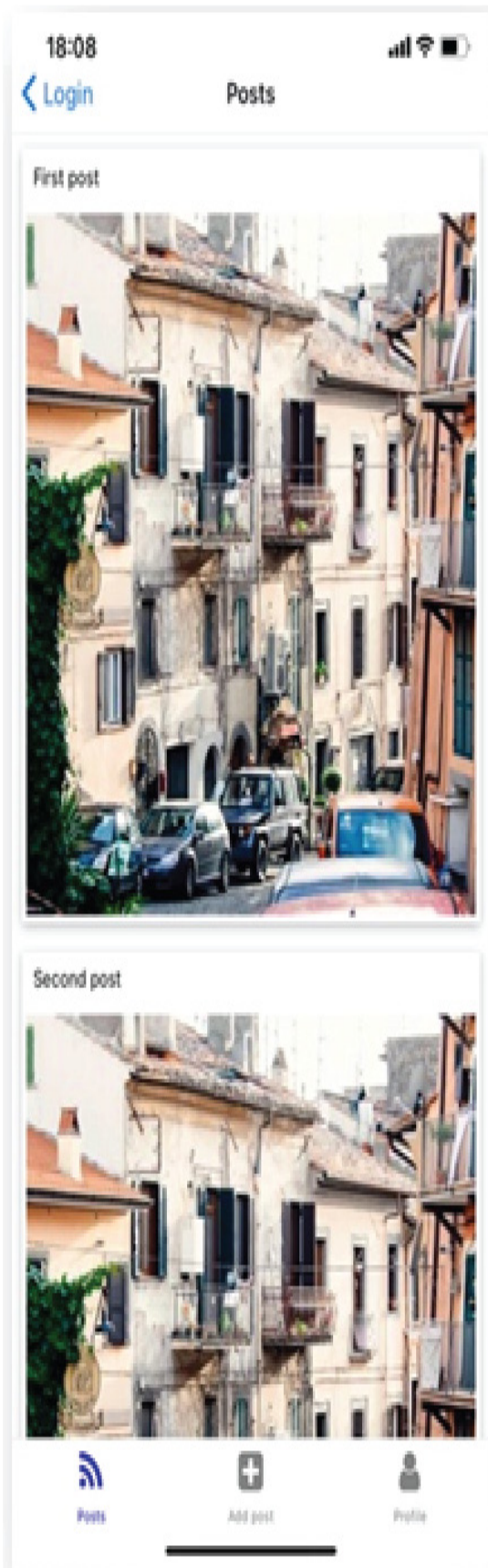


Figure 9.7 – Differences in styling on iOS and Android

Depending on your type of phone you can also rename this file from `components/PostItem.ios.js` to `components/PostItem.android.js` to see the same changes on Android.

That's it. With these final changes, you've created a React Native application that will run on both Android and iOS devices and has differences in styling between these two platforms.

Summary

In this chapter, you've created a mobile social media application with React Native and Expo that uses a local API to send and receive data as well for authentication. To handle authentication, multiple types of navigators are combined. We've learned how to use the camera and the camera roll of a mobile device, after getting the permissions to use them. Also, the differences in styling between iOS and Android were explained.

In completing this social media application, you've completed the final React Native chapter of this book and are now ready to start the very last chapter. In the last chapter, you'll be exploring another use case of React, which is VR. By combining React with Three.js, you can create 360-degree 2D and 3D experiences by writing React components.

Further reading

- Expo camera: <https://docs.expo.io/versions/latest/sdk/camera/>
- Platform-specific code: <https://reactnative.dev/docs/platform-specific-code>

Chapter 10: Creating a Virtual Reality Application with React and Three.js

You're almost there—only one more chapter to go and then you can call yourself a React expert that has experienced React on every platform. Throughout this book, you've built nine applications with React and React Native. In this final chapter, we won't be creating a web or mobile application, but a **Virtual Reality (VR)** application with React and **three.js**. With three.js, you can create dynamic 2D, 3D, and VR experiences using JavaScript, and with the use of another library apply it within React. Although VR is still an emerging technology, the best use cases for VR are, for example, retail stores that want their customers to experience their stores or games online.

In this chapter, you'll explore the very basics of what's possible with React together with three.js and how it relates to React. The application you will build will be able to render 360-degree panorama images and use state management to render between screens. Animated 3D objects will also be displayed by combining React and three.js with other libraries.

The following topics will be covered in this chapter:

- Getting started with three.js
- Creating a panorama viewer with React and three.js
- Animating 3D objects

Project overview

In this chapter, you will build a VR application with React and three.js that uses principles from both JavaScript and React. Both 2D panorama images and 3D objects will be added to this application, and the project can be run in the browser using Create React App.

The build time is 1.5 hours.

Getting started

The application for this chapter will be built from scratch and uses assets that can be found on GitHub at <https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter10-assets>.

These assets should be downloaded to your computer so that you can use them later on in this chapter. The complete code for this chapter can be found on GitHub at <https://github.com/PacktPublishing/React-Projects-Second-Edition/tree/main/Chapter10>.

Creating a VR application with React and Three.js

You can write 2D and 3D VR applications in React by combining it with other libraries. Previously, you could write VR applications in React directly with React 360. But due to the emergence of other popular libraries, such as three.js, which is based on JavaScript, its development was discontinued. Three.js allows you to create applications with both 2D and 3D UI components without having to deal with complex setups for mobile or VR devices, which is similar to how React works.

To render both 2D and 3D in browsers, three.js uses **WebGL**, which is a JavaScript API that runs directly in the browser. It's supported by all recent versions of popular browsers, such as Chrome, Firefox, and Microsoft Edge.

Getting started with Three.js

Three.js is based on JavaScript and can be used together with React using a different library called `@react-three/fiber`, which is a React renderer for three.js that creates a link between the two.

As we already did for our previous React applications that render in the browser, we can use Create React App as the starting point for this application. To get started with building 2D and 3D applications in React with three.js, we first need to create a new project with Create React App:

```
npx create-react-app chapter-10
```

Secondly, we need to install both three.js and **@react-three/fiber** from npm:

```
npm install three @react-three/fiber
```

We don't need any extra dependencies or configuration as Create React App already has the right configuration out of the box. If we move into the project's root directory, which is named after our project name, we will see that it has the following structure:

```
chapter-10
├── /node_modules
├── package.json
├── /public
│   └── index.html
├── /src
│   ├── App.css
│   ├── App.js
│   └── index.css
```

```
| - index.js
```

NOTE

Not all files that were created by Create React App are mentioned above; instead, only the ones used in this chapter are listed.

Creating 3D objects with Three.js

The base for the application has now been created with Create React App, and we've also installed three.js together with `@react-three/fiber`. This last library lets us render three.js elements as components inside React and makes multiple Hooks available to make changes. This way, we can use three.js in the same declarative and predictive way that we're already used to from learning React. There is no additional overhead incurred by using this library instead of three.js directly, as components are rendered outside the render loop of React.

To create 3D objects in React with three.js, we need to take the following steps:

1. Replace the contents of `src/App.js` with the following, so that it will return a `Canvas` component from `@react-three/fiber`:

```
import { Canvas } from '@react-  
three/fiber';  
import './App.css';
```

```
export default function App() {  
  return (  
    <div id="Canvas-container">  
      <Canvas><Canvas />  
    </div>  
  )  
}
```

This code adds a **Canvas** component, which is important when we want to render three.js elements in React for multiple reasons. With the **Canvas** component, both a **scene** and a **camera** are created, the most important building blocks for this application. The scene handles whatever is rendered by three.js, while the camera is the perspective from which we're looking at the scene. This **Canvas** component will render our three.js components and elements outside of the DOM and automatically handles resizing.

2. The **Canvas** component will be resized to fit the **div** element it's rendered in, so you can control its size by changing the width and height of **#canvas-container** in CSS. This can be done by replacing the contents of **src/App.css** with the following:

```
#Canvas-container {  
  height: 100vh;  
}
```

3. To render something on the **Canvas**, we need to add a mesh element to this file, for which we don't need to import anything. The same as we can add **div** or any other elements in React, **three.js** elements will be treated as JSX elements automatically when placed inside a **Canvas** from **@react-three/fiber**:

```
import { Canvas } from '@react-  
three/fiber'  
  
export default function App() {  
  return (  
    <div id="Canvas-container">  
      <Canvas>  
+      <mesh>  
+      <boxGeometry />  
+      </mesh>  
    </Canvas>  
    </div>  
  )  
}
```

NOTE

*When we're adding a **mesh** element inside a **Canvas** component from **@react-three/fiber**, under the hood, it will create a **THREE.Mesh** object.*

4. This will render a small gray square using the **boxGeometry** element from **three.js** but has no additional features yet. Also, the

square we have now is quite small. By adding the **scale** prop to the **mesh** element, we can increase the size of this element:

```
// ...  
export default function App() {  
  return (  
    <div id="Canvas-container">  
      <Canvas>  
-      <mesh>  
+      <mesh scale={2}>  
        <boxGeometry />  
      // ...  
    )  
  )  
}
```

5. To give the element some color, we first need to add a **meshStandardMaterial** with a **color** prop within our mesh element, and add another element called **ambientLight**. This element will add light to the component to make the color of the **boxGeometry** visible. On this **ambientLight** element, we can configure how bright the light must shine using the **intensity** prop:

```
// ...  
export default function App() {  
  return (  
    <div id='canvas-container'>  
      <Canvas>  
        <mesh scale={2}>  
          <boxGeometry />  
        <meshStandardMaterial color='red' />  
      <ambientLight intensity={0.5} />  
    </div>  
  )  
}
```



```

+      <meshStandardMaterial
color='blue' />
+      <ambientLight intensity={0.5}
/>

      </mesh>
    </Canvas>
  </div>
);
}

```

In our application, we can now see a blue square being rendered instead of a gray one. You can see the effect of the **ambientLight** element by changing the intensity to see the square getting lighter or darker depending on the value of the intensity.

Having a 2D square is cool, but with three.js, we can also build 3D components. For this, we need to make some changes to the component to interact with three.js directly outside of React to prevent performance issues. To make the element 3D, make the following changes:

1. Let's create a separate component for the **boxGeometry** element, so we can separate concerns and make it reusable. We can do this in a new file called **Box.js** under a new directory called **components** in our **src** directory:

```

export default function Box() {
  return (
    <mesh scale={2}>
      <boxGeometry />
      <meshStandardMaterial color='blue' />
    </mesh>
  );
}

```

2. We need to add a **ref** to the **mesh** so we can get access to it outside the scope of React, and alter it using **three.js**:

```

+ import { useRef } from 'react';
  export default function Box() {
+   const mesh = useRef();

    return (
      <mesh
        scale={2}
+       ref={mesh}
      >
        <boxGeometry />
        <meshStandardMaterial color='blue'
      />
      </mesh>
    );
  }

```

```
}
```

3. Altering the `mesh` element with `three.js` can be done by changing the values of the `mesh` that we can now access using the `ref`.

These alterations must be done within a `useFrame` Hook from `@react-three/fiber`, which is triggered on every frame render by `three.js`. When a new frame is rendered, we can slightly alter the rotation of the mesh making it rotate:

```
import { useRef } from 'react';
+ import { useFrame } from '@react-
three/fiber';
export default function Box() {
  const mesh = useRef();
+  useFrame(() => {
+    mesh.current.rotation.x =
      mesh.current.rotation.y += 0.01;
+  });
  return (
    // ...
```

NOTE

If you don't want to rotate the `Box` component continuously on every frame render, you can also use a `useEffect` Hook to rotate the mesh just once on the initial render or on a set interval.

4. In the `src/App.js` file, we need to replace the `boxGeometry` element with this new component to make it visible in the application:

```

import { Canvas } from '@react-
three/fiber';
+ import Box from './components/Box';
export default function App() {
  return (
    <div id='canvas-container'>
      <Canvas>
-        <mesh>
-          <boxGeometry />
-          <meshStandardMaterial
color='blue' /> */}
+          <Box />
          <ambientLight intensity={0.5}
/>
-        </mesh>
      </Canvas>
    </div>
  );
}

```

5. Finally, we need to add two more light elements to highlight that we're rendering a 3D element, which are the **spotLight** and **pointLight** elements:

```

// ...
export default function App() {

```

```

    return (
      <div id='canvas-container'>
        <Canvas>
          <Box />
          <ambientLight intensity={0.5} />
+      <spotLight position={[10, 10,
10]}
          angle={0.15} penumbra={1} />
+      <pointLight position={[-10, -10,
-10]} />
        </Canvas>
      </div>
    );
  }

```

NOTE

*The value for **position** is an array with three numbers. These numbers represent a **Vector3** position. This is a format to describe the position of an object in 3D space where the numbers are the x, y, and z values.*

By opening the application in the browser, you can now see a blue square box that is being rotated in 3D:

Figure 10.1 – Rendering a 3D element with Three.js

Something else that we can do with three.js is control the **Canvas** using our mouse. The **Box** component is now rotating on every frame render, but we could also control the rotation of the entire **Canvas** using three.js. The **Canvas** component already sets up a camera that we can control using the **OrbitControls** component from three.js. To do this for our application, we need to do the following:

1. Disable the rotation of the **Box** component by adding a prop called **rotate**, which can be either **true** or **false**. If no value is provided, the default value will be **false**, meaning the **Box** component isn't rotating:

```
// ...  
  
- export default function Box() {  
+ export default function Box({ rotate =  
false }) {  
    const mesh = useRef();  
    useFrame(() => {  
+      if (rotate) {  
        mesh.current.rotation.x =  
        mesh.current.rotation.y += 0.01;  
+      }  
    });  
    return (  
      // ...
```

2. From the **src/App.js** file, we don't need to set this prop as we don't want the **Box** component to rotate. Instead, we will create a new component in the **src/components/Controls.js** file to control the rotation of the entire **Canvas** and thereby the camera of the application. To do this, we need to add the following content to this file:

```
import { useEffect } from 'react';
import { useThree } from '@react-
three/fiber';
import { OrbitControls } from
  'three/examples/jsm/controls/OrbitControl
s';
export default function Controls() {
  const { camera, gl } = useThree();
  useEffect(() => {
    const controls = new
OrbitControls(camera,
    gl.domElement);
    return () => {
      controls.dispose();
    };
  }, [camera, gl]);
  return null;
};
```

This will create the **Controls** component and use the **OrbitControls** component from three.js as its base. From the **useThree** Hook, it will take the **camera** and **gl** from three.js, where the first one is the view perspective and the second one is the render for WebGL. In the **useEffect** Hook, the **OrbitControls** component will be created and also cleaned up with the **dispose** method when it's no longer needed.

3. We need to import this new **Controls** component in the **src/App.js** file and place it inside the **Canvas** component:

```
import { Canvas } from '@react-  
three/fiber';  
import Box from './components/Box';  
+ import Controls from  
  './components/Controls';  
export default function App() {  
  return (  
    <div id='canvas-container'>  
      <Canvas>  
+      <Controls />  
      <Box />  
      // ...  
    </div>  
  );  
}
```

4. With the previous change, we could already rotate the **Box** component while clicking and dragging around this component from the browser. To make this experience smoother, we can add a mini-

mum and maximum control distance to this component in `src/components/Box.js`:

```
// ...
export default function Controls() {
  // ...
  useEffect(() => {
    const controls = new
OrbitControls(camera,
    gl.domElement);
+   controls.minDistance = 2;
+   controls.maxDistance = 20;

    // ...
  }, [camera, gl]);
  return null;
}
```

5. Finally, we can allow the **Controls** component, for example, to zoom or pan. This can be done by setting the following values:

```
// ...
export default function Controls() {
  // ...
  useEffect(() => {
    const controls = new
OrbitControls(camera,
```

```
        gl.domElement);  
        controls.minDistance = 2;  
        controls.maxDistance = 20;  
+       controls.enableZoom = true;  
+       controls.enablePan = true;  
  
        // ...
```

After adding these last two values, you can rotate, zoom, and pan the **Box** component in 3D from the browser with **three.js** and **React**. In the next section of this chapter, we'll be rendering 360-degree panorama images to interact with as well.

Rendering 360-degree panorama images

The application is using a default background that is displayed for the scene, but it's also possible to dynamically change the background of our scene. For this application, we want our scene background to be either 360 degrees or in 3D. Online images can be found on numerous stock photo websites that meet the requirements.

In this book's GitHub repository, you can find a selection of assets for this chapter under the **chapter-10-assets** directory, including two 360-degree panorama images. You need to download both the **beach.jpeg** and **mountain.jpeg** files and place them in the **public** di-

rectory of the application. The file structure for this chapter will therefore become the following:

```
chapter-10
|- /node_modules
|- package.json
|- /public
    |- index.html
    |- beach.jpeg
    |- mountain.jpeg
|- /src
    |- /components
        |- Box.js
        |- Controls.js
    |- App.css
    |- App.js
    |- index.css
    |- index.js
```

After we've added the 360-degree panorama images to the project, we can proceed by rendering them in the background of our scene on the **Canvas**. Using components and Hooks from both **three.js** and **@react-three/fiber**, we can create a 360-degree view in which we can also render the 3D object that we created in the previous section of this chapter.

To add a 360-degree background, we need to follow a couple of steps:

1. Create a new file called **Panorama.js** in the **components** directory in the **src** file of the project. In this file, the setup to create a new mesh with the 360-degree image as a texture is added. First, we need to import the dependencies:

```
import { useLoader } from '@react-three/fiber';  
import * as THREE from 'three';  
import Box from './Box';
```

2. Below the imports, we need to define the backgrounds that we want to use for this application:

```
// ...  
const backgrounds = [  
  {  
    id: 1,  
    url: '/mountain.jpeg',  
  },  
  {  
    id: 2,  
    url: '/beach.jpeg',  
  },  
];
```

3. At the bottom of this file, the actual **Panorama** component must be created, which uses a Hook from **@react-three/fiber** and returns a **mesh** element from **three.js** with two other **three.js** elements:

```
// ...  
export default function Panorama() {  
  const background =  
  useLoader(THREE.TextureLoader,  
    backgrounds[0].url);  
  return (  
    <mesh>  
      <sphereBufferGeometry args={[500, 60,  
40]} />  
      <meshBasicMaterial map={background}  
        side={THREE.BackSide} />  
    </mesh>  
  );  
}
```

The **useLoader** Hook takes the **THREE.TextureLoader** and the background image to create an object that can be used by **meshBasicMaterial** as a texture. It will use the first entry of the **backgrounds** array, something we can make dynamic later on. The **sphereBufferGeometry** defines our 360-degree view within our scene on the **Canvas**.

4. This new **Panorama** component must be imported in **src/App.js** so it can be rendered. Make sure to render this component within a **Suspense** component from React, as it's a dynamic component due to the usage of the **useLoader** Hook:

```
+ import { Suspense } from 'react';
  import { Canvas } from '@react-
three/fiber';
+ import Panorama from
'./components/Panorama';
  // ...
  export default function App() {
    return (
      <div id='Canvas-container'>
        <Canvas>
          <Controls />
+          <Suspense fallback={null}>
+            <Panorama />
+          </Suspense>
          <Box />
          // ...
```

You can now open the application in the browser again to see how the 360-degree panorama is rendered, also with our previously created 3D object:

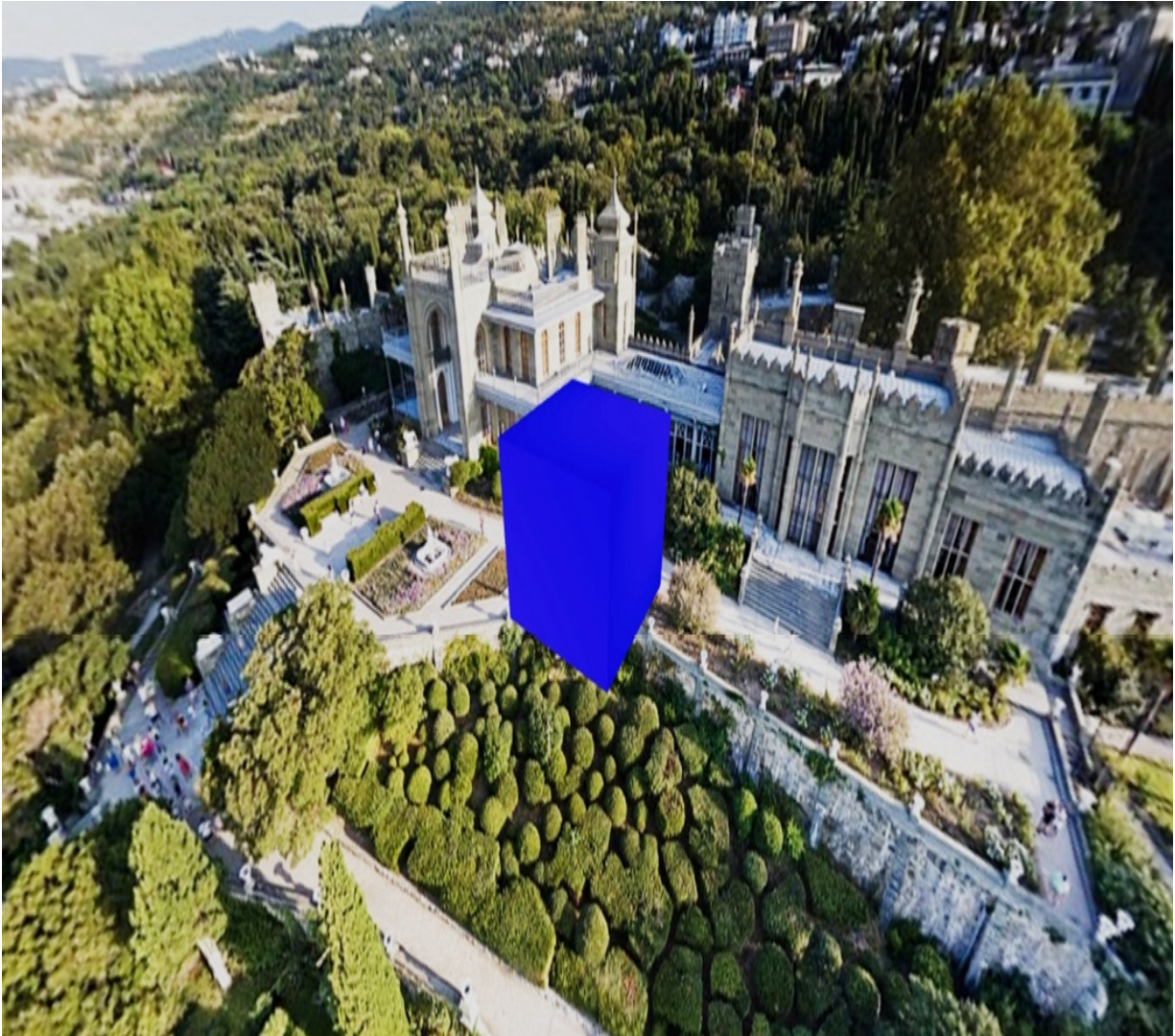


Figure 10.2 – Rendering a 360-degree panorama image

NOTE

*If you try zooming in and out of the 360-degree panorama images, you'll notice that only the **Box** component is changing its size. The background images are set to cover the full background and are not in 3D.*

Besides rendering a 360-degree panorama image, we can also make this interactive. By adding more three.js elements and using React, we can let the user change the background image by clicking on, for example, the 3D box.

To change the backgrounds, we need to combine three.js with React and use local state management to keep track of which 360-degree panorama image should be rendered. Three.js elements rendered by `@react-three/fiber` can also handle `onClick` events to make them clickable components. Let's implement this:

1. In the `src/components/Panorama.js` file, we need to import the `useState` Hook from React, and create a local state variable with it:

```
+ import { useState } from 'react';
  import { useLoader } from '@react-
three/fiber';
  import * as THREE from 'three';
  // ...

  export default function Panorama() {
+   const [activeBackground,
setActiveBackground] =
      useState(1);

  // ...
```


2. Based on the value for **activeBackground**, we can select the 360-degree panorama image that should be rendered as the background. The **id** field of the **backgrounds** array is used to match the local state variable to the correct background:

```
// ...  
export default function Panorama() {  
  const [activeBackground,  
setActiveBackground] =  
    useState(1);  
  
-   const background =  
useLoader(THREE.TextureLoader,  
          backgrounds[0].url);  
+   const { url } = backgrounds.find(({ id  
    }) =>  
      id === activeBackground);  
+   const background = useLoader(  
+     THREE.TextureLoader,  
+     url  
+   );  
  return (  
    // ...
```

3. In the return statement of this **Panorama** component, we need to wrap the returned **mesh** element in a **group** element. This **group** ele-

ment lets three.js return multiple interactive elements at once:

```
// ...
export default function Panorama() {
  // ...
  return (
+    <group>
      <mesh>
        <sphereBufferGeometry args={[500,
60, 40]} />
        <meshBasicMaterial map=
{background}
          side={THREE.BackSide} />
      </mesh>
+    </group>
  );
}
```

4. In this **group** element, we can add another clickable **group** element with the **onClick** event that will update the value for **activeBackground** when clicked on: // ...And add the **group** element with the **onClick** event, which will update the value for **activeBackground** when clicked on:

```
// ...
export default function Panorama() {
  // ...
```

```

    return (
      <group>
        // ...
+      <group
+        onClick={(e) => {
+          e.stopPropagation();
+          setActiveBackground(activeBackg
round ===
                                1 ? 2 : 1);
+        }}
+      >
+        <Box />
+      </group>
    </group>
  );
}

```

5. To prevent the **Box** component from being rendered multiple times, we need to remove it from the **src/App.js** file:

```

import { Suspense } from 'react';
import { Canvas } from '@react-
three/fiber';
- import Box from './components/Box';
// ...
export default function App() {

```

```

    return (
      <div id='Canvas-container'>
        <Canvas>
          <Controls />
-         <Box />
          // ...
        </div>
      );
    }

```

From our application, you can now change the 360-degree panorama image that is being rendered by clicking on the 3D square. We can improve the user experience more by making the `mesh` element interactive, for example, when the user hovers over the `Box`.

6. In the `src/components/Box.js` file, we can add a local state variable to check whether the component is `hovered`, which is triggered from the `mesh` element:

```

- import { useRef } from 'react';
+ import { useRef, useState } from 'react';
  import { useFrame } from '@react-
three/fiber';

  export default function Box({ rotate =
false }) {
    const mesh = useRef();

```

```

+   const [hovered, setHovered] =
useState(false);
    // ...
    return (
      <mesh
        scale={2}
        ref={mesh}
+      onPointerOver={(e) =>
setHovered(true)}
+      onPointerOut={(e) =>
setHovered(false)}
      >
        <boxGeometry />
        <meshStandardMaterial color='blue'
/>
      </mesh>
    );
  }

```

7. When the local state variable **hovered** is **true**, we want the **color** prop on the **meshStandardMaterial** element to change to a different color:

```

    // ...
    return (
      // ...

```

```
        <boxGeometry />
-      <meshStandardMaterial color='blue'
/>
+      <meshStandardMaterial color=
{hovered ?
      'purple' : 'blue'} />
    </mesh>
  );
}
```

If you now open the application on <http://localhost:3000>, you can see the **Box** component changes from blue to purple when hovered over. Clicking on it will render a different 360-degree panorama image, which is the beach:

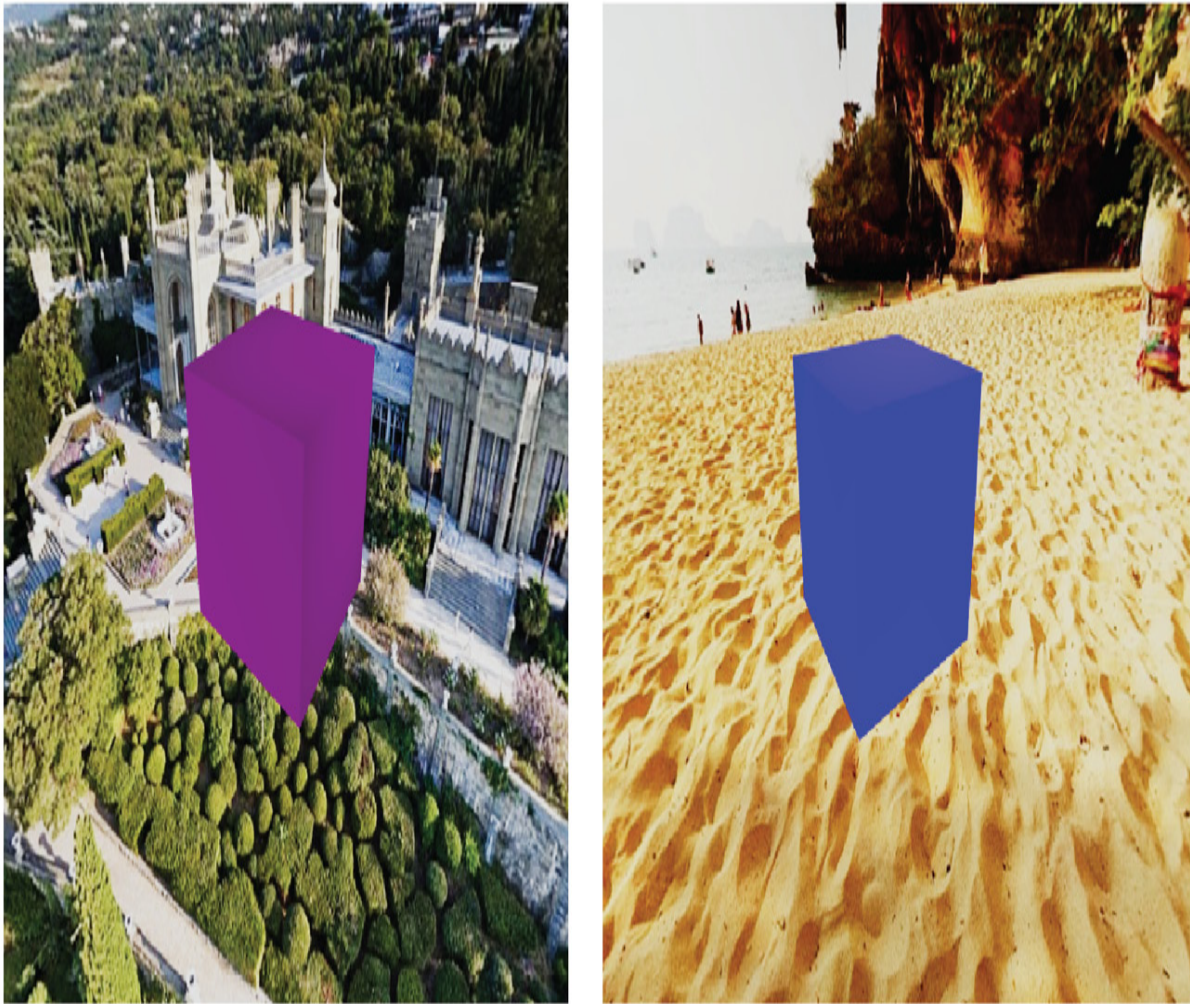


Figure 10.3 – Hovering over and clicking on 3D elements

Besides rendering 360-degree backgrounds and creating interactive 3D components, we can also import external 3D objects in React with `three.js` and animate them. This will be shown in the next section, by adding `react-spring` to our application.

Animating 3D objects

So far, all the components you've added in this chapter that were created with `three.js` didn't have animations. With `three.js`, you can also import external 3D objects and animate components with **`react-spring`**. This library works similar to the Animated API that we used for React Native earlier in this book.

Importing 3D objects

Before getting into animating 3D objects in React, let's import an external 3D object with `three.js` first. `Three.js` can import multiple file formats for 3D objects, including **`.obj`**, **`.gltf`**, and **`.gltb`**. These file formats are the most common ones for creating 3D objects that can be used in other programs. For this chapter, we'll be using a **`.gltb`** file with a 3D version of the Ingenuity Mars Helicopter from NASA. This file can be found in the repository for this book in the **`chapter-10-assets`** directory, and you can place it in the **`public`** directory next to the 360-degree panorama images that you downloaded in the previous section.

Both **`.gltf`** and **`.gltb`** files can be loaded into `three.js` with **`GLTFLoader`**, which can load GLTF objects. GLTF is one of the most popular formats for 3D objects, also called the **JPEG of 3D**. The 3D model from NASA that you've placed in the **`public`** directory can be imported into a component, in a new file called **`Helicopter.js`** in the **`components`** directory:


```

import { useLoader } from '@react-
three/fiber';
import { GLTFLoader } from
  'three/examples/jsm/loaders/GLTFLoader';
export default function Helicopter() {
  const gltf = useLoader(GLTFLoader, './
Ingenuity_v3.glb'
);
  return (
    <group position={[2, 2, 1]}>
      <primitive object={gltf.scene} />
    </group>
  );
}

```

This component again uses the **useLoader** Hook from **@react-three/fiber**, and also imports the **GLTFLoader** that it needs to render the 3D GLTF object. A primitive element with the GLTF object is returned within a **group** element that has a custom position.

In **src/App.js**, we can return this new **Helicopter** component from within a **Suspense** component, as the **useLoader** Hook makes this a dynamic component:

```

import { Suspense } from 'react';

```

```

    import { Canvas } from '@react-
three/fiber';
+ import Helicopter from
'./components/Helicopter';
    // ...
    export default function App() {
      return (
        <div id='Canvas-container'>
          <Canvas>
            // ...
            <Suspense fallback={null}>
              <Panorama />
+              <Helicopter />
            </Suspense>
          </Canvas>
        </div>
      );
    }

```

This will add the Ingenuity Mars Helicopter from NASA to our application, rendered in a position close to our 3D box. You can see what this looks like in the following screenshot:



Figure 10.4 – Rendering an external 3D object with three.js

In the next part of this section, we'll animate this 3D object using a popular React library called `react-spring`.

Animating 3D objects with React

In a previous chapter of this book, we animated components in React Native with the Animated API. For web-based React applications, we

can use another library for this, which is **react-spring**. Using this library, we can add animations to, for example, rotate, move, or fade components in and out of a frame. As has been the case in other examples of using React, this library provides Hooks to add these interactions.

There is a special library from **react-spring** that works well with **@react-three/fiber**, which we can install from npm:

```
npm install @react-spring/three
```

After the installation is complete, we can import the **useSpring** Hook and the **animated** element from this library in our **Helicopter** component in **src/components/Helicopter.js**:

```
import { useLoader } from '@react-  
three/fiber';  
import { GLTFLoader } from  
  'three/examples/jsm/loaders/GLTFLoader';  
+ import { useSpring, animated } from  
  '@react-spring/three';  
export default function Helicopter() {  
  // ...
```

We can pass the configuration for our animation to the **useSpring** Hook, so it will create the props we can pass to the element that we want to animate:

```

// ...
export default function Helicopter() {
  const gltf =
    useLoader(GLTFLoader,
      './Ingenuity_v3.glb');
+   const props = useSpring({
+     loop: true,
+     to: [
+       { position: [2, 2, 3] },
+       { position: [2, 2, 6] },
+       { position: [2, 2, 9] },
+       { position: [2, 4, 9] },
+       { position: [2, 6, 9] },
+     ],
+     from: { position: [2, 2, 1] },
+   });
  return (
    // ...

```

The object with our animation configuration describes that we want to change the **position** prop of our 3D object. The starting position is described and the different positions it should also move to. This animation will also be looped.

The animated element from **react-spring** can then be used to extend the **group** element from **three.js** that is wrapping our 3D object.

This **group** element will become an animated element that also takes the props that were created by the **useSpring** Hook:

```
// ...
return (
-   <group position={[2, 2, 1]}>
+   <animated.group {...props}>
      <primitive object={gltf.scene} />
+   </animated.group>
-   </group>
  );
}
```

Now, when you open the application, the helicopter will be moving around in different positions on the 360-degree panorama image.

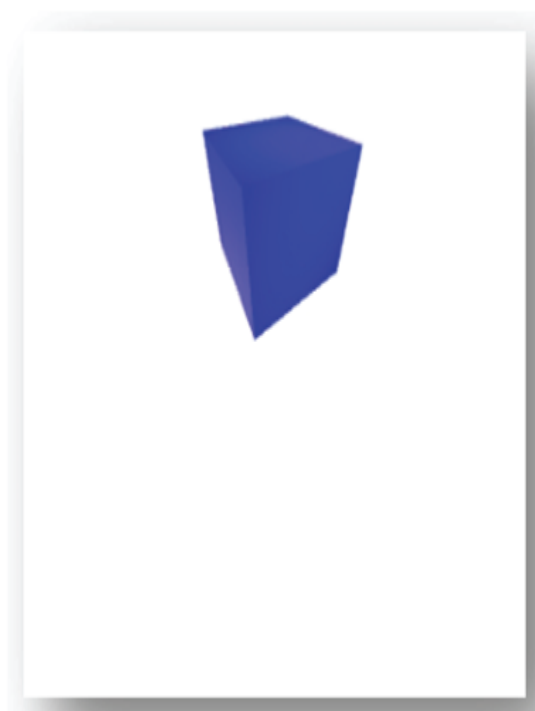
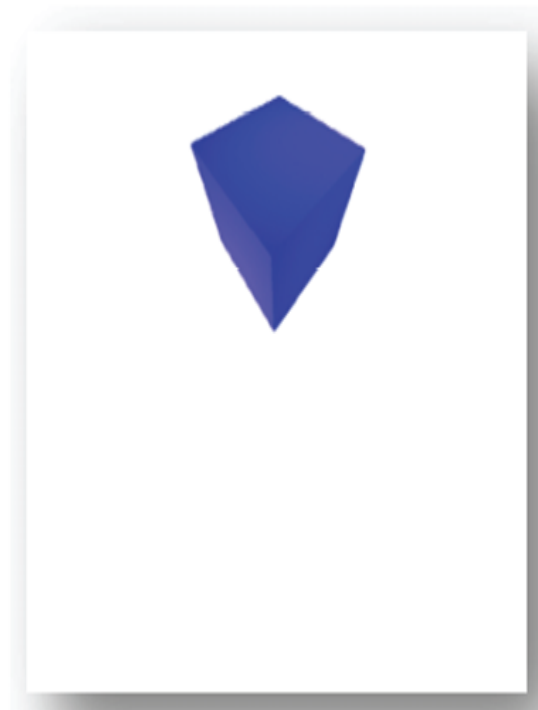
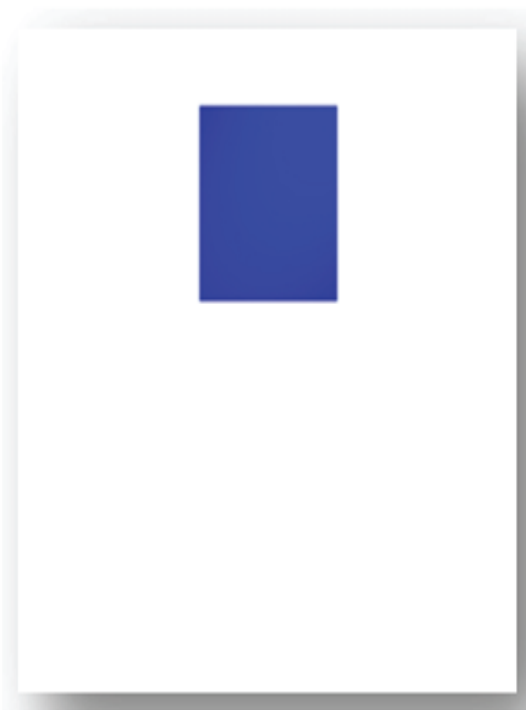
Summary

In this final chapter, you've combined all of the knowledge you have gathered from this book to create a VR application with React. We were able to do this by combining it with `three.js`, which is a JavaScript library for creating 3D applications. The project we created in this chapter serves a different and more niche use case than the other React projects in this book. It has basic animations, as well as a 3D helicopter object that flies away into the distance.

With this final chapter, you've completed all 10 chapters of this book and have created 10 projects with React and React Native. Now, you have a solid understanding of everything that you can do with React and how to use it across different platforms. While React and React Native are already mature libraries, new features are added continuously. Even as you finish reading this book, there will probably be new features you can check out. My main advice would be to never stop learning and keep a close eye on the documentation whenever a new feature is announced.

Further reading

- Three.js: <https://threejs.org/>
- @react-three/fiber: <https://docs.pmnd.rs/react-three-fiber/getting-started/introduction>
- NASA 3D images: <https://mars.nasa.gov/3d/images/>





[Packt.com](https://www.packt.com)

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

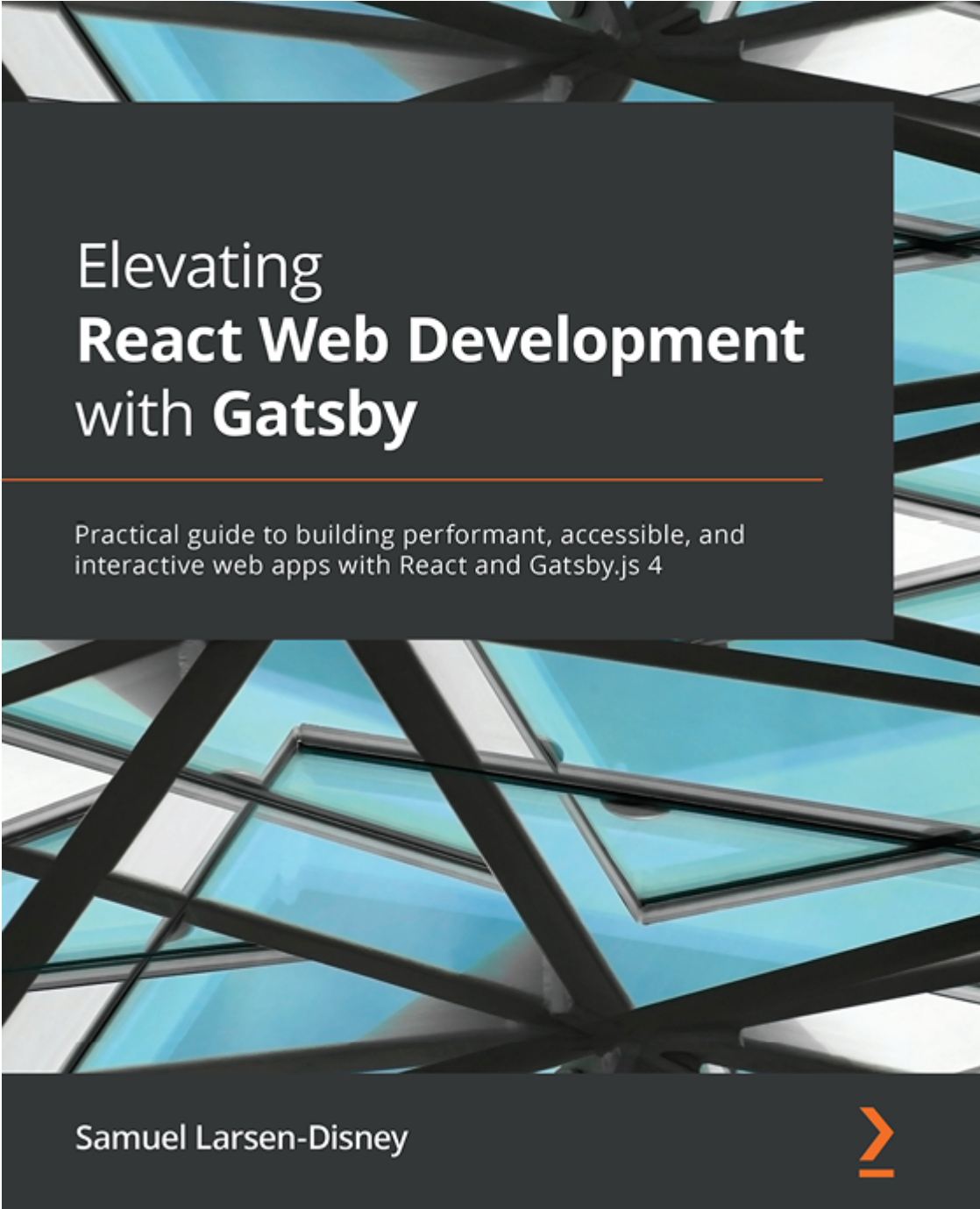
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Elevating **React Web Development** with **Gatsby**

Practical guide to building performant, accessible, and
interactive web apps with React and Gatsby.js 4

Samuel Larsen-Disney



Elevating React Web Development with Gatsby

Samuel Larsen-Disney

ISBN: 978-1-80020-909-1

- Understand what GatsbyJS is, where it excels, and how to use it
- Structure and build a GatsbyJS site with confidence
- Elevate your site with an industry-standard approach to styling
- Configure your GatsbyJS projects with search engine optimization to improve their ranking
- Get to grips with advanced GatsbyJS concepts to create powerful and dynamic sites
- Supercharge your site with translations for a global audience
- Discover how to use third-party services that provide interactivity to users

React 17

Design Patterns and Best Practices

Third Edition

Design, build, and deploy production-ready web applications using industry-standard practices

Carlos Santana Roldán



React 17 Design Patterns and Best Practices – Third Edition

Carlos Santana Roldán

ISBN: 978-1-80056-044-4

- Get to grips with the techniques of styling and optimizing React components
- Create components using the new React Hooks
- Get to grips with the new React Suspense technique and using GraphQL in your projects
- Use server-side rendering to make applications load faster
- Write a comprehensive set of tests to create robust and maintainable code
- Build high-performing applications by optimizing components

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *React Projects*, we'd love to hear your thoughts!

If you purchased the book from Amazon, please [click here to go straight to the Amazon review page](#) for this book and share your feedback or leave a review on the site that you purchased it from.

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